

Introdução ao Pandas: DataFrames e Séries

1. DataFrame

1.2 O que é um DataFrame?

Um **DataFrame** é uma estrutura de dados do Pandas que funciona como uma **tabela**. Ele tem linhas e colunas, e cada coluna pode ter um tipo diferente de dado (números, textos, datas, etc.). Pense nele como uma planilha do Excel, onde você organiza seus dados para análises.

1.2 Características principais:

- **Objeto:** O DataFrame é um **objeto** que faz parte do Pandas, que facilita a manipulação de dados.
- **Facilidade de Manipulação:** Com ele, você pode fazer diversas operações, como:
 - Gerar gráficos
 - Filtrar linhas ou colunas (slicing)
 - Limpar e preparar dados para análise

1.3 Exemplo de Criação de um DataFrame:

```
import pandas as pd
```

```
# Criando um DataFrame a partir de um dicionário
```

```
dados = {  
    'Nome': ['João', 'Maria', 'Pedro'],  
    'Idade': [25, 30, 22],  
    'Cidade': ['Lisboa', 'Porto', 'Braga']  
}
```

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

```
print(df)
```

1.4 Resultado da Tabela no Terminal:

Nome	Idade	Cidade
João	25	Lisboa
Maria	30	Porto
Pedro	22	Braga

2. Operações Comuns com DataFrames

2.1 Slicing: Selecionar Partes do DataFrame

Você pode selecionar partes de um DataFrame, como colunas ou linhas, utilizando o slicing.

Exemplo - Selecionando apenas a coluna "Nome":

```
nomes = df['Nome']  
  
print(nomes)
```

Resultado no Terminal:

```
0  João  
1  Maria  
2  Pedro  
  
Name: Nome, dtype: object
```

3. Importação de Bibliotecas

Para começar a trabalhar com **DataFrames** e **Séries**, é necessário importar as bibliotecas que serão usadas.

```
import pandas as pd # Para trabalhar com DataFrames e Séries
```

```
import numpy as np # Para trabalhar com arrays numéricos e funções matemáticas
```

4. Séries

4.1 O que é uma Série?

Uma **Série** no Pandas é como uma **coluna de uma tabela**. Ela é uma lista de dados com um rótulo (ou índice) para cada valor, assim como em uma tabela onde cada linha tem um identificador.

- **Dados:** Conteúdo da Série (números, textos, etc.).
- **Índice:** O rótulo que identifica cada elemento.

Exemplo de Criação de uma Série:

```
# Criando uma Série com Pandas
```

```
idades = pd.Series([25, 30, 22], index=['João', 'Maria', 'Pedro'])
```

```
print(idades)
```

Resultado no Terminal

```
João    25
```

```
Maria   30
```

```
Pedro   22
```

```
dtype: int64
```

Explicação

- A Série tem os valores [25, 30, 22].
- O índice ou rótulo é ['João', 'Maria', 'Pedro'].

4.2 Operações com Séries

- Acessar dados de uma Série usando o índice:

```
idade_maria = idades['Maria']  
  
print(idade_maria)
```

- **Resultado:** 30
- Fazer operações matemáticas nas Séries:

```
# Somando 1 em todas as idades  
  
idades += 1  
  
print(idades)
```

- **Resultado:**

```
João    26  
Maria   31  
Pedro   23  
  
dtype: int64
```

5. Estruturar Dados com Séries

Exemplo de Criação de uma Série com Valores Ausentes:

```
# Criando uma série com valores e NaN  
  
series = pd.Series([7, 4, 2, np.nan, 6, 9])  
  
print(series)
```

Resultado no Terminal:

```
0    7.0
1    4.0
2    2.0
3    NaN
4    6.0
5    9.0

dtype: float64
```

- **NaN**: Representa valores ausentes ou nulos.
 - A Série é de tipo `float64`, pois contém números decimais.
-

6. Trabalhar com Datas

Para gerar uma sequência de datas, usamos o método `pd.date_range()`. Isso é especialmente útil para séries temporais.

Exemplo de Geração de Datas:

```
# Gerando uma sequência de 6 datas a partir de 1º de janeiro de 2024

datas = pd.date_range('2024-01-01', periods=6)

print(datas)
```

Resultado no Terminal:

```
DatetimeIndex(['2024-01-01', '2024-01-02', '2024-01-03', '2024-01-04',
               '2024-01-05', '2024-01-06'],
              dtype='datetime64[ns]', freq='D')
```

- `pd.date_range()`: Cria uma sequência de datas.
 - `periods=6`: Define o número de datas consecutivas.
-

7. Gerar Dados Aleatórios

Agora vamos criar um **DataFrame** com dados aleatórios. Usamos a função `np.random.randn()` para gerar uma matriz de números aleatórios com 6 linhas e 4 colunas.

Exemplo de Criação de DataFrame com Dados Aleatórios:

```
# Criando um DataFrame com dados aleatórios

df = pd.DataFrame(np.random.randn(6, 4), index=datas, columns=list("ABCD"))

print(df)
```

Resultado no Terminal (exemplo de tabela com números aleatórios):

	A	B	C	D
2024-01-01	0.578161	0.296047	-1.178161	1.422147
2024-01-02	-0.865874	0.239582	-0.464167	0.112958
2024-01-03	-0.347586	-0.146395	-1.031576	0.935436
2024-01-04	0.792681	-0.245680	0.474913	0.267528
2024-01-05	0.295413	1.475845	-0.203665	0.707858
2024-01-06	-0.316275	0.491142	-0.489128	0.118413

- `np.random.randn(6, 4)`: Gera uma matriz de números aleatórios com 6 linhas e 4 colunas.
- `index=datas`: Define o índice do DataFrame como as datas geradas.
- `columns=list("ABCD")`: Nomeia as colunas como 'A', 'B', 'C', 'D'.

8. Verificar Tipos de Dados

Para verificar os tipos de dados de cada coluna em um DataFrame no Pandas, você pode utilizar o método `dtypes`. Isso retornará uma série com o nome de cada coluna e seu respectivo tipo de dados. Veja um exemplo simples:

```
import pandas as pd

import numpy as np

# Criando um DataFrame de exemplo

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [20, 21, 22],
    'Grade': [85.5, 90.2, 95.1],
    'Passed': [True, False, True]
}

df = pd.DataFrame(data)

# Verificando os tipos de dados de cada coluna

print("Tipos de dados de cada coluna:")

print(df.dtypes)
```

Isso produzirá uma saída com os tipos de dados de cada coluna, como:

```
Name    object
```

```
Age      int64
Grade    float64
Passed    bool
dtype: object
```

Esses tipos de dados podem incluir:

- `int64`: números inteiros
- `float64`: números decimais
- `object`: geralmente utilizado para strings (textos)
- `bool`: valores booleanos (True ou False)

Se precisar de uma visão mais detalhada sobre os dados, como contagem de valores nulos e não nulos por tipo, você pode usar o método `info()`:

```
print(df.info())
```

A saída mostrará informações detalhadas sobre o DataFrame, como:

```
<class 'pandas.core.frame.DataFrame'>
```

```
Index: 3 entries, Alice to Charlie
```

```
Data columns (total 4 columns):
```

```
#   Column  Non-Null Count  Dtype
```

```
---  -
```

```
0  Age      3 non-null    int64
```

```
1  Grade    3 non-null    float64
```

```
2  Passed   3 non-null     bool
```

```
3  Name     3 non-null     object
```

```
dtypes: float64(1), int64(1), object(1), bool(1)
```


memory usage: 204.0 bytes

Essa saída fornece uma visão completa sobre os dados, incluindo o uso de memória e informações sobre valores nulos.

9. Verificar Linhas e Colunas em um DataFrame

Para verificar o número de linhas e colunas em um DataFrame usando o Pandas, você pode usar o atributo `shape`. Aqui está um exemplo rápido:

```
# Importando Bibliotecas
```

```
import pandas as pd
```

```
import numpy as np
```

```
# Gerando uma sequência de datas
```

```
datas = pd.date_range('2024-04-04', periods=60, freq='D')
```

```
# Criando um DataFrame com números aleatórios
```

```
df = pd.DataFrame(np.random.randn(60, 5), index=datas, columns=list("ABCDE"))
```

```
# Verificando a forma do DataFrame
```

```
print("Número de linhas e colunas:")
```

```
print(df.shape) # Saída: (60, 5)
```

Explicação

1. **Importando Bibliotecas:** Importamos as bibliotecas necessárias, `pandas` e `numpy`.
2. **Gerando Datas:** Criamos uma faixa de 60 datas diárias começando em 4 de abril de 2024.

3. **Criando o DataFrame:** Um DataFrame é criado com 60 linhas e 5 colunas, preenchido com números aleatórios.
4. **Verificando a Forma:** O atributo `shape` retorna uma tupla que indica o número de linhas e colunas, que neste caso é `(60, 5)`.

Saída

A saída de `df.shape` será:

scss

Copiar código

`(60, 5)`

Isso significa que o DataFrame tem **60 linhas e 5 colunas**.

10. Adicionar Colunas

Adicionar colunas a um DataFrame no Pandas é uma tarefa simples. Você pode fazer isso atribuindo um novo valor a uma nova coluna ou usando métodos como `assign()`. Veja alguns exemplos:

10.1. Adicionando uma Coluna com um Valor Fixo

Você pode adicionar uma coluna ao DataFrame com um valor fixo para todas as linhas:

```
import pandas as pd
```

```
# Criando um DataFrame de exemplo
```

```
data = {  
    'Name': ['Alice', 'Bob', 'Charlie'],  
    'Age': [20, 21, 22]  
}
```

```
df = pd.DataFrame(data)
```

```
# Adicionando uma nova coluna 'City' com um valor fixo
```

```
df['City'] = 'Unknown'

print("DataFrame após adicionar a coluna 'City':")
print(df)
```

Saída:

```
   Name  Age  City
0  Alice   20  Unknown
1   Bob   21  Unknown
2 Charlie   22  Unknown
```

10.2. Adicionando uma Coluna com uma Lista de Valores

Se você tiver uma lista de valores, pode usá-la para adicionar uma nova coluna:

```
# Adicionando uma nova coluna 'Grade' com valores diferentes
df['Grade'] = [85.5, 90.2, 95.1]

print("\nDataFrame após adicionar a coluna 'Grade':")
print(df)
```

Saída:

```
   Name  Age  City  Grade
0  Alice   20  Unknown   85.5
1   Bob   21  Unknown   90.2
2 Charlie   22  Unknown   95.1
```

10.3. Usando o Método `assign()`

Você também pode usar o método `assign()` para adicionar uma nova coluna de forma mais funcional:

```
# Usando o método assign para adicionar a coluna 'Passed'
df = df.assign(Passed=[True, False, True])

print("\nDataFrame após usar assign para adicionar a coluna 'Passed':")
print(df)
```

Saída:

```
   Name Age  City Grade Passed
0  Alice  20 Unknown  85.5   True
1   Bob   21 Unknown  90.2  False
2 Charlie  22 Unknown  95.1   True
```

11. Operações entre Colunas

As operações entre colunas de um DataFrame no Pandas permitem realizar cálculos e manipulações de dados de maneira eficiente. Você pode somar, subtrair, multiplicar e dividir colunas, além de aplicar funções mais complexas. Vamos ver alguns exemplos práticos:

11.1. Soma de Colunas

Você pode somar duas ou mais colunas e armazenar o resultado em uma nova coluna.

```
import pandas as pd

# Criando um DataFrame de exemplo
data = {
    'Math': [85, 90, 95],
    'English': [80, 85, 90]
}

df = pd.DataFrame(data)

# Somando as colunas 'Math' e 'English' para criar uma nova coluna 'Total'
df['Total'] = df['Math'] + df['English']

print("DataFrame após somar as colunas:")
print(df)
```

Saída:

```
   Math English Total
0   85     80   165
```

```
1  90    85  175
2  95    90  185
```

11.2. Subtração de Colunas

Você pode subtrair os valores de uma coluna a partir de outra:

```
# Subtraindo a coluna 'English' da coluna 'Math'
df["Difference"] = df["Math"] - df["English"]

print("\nDataFrame após subtrair as colunas:")
print(df)
```

Saída:

```
   Math  English  Total  Difference
0   85     80    165         5
1   90     85    175         5
2   95     90    185         5
```

11.3. Multiplicação de Colunas

Você pode multiplicar duas colunas para obter um produto:

```
# Multiplicando as colunas 'Math' e 'English'
df["Product"] = df["Math"] * df["English"]

print("\nDataFrame após multiplicar as colunas:")
print(df)
```

Saída:

```
   Math  English  Total  Difference  Product
0   85     80    165         5    6800
1   90     85    175         5    7650
2   95     90    185         5    8550
```

11.4. Divisão de Colunas

Você pode dividir uma coluna pela outra:

```
# Dividindo a coluna 'Math' pela coluna 'English'  
df['Division'] = df['Math'] / df['English']
```

```
print("\nDataFrame após dividir as colunas:")  
print(df)
```

Saída:

	Math	English	Total	Difference	Product	Division
0	85	80	165	5	6800	1.0625
1	90	85	175	5	7650	1.0588
2	95	90	185	5	8550	1.0556

11.5. Aplicando Funções

Você também pode aplicar funções a colunas usando o método `apply()`. Por exemplo, vamos calcular a média das notas:

```
# Calculando a média das notas  
df['Average'] = df[['Math', 'English']].mean(axis=1)  
  
print("\nDataFrame após calcular a média das notas:")  
print(df)
```

Saída:

	Math	English	Total	Difference	Product	Division	Average
0	85	80	165	5	6800	1.0625	82.5
1	90	85	175	5	7650	1.0588	87.5
2	95	90	185	5	8550	1.0556	92.5

12. Editar Valores

Editar valores em um DataFrame do Pandas é uma tarefa comum e essencial para a manipulação de dados. Você pode alterar valores específicos, atualizar uma coluna inteira, ou aplicar condições para modificar os dados. Vamos explorar algumas abordagens para editar valores em um DataFrame.

12.1. Editando Valores Específicos

Para editar um valor específico em um DataFrame, você pode usar o método `.loc[]` ou `.iloc[]`.

12.2. Usando o Método `.iloc[]`

O método `.loc[]` é utilizado para acessar um grupo de linhas e colunas pelo rótulo.

```
import pandas as pd

# Criando um DataFrame de exemplo
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [20, 21, 22],
    'Grade': [85.5, 90.2, 95.1]
}

df = pd.DataFrame(data)

# Editando a idade de Bob
df.loc['Bob', 'Age'] = 44

print("DataFrame após editar a idade de Bob:")
print(df)
```

Saída:

```
   Name  Age  Grade
0  Alice   20   85.5
1   Bob   44   90.2
2 Charlie   22   95.1
```

12.3. Usando o Método `.iloc[]`

O método `.iloc[]` é utilizado para acessar um grupo de linhas e colunas pelo índice posicional. Isso é útil quando você deseja acessar linhas e colunas com base na sua posição numérica.

```
# Editando a idade de Charlie usando .iloc[]
df.iloc[2, 1] = 23 # 2 é o índice da linha de Charlie, 1 é o índice da coluna 'Age'
```

```
print("\nDataFrame após editar a idade de Charlie usando .iloc[2]:")
print(df)
```

Saída:

```
   Name Age Grade
0  Alice  20  85.5
1   Bob   22  90.2
2 Charlie  23  95.1
```

12.4. loc x iloc

- O método `.loc[]` é útil quando você precisa acessar ou modificar dados com base nos rótulos das linhas e colunas. Por exemplo, `df.loc[df["Name"] == 'Bob', 'Age'] = 22` altera a idade de Bob para 22.
- O método `.iloc[]` é útil quando você precisa acessar ou modificar dados com base nas posições numéricas das linhas e colunas. Por exemplo, `df.iloc[2, 1] = 23` altera a idade de Charlie para 23, utilizando o índice posicional.

Ambos os métodos são essenciais para manipulação de dados em um DataFrame e podem ser utilizados conforme a necessidade da análise.

12.5. Perguntas e Respostas sobre Índices em Pandas

Os índices começam a contar por zero?

- Sim, em Pandas, os índices começam a contar a partir de zero. Isso significa que o primeiro elemento tem o índice 0, o segundo tem o índice 1, e assim por diante.

O index pode ser referência em iloc?

- Não, o método `.iloc[]` utiliza a posição numérica dos índices e não os rótulos dos índices. Portanto, ao usar `.iloc[]`, você deve fornecer um número inteiro correspondente à posição da linha, enquanto que `.loc[]` permite acessar os dados usando os rótulos do índice.

O índice label é considerado um valor?

- Sim, o índice label é considerado um valor no contexto do acesso e modificação de dados. Ele identifica de forma única uma linha específica no DataFrame, permitindo que você acesse ou altere dados diretamente usando esses rótulos.

12.5. Atualizando uma Coluna Inteira

Se você quiser atualizar todos os valores de uma coluna, pode fazer isso atribuindo um novo valor diretamente.

```
# Atualizando todas as notas para adicionar 5 pontos
df['Grade'] += 5

print("\nDataFrame após atualizar as notas:")
print(df)
```

Saída:

```
   Name  Age  Grade
0  Alice   20  90.5
1   Bob   22  95.2
2 Charlie   22 100.1
```

12.6. Usando Condições para Editar Valores

Você pode usar condições para editar valores em um DataFrame. Por exemplo, vamos definir um novo valor para as notas abaixo de 90.

```
# Definindo as notas abaixo de 90 como 'Reprovado'
df.loc[df['Grade'] < 90, 'Grade'] = 'Reprovado'

print("\nDataFrame após aplicar condição nas notas:")
print(df)
```

Saída:

```
   Name  Age  Grade
0  Alice   20  Reprovado
1   Bob   22   95.2
2 Charlie   22  100.1
```

12.7. Editando Valores com Funções

Você também pode aplicar funções para modificar valores em uma coluna. Por exemplo, vamos usar uma função para aumentar a idade em 1 ano.

```
# Aumentando a idade de todos em 1 ano
df['Age'] = df['Age'].apply(lambda x: x + 1)

print("\nDataFrame após aumentar a idade em 1 ano:")
print(df)
```

Saída:

```
   Name Age  Grade
0  Alice  21  Reprovado
1   Bob  23    95.2
2 Charlie  23   100.1
```

12.8 Aviso de Incompatibilidade de Tipo de Dados no Pandas

Ao trabalhar com o Pandas, você pode encontrar a seguinte mensagem de aviso:

FutureWarning: Setting an item of incompatible dtype is deprecated and will raise an error in a future version of pandas. Value 'Accept' has dtype incompatible with int64, please explicitly cast to a compatible dtype first.

O que isso significa?

Esse aviso indica que você está tentando atribuir um valor de tipo incompatível a uma coluna do DataFrame. No seu caso, você está tentando atribuir a string `"Accept"` a elementos da coluna `'Grade'`, que contém valores do tipo `int64` (números inteiros). Essa operação não é permitida em versões futuras do Pandas, e seu código poderá falhar.

Como resolver?

Para evitar esse aviso e garantir que seu código funcione corretamente nas futuras versões do Pandas, você pode seguir uma das abordagens abaixo:

Converter a coluna para um tipo de dados compatível: Se você deseja misturar números e strings na coluna, pode converter a coluna `'Grade'` para o tipo `object`, que pode armazenar tanto valores numéricos quanto strings:

```
df['Grade'] = df['Grade'].astype(object)

df.loc[df['Grade'] > 90, 'Grade'] = "Accept"
```

1.

Usar uma nova coluna: Se a coluna 'Grade' deve permanecer apenas com valores numéricos, considere criar uma nova coluna para armazenar os valores de aceitação.

Por exemplo:

```
df['Status'] = "Not Accept"

df.loc[df['Grade'] > 90, 'Status'] = "Accept"
```

13 Visualizando Dataframes

A visualização de Data Frames é uma parte essencial da análise de dados, pois permite que você obtenha uma visão rápida dos dados que está manipulando. O Pandas oferece métodos simples para visualizar as primeiras e as últimas linhas de um DataFrame.

13.1. Visualizar as Primeiras X Linhas

O método `.head(x)` é usado para exibir as primeiras `x` linhas de um DataFrame. Se você não especificar um valor, o padrão será 5.

```
import pandas as pd

# Criando um DataFrame de exemplo
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank'],
    'Age': [20, 21, 22, 23, 24, 25],
    'Grade': [85.5, 90.2, 95.1, 88.5, 92.3, 89.0]
}

df = pd.DataFrame(data)

# Visualizando as primeiras 3 linhas do DataFrame
print("Primeiras 3 linhas do DataFrame:")
print(df.head(3))
```

Saída:

```
   Name Age Grade
0  Alice  20  85.5
1   Bob   21  90.2
2 Charlie  22  95.1
```

13.2. Visualizar as Últimas X Linhas

O método `.tail(x)` é utilizado para exibir as últimas `x` linhas de um `DataFrame`. Assim como no método `.head()`, se você não especificar um valor, o padrão será 5.

```
# Visualizando as últimas 2 linhas do DataFrame
print("\nÚltimas 2 linhas do DataFrame:")
print(df.tail(2))
```

Saída:

```
   Name Age Grade
4   Eva  24  92.3
5  Frank  25  89.0
```

13.3 Obter Nome das Colunas

Para obter os nomes das colunas de um `DataFrame` no Pandas, você pode usar o atributo `.columns`. Isso retorna um objeto do tipo `Index`, que contém os nomes das colunas.

```
import pandas as pd

# Criando um DataFrame de exemplo
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [20, 21, 22],
    'Grade': [85.5, 90.2, 95.1]
}

df = pd.DataFrame(data)

# Obtendo os nomes das colunas
colunas = df.columns
print("Nomes das colunas:")
```

```
print(colunas)
```

Saída:

```
Nomes das colunas:  
Index(['Name', 'Age', 'Grade'], dtype='object')
```

13.4 Obter Apenas Números

Para obter apenas os valores numéricos de um DataFrame, você pode usar o método `.to_numpy()`, que converte o DataFrame em um array NumPy. Isso é útil quando você deseja trabalhar apenas com os dados numéricos sem a estrutura de DataFrame.

```
# Obtendo apenas os valores numéricos como um array NumPy  
numeros = df[['Age', 'Grade']].to_numpy()  
print("\nValores numéricos como array NumPy:")  
print(numeros)
```

Saída:

```
Valores numéricos como array NumPy:  
[[20. 85.5]  
 [21. 90.2]  
 [22. 95.1]]
```

13.5. Inverter Linhas em Colunas

Para inverter linhas em colunas, você pode usar o método `.T` (transposição) do DataFrame. Isso transforma as linhas em colunas e vice-versa.

```
# Invertendo linhas em colunas  
df_transposto = df.T  
print("\nDataFrame transposto (linhas em colunas):")  
print(df_transposto)
```

Saída:

```
      0   1   2  
Name  Alice Bob Charlie
```

```
Age    20  21   22
Grade  85.5 90.2 95.1
```

14. Combinar DataFrames

Combinar DataFrames é uma tarefa comum no Pandas, e você pode fazer isso de várias maneiras, sendo a concatenação uma das mais simples. Vamos explorar como concatenar DataFrames e definir chaves.

14.1. Concatenar DataFrames

Para concatenar Data Frames, você pode usar a função `pd.concat()`. Abaixo, você verá um exemplo básico:

```
import pandas as pd

# Criando DataFrames de exemplo
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

# Concatenando os DataFrames
result = pd.concat([df1, df2])

print("DataFrame Concatenado:")
print(result)
```

Saída:

```
   A  B
0  1  3
1  2  4
0  5  7
1  6  8
```

Neste exemplo, os dois DataFrames (`df1` e `df2`) foram concatenados verticalmente.

14.2. Transformar em Objeto tipo DataFrame

Você pode transformar o resultado da concatenação em um novo DataFrame. A função `pd.concat()` já retorna um DataFrame, mas você pode armazená-lo em uma nova variável, como fizemos acima com `result`.

14.3. Definir Chaves

Se você quiser adicionar uma chave para identificar a origem de cada DataFrame na concatenação, pode usar o parâmetro `keys`:

```
# Concatenando com chaves
result_with_keys = pd.concat([df1, df2], keys=['df1', 'df2'])

print("\nDataFrame Concatenado com Chaves:")
print(result_with_keys)
```

Saída:

```
      A  B
df1 0  1  3
    1  2  4
df2 0  5  7
    1  6  8
```

Neste exemplo, as chaves `'df1'` e `'df2'` foram adicionadas, permitindo identificar facilmente a origem de cada linha no DataFrame resultante.

Resumo

- **Concatenar:** Use `pd.concat([list_of_dataframes])` para combinar DataFrames.
- **Transformar em DataFrame:** O resultado da concatenação é um DataFrame.
- **Definir Chaves:** Use o parâmetro `keys` para adicionar identificadores ao resultado da concatenação.

14.4 Obter elementos de determinado grupo

Para obter elementos de um determinado grupo em um DataFrame concatenado com chaves, você pode usar o método `.loc[]` ou selecionar diretamente os grupos pela chave definida durante a concatenação.

Usando `.loc[]` para acessar o grupo "df1":

```
# Acessando o grupo 'df1'
df1_group = result_with_keys.loc['df1']
print("\nElementos do grupo 'df1':")
print(df1_group)
```

Resultado:

	A	B
0	A0	B0
1	A1	B1
2	A2	B2

Usando `.loc[]` para acessar o grupo "df2":

```
# Acessando o grupo 'df2'
df2_group = result_with_keys.loc['df2']
print("\nElementos do grupo 'df2':")
print(df2_group)
```

Resultado:

	A	B
0	A3	B3
1	A4	B4
2	A5	B5

Explicação:

- **`result_with_keys.loc['df1']`**: Retorna todas as linhas pertencentes ao grupo 'df1', que foi definido durante a concatenação.

- `result_with_keys.loc['df2']`: Da mesma forma, retorna as linhas do grupo 'df2'.

14.5 Obter Elementos por Chave

Para acessar elementos de um DataFrame concatenado com chaves, você pode usar o método `.loc[]`. As chaves são definidas durante a concatenação e permitem segmentar os dados com facilidade.

Exemplo de Concatenar DataFrames com Chaves

```
import pandas as pd

# Criando dois DataFrames de exemplo
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']})
df2 = pd.DataFrame({'A': ['A3', 'A4', 'A5'], 'B': ['B3', 'B4', 'B5']})

# Concatenando e definindo chaves
result_with_keys = pd.concat([df1, df2], keys=['df1', 'df2'])
print("DataFrame concatenado com chaves:")
print(result_with_keys)
```

Acessando Elementos por Chave

Acessar o grupo 'df1':

```
df1_group = result_with_keys.loc['df1']
print("\nElementos do grupo 'df1':")
print(df1_group)
```

Acessar o grupo 'df2':

```
df2_group = result_with_keys.loc['df2']
print("\nElementos do grupo 'df2':")
print(df2_group)
```

Resultado Esperado

DataFrame concatenado com chaves:

		A	B
df1	0	A0	B0
	1	A1	B1
	2	A2	B2
df2	0	A3	B3
	1	A4	B4
	2	A5	B5

Elementos do grupo 'df1':

	A	B
0	A0	B0
1	A1	B1
2	A2	B2

Elementos do grupo 'df2':

	A	B
0	A3	B3
1	A4	B4
2	A5	B5

Explicação

- **Chaves Definidas:** As chaves 'df1' e 'df2' foram definidas na concatenação, permitindo acesso fácil aos grupos.
- **.loc[]:** O método `.loc[]` é utilizado para acessar os elementos correspondentes a uma chave específica no DataFrame.

CÓDIGO

```
import pandas as pd
import numpy as np
```

```
# Creating the data dictionary
data = {
    'Student Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [20, 21, 22],
    'Grade': [85, 90, 95]
```

```
}
```

```
# Creating the DataFrame
```

```
df = pd.DataFrame(data)
```

```
# Creating the Series with indices corresponding to the student names
```

```
city_series = pd.Series(['Lisbon', 'Porto', 'Braga'], index=['Alice', 'Bob', 'Charlie'])
```

```
# Setting "Student Name" as the index of the DataFrame and using inplace=True to replace last  
Dataframe for the new one
```

```
df.set_index('Student Name', inplace=True)
```

```
# Adding the Series to the DataFrame as a new column
```

```
df['City'] = city_series
```

```
# Accessing data from a Serie
```

```
ageAlice = df.loc['Alice', 'Age']
```

```
# Operations using series
```

```
ages = df['Age']
```

```
# Creating serie with missing values
```

```
series = pd.Series([7, 4, 2, np.nan, 6, 9])
```

```
# Generating sequence of dates
```

```
dates = pd.date_range('2024-01-01', periods=7)
```

```
# Generating random date
```

```
randomDf = pd.DataFrame(np.random.randn(7, 4), index=dates, columns=list("ABCD")) # 7  
rows and 4 columns
```

```
# Checking data types
```

```
print(df.dtypes)
```

```
# Checking data types with more details
```

```
print(df.info())
```

```
# Checking rows and columns
```

```
print(df.shape)
```

```
# Add Columns
```

```
df['Gender'] = "Unkonown"
```

```
# Add Columns using value list
```

```
df['Id'] = [1, 2, 3]

# Adding columns using assign() method
df = df.assign(Passed=[True, False, True])

# Operations between Columns
df['Grade * Age'] = df['Grade'] * df['Age']

# Editing specific values using loc
df.loc['Alice', 'Id'] = 33

# Editing specific values using iloc
df.iloc[2, 4] = 33 # Remember that the index starts at ZERO

# Updating an entire column
df['Grade'] = 100

# FutureWarning: Setting an item of incompatible dtype is deprecated and will raise an error in a
future version of pandas.
# Value 'Accept' has dtype incompatible with int64, please explicitly cast to a compatible dtype
first.
# Using conditions to edit values
df['Status'] = df.loc[df['Grade'] > 90, 'Grade'] = "Accept"

# Editing values with functions
df['Age'] = df['Age'].apply(lambda x:x +1)

# Viewing the first few lines
print(df.head(2))

# Viewing the last few lines
print(df.tail(2))

# Get column name
print("\nColumn name")
print(df.columns)

# Get only numbers
numbers = df[['Age','Id']].to_numpy()
print(numbers)

# Flipping columns to rows
dfTranspor = df.T
```

```
# Combine Data Frames
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

resultConcat = pd.concat([df1, df2])

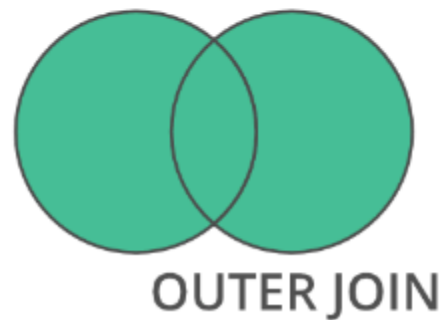
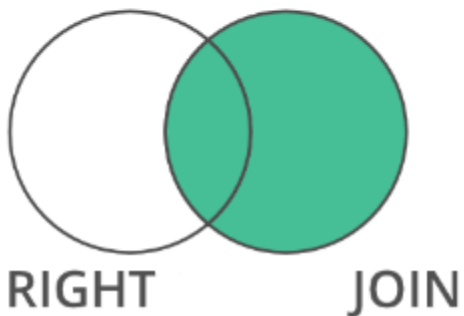
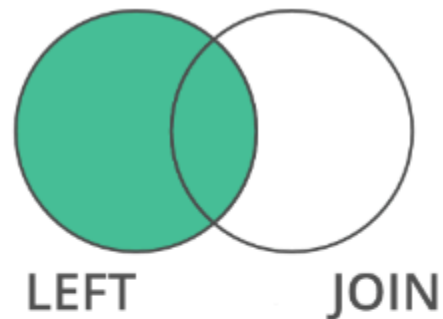
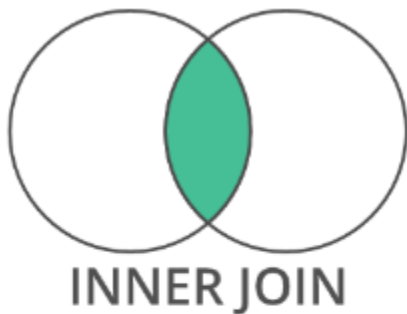
# Combining using keys
resultWithKeys = pd.concat([df1, df2], keys=['df1', 'df2'])

# Getting elements of a group
df1Group = resultWithKeys.loc['df1']
df2Group = resultWithKeys.loc['df2']

# Getting elements of a group for keys
```

Merge

Merge de Dados



Cadastro da Loja A

```
cadastro_a = {'Id': ['AA2930', 'BB4563', 'CC2139', 'DE2521', 'GT3462', 'HH1158'],
              'Nome': ['Victor', 'Amanda', 'Bruna', 'Carlos', 'Ricardo', 'Maria'],
              'Idade': [20, 35, 40, 54, 30, 27],
              'CEP': ['00092-029', '11111-111', '22222-888', '00000-999', '88888-111', '77777-666']}

cadastro_a = pd.DataFrame(cadastro_a, columns = ['Id', 'Nome', 'Idade', 'CEP'])
cadastro_a
```

Cadastro da Loja B

```
cadastro_b = {'Id': ['CC2930', 'EF4563', 'DD2139', 'GT2521', 'HH3462'],
              'Nome': ['Marcos', 'Patrícia', 'Ericka', 'Ricardo', 'Maria'],
              'Idade': [19, 30, 22, 30, 27],
              'CEP': ['00092-029', '11111-111', '22222-888', '00000-999', '88888-111']}

cadastro_b = pd.DataFrame(cadastro_b, columns = ['Id', 'Nome', 'Idade', 'CEP'])
```

Registro de Compras de Todas as Unidades

```
compras = {
    'Id': ['AA2930', 'EF4488', 'CC2139', 'EF4488', 'CC9999', 'AA2930', 'HH1158', 'HH1158'],
    'Data': ['2019-01-01', '2019-01-30', '2019-01-30', '2019-02-01', '2019-02-20', '2019-03-15',
            '2019-04-03', '2019-04-04'],
    'Valor': [200, 100, 40, 150, 300, 25, 50, 500]}

compras = pd.DataFrame(compras, columns = ['Id', 'Data', 'Valor'])
compras
```

INNER JOIN

Interseção entre duas tabelas.

```
pd.merge(tabela_1, tabela_2, on=[PK], how="merge type")
```

Exemplo

Encontrar clientes que frequentam as **duas lojas**.

```
pd.merge(cadastro_a, cadastro_b, on=['Id'], how='inner')
```

	Id	Nome_x	Idade_x	CEP_x	Nome_y	Idade_y	CEP_y
0	GT3462	Ricardo	30	88888-111	Ricardo	30	00000-999
1	HH1158	Maria	27	77777-666	Maria	27	88888-111

Tab I Tab II

	Id	Nome_x	Idade_x	CEP_x	Nome_y	Idade_y	CEP_y
0	GT3462	Ricardo	30	88888-111	Ricardo	30	00000-999
1	HH1158	Maria	27	77777-666	Maria	27	88888-111

Escolher Colunas

```
pd.merge(cadastro_a, cadastro_b[['Id', 'Idade', 'CEP']], on=['Id'], how='inner')
```

	<u>Id</u>	Nome	Idade_x	CEP_x	<u>Idade_y</u>	<u>CEP_y</u>
0	GT3462	Ricardo	30	88888-111	30	00000-999
1	HH1158	Maria	27	77777-666	27	88888-111

Trocar Nomes de Colunas

```
pd.merge(cadastro_a, cadastro_b[['Id', 'Idade', 'CEP']], on=['Id'], how='inner',  
suffixes=('_Cadastro_a', '_Cadastro_b'))
```

	Id	Nome	Idade_Cadastro_a	CEP_Cadastro_a	Idade_Cadastro_b	CEP_Cadastro_b
0	GT3462	Ricardo	30	88888-111	30	00000-999
1	HH1158	Maria	27	77777-666	27	88888-111

FULL JOIN

Junta todos os dados.

Juntar com Duplicatas

```
loja = pd.concat([cadastro_a, cadastro_b], ignore_index=True)
```

	Id	Nome	Idade	CEP
0	AA2930	Victor	20	00092-029
1	BB4563	Amanda	35	11111-111
2	CC2139	Bruna	40	22222-888
3	DE2521	Carlos	54	00000-999
4	GT3462	<u>Ricardo</u>	30	88888-111
5	HH1158	<u>Maria</u>	27	77777-666
6	CC2930	Marcos	19	00092-029
7	EF4563	Patrícia	30	11111-111
8	DD2139	Ericka	22	22222-888
9	GT3462	<u>Ricardo</u>	30	00000-999
10	HH1158	<u>Maria</u>	27	88888-111

Remover Duplicatas

```
singleClients = lojas.drop_duplicates(subset=['Id'])
```

subset

Coluna utilizada como referência, assim gera um subconjunto que **não aceita duplicatas**.

	Id	Nome	Idade	CEP
0	AA2930	Victor	20	00092-029
1	BB4563	Amanda	35	11111-111
2	CC2139	Bruna	40	22222-888
3	DE2521	Carlos	54	00000-999
4	GT3462	Ricardo	30	88888-111
5	HH1158	Maria	27	77777-666
6	CC2930	Marcos	19	00092-029
7	EF4563	Patrícia	30	11111-111
8	DD2139	Ericka	22	22222-888

LEFT JOIN

Todos os dados da **primeira** tabela e a interseção entre elas.

Exemplo

Pegar clientes que **fizeram compra** e estão **cadastrados** na Loja A

cadastro_a				
	Id	Nome	Idade	CEP
0	AA2930	Victor	20	00092-029
1	BB4563	Amanda	35	11111-111
2	CC2139	Bruna	40	22222-888
3	DE2521	Carlos	54	00000-999
4	GT3462	Ricardo	30	88888-111
5	HH1158	Maria	27	77777-666

Compra

compras			
	Id	Data	Valor
0	AA2930	2019-01-01	200
1	EF4488	2019-01-30	100
2	CC2139	2019-01-30	40
3	EF4488	2019-02-01	150
4	CC9999	2019-02-20	300
5	AA2930	2019-03-15	25
6	HH1158	2019-04-03	50
7	HH1158	2019-04-04	500

Left Join

```
pd.merge(cadastro_a, compras, on=['Id'], how='left')
```

	Id	Nome	Idade	CEP	Data	Valor
0	AA2930	Victor	20	00092-029	2019-01-01	200.0
1	AA2930	Victor	20	00092-029	2019-03-15	25.0
2	BB4563	Amanda	35	11111-111	NaN	NaN
3	CC2139	Bruna	40	22222-888	2019-01-30	40.0
4	DE2521	Carlos	54	00000-999	NaN	NaN
5	GT3462	Ricardo	30	88888-111	NaN	NaN
6	HH1158	Maria	27	77777-666	2019-04-03	50.0
7	HH1158	Maria	27	77777-666	2019-04-04	500.0

NaN

Não fizeram compra.

RIGHT JOIN

Todos os dados da **segunda** tabela e a interseção entre elas.

OUTER

Juntar Dataframes e atribuir a apenas **um**.

```
# Outer
```

```
outer = pd.merge(cadastro_a, cadastro_b, on=['Id'], how='outer')
```

```
outer
```

	Id	Nome_x	Idade_x	CEP_x	Nome_y	Idade_y	CEP_y
0	AA2930	Victor	20.0	00092-029	NaN	NaN	NaN
1	BB4563	Amanda	35.0	11111-111	NaN	NaN	NaN
2	CC2139	Bruna	40.0	22222-888	NaN	NaN	NaN
3	DE2521	Carlos	54.0	00000-999	NaN	NaN	NaN
4	GT3462	Ricardo	30.0	88888-111	Ricardo	30.0	00000-999
5	HH1158	Maria	27.0	77777-666	Maria	27.0	88888-111
6	CC2930	NaN	NaN	NaN	Marcos	19.0	00092-029
7	EF4563	NaN	NaN	NaN	Patrícia	30.0	11111-111
8	DD2139	NaN	NaN	NaN	Ericka	22.0	22222-888

Indicator

Argumento que mostra quais elementos estão disponíveis em ambas as tabelas.

cadastro_a				
	Id	Nome	Idade	CEP
0	AA2930	Victor	20	00092-029
1	BB4563	Amanda	35	11111-111
2	CC2139	Bruna	40	22222-888
3	DE2521	Carlos	54	00000-999
4	GT3462	Ricardo	30	88888-111
5	HH1158	Maria	27	77777-666

cadastro_b				
	Id	Nome	Idade	CEP
0	CC2930	Marcos	19	00092-029
1	EF4563	Patrícia	30	11111-111
2	DD2139	Ericka	22	22222-888
3	GT3462	Ricardo	30	00000-999
4	HH1158	Maria	27	88888-111

```
# Outer
```

```
outer = pd.merge(cadastro_a, cadastro_b, on=['Id'], how='outer', indicator=True)
outer
```

	Id	Nome_x	Idade_x	CEP_x	Nome_y	Idade_y	CEP_y	_merge
0	AA2930	Victor	20.0	00092-029	NaN	NaN	NaN	left_only
1	BB4563	Amanda	35.0	11111-111	NaN	NaN	NaN	left_only
2	CC2139	Bruna	40.0	22222-888	NaN	NaN	NaN	left_only
3	DE2521	Carlos	54.0	00000-999	NaN	NaN	NaN	left_only
4	GT3462	Ricardo	30.0	88888-111	Ricardo	30.0	00000-999	both
5	HH1158	Maria	27.0	77777-666	Maria	27.0	88888-111	both
6	CC2930	NaN	NaN	NaN	Marcos	19.0	00092-029	right_only
7	EF4563	NaN	NaN	NaN	Patrícia	30.0	11111-111	right_only
8	DD2139	NaN	NaN	NaN	Ericka	22.0	22222-888	right_only

Group By

Faz a estatística descritiva por grupos, ou seja, uma coluna que tenha **valores repetidos** pode ser agrupada conforme esses valores, enquanto uma coluna, a seguir, que contenha **quantidades**, pode ser **calculada e agrupada** conforme valores da primeira coluna.

```
df = pd.DataFrame({
    'A': ['verdadeiro', 'falso', 'verdadeiro', 'falso', 'verdadeiro', 'falso', 'verdadeiro', 'falso'],
    'B': ['um', 'um', 'dois', 'tres', 'dois', 'dois', 'um', 'tres'],
    'C': np.random.randn(8),
```

```
'D': np.random.randn(8))
df
```

	A	B	C	D
0	verdadeiro	um	-0.436887	-0.109568
1	falso	um	0.726279	1.600136
2	verdadeiro	dois	-1.024609	0.708362
3	falso	tres	-0.166986	0.334238
4	verdadeiro	dois	-1.038773	-2.318863
5	falso	dois	-1.804788	-0.079219
6	verdadeiro	um	-0.167240	-2.258025
7	falso	tres	-0.482972	-0.727926

Uma Coluna

```
df.groupby(['A']).sum()
```

```
df.groupby(['A']).sum()
```

		B	C	D
A				
falso	umtresdoistres	-1.728467	1.127229	
verdadeiro	umdoisdoisum	-2.667509	-3.978094	

Duas Colunas

```
df.groupby(['A','B']).sum()
```

		C	D
A	B		
falso	dois	-1.004833	1.242082
	tres	-1.402096	0.666521
	um	0.322921	1.030595
verdadeiro	dois	-0.490761	-1.071528
	um	-1.778852	-0.549634

Indexações

Indexação

Muito útil para trabalhar com **vetores, arrays, listas, tuplas**.

Vetores

$$\underline{V} = [a_1, a_2, a_3, \dots, a_n]$$

Diagram illustrating vector indexing. The vector V is shown as a list of elements $a_1, a_2, a_3, \dots, a_n$. Below the first three elements, there are handwritten annotations: $[0]$ under a_1 , $[1]$ under a_2 , and $[2]$ under a_3 , each with an upward arrow pointing to the corresponding element. A final upward arrow points to a_n .

Multi Index

Aparentemente, faz combinações com todos os resultados possíveis.

```
arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
pd.MultiIndex.from_arrays(arrays, names=['number', 'color'])
```

```
MultiIndex([(1, 'red'),
            (1, 'blue'),
            (2, 'red'),
            (2, 'blue')],
           names=['number', 'color'])
```

Levels

Valores únicos para a estrutura de Array.

Codes

São a **indexação de localização**, ou seja, a **posição** dos índices na estrutura de dados.

Multi Index com Produto Cartesiano

Produto Cartesiano

Basicamente são todas as combinações possíveis em um plano cartesiano. Para entender melhor, faça uma **multiplicação distributiva** entre os elementos das arrays.

Reshaping de Dados

Reshaping de Dados

Reorganização de dados, transformar dados à maneira que possibilite uma análise mais precisa.

```
datas = pd.date_range('20240404', periods=6)
df = pd.DataFrame(np.random.randn(6, 4), index=datas, columns=['Var_A', 'Var_B', 'Var_C', 'Var_D'])
df
```

	Var_A	Var_B	Var_C	Var_D
2024-04-04	0.268976	-0.815647	1.839551	0.228990
2024-04-05	0.544691	-0.888098	-0.199232	0.177273
2024-04-06	0.789130	0.056554	0.448310	0.403587
2024-04-07	-0.701470	0.686439	1.461771	-0.074972
2024-04-08	-0.866254	1.040389	-0.280291	-0.557860
2024-04-09	-1.338732	0.485034	0.877773	1.697296

Transposed

Atributo que Inverte linhas para colunas e colunas para linhas.

```
dftTransposed= df.T
```

```
dftTransposed
```

	2024-04-04	2024-04-05	2024-04-06	2024-04-07	2024-04-08	2024-04-09
Var_A	0.268976	0.544691	0.789130	-0.701470	-0.866254	-1.338732
Var_B	-0.815647	-0.888098	0.056554	0.686439	1.040389	0.485034
Var_C	1.839551	-0.199232	0.448310	1.461771	-0.280291	0.877773
Var_D	0.228990	0.177273	0.403587	-0.074972	-0.557860	1.697296

Shape

Atributo que todo Dataframe tem, ele verifica o número de Linhas e Colunas.

```
df.shape
```

```
(6, 4)
```

```
dftTransposed.shape
```

```
(4, 6)
```

Values

Atributo que permite extrair valores do Dataframe.

	2024-04-04	2024-04-05	2024-04-06	2024-04-07	2024-04-08	2024-04-09
Var_A	-0.043038	0.777544	-0.053062	0.390784	0.935827	0.865199
Var_B	1.569261	-1.387027	-1.587814	0.558133	0.276006	-0.423601
Var_C	0.010883	0.634325	-0.384465	1.589187	0.445612	-0.014063
Var_D	-1.508182	0.701169	-0.057362	-1.919886	0.029332	0.066220

```
dftTransposed.values
```



```
array([[ -0.04303816,  0.77754435, -0.05306217,  0.39078388,  0.93582726,
         0.86519887],
       [ 1.56926068, -1.38702676, -1.58781373,  0.55813267,  0.2760056 ,
        -0.42360071],
       [ 0.01088287,  0.63432527, -0.3844647 ,  1.5891867 ,  0.44561167,
        -0.01406273],
       [-1.50818236,  0.7011694 , -0.05736214, -1.91988589,  0.02933159,
         0.06621983]])
```

Size

Atributo que calcula o tamanho do Dataframe.

```
dftTransposed.size
```

```
24
```

Reshape

Método que permite reorganizar o Dataframe. Sempre atente-se para o número de linhas e colunas, de modo que seja conforme o números de elementos do Dataframe.

```
v = dftTransposed.values
```

```
v
```

```
array([[ -0.04303816,  0.77754435, -0.05306217,  0.39078388,  0.93582726,
         0.86519887],
       [ 1.56926068, -1.38702676, -1.58781373,  0.55813267,  0.2760056 ,
        -0.42360071],
       [ 0.01088287,  0.63432527, -0.3844647 ,  1.5891867 ,  0.44561167,
        -0.01406273],
       [-1.50818236,  0.7011694 , -0.05736214, -1.91988589,  0.02933159,
         0.06621983]])
```

```
v.reshape((2, 12))
```

```
array([[ -0.04303816,  0.77754435, -0.05306217,  0.39078388,  0.93582726,
         0.86519887,  1.56926068, -1.38702676, -1.58781373,  0.55813267,
         0.2760056 , -0.42360071],
       [ 0.01088287,  0.63432527, -0.3844647 ,  1.5891867 ,  0.44561167,
        -0.01406273, -1.50818236,  0.7011694 , -0.05736214, -1.91988589,
         0.02933159,  0.06621983]])
```

Função Pivot

Função utilizada para manipular, modificar tabelas.

Método Range

Serve para gerar uma sequência de números. Utiliza-se com **for** para iterar sobre uma sequência de números.

```
range(12)
```

```
range(0, 12)
```

```
for i in range(12):
```

```
    print(i)
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
10
```

```
11
```

Método Choice

Pega uma lista e escolhe um elemento de forma aleatória.

```
Pessoa = ['George', 'Victor', 'Lucas']
```

```
np.random.choice(Pessoa)
```

Método Pivot

Serve para transformar a tabela, agrupar dados por categorias e torná-la dinâmica e interativa.

	Dia	Nome	Gasto
0	2019-01-01	Lucas	64.34
1	2019-01-02	Victor	36.36
2	2019-01-03	Victor	14.93
3	2019-01-04	George	45.59
4	2019-01-05	Victor	94.20
5	2019-01-06	Lucas	30.36
6	2019-01-07	Lucas	7.13
7	2019-01-08	Victor	20.21
8	2019-01-09	Lucas	14.42
9	2019-01-10	Victor	74.69
10	2019-01-11	Victor	21.17
11	2019-01-12	George	41.67

Index

Coluna referência da tabela.

Columns

Define quais são os **valores** da tabela antiga que serão usados como **colunas** para organizar os **valores** da tabela nova.

Values

Define os **novos valores** que serão **reorganizados** na nova tabela.

```
df.pivot(index='Dia', columns='Nome', values='Gasto')
```

Nome	George	Lucas	Victor
Dia			
2019-01-01	NaN	64.34	NaN
2019-01-02	NaN	NaN	36.36
2019-01-03	NaN	NaN	14.93
2019-01-04	45.59	NaN	NaN
2019-01-05	NaN	NaN	94.20
2019-01-06	NaN	30.36	NaN
2019-01-07	NaN	7.13	NaN
2019-01-08	NaN	NaN	20.21
2019-01-09	NaN	14.42	NaN
2019-01-10	NaN	NaN	74.69
2019-01-11	NaN	NaN	21.17
2019-01-12	41.67	NaN	NaN

Função Pivot Table

Pivot x Pivot Table

```
Carros = [7, 4, 3, 2, 8]
dias = pd.date_range('20190101', '20190101', periods=5)
vendedor = ['George', 'Vagner', 'Pedro', 'Vagner', 'George']

df = pd.DataFrame({'Vendas': Carros, 'Data': dias, 'Vendedor': vendedor})
df
```

Pivot

A função **Pivot** **não aceita** valores duplicados como **index**.

	Vendas	Data	Vendedor
0	7	2019-01-01	George
1	4	2019-01-01	Vagner
2	3	2019-01-01	Pedro
3	2	2019-01-01	Vagner
4	8	2019-01-01	George

```
pd.pivot(df, index='Data', columns='Vendedor', values='Vendas')
```

```
pd.pivot(df, index='Data', columns='Vendedor', values='Vendas')
-----
ValueError                                Traceback (most recent call last)
<ipython-input-10-4a2fe097b3b1> in <cell line: 1>()
----> 1 pd.pivot(df, index='Data', columns='Vendedor', values='Vendas')

4 frames
/usr/local/lib/python3.10/dist-packages/pandas/core/reshape/reshape.py in _make_selectors(self)
    186
    187     if mask.sum() < len(self.index):
--> 188         raise ValueError("Index contains duplicate entries, cannot reshape")
    189
    190     self.group_index = comp_index

ValueError: Index contains duplicate entries, cannot reshape
```

Pivot Table

A função **Pivot Table** aceita esses valores duplicados no **index**.

```
pd.pivot_table(df, index='Data', columns='Vendedor', values='Vendas')
```

Vendedor	George	Pedro	Vagner
Data			
2019-01-01	7.5	3.0	3.0



É o tipo de **agregamento** padrão da função Pivot Table, que no caso acima é a média.

Trocar o Tipo de Agregamento

aggfunc

Parâmetro da função **Pivot Table**

```
pd.pivot_table(df, index='Data', columns='Vendedor', values='Vendas', aggfunc='sum')
```

Vendedor	George	Pedro	Vagner
Data			
2019-01-01	15	3	6

Stack e Unstack de Dados

Stack significa **empilhar** os valores, basicamente pega todas as colunas e empilha em uma **única coluna**. **Ajuda a visualizar** melhor alguns tipos de Dataframe.

read

Método que lê um data frame:

```
df = pd.read_csv("https://cdncontribute.geeksforgeeks.org/wp-content/uploads/nba.csv")
```

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0
...
453	Shelvin Mack	Utah Jazz	8.0	PG	26.0	6-3	203.0	Butler	2433333.0
454	Raul Neto	Utah Jazz	25.0	PG	24.0	6-1	179.0	NaN	900000.0
455	Tibor Pleiss	Utah Jazz	21.0	C	26.0	7-3	256.0	NaN	2900000.0
456	Jeff Withey	Utah Jazz	24.0	C	26.0	7-0	231.0	Kansas	947276.0
457	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

head

Método que lê as **primeiras linhas** de um Dataframe:

```
df.head(4)
```

```
df.head(4) ?
```

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0

stack

Método que **empilha** os dados em apenas **uma coluna**.

```
stack_df = df.stack()
```

```
stack_df
```

```
0    Name    Avery Bradley
    Team    Boston Celtics
    Number    0.0
    Position    PG
    Age    25.0
    ...
456    Age    26.0
    Height    7-0
    Weight    231.0
    College    Kansas
    Salary    947276.0
Length: 4018, dtype: object
```

unstack

Método que **reverte** o método stack.

```
udf = stack_df.unstack()
```

```
udf
```

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0
...
452	Trey Lyles	Utah Jazz	41.0	PF	20.0	6-10	234.0	Kentucky	2239800.0
453	Shelvin Mack	Utah Jazz	8.0	PG	26.0	6-3	203.0	Butler	2433333.0
454	Raul Neto	Utah Jazz	25.0	PG	24.0	6-1	179.0	NaN	900000.0
455	Tibor Pleiss	Utah Jazz	21.0	C	26.0	7-3	256.0	NaN	2900000.0
456	Jeff Withey	Utah Jazz	24.0	C	26.0	7-0	231.0	Kansas	947276.0

Método Melt

Basicamente, faz um tipo de **fundição**, **reformatação** dos dados.

```
df = pd.DataFrame(
    {
        'A': {0: 'a', 1: 'b', 2: 'c'},
        'B': {0: 1, 1: 3, 2: 5},
        'C': {0: 2, 1: 4, 2: 6}
    }
)
df
```

id_vars

Parâmetro que define a **referência**, **index** da tabela. É a referência para **combinações** possíveis, ou seja, todas as combinações são feitas em volta dessa referência.

values_vars

Parâmetro que define os **valores** da tabela.

Uma Coluna

```
pd.melt(df, id_vars=['A'], value_vars=['B'])
```

	A	variable	value
0	a	B	1
1	b	B	3
2	c	B	5

Duas Colunas

```
pd.melt(df, id_vars=['A'], value_vars=['B', 'C'])
```

	A	variable	value
0	a	B	1
1	b	B	3
2	c	B	5
3	a	C	2
4	b	C	4
5	c	C	6

var_name

Customiza colunas.

```
pd.melt(df, id_vars=['A'], value_vars=['B', 'C'], var_name='VarTeste')
```

	A	VarTeste	value
0	a	B	1
1	b	B	3
2	c	B	5
3	a	C	2
4	b	C	4
5	c	C	6

```
pd.melt(df, id_vars=['A'], value_vars=['B', 'C'], var_name='VarTeste', value_name='Nome do Valor')
```

	A	VarTeste	Nome do Valor
0	a	B	1
1	b	B	3
2	c	B	5
3	a	C	2
4	b	C	4
5	c	C	6

Exemplo Real

```
data = {
    'localizacao': ['CidadeA', 'CidadeB'],
    'temperatura': ['Prevista', 'Atual'],
    'set-2019': [30, 32],
    'out-2019': [45, 43],
    'nov-2019': [24, 22]
}
df = pd.DataFrame(data, columns=['localizacao', 'temperatura', 'set-2019', 'out-2019', 'nov-2019'])
df
```

	localizacao	temperatura	set-2019	out-2019	nov-2019
0	CidadeA	Prevista	30	45	24
1	CidadeB	Atual	32	43	22

```
df2 = pd.melt(df, id_vars=['localizacao', 'temperatura'], var_name='Date', value_name='Valor')
df2
```

	localizacao	temperatura	Date	Valor
0	CidadeA	Prevista	set-2019	30
1	CidadeB	Atual	set-2019	32
2	CidadeA	Prevista	out-2019	45
3	CidadeB	Atual	out-2019	43
4	CidadeA	Prevista	nov-2019	24
5	CidadeB	Atual	nov-2019	22

Seleção de Linhas e Colunas

Seleção de Linhas e Colunas

Selecionar uma Coluna

```
data = pd.date_range('20240404', periods = 600, freq='D')
df = pd.DataFrame(np.random.randn(600, 5), index=data, columns=list('ABCDE'))
```

	A	B	C	D	E
2024-04-04	-0.119895	-0.632230	-0.445982	-0.156526	1.828210
2024-04-05	-1.904037	0.724080	0.943096	-0.183600	-1.951417
2024-04-06	0.114000	-0.082972	-1.027746	0.134998	0.870852
2024-04-07	0.833892	0.009063	-1.595851	0.067835	-0.425049
2024-04-08	-0.700873	0.637205	0.059915	-1.699029	1.200475
...
2025-11-20	0.568745	0.643103	-0.437746	-0.254525	-0.102751
2025-11-21	-0.509273	0.186878	-1.115354	-0.265476	0.102002
2025-11-22	-0.218401	-0.893301	0.506854	0.482486	-0.655852
2025-11-23	1.687144	-0.903900	-0.783969	-0.041586	1.053660
2025-11-24	0.929284	-0.176268	-1.858874	0.402894	-0.172809

600 rows × 5 columns

```
# Just One Column
```

```
df['D']
```

```

2024-04-04    -0.156526
2024-04-05    -0.183600
2024-04-06     0.134998
2024-04-07     0.067835
2024-04-08    -1.699029
...
2025-11-20    -0.254525
2025-11-21    -0.265476
2025-11-22     0.482486
2025-11-23    -0.041586
2025-11-24     0.402894
Freq: D, Name: D, Length: 600, dtype: float64

```

Selecionar Todas as Linhas de Colunas Específicas

```
df.loc[:, ['B', 'C', 'D']]
```

	A	B	C	D	E
2024-04-04	-0.119895	-0.632230	-0.445982	-0.156526	1.828210
2024-04-05	-1.904037	0.724080	0.943096	-0.183600	-1.951417
2024-04-06	0.114000	-0.082972	-1.027746	0.134998	0.870852
2024-04-07	0.833892	0.009063	-1.595851	0.067835	-0.425049
2024-04-08	-0.700873	0.637205	0.059915	-1.699029	1.200475
...
2025-11-20	0.568745	0.643103	-0.437746	-0.254525	-0.102751
2025-11-21	-0.509273	0.186878	-1.115354	-0.265476	0.102002
2025-11-22	-0.218401	-0.893301	0.506854	0.482486	-0.655852
2025-11-23	1.687144	-0.903900	-0.783969	-0.041586	1.053660
2025-11-24	0.929284	-0.176268	-1.858874	0.402894	-0.172809

600 rows x 5 columns

Selecionar Intervalo de Linhas

Lembrando que a **indexação** começa em **zero**

	A	B	C	D	E
0 2024-04-04	-0.119895	-0.632230	-0.445982	-0.156526	1.828210
1 2024-04-05	-1.904037	0.724080	0.943096	-0.183600	-1.951417
2 2024-04-06	0.114000	-0.082972	-1.027746	0.134998	0.870852
3 2024-04-07	0.833892	0.009063	-1.595851	0.067835	-0.425049
4 2024-04-08	-0.700873	0.637205	0.059915	-1.699029	1.200475

```
df[1:5]
```

	A	B	C	D	E
2024-04-05	-1.904037	0.724080	0.943096	-0.183600	-1.951417
2024-04-06	0.114000	-0.082972	-1.027746	0.134998	0.870852
2024-04-07	0.833892	0.009063	-1.595851	0.067835	-0.425049
2024-04-08	-0.700873	0.637205	0.059915	-1.699029	1.200475

Selecionar por Intervalo de Datas

```
# Select for Dates
```

```
df.loc['20240404': '20241017']
```

	A	B	C	D	E
2024-04-04	-0.119895	-0.632230	-0.445982	-0.156526	1.828210
2024-04-05	-1.904037	0.724080	0.943096	-0.183600	-1.951417
2024-04-06	0.114000	-0.082972	-1.027746	0.134998	0.870852
2024-04-07	0.833892	0.009063	-1.595851	0.067835	-0.425049
2024-04-08	-0.700873	0.637205	0.059915	-1.699029	1.200475
...
2024-10-13	0.937643	-2.365223	-0.831370	1.172778	0.371069
2024-10-14	0.960666	-0.690932	-0.504381	0.461184	0.147977
2024-10-15	0.272189	0.417760	-0.955876	-1.497914	-0.143574
2024-10-16	-1.412115	-0.540509	0.577998	0.630402	0.448853
2024-10-17	-0.858036	1.549097	1.570871	0.942870	-0.636240


Selecionar Colunas Específicas por Intervalo de Datas

```
# Select for Dates of some Columns
```

```
df.loc['20240404': '20241017', ['A', 'C', 'E']]
```

	A	C	E
2024-04-04	-0.119895	-0.445982	1.828210
2024-04-05	-1.904037	0.943096	-1.951417
2024-04-06	0.114000	-1.027746	0.870852
2024-04-07	0.833892	-1.595851	-0.425049
2024-04-08	-0.700873	0.059915	1.200475
...
2024-10-13	0.937643	-0.831370	0.371069
2024-10-14	0.960666	-0.504381	0.147977
2024-10-15	0.272189	-0.955876	-0.143574
2024-10-16	-1.412115	0.577998	0.448853
2024-10-17	-0.858036	1.570871	-0.636240

Obter Valores de Uma Linha




	A	B	C	D	E
2024-04-04	-1.195046	-1.594711	-1.010437	0.986140	0.191065
2024-04-05	-0.068435	-0.237879	0.717221	0.632073	-1.771705
2024-04-06	-1.159017	0.522043	0.632084	-0.661912	-0.236849
2024-04-07	-0.897142	-0.290861	0.606141	-1.248135	-0.951267
2024-04-08	-1.135759	0.067661	0.907841	0.185759	-1.483362
...
2025-11-20	-0.896075	-0.190532	-0.273760	0.651769	-0.598784
2025-11-21	0.112284	0.915047	-0.822601	0.208620	-0.673778
2025-11-22	0.897002	0.919017	0.054211	-1.940484	2.582923
2025-11-23	-2.663626	0.387104	-1.051128	0.459156	1.413099
2025-11-24	-0.471499	-0.213823	-1.039168	1.024273	-1.010700
600 rows × 5 columns					

```
df.iloc[1]
```

```
A    -0.068435
B    -0.237879
C     0.717221
D     0.632073
E    -1.771705
Name: 2024-04-05 00:00:00, dtype: float64
```

Obter Valores Seleccionando Células



	A	B	C	D	E
2024-04-04	-1.195046	-1.594711	-1.010437	0.986140	0.191065
2024-04-05	-0.068435	-0.237879	0.717221	0.632073	-1.771705
2024-04-06	-1.159017	0.522043	0.632084	-0.661912	-0.236849
2024-04-07	-0.897142	-0.290861	0.606141	-1.248135	-0.951267
2024-04-08	-1.135759	0.067661	0.907841	0.185759	-1.483362
...
2025-11-20	-0.896075	-0.190532	-0.273760	0.651769	-0.598784
2025-11-21	0.112284	0.915047	-0.822601	0.208620	-0.673778
2025-11-22	0.897002	0.919017	0.054211	-1.940484	2.582923
2025-11-23	-2.663626	0.387104	-1.051128	0.459156	1.413099
2025-11-24	-0.471499	-0.213823	-1.039168	1.024273	-1.010700

600 rows × 5 columns

```
df.iloc[2:4, 0:2]
```

	A	B
2024-04-06	-1.159017	0.522043
2024-04-07	-0.897142	-0.290861

Obter Posições Específicas de um Dataframe

	A	B	C	D	E	
0	2024-04-04	-1.195046	-1.594711	-1.010437	0.986140	0.191065
1	2024-04-05	-0.068435	-0.237879	0.717221	0.632073	-1.771705
2	2024-04-06	-1.159017	0.522043	0.632084	-0.661912	-0.236849
3	2024-04-07	-0.897142	-0.290861	0.606141	-1.248135	-0.951267
4	2024-04-08	-1.135759	0.067661	0.907841	0.185759	-1.483362
5	2024-04-09	-1.504186	-0.066096	-0.187158	1.771984	0.647943
6	2024-04-10	1.215140	-1.183943	0.647133	-1.655093	-1.182784
	2024-04-11	-0.292092	-0.935341	-1.126067	-0.119163	1.060261
	2024-04-12	-1.037691	-1.159713	-0.502351	-0.106402	-1.442331
	2024-04-13	-1.807584	0.024583	0.411324	1.749290	0.470552

```
# Get specific positions
```

```
df.iloc[[1, 5, 6], [0, 3]]
```

	A	D
2024-04-05	-0.068435	0.632073
2024-04-09	-1.504186	1.771984
2024-04-10	1.215140	-1.655093

Obter valores de intervalo de linhas e todas as colunas

```
# Get row interval and all columns
```

```
df.iloc[1:3, :]
```

	A	B	C	D	E
2024-04-05	-0.068435	-0.237879	0.717221	0.632073	-1.771705
2024-04-06	-1.159017	0.522043	0.632084	-0.661912	-0.236849

Filtros Booleanos

Outra Forma de Obter Dados por Coluna

df.A

	A	B	C	D	E
2024-04-04	-1.195046	-1.594711	-1.010437	0.986140	0.191065
2024-04-05	-0.068435	-0.237879	0.717221	0.632073	-1.771705
2024-04-06	-1.159017	0.522043	0.632084	-0.661912	-0.236849

Próximas etapas: [Gerar código com df](#) [Ver gráficos recomendados](#)

Booleans
df.A

```
2024-04-04    -1.195046
2024-04-05    -0.068435
2024-04-06    -1.159017
2024-04-07    -0.897142
2024-04-08    -1.135759
...
2025-11-20    -0.896075
2025-11-21     0.112284
2025-11-22     0.897002
2025-11-23    -2.663626
2025-11-24    -0.471499
Freq: D, Name: A, Length: 600, dtype: float64
```

Condição para todo o DF

```
# All DF just Positive Values
df[df > 0]
```

	A	B	C	D	E
2024-04-04	NaN	NaN	NaN	0.986140	0.191065
2024-04-05	NaN	NaN	0.717221	0.632073	NaN
2024-04-06	NaN	0.522043	0.632084	NaN	NaN
2024-04-07	NaN	NaN	0.606141	NaN	NaN
2024-04-08	NaN	0.067661	0.907841	0.185759	NaN
...
2025-11-20	NaN	NaN	NaN	0.651769	NaN
2025-11-21	0.112284	0.915047	NaN	0.208620	NaN
2025-11-22	0.897002	0.919017	0.054211	NaN	2.582923
2025-11-23	NaN	0.387104	NaN	0.459156	1.413099
2025-11-24	NaN	NaN	NaN	1.024273	NaN

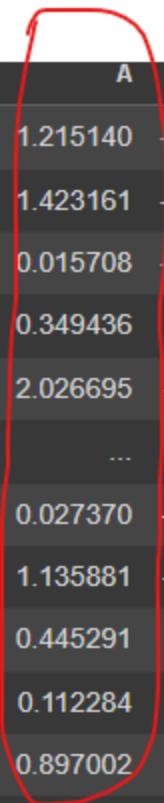
Condição Por Coluna

Todas as Colunas por Resultado

Serão selecionadas **todas as linhas** de **todo o Dataframe** que atendam a condição:

```
# Condition for Column
```

```
df[df.A > 0]
```



	A	B	C	D	E
2024-04-10	1.215140	-1.183943	0.647133	-1.655093	-1.182784
2024-04-14	1.423161	-1.323452	-0.128855	1.042204	-0.607324
2024-04-17	0.015708	-0.526147	0.654818	-0.715164	-2.354937
2024-04-20	0.349436	0.434188	1.783528	-1.175023	-0.892859
2024-04-22	2.026695	-0.538293	-1.464009	-0.736399	0.405315
...
2025-11-17	0.027370	-0.881654	-0.523321	0.308872	1.059784
2025-11-18	1.135881	-0.114435	-0.319849	1.288983	-1.136965
2025-11-19	0.445291	0.785876	-0.451441	-0.434800	-0.543921
2025-11-21	0.112284	0.915047	-0.822601	0.208620	-0.673778
2025-11-22	0.897002	0.919017	0.054211	-1.940484	2.582923

Limpeza de Dados

Data Wrangling

Estudo relacionado com a **Limpeza e Estruturação** de Dados. Depois de efetuar a limpeza necessária, é necessário que haja a estruturação adequada para serem efetuadas as análises.

Big Data

É um grande volume de dados que não pode ser processado por métodos tradicionais devido à sua quantidade, variedade e velocidade. Forma de obtenção de dados fundamentada em 4 V's:

Volume

Volume grande de dados.

Velocidade

Gerados em uma velocidade alta.

Variedade

Diversos tipos de dados.

Veracidade

Qualidade e confiabilidade dos dados.

IOT

Internet of Things, plataformas que estão relacionadas a internet.

Entender os Dados

- Saber o Objetivo,
- Transformar em estruturas adequadas para análise,
- Salvar Arquivos e
- Separar Diretórios.

Fluxo Completo

Dado -> Estrutura -> Arquivo

Pipeline

É uma **sequência de passos que os dados seguem desde a coleta até a análise**. Ele pode incluir a **extração dos dados, limpeza, transformação, carregamento e análise**. Cada etapa do pipeline **prepara os dados** para a próxima, garantindo que sejam processados de forma eficiente e organizada.

Sumarizando Dados

Entender como estão estruturados os dados obtidos.

Descobrir Tipos de Dados em um Data Frame

Somente é possível realizar uma boa limpeza e estruturação, depois de identificar os tipos de dados.

dtype

```
[8] df
```



	Var_A	Var_A	Var_C	Var_D
2020-01-01	-1.599216	-1.488703	0.772840	0.453271
2020-01-02	0.326214	0.154869	-0.553377	0.856565
2020-01-03	0.398718	0.151803	-1.051645	0.698003
2020-01-04	-1.448266	-2.304505	-0.765897	1.062226
2020-01-05	-0.055839	1.159943	-0.511918	-0.716154
2020-01-06	-1.528779	-0.196023	-0.371666	0.225639

```
df.dtypes
```

```
Var_A    float64
Var_A    float64
Var_C    float64
Var_D    float64
dtype: object
```

```
[10] df2
```



	A	B	C	D	E
0	1.0	2013-01-02	1.0	3	Python
1	1.0	2013-01-02	1.0	3	Python
2	1.0	2013-01-02	1.0	3	Python
3	1.0	2013-01-02	1.0	3	Python

Próximas etapas:

[Gerar código com df2](#)

```
[14] df2.dtypes
```



```
A          float64
B    datetime64[ns]
C          float32
D           int32
E           object
dtype: object
```

Sumarização

Resumir os dados.

describe()

É feito apenas em **dados numéricos**.

df2

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	Python
1	1.0	2013-01-02	1.0	3	train	Python
2	1.0	2013-01-02	1.0	3	test	Python
3	1.0	2013-01-02	1.0	3	train	Python

df2.describe()

	A	B	C	D
count	4.0	4	4.0	4.0
mean	1.0	2013-01-02 00:00:00	1.0	3.0
min	1.0	2013-01-02 00:00:00	1.0	3.0
25%	1.0	2013-01-02 00:00:00	1.0	3.0
50%	1.0	2013-01-02 00:00:00	1.0	3.0
75%	1.0	2013-01-02 00:00:00	1.0	3.0
max	1.0	2013-01-02 00:00:00	1.0	3.0
std	0.0	NaN	0.0	0.0

Count

Contagem do Número de Linhas.

Mean

Média.

Min

Valor mínimo.

25%, 50%, 75%

Estatística Descritiva que dá inúmeras ideias quanto ao estado dos dados em cada parte do Dataframe. Como por exemplo: Assimetria, média, etc.

Max

Valor Máximo.

Std

Desvio Padrão

****REFAZER Reindexação**

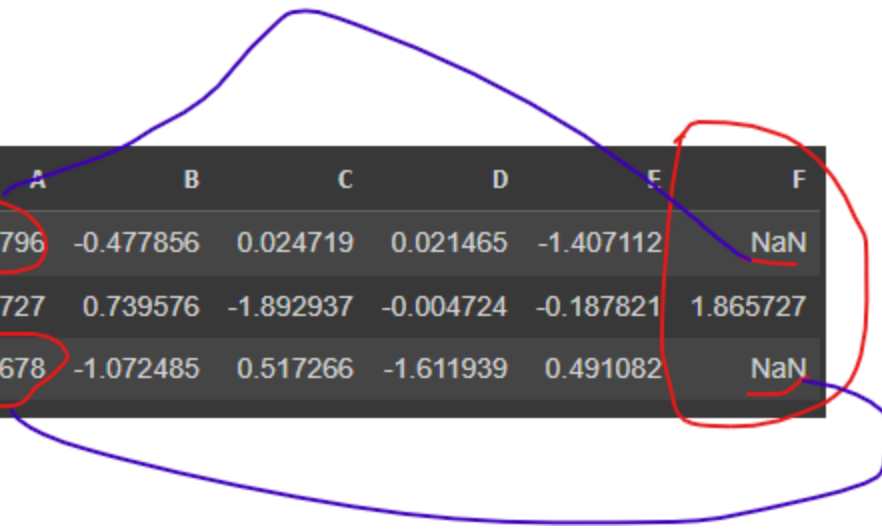
7:00

Dados Missing - NaN

```
datas = pd.date_range('20190101', periods = 60, freq="D")
df = pd.DataFrame(np.random.randn(60, 5), index=datas, columns=list('ABCDE'))
df
```

	A	B	C	D	E
2019-01-01	-0.169796	-0.477856	0.024719	0.021465	-1.407112
2019-01-02	1.865727	0.739576	-1.892937	-0.004724	-0.187821
2019-01-03	-0.219678	-1.072485	0.517266	-1.611939	0.491082
2019-01-04	0.347446	-0.821685	1.591727	-0.915793	-0.102149

```
df['F'] = df.A[df.A > 0]
```

	A	B	C	D	E	F
2019-01-01	-0.169796	-0.477856	0.024719	0.021465	-1.407112	NaN
2019-01-02	1.865727	0.739576	-1.892937	-0.004724	-0.187821	1.865727
2019-01-03	-0.219678	-1.072485	0.517266	-1.611939	0.491082	NaN

Corrigindo

Há maneiras de corrigir isso.

Remover Linhas NaN

Dropna

Método para **remover valores NaN**, mas também remove as **linhas**.

```
df2 = df.copy()
df2
```

	A	B	C	D	E	F
2019-01-01	-0.169796	-0.477856	0.024719	0.021465	-1.407112	NaN
2019-01-02	1.865727	0.739576	-1.892937	-0.004724	-0.187821	1.865727
2019-01-03	-0.219678	-1.072485	0.517266	-1.611939	0.491082	NaN
2019-01-04	0.347446	-0.821685	1.591727	-0.915793	-0.102149	0.347446
2019-01-05	0.368275	0.457689	-0.640344	-0.400149	-1.354794	0.368275
2019-01-06	-0.725896	-0.039305	2.287562	0.578945	-0.053486	NaN
2019-01-07	-0.286739	-0.797215	1.449312	1.236864	-1.662811	NaN
2019-01-08	0.452812	-0.847263	0.159311	0.851509	-1.757498	0.452812
2019-01-09	-1.610690	-0.790924	0.018338	0.791481	-0.471551	NaN

```
[11] df2.shape
```

```
(60, 6)
```

```
df2.dropna()
```

df2.dropna()

	A	B	C	D	E	F
2019-01-02	1.865727	0.739576	-1.892937	-0.004724	-0.187821	1.865727
2019-01-04	0.347446	-0.821685	1.591727	-0.915793	-0.102149	0.347446
2019-01-05	0.368275	0.457689	-0.640344	-0.400149	-1.354794	0.368275
2019-01-08	0.452812	-0.847263	0.159311	0.851509	-1.757498	0.452812
2019-01-11	0.309005	0.225272	0.096822	0.758550	0.813856	0.309005
2019-01-13	1.272017	-0.552437	1.104164	1.798212	0.472958	1.272017

0
1
2
3

```
[12] df2.dropna().shape
```

```
(22, 6)
```

Substituir NaN pela Média

fillna

Método que preenche com um valor os NaN's

```
[14] df3 = df.copy()
df3.head(3)
```



	A	B	C	D	E	F
2019-01-01	-0.169796	-0.477856	0.024719	0.021465	-1.407112	NaN
2019-01-02	1.865727	0.739576	-1.892937	-0.004724	-0.187821	1.865727
2019-01-03	-0.219678	-1.072485	0.517266	-1.611939	0.491082	NaN

Próximas etapas:

[Gerar código com df3](#)

[Ver gráficos recomendados](#)

```
[18] df3.fillna(np.mean(df3.A)).head(3)
```



	A	B	C	D	E	F
2019-01-01	-0.169796	-0.477856	0.024719	0.021465	-1.407112	-0.212550
2019-01-02	1.865727	0.739576	-1.892937	-0.004724	-0.187821	1.865727
2019-01-03	-0.219678	-1.072485	0.517266	-1.611939	0.491082	-0.212550

outra forma

```
df4.fillna(value = 777).head(3)
```

	A	B	C	D	E	F
2019-01-01	-0.169796	-0.477856	0.024719	0.021465	-1.407112	777.000000
2019-01-02	1.865727	0.739576	-1.892937	-0.004724	-0.187821	1.865727
2019-01-03	-0.219678	-1.072485	0.517266	-1.611939	0.491082	777.000000

Dados Únicos

Duplicatas

Verificar se há dados repetidos em um Data Frame

```
df2 = pd.DataFrame({
    'A': 1.,
    'B': pd.Timestamp('20130102'),
    'C': pd.Series(1, index=list(range(4)), dtype='float32'),
    'D': np.array([3] * 4, dtype='int32'),
    'E': pd.Categorical(["test", "train", "test", "train"]),
    'F': 'Python',
    'G': [2, 2, 4, 4],
    'H': [np.nan, 2, 4, np.nan]
})
```

	A	B	C	D	E	F	G	H
0	1.0	2013-01-02	1.0	3	test	Python	2	NaN
1	1.0	2013-01-02	1.0	3	train	Python	2	2.0
2	1.0	2013-01-02	1.0	3	test	Python	4	4.0
3	1.0	2013-01-02	1.0	3	train	Python	4	NaN

nunique(axis, dropna) ou ()

Contar valores distintos por **coluna**.

df2

	A	B	C	D	E	F	G	H
0	1.0	2013-01-02	1.0	3	test	Python	2	NaN
1	1.0	2013-01-02	1.0	3	train	Python	2	2.0
2	1.0	2013-01-02	1.0	3	test	Python	4	4.0
3	1.0	2013-01-02	1.0	3	train	Python	4	NaN

Próximas etapas: [Gerar código com df2](#) [Ver gráficos](#)

df2.nunique()

A	1
B	1
C	1
D	1
E	2
F	1
G	2
H	2

dtype: int64

axis

Busca as linhas, ou seja, os **indices**.

0

Valor **default**.

1

✓
0s

[4] df2

↔

	A	B	C	D	E	F	G	H
0	1.0	2013-01-02	1.0	3	test	Python	2	NaN
1	1.0	2013-01-02	1.0	3	train	Python	2	2.0
2	1.0	2013-01-02	1.0	3	test	Python	4	4.0
3	1.0	2013-01-02	1.0	3	train	Python	4	NaN

Próximas etapas: [Gerar código com df2](#) [Ver gráficos re](#)

✓
0s

[8] df2.nunique(axis=0)

↔

A	1
B	1
C	1
D	1
E	2
F	1
G	2
H	2

dtype: int64

✓
0s

```
# df2.nunique(dropna=False)  
df2.nunique(axis=1)
```

↔

0	6
1	6
2	6
3	6

dtype: int64

dropna

True

Valor **default**, **Não** conta o NaN

	A	B	C	D	E	F	G	H
0	1.0	2013-01-02	1.0	3	test	Python	2	NaN
1	1.0	2013-01-02	1.0	3	train	Python	2	2.0
2	1.0	2013-01-02	1.0	3	test	Python	4	4.0
3	1.0	2013-01-02	1.0	3	train	Python	4	NaN

Próximas etapas:

Gerar código com df2

Ver gráfico

[5] df2.nunique()💡

A1
B1
C1
D1
E2
F1
G2
H2
dtype: int64

df2.nunique(dropna=True)

A1
B1
C1
D1
E2
F1
G2
H2
dtype: int64

False

Conta o NaN

[4] df2

	A	B	C	D	E	F	G	H
0	1.0	2013-01-02	1.0	3	test	Python	2	NaN
1	1.0	2013-01-02	1.0	3	train	Python	2	2.0
2	1.0	2013-01-02	1.0	3	test	Python	4	4.0
3	1.0	2013-01-02	1.0	3	train	Python	4	NaN

Próximas etapas: [Gerar código com df2](#) [Ver gráficos](#)

[5] df2.nunique()

	A	B	C	D	E	F	G	H
1	1	1	1	1	2	1	2	2

dtype: int64

df2.nunique(dropna=False)

	A	B	C	D	E	F	G	H
1	1	1	1	1	2	1	2	3

dtype: int64

Importando e Exportando

Tipos de Arquivos

txt

Arquivo imples, tipo texto. Geralmente utilizado para **dados não estruturados**.

CSV

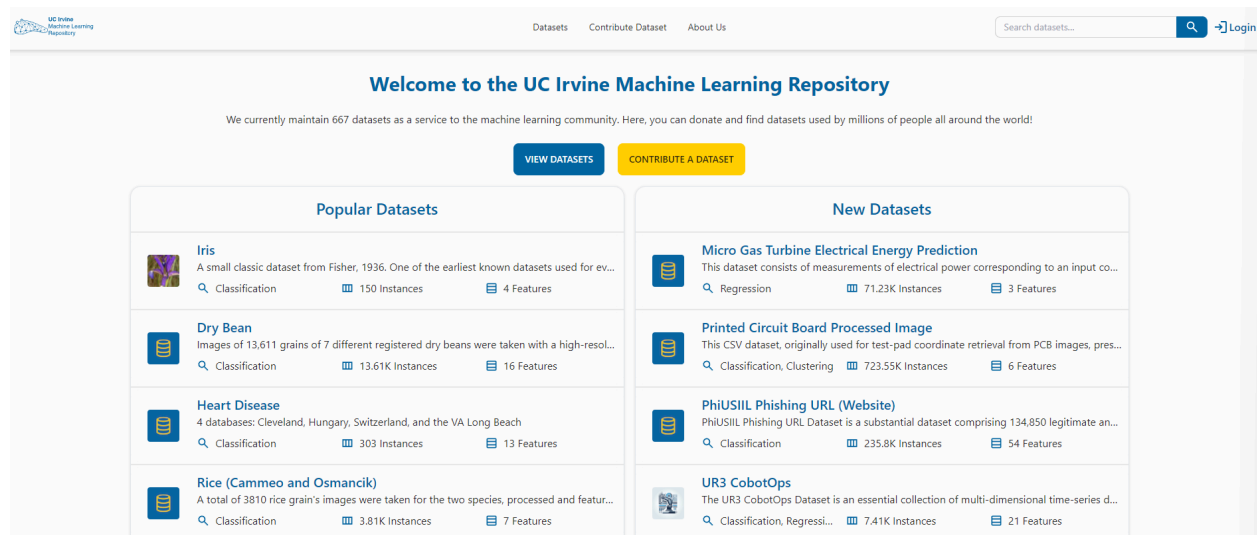
Comma Separated Values, Valores Separados por Vírgulas, mais comum.

xlsx

Arquivos Excel.

Repositório de Dados Gratuitos

[UCI Machine Learning Repository](https://uciml.github.io/)



The screenshot displays the homepage of the UCI Machine Learning Repository. At the top, there is a navigation bar with links for 'Datasets', 'Contribute Dataset', and 'About Us', along with a search bar and a 'Login' button. The main heading reads 'Welcome to the UC Irvine Machine Learning Repository'. Below this, a message states: 'We currently maintain 667 datasets as a service to the machine learning community. Here, you can donate and find datasets used by millions of people all around the world!'. Two buttons, 'VIEW DATASETS' and 'CONTRIBUTE A DATASET', are prominently displayed. The page is divided into two columns: 'Popular Datasets' and 'New Datasets'. The 'Popular Datasets' column lists four datasets: 'Iris' (a small classic dataset from Fisher, 1936, used for classification with 150 instances and 4 features), 'Dry Bean' (images of 13,611 grains of 7 different registered dry beans, used for classification with 13,61K instances and 16 features), 'Heart Disease' (4 databases: Cleveland, Hungary, Switzerland, and the VA Long Beach, used for classification with 303 instances and 13 features), and 'Rice (Cammeo and Osmancik)' (a total of 3810 rice grain's images, used for classification with 3,81K instances and 7 features). The 'New Datasets' column lists four datasets: 'Micro Gas Turbine Electrical Energy Prediction' (measurements of electrical power, used for regression with 71,23K instances and 3 features), 'Printed Circuit Board Processed Image' (a CSV dataset for test-pad coordinate retrieval, used for classification and clustering with 723,55K instances and 6 features), 'PhiUSIIL Phishing URL (Website)' (a substantial dataset comprising 134,850 legitimate and malicious URLs, used for classification with 235,8K instances and 54 features), and 'UR3 CobotOps' (an essential collection of multi-dimensional time-series data, used for classification and regression with 7,41K instances and 21 features).

Abrindo Arquivos de Dados Externos