Pandas

É muito comum trabalharmos com processamento de dados que se encontram no formato de tabelas (também chamados dados "tabulares", ou dados es
empresas. Assim, se quisermos fazer trabalhos básicos em dados tabulares de forma automatizada, com o Python, precisamos de uma biblioteca com fur

O **Pandas** (derivado de *panel data*) é uma biblioteca que implementa essas funcionalidades [1]. Criada por Wes McKinney em torno de 2009, e tendo vir
dados e estatística computacional. O pandas é muito versátil e simples de usar, facilitando muito o trabalho com dados estruturados.

Para entender o funcionamento do `pandas`, é importante conhecer os objetos fundamentais da biblioteca: `Series` e `DataFrame`. Esses objetos representa
maior parte das operações do pandas.

Além disso, é importante também ter conhecimentos básicos de NumPy, visto que ele é a biblioteca principal na qual o pandas se apoia. Assim, toda a est

1. Instalação

O pandas é uma biblioteca para linguagem Python, e para ser instalada nós procedemos da mesma forma que qualquer outra biblioteca. A forma mais fá

A distribuição Anaconda (https://www.anaconda.com/distribution/) vem com o Python e diversas bibliotecas científicas e para ciência de dados instalad
install pandas

```
Também é possível baixar a partir do servidor da Python Software Foundation, conhecido como PyPI, através do gerenciador de pacotes `
```

$ pip install pandas

```
## 2. Introdução ao Pandas  <a name="2introducao"></a>

Antes de mais nada, precisamos importar o Pandas para poder usá-lo. Nós fazemos isso da mesma forma que qualquer outra biblioteca Pyt

```python
>>> import Pandas as pd
```

Normalmente, dados costumam vir na forma de tabelas. Nós chamamos dados com esse formato de **dados estruturados**. Isso significa que em geral, dad
linha. A unidade básica que costumamos pensar é a de células da tabela. Uma célula é uma interseção entre uma dada linha e uma dada coluna da tabela.

Por exemplo, na tabela abaixo, nós podemos dizer que na célula [2,1] nós temos o valor 1. No Python, a linha 2 é a terceira, contando de cima pra baixo, e

```
[[1, 2, 3],
 [2, 2, 2],
 [0, 1, 0]]
```

Perceba que falar que a linha 2 é a "terceira" (devido ao Python começar a indexação do 0) é algo arbitrário. Se a gente quisesse dar nomes pras linhas, p
reescrever a célula como sendo ["c","B"].

O que o Pandas faz é exatamente isso: Ele cria um array de numpy, porém com linhas (ou índices) com nomes específicos que podemos definir como quer

Essa estrutura, com uma tabela, uma lista de índices (nomes das linhas) e uma lista de colunas (nomes das colunas), é o que chamamos de um `DataFrame` c

```
>>> dados = [[1.69, 87.0],
             [1.59, 56.5],
             [1.69, 90.3],
             [1.74, 78.6]]
>>> df = pd.DataFrame(dados, columns=['Nome', 'altura', 'peso'],
```
```

```
                    index=['Fulano', 'Sicrana', 'Beltrana', 'João'])
>>> df
          altura  peso
Fulano      1.69  87.0
Sicrana     1.59  56.5
Beltrana    1.69  90.3
João        1.74  78.5
```

Uma observação: É comum não passarmos uma lista de índices. Nesses casos, o pandas usa um padrão que é usar índices numéricos em sequência (por e

```
>>> dados = [['Fulano',   1.69, 87.0],
             ['Sicrana',  1.59, 56.5],
             ['Beltrana', 1.69, 90.3],
             ['João',     1.74, 78.6]]
>>> df = pd.DataFrame(dados, columns=['Nome', 'altura', 'peso'])
>>> df
       Nome  altura  peso
0    Fulano    1.69  87.0
1   Sicrana    1.59  56.5
2  Beltrana    1.69  90.3
3      João    1.74  78.6
```

É interessante ter em mente que o pandas tem uma inspiração fortíssima na linguagem de programação R, muito usada para estatística e análise de dadc mais fácil de se acostumar com o pandas.

Agora, se olharmos o tipo da nossa variável `df`, veremos que ele é um DataFame do pandas.

```
>>> print(type(df))
<class 'pandas.core.frame.DataFrame'>
```

O outro objeto fundamental em pandas é o objeto `Series`, que nada mais é que uma lista (unidimensional) indexada. Assim como no dataframe, esse índi (assim como o faz variáveis do tipo `DataFrame`).

```
>>> minha_lista = [87.0, 56.5, 90.3, 78.6]
>>> serie = pd.Series(minha_lista, index=['Fulano', 'Sicrana', 'Beltrana', 'João'])
>>> serie
Fulano      87.0
Sicrana     56.5
Beltrana    90.3
João        78.6
dtype: float64
>>> serie = pd.Series(minha_lista) # Se não passarmos um índice, ele vira uma sequência
>>> serie
0    87.0
1    56.5
2    90.3
3    78.6
dtype: float64
```

Ao trabalhar com o pandas, estamos sempre atuando com esses dois tipos de objeto (`DataFrame` e `Series`).

Como exemplo, imagina que queiramos uma lista com os valores de uma dada coluna (ou de uma dada linha) do nosso dataframe. Nesse caso, podemos u

```
>>> serie = df['peso']
>>> serie
0    87.0
1    56.5
2    90.3
3    78.6
Name: peso, dtype: float64
>>> print(type(serie))
<class 'pandas.core.series.Series'>
```

Agora que vimos o que são os objetos fundamentais do pandas, vamos nos aprofundar um pouco mais em suas características.

2.1. Pandas Series

Como vimos, podemos criar um objeto `pd.Series` a partir de uma lista (ou de um array do numpy). Assim como com o NumPy, nós podemos controlar o ti

```
>>> import pandas as pd
>>> import numpy as np
>>> minha_lista = [10, 20, 30, 40]
>>> serie = pd.Series(minha_lista, index=['a', 'b', 'c', 'd'], dtype=np.float32)
>>> serie
a    10.0
b    20.0
c    30.0
d    40.0
dtype: float32
>>> serie.dtype
dtype('float32')
```

O nome de cada linha (o "index" da série) pode ser alterado após a criação da série. Basta atribuirmos um novo valor ao atributo `index`. O mesmo vale pa

```
>>> minha_lista = [10, 20, 30, 40]
>>> serie = pd.Series(minha_lista)
>>> serie
0    10
1    20
2    30
3    40
dtype: int64
>>> serie.index = ['a', 'b', 'c', 'd']
>>> serie
a    10
b    20
c    30
d    40
dtype: int64
>>> serie.astype(np.float32)
a    10.0
b    20.0
c    30.0
d    40.0
dtype: float32
```

Vale notar, contudo, que a função `astype` retorna uma cópia da série. Ela não salva o resultado dentro da mesma variável que tínhamos. Isso significa que

Uma última propriedade das séries do pandas que podemos controlar é o nome delas.

```
>>> minha_lista = [10, 20, 30, 40]
>>> serie = pd.Series(minha_lista, index=['a', 'b', 'c', 'd'], name='Números')
>>> serie
a    10
b    20
c    30
d    40
Name: Números, dtype: int64
```

Manipulando séries Pandas

Quando usamos listas básicas do Python, nós podemos querer usar um elemento numa posição específica. Para isso, usamos um índice ao lado da lista, c

Com uma série do pandas, a ideia é a mesma. Seja para usar um elemento específico, ou para usar um intervalo de valores da série, a sintaxe do pandas é

```
>>> serie
a    10.0
b    20.0
```

```
c    30.0
d    40.0
dtype: float32
>>> serie['c']
30.0
>>> serie[['c','a']]   # Se pegarmos vários índices, ele nos retorna uma Série Pandas
c    30.0
a    10.0
dtype: float32
>>> serie['a':'c']   # Podemos pegar um slicing pela ordem dos índices
a    10.0
b    20.0
c    30.0
dtype: float32
```

Um outro método para selecionar os elementos é usando as funções `iloc`, que seleciona pela posição do índice (começando do zero), e `loc`, que selecion

```
>>> serie.iloc[2]
30.0
>>> serie.iloc[1:3]
b    20.0
c    30.0
dtype: float32
>>> serie.loc['c']
30.0
>>> serie.loc['a':'c']
a    10.0
b    20.0
c    30.0
dtype: float64
>>> serie.loc['c':'a']
Series([], dtype: float64)
```

Como não existe índice depois de "c" mas antes de "a", o resultado da última linha do bloco de código acima foi uma série vazia.

Por fim, de forma análoga ao NumPy, nós podemos usar "filtros lógicos" (também chamados máscaras booleanas). Eles nada mais são que uma lista (ou n

Podemos selecionar assim apenas os elementos que satisfaçam determinada condição.

```
>>> serie
a    10.0
b    20.0
c    30.0
d    40.0
dtype: float32
>>> serie[[True, False, True, False]]
a    10.0
c    30.0
dtype: float32
>>> serie[serie > 15]
b    20.0
c    30.0
d    40.0
dtype: float32
```

De forma semelhante, podemos passar essa máscara para o método `loc` e obter o mesmo comportamento.

```
>>> mask = serie > 15
>>> mask
a    False
b     True
c     True
d     True
dtype: bool
```

```
>>> serie.loc[mask]
b    20.0
c    30.0
d    40.0
dtype: float32
```

Note que a variável `mask` é uma série do Pandas, resultante da operação `> 15` elemento a elemento. Por trás dos panos, o mesmo estava acontecendo no

Métodos e operações

Como já dito no texto, o pandas é construído em cima do NumPy. Os objetos fundamentais do pandas sempre tem um ndarray por trás. Por isso, muitas c

As operações matemáticas (`+`, `-`, `*`, `/`, `**`), por exemplo, são realizadas elemento a elemento assim como no NumPy. Uma diferença crucial é que o Panda

```
>>> serie
a    10.0
b    20.0
c    30.0
d    40.0
dtype: float32
>>> values = pd.Series([0.5, 1.0, 1.5, 2.0], index=['b', 'd', 'a', 'c'], dtype=np.float32)
>>> values
b    0.5
d    1.0
a    1.5
c    2.0
dtype: float32
>>> serie * values
a    15.0
b    10.0
c    60.0
d    40.0
dtype: float32
```

No exemplo acima, note que os elementos com índice "a" são o 10, na variável `serie`, e 1.5, na variável `values`. Logo, o resultado da multiplicação para o í

Se as séries não tiverem o mesmo conjunto de valores nos índices, as operações darão valores absurdos (o pandas não irá levantar um erro ou uma exceç

```
>>> serie
a    10.0
b    20.0
c    30.0
d    40.0
dtype: float32
>>> other_values = pd.Series([0.5, 1.0, 1.5, 2.0], dtype=np.float32)
>>> other_values
0    0.5
1    1.0
2    1.5
3    2.0
dtype: float32
>>> serie * other_values
a   NaN
b   NaN
c   NaN
d   NaN
0   NaN
1   NaN
2   NaN
3   NaN
dtype: float32
```

Todos os valores resultantes no exemplo acima são `NaN` (Not a Number - o que significa que não há resultado ou é inválido), e temos os índices de ambas a `other_values`, e logo seu valor é "vazio").

Existem também alguns métodos para operar em cima da série de forma agregada. Eles podem retornar algum valor estatístico dos valores da série (com

Abaixo estão alguns exemplos de métodos que as séries possuem para esse tipo de operação.

Método	Descrição
`sum`	soma os valores
`mean`	média dos valores
`std`	desvio padrão dos valores
`mode`	moda dos valores
`max`	valor máximo na série
`min`	valor mínimo na série
`value_counts`	conta repetições de cada valor
`describe`	resumo de estatísticas básicas

Veja alguns exemplos abaixo.

```
>>> valores = [1, 1, 2, 3, 5, 8, 13]
>>> fibonacci = pd.Series(valores)
>>> fibonacci.sum()
33
>>> fibonacci[fibonnaci > 4].sum()
26
>>> valores = [1, 1, 0, 0, 1, 0, 1, 1, 1]
>>> serie = pd.Series(valores)
>>> serie.value_counts()
1    6
0    3
dtype: int64
>>> serie.describe()
count    9.000000
mean     0.666667
std      0.500000
min      0.000000
25%      0.000000
50%      1.000000
75%      1.000000
max      1.000000
dtype: float64
```

Quando as séries são compostas por elementos do tipo "string" (`str`, no Python), o pandas entenderá seu `dtype` como sendo `object`.

Nestes casos, a série tem um atributo `str`, que permite que façamos operações próprias de strings ao longo dos lementos da série.

```
>>> pronomes = pd.Series(['eu', 'tu', 'ele/ela', 'nós', 'vós', 'eles/elas'])
>>> pronomes
0          eu
1          tu
2      ele/ela
3         nós
4         vós
5    eles/elas
dtype: object
>>> pronomes.str
<pandas.core.strings.StringMethods at 0x7fe0d30292e8>
>>> pronomes.str.upper()
0          EU
1          TU
2      ELE/ELA
3         NÓS
4         VÓS
```

```
5    ELES/ELAS
dtype: object
```

Se tentássemos usar o método `upper` diretamente com a série, veríamos o erro abaixo.

```
>>> pronomes.upper()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-50-6ecc768b079a> in <module>
----> 1 pronomes.upper()

~/miniconda3/envs/data_science/lib/python3.7/site-packages/pandas/core/generic.py in __getattr__(self, name)
   5272             if self._info_axis._can_hold_identifiers_and_holds_name(name):
   5273                 return self[name]
-> 5274             return object.__getattribute__(self, name)
   5275
   5276     def __setattr__(self, name: str, value) -> None:

AttributeError: 'Series' object has no attribute 'upper'
```

2.2. Pandas DataFrame

O `DataFrame` é uma representação de uma tabela. Ele possui dois eixos rotulados, que são as linhas (rotuladas pelo índice, ou `index`) e as colunas (rotulad

Existem diversas maneiras de se criar um dataframe, podendo ser a partir de listas, dicionários etc.
Um dos modos mais comuns é a criação a partir da leitura de um arquivo no formato `.csv`, como veremos a seguir para o caso do dataset `titanic`, muito

```
>>> df_titanic = pd.read_csv('../datasets/titanic.csv')
>>> print(type(df_titanic))
<class 'pandas.core.frame.DataFrame'>
```

Note que `df_titanic` é um `DataFrame` com os dados do arquivo `titanic.csv`. Com relação à pasta na qual o código foi executado, este arquivo está locali

Vemos que o parâmetro do método `pd.read_csv` é o nome do arquivo que se deseja ler, com o caminho relativo até o arquivo.

Existem diversos parâmetros relevantes. Todos podem ser encontrados na documentação do Pandas.

> **Observação:**
> Note que podemos também utilizar um arquivo do formato `.xlsx`, natural do excel.
> Para tanto, devemos utilizar o método `pd.read_excel`.
>
> É possível também milhares de outros formatos menos comuns, como `.data`, `.txt`, e outros. O método mais geral para leitura destes formatos é `pd.re`

A segunda forma de construir um DataFrame é a partir de variáveis do Python. O primeiro exemplo que veremos é como construir um DataFrame a part

Este é um método muito útil, pois a estrutura de `dict`, nativa do Python, é bem semelhante a de um `DataFrame`.

Neste caso, cada chave do nosso dicionário se tornará uma **coluna** enquanto os valores (que podem ser elementos de listas, arrays, series...) serão os eler

```
>>> dicionario = {
                  'coluna_A': [1, 2, 3, 4, 5],
                  'coluna_B': ['a', 'b', 'c', 'd', 'e'],
                  'coluna_C': [0.5, 1.5, 4.5, 6.5, 8.5]
                }
>>> df = pd.DataFrame(dicionario)
>>> df
   coluna_A coluna_B  coluna_C
0         1        a       0.5
1         2        b       1.5
2         3        c       4.5
3         4        d       6.5
4         5        e       8.5
```

Um outro método, que já vimos antes neste material, é passando uma lista de listas (ou um array bidimensional do NumPy).

```
>>> dados = [[1.69, 87.0],
             [1.59, 56.5],
             [1.69, 90.3],
             [1.74, 78.6]]
>>> df = pd.DataFrame(dados, columns=['altura', 'peso'])
>>> df
   altura  peso
0    1.69  87.0
1    1.59  56.5
2    1.69  90.3
3    1.74  78.6
```

Muitas das propriedades das séries Pandas se mantêm para DataFrames. A diferença é que agora, além de um atributo `index` com os nomes das linhas, t

```
>>> df
   altura  peso
0    1.69  87.0
1    1.59  56.5
2    1.69  90.3
3    1.74  78.6
>>> df.index      # O índice padrão é uma lista de números igualmente espaçados de 0 até o tamanho
RangeIndex(start=0, stop=4, step=1)
>>> df.columns
Index(['altura', 'peso'], dtype='object')
```

Manipulando dataframes Pandas

As formas de se acessar os valores de um `DataFrame` são análogas ao que vimos antes com séries do pandas. A única diferença é que agora teremos de us

```
>>> df
   altura  peso
0    1.69  87.0
1    1.59  56.5
2    1.69  90.3
3    1.74  78.6
>>> df['altura'][1]        # se tentarmos fazer como Python, fazemos [coluna][linha], usando os rótulos
1.59
>>> df.iloc[1, 0]          # No caso do iloc, fazemos [linha, coluna], usando a posição
1.59
>>> df.loc[1, 'altura']    # No caso do loc, fazemos [linha, coluna], usando os rótulos
1.59
```

No caso que fizemos `df['altura'][1]`, no fundo estamos primeiro acessando uma série do pandas, `df['altura']`, e depois usando o que vimos na sessã

Se nosso objetivo for selecionar um único elemento, também podemos usar os métodos `iat` e `at`. O método `iat` recebe como argumento a posição do el
e das colunas.

```
>>> df
   altura  peso
0    1.69  87.0
1    1.59  56.5
2    1.69  90.3
3    1.74  78.6
>>> df.iat[1, 0]        # No caso do iat, acessamos [linha, coluna], usando a posição
1.59
>>> df.at[1, 'altura']    # No caso do at, acessamos [linha, coluna], usando os rótulos
1.59
```

Podemos também selecionar diferentes subconjuntos do DataFrame.

```
>>> df
   altura  peso     nomes
```

```
0    1.69  77.2    Fulano
1    1.50  64.2   Sicrana
2    1.69  85.0  Beltrana
3    1.74  78.6      João
>>> df.loc[1:3, ['peso', 'nomes']]
   peso     nomes
1  64.2   Sicrana
2  85.0  Beltrana
3  78.6      João
>>> df[['nomes', 'altura']]
      nomes  altura
0    Fulano    1.69
1   Sicrana    1.50
2  Beltrana    1.69
3      João    1.74
```

Podemos usar máscaras booleanas da mesma forma que fizemos com `Series` no pandas e `ndarray` no NumPy.

```
>>> df
   altura  peso     nomes
0    1.69  77.2    Fulano
1    1.50  64.2   Sicrana
2    1.69  85.0  Beltrana
3    1.74  78.6      João
>>> df[df['peso'] > 78.0]
   altura  peso     nomes
2    1.69  85.0  Beltrana
3    1.74  78.6      João
```

Porém, objetos do tipo `DataFrame` possuem também outro método para fazer esse tipo de filtro. Esse é o método `query`, que recebe uma string represent

```
>>> df.query('peso > 78.0')
   altura  peso     nomes
2    1.69  85.0  Beltrana
3    1.74  78.6      João
```

Sempre que nós buscarmos alterar elementos de um `DataFrame`, devemos acessar os elementos a serem alterados através de um método dentre `at`, `iat`,

```
>>> df
   altura  peso
0    1.69  87.0
1    1.59  56.5
2    1.69  90.3
3    1.74  78.6
>>> df.at[1, 'altura'] = 1.50
>>> df
   altura  peso
0    1.69  87.0
1    1.50  56.5
2    1.69  90.3
3    1.74  78.6
>>> df.loc[:2, 'peso'] = [77.2, 64.2, 85.0]
>>> df
   altura  peso
0    1.69  77.2
1    1.50  64.2
2    1.69  85.0
3    1.74  78.6
```

Uma última manipulação que fazemos com frequência é a criação de colunas novas. A sintaxe relevante se encontra abaixo.

```
>>> df
   altura  peso
```

```
0    1.69  77.2
1    1.50  64.2
2    1.69  85.0
3    1.74  78.6
>>> df['nomes'] = ['Fulano', 'Sicrana', 'Beltrana', 'João']
>>> df
   altura  peso     nomes
0    1.69  77.2    Fulano
1    1.50  64.2   Sicrana
2    1.69  85.0  Beltrana
3    1.74  78.6      João
```

Por fim, para salvar o nosso dataframe, podemos usar métodos como `to_csv`, `to_json`, `to_html`, entre outros.

```
>>> df
   altura  peso
0    1.69  77.2
1    1.50  64.2
2    1.69  85.0
3    1.74  78.6
>>> df.to_csv('../datasets/alturas_e_pesos.csv')
```

Outras transformações e métodos

Existem diversas outras transformações e métodos que podemos usar com nosso DataFrame. Como exemplo, nós podemos usar os dados contidos em u

```
>>> df
   altura  peso     nomes
0    1.69  77.2    Fulano
1    1.50  64.2   Sicrana
2    1.69  85.0  Beltrana
3    1.74  78.6      João
>>> df.drop(columns='nomes').to_numpy() # A função "drop" remove as colunas especificadas
array([[ 1.69, 77.2 ],
       [ 1.5 , 64.2 ],
       [ 1.69, 85.  ],
       [ 1.74, 78.6 ]])
>>> df.to_numpy()
array([[1.69, 77.2, 'Fulano'],
       [1.5, 64.2, 'Sicrana'],
       [1.69, 85.0, 'Beltrana'],
       [1.74, 78.6, 'João']], dtype=object)
>>> df.values  # O atributo "values" funciona igual a função to_numpy().
array([[1.69, 77.2, 'Fulano'],
       [1.5, 64.2, 'Sicrana'],
       [1.69, 85.0, 'Beltrana'],
       [1.74, 78.6, 'João']], dtype=object)
```

É possível também fazer operações matemáticas entre dataframes. No caso, o Pandas sempre vai operar entre células de mesmo índice e mesma coluna.

```
>>> df1
   x  y
0  1  2
1  2  4
2  3  6
3  4  8
>>> df2
   x   y
0  1   3
1  2   6
2  3   9
3  4  12
>>> df3
```

```
   a   y
0  1   3
1  2   6
2  3   9
3  4  12
>>> df1 + df2
   x   y
0  2   5
1  4  10
2  6  15
3  8  20
>>> df1 + df3
    a   x   y
0 NaN NaN   5
1 NaN NaN  10
2 NaN NaN  15
3 NaN NaN  20
```

Além das operações matemáticas, podemos também rearrumar a ordem das linhas da tabela. O método `sort_index` organiza a tabela pelo índice, indo d

```
>>> df
   altura  peso     nomes
b    1.69  77.2    Fulano
a    1.50  64.2   Sicrana
c    1.69  85.0  Beltrana
f    1.74  78.6      João
>>> df.sort_index()
   altura  peso     nomes
a    1.50  64.2   Sicrana
b    1.69  77.2    Fulano
c    1.69  85.0  Beltrana
f    1.74  78.6      João
```

De forma semelhante, podemos ordenar as linhas através dos valores de uma dada coluna, usando o método `sort_values`. É possível organizar indo do m

```
>>> df
   altura  peso     nomes
b    1.69  77.2    Fulano
a    1.50  64.2   Sicrana
c    1.69  85.0  Beltrana
f    1.74  78.6      João
>>> df.sort_values('peso')
   altura  peso     nomes
a    1.50  64.2   Sicrana
b    1.69  77.2    Fulano
f    1.74  78.6      João
c    1.69  85.0  Beltrana
>>> df.sort_values('peso', ascending=False)
   altura  peso     nomes
c    1.69  85.0  Beltrana
f    1.74  78.6      João
b    1.69  77.2    Fulano
a    1.50  64.2   Sicrana
```

Podemos também reorganizar as colunas. Para isso, a sintaxe mais comum é dada abaixo.

```
>>> df
   altura  peso     nomes
b    1.69  77.2    Fulano
a    1.50  64.2   Sicrana
c    1.69  85.0  Beltrana
f    1.74  78.6      João
>>> df[['nomes', 'peso', 'altura']]
```

```
       nomes  peso  altura
b     Fulano  77.2    1.69
a    Sicrana  64.2    1.50
c   Beltrana  85.0    1.69
f       João  78.6    1.74
```

3. Tratamento e limpeza de dados

É muito comum em um conjunto de dados, seja ele proveniente de um banco dados ou de um arquivo `csv`, existirem valores nulos (ou seja, valores inváli

Para fins de análises/modelos é muito importante identificar a incidência desses valores e tomar uma decisão, seja a de remover os valores nulos, ou a de

Identificando Elementos Nulos por Coluna: Identificar a quantidade de valores nulos por coluna é muito importante, pois assim podemos identificar qua

```
>>> df
   altura  peso     nomes
b    1.69  77.2    Fulano
a    1.50   NaN   Sicrana
c    1.69  85.0  Beltrana
f     NaN  78.6      João
>>> df.isnull().sum()
altura    1
peso      1
nomes     0
dtype: int64
```

Removendo os valores nulos: Para remover os nulos, iremos utilizar o comando `dropna`, como segue:

```
>>> df
   altura  peso     nomes
b    1.69  77.2    Fulano
a    1.50   NaN   Sicrana
c    1.69  85.0  Beltrana
f     NaN  78.6      João
>>> df.dropna()
   altura  peso     nomes
b    1.69  77.2    Fulano
c    1.69  85.0  Beltrana
```

Ou seja, ele removeu todas as linhas que contém algum valor nulo.

Substituindo valores nulos: Como muitas vezes não queremos diminuir o tamanho do nosso conjunto de dados, então uma abordagem é substituir esses

Para fazer isso, podemos utilizar o método `fillna`, em que o parâmetro passado será o valor de substituição. Neste exemplo iremos substituir os valores

```
>>> df
   altura  peso     nomes
b    1.69  77.2    Fulano
a    1.50   NaN   Sicrana
c    1.69  85.0  Beltrana
f     NaN  78.6      João
>>> df.fillna(-1)
   altura  peso     nomes
b    1.69  77.2    Fulano
a    1.50  -1.0   Sicrana
c    1.69  85.0  Beltrana
f   -1.00  78.6      João
```

Se quisermos alterar um valor específico na nossa tabela (não apenas o NaN), podemos usar o método `replace`.

```
>>> df
   altura  peso
0    1.69  77.2
1    1.50  -1.0
```

```
2    1.69  85.0
3   -1.00  78.6
>>> df.replace(-1, 60)
   altura  peso
0    1.69  77.2
1    1.50  60.0
2    1.69  85.0
3   60.00  78.6
```

Veja que alteramos os valores de todo o dataframe e não só de uma coluna.

Outro tratamento importante é remover duplicatas da nossa tabela.

```
>>> df_dup
   a  b
0  1  0
1  1  0
2  2  1
3  2  2
>>> df_dup.drop_duplicates()
   a  b
0  1  0
2  2  1
3  2  2
```

4. Exploração de Dados

Podemos aproveitar o Pandas para ajudar na exploração dos dados. Existem diversos métodos para nos ajudar a sumarizar dados e calcular algumas esta

Como um exemplo, podemos ver a média de cada coluna de um dataframe.

```
>>> df
   altura  peso
0    1.69  77.2
1    1.50  68.0
2    1.58  65.0
3    1.69  85.0
4    1.74  78.6
5    1.73  69.2
>>> df.mean()
altura     1.655000
peso      73.833333
dtype: float64
```

Também podemos achar outras informações, como o maior e o menor valor das colunas.

```
>>> df
   altura  peso
0    1.69  77.2
1    1.50  68.0
2    1.58  65.0
3    1.69  85.0
4    1.74  78.6
5    1.73  69.2
>>> df.min()
altura     1.5
peso      65.0
dtype: float64
>>> df.max()
altura     1.74
peso      85.00
dtype: float64
```

Um método muito útil, que reune várias descrições ao mesmo tempo é o `describe`.

```
>>> df
   altura  peso
0    1.69  77.2
1    1.50  68.0
2    1.58  65.0
3    1.69  85.0
4    1.74  78.6
5    1.73  69.2
>>> df.describe()
         altura       peso
count  6.000000   6.000000
mean   1.655000  73.833333
std    0.094816   7.645565
min    1.500000  65.000000
25%    1.607500  68.300000
50%    1.690000  73.200000
75%    1.720000  78.250000
max    1.740000  85.000000
```

Quando estamos fazendo análises em um conjunto de dados, é muito útil saber alguns comportamentos separados por grupos. Para tanto, vamos utilizar

Neste exemplo, iremos analisar as médias de altura e peso por genero.

```
>>> df
   altura  peso  genero
0    1.69  77.2     0.0
1    1.50  68.0     0.0
2    1.58  65.0     0.0
3    1.69  85.0     1.0
4    1.74  78.6     1.0
5    1.73  69.2     1.0
>>> df.groupby('genero').mean()
        altura       peso
genero
0.0       1.59  70.066667
1.0       1.72  77.600000
```

Também podemos cruzar informações de diferentes tipos de agrupamentos, para avaliar os dados. Para tanto, utilizamos o método `pivot_table`. É possí

```
>>> df
   altura  peso  genero  carioca
0    1.69  77.2     0.0        1
1    1.50  68.0     0.0        0
2    1.58  65.0     0.0        1
3    1.69  85.0     1.0        0
4    1.74  78.6     1.0        1
5    1.73  69.2     1.0        0
>>> df.pivot_table(index='genero', columns='carioca', values='peso', aggfunc=np.max)
carioca     0     1
genero
0.0      68.0  77.2
1.0      85.0  78.6
>>> df.pivot_table(index='genero', columns='carioca', values='peso', aggfunc=np.sum)
carioca      0      1
genero
0.0       68.0  142.2
1.0      154.2   78.6
```

Referências:

1. pandas: a Foundational Python Library for Data Analysis and Statistics; McKinney, W.; Workshop Python for High Performance and Scientific Comp
df.

2. Tutorial de instalação do Pandas, disponível em https://pandas.pydata.org/docs/getting_started/install.html.
3. Python for Data Analysis: Data Wrangling with pandas, Numpy & Jupyter; McKinney W.; 3 ed., O'Reilly. Disponível em https://wesmckinney.com/bc