Módulo de Programação Python

Trilha Python - Aula 14: Utilizando Pandas - Introdução

Objetivo: Trabalhar com pacotes e módulos disponíveis em **Python**: **Pandas**: Discutir a importância de obter, carregar e organizar grandes volumes de dados. Apresentar Pandas e suas funcionalidades e características básicas.

Contextualização

Até aqui discutimos sobre a importância de trabalhar com estruturas de dados eficientes para armazenar grandes volumes de dados. Nas aulas anteriores foram apresentados os arrays de tipo fixo implementados na forma de *ndarrays* da **NumPy**.

Ainda que muito eficientes para armazenar e processar dados numéricos, os *ndarrays* apresentam limitações para análise da dados não numéricos.

Imaginem, no exemplo que construímos na aula anterior, que queremos adicionar uma etiqueta ou rótulo a cada aluno com o nome ou o e-mail.

O Pandas, e em particular seus objetos `Series` e `DataFrame` , baseia-se no uso de *ndarrays* de **NumPy** e fornece acesso eficiente a esses tipos de tarefas de "gestão de dados" que ocupam muito do tempo de um cientista de dados.

Vamos abordar então em como utilizar `Series` , `DataFrame` e estruturas relacionadas de forma eficaz.

No ambiente virtual que utilizamos até qui temos os pacotes e módulos para rodar o *jupyter notebook* e **NumPY**. Vamos começar então por instalar **Pandas**

In [ ]:
```
#pip list
#pip freeze > requirements.txt
#cat requirements.txt
#pip install pandas
```

In [1]:
```
import numpy as np
import pandas as pd
print("Numpy version: ", np.__version__)
print("Pandas version: ", pd.__version__)
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Adva
nced Vector Extensions (Intel(R) AVX) instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Adva
nced Vector Extensions (Intel(R) AVX) instructions.
Numpy version:  1.26.2
Pandas version:  2.1.4
```

Os objetos Pandas podem ser considerados versões aprimoradas de matrizes *ndarrays* de **NumPy** nas quais as linhas e colunas são identificadas com rótulos em vez de simples índices inteiros.

Da mesma forma que **NumPy**, _**Pandas** fornece, além das estruturas de dados, uma série de ferramentas, métodos e funcionalidades úteis .

Vamos começar aprestando as estruturas básicas de **Pandas**.

In [ ]:
```
from random import uniform
lista = [uniform(4, 10) for _ in range(5)]
for val in lista:
    print(f"{val:.2f}", end=" ")
```

Pandas `Series`

Uma `Series` **Pandas** é uma matriz unidimensional de dados indexados.

De forma simples um objeto da classe `Series` pode ser criado a partir de uma lista ou de um *ndarray*.

In [ ]:
```
dSerie = pd.Series(lista)
dSerie
```

In [ ]:
```
npArray = np.array(lista)
dSerie = pd.Series(npArray)
dSerie
```

Reparem que um objeto `Series` consiste em uma sequência de valores e sua correspondente sequência de índices, que podemos acessar com os atributos `values` e `index` .

In [ ]:
```
print(dSerie.values)
print(type(dSerie.values))
```

Já o tributo `index` é um objeto semelhante a um *ndarray*, de tipo `pd.Index` .

In [ ]:
```
print(dSerie.index)
print(type(dSerie.index))
```

Os elementos de `dSerie` podem ser acessado via indexação.

In [ ]:
```
dSerie[0]
```

In [ ]:
```
dSerie[1:3]
```

Pode parecer que um objeto da classe `Sreies` é semelhante a um *ndarrays*, podendo usar um o outro. Mas ...

In [ ]:
```
print(dSerie.values[-1])
try:
    print(dSerie[-1])
except Exception as e:
    print(e)
```

Entretanto, enquanto o *ndarray* de **NumPy** possui um índice inteiro, definido implicitamente, usado para acessar os valores, os objetos `Series` de **Pandas** possuem um índice definido explicitamente, associado ao conjunto de valores.

Essa definição explícita de índice fornece recursos adicionais como, por exemplo, o fato de que o índice não precisa ser um número inteiro, mas pode consistir em valores de qualquer tipo desejado.

Por exemplo, se desejarmos, podemos usar strings como índice:

In [ ]:
```
dSerie = pd.Series(lista, index=['alpha', 'beta', 'gamma', 'delta',
dSerie
```

In [ ]:
```
print(dSerie.values[-1])
print(dSerie['epsilon'])
```

Podemos inclusive usar índices inteiros não contíguos ou não sequenciais.

In [ ]:
```
dSerie = pd.Series(lista, index=[99, 87, 65, 43, 21])
dSerie
```

In [ ]:
```
print(dSerie.values[-1])
print(dSerie[21])
```

In [ ]:
```
dSerie = pd.Series(lista, index=[0.1, 0.01, 0.001, 0.0001, 0.00001]
dSerie
```

Podemos então pensar as `Sreies` **Pandas** como uma forma particular e específica de dicionário **Python**.

- Um dicionário **Python** é uma estrutura que mapeia chaves arbitrárias para um conjunto de valores arbitrários
- Um objeto `Series` é uma estrutura que mapeia chaves de tipo fixo para um conjunto de valores de tipo fixo.

O fato de tratar de chaves e valores tipados é importante: assim como o código compilado, específico de cada tipo, por trás de um *ndarray* **NumPy**, o torna mais eficiente do que uma lista **Python** para determinadas operações, as informações de tipo de um `Series` **Pandas** o tornam muito mais eficiente do que os dicionários **Python** para determinadas operações.

A analogia da série como dicionário pode ficar ainda mais evidente quando constatamos que podemos construir um objeto `Series` diretamente de um dicionário **Python**:

In [ ]:
```
popPorEstadoDic = { 'São Paulo': 44411238, 'Minas Gerais':20538718,
                    'Bahia':14141626, 'Paraná':11444380, 'Rio Grande do
                    'Pernambuco':9058931, 'Ceará':8794957}
popPorEstadoSer = pd.Series(popPorEstadoDic)
popPorEstadoSer
```

Nesta construção é criada um objeto `Series` onde o índice é extraído das chaves do dicionário.

Uma vez criado o acesso aos elementos repete a sintaxes típica dos dicionário **Python**.

In [ ]:
```
print("A população da Bahia, segundo o IBGE, é de", popPorEstadoSer
```

Por outro lado, diferente de um dicionário, `Series` também suportam operações no estilo array, como *slicing*.

In [ ]:
```
print(popPorEstadoSer['Bahia':'Pernambuco'])
```

Vamos explorar então as formas de ciar `Series` . Já vimos que podemos passar um conjunto de dados, na forma de uma lista ou de um *ndarray*. Neste caso os índices são criados como de forma sequencial como inteiros que correspondem aos índices do *ndarray*.

In [ ]:
```
títulos = [12, 8, 8, 7, 6, 4, 4, 4, 3, 3, 2, 2, 2, 1, 1, 1, 1]
brasileirão = pd.Series(títulos)
brasileirão
```

Mas podemos acrescentar os índices na forma de uma lista.

In [ ]:
```
campBrasileiros = ['Palmeiras', 'Santos', 'Flamengo', 'Corinthians'
                   'Cruzeiro', 'Fluminense', 'Vasco', 'Internaciona
                    'Bahia', 'Botafogo', 'Grêmio', 'Athletico-PR',
brasileirão = pd.Series(títulos, index=campBrasileiros)
print(brasileirão)
```

Quando a lista de índices é fornecida o valor pode ser apenas um escalar. Neste caso o valor é repetido para cada índice.

In [ ]:
```
brasileirão = pd.Series(1, index=campBrasileiros)
brasileirão
```

Como já vimos, os dados podem ser fornecidos na forma de um dicionário **Python**.

In [ ]:
```
dicBrasileirao = {2:'Santos', 4:'Corinthians', 3:'Flamengo',  1:'Pa
brasileirão = pd.Series(dicBrasileirao)
print(brasileirão)
print(type(brasileirão.values))
```

Mesmo quando fornecido um dicionário podemos escolher apenas um subconjunto dos elementos especificando uma lista de iíndices.

In [ ]:
```
brasileirão = pd.Series(dicBrasileirao, index=[2,3,4])
print(brasileirão)
```

In [ ]:
```
lCampBrasileiros = [('Palmeiras', 12, [1960, 1967, 1967, 1969, 1972
 ('Santos', 8, [1961, 1962, 1963, 1964, 1965, 1968, 2002, 2004]),
 ('Flamengo',   8, [1980, 1982, 1983, 1987, 1992, 2009, 2019, 2020]
 ('Corinthians', 7, [1990, 1998, 1999, 2005, 2011, 2015, 2017]),
 ('São Paulo', 6,   [1977, 1986, 1991, 2006, 2007, 2008]),
 ('Cruzeiro',   4, [1993, 1996, 2000, 2003, 2017, 2018]),
 ('Fluminense', 4, [1970, 1984, 2010, 2012]),
 ('Vasco', 4, [1974, 1989, 1997, 2000]),
 ('Internacional', 3,   [1975, 1976, 1979]),
 ('Atlético-MG', 3, [1937, 1971, 2021]),
 ('Bahia', 2, [1959, 1988]),
 ('Botafogo', 2, [1968, 1995]),
 ('Grêmio', 2, [1981, 1996]),
 ('Athletico-PR', 1, [2001]),
 ('Coritiba', 1, [1985]),
 ('Guarani', 1, [1978]),
 ('Sport', 1, [1987])]
```

Pandas `DataFrame`

Da mesma forma que os objetos `Series`, o `DataFrame` pode ser pensado como uma generalização de um *ndarray* **NumPy** ou como uma especialização de um dicionário **Python**.

Se um objeto `Series` é análogo a uma matriz unidimensional com índices flexíveis, um `DataFrame` pode ser visto como uma estrutura análoga a uma matriz bidimensional com índices de linha e nomes de colunas flexíveis. Você pode pensar em um `DataFrame` como uma sequência de objetos `Series` alinhados. Aqui, por "alinhado" queremos dizer que eles compartilham o mesmo índice.

In [ ]:
```
torcidas= {'Flamengo':46953599, 'Corinthians':30444799, 'Palmeiras'
torcidaSer = pd.Series(torcidas)
torcidaSer
```

In [ ]:
```
dicBrasileirao = {'Santos':8, 'Corinthians':7, 'Flamengo':8,  'Palm
brasileirão = pd.Series(dicBrasileirao)
brasileirão
```

In [ ]:
```
times = pd.DataFrame({'Títulos':brasileirão, 'Torcida':torcidaSer})
times
```

Assim como o objeto `Series`, o `DataFrame` possui um atributo de índice que dá acesso aos rótulos das linhas.

In [ ]:
```
times.index
```

Além disso, o `DataFrame` possui um atributo `columns`, que é um objeto Index que contém os rótulos das colunas.

In [ ]:
```
times.columns
```

Desta forma, o `DataFrame` pode ser pensado como uma generalização de um *ndarray* **NumPy** bidimensional, onde tanto as linhas quanto as colunas possuem um índice generalizado para acessar os dados.

O primeiro exemplo mostrou como criar um `DataFrame` a partir de duas `Series`. Também podemos construir um `DataFrame` a partir de uma lista de dicionários.

In [ ]:
```
dicBrasileirao = {'Santos':8, 'Corinthians':7, 'Flamengo':8,  'Palm
torcidas= {'Flamengo':46953599, 'Corinthians':30444799, 'Palmeiras'

times = pd.DataFrame([torcidas, dicBrasileirao], index=['Torcida',
times
```

No caso de faltarem algumas chaves nos dicionários, o **Pandas** irá preenchê-las com valores **NaN**.

In [ ]:
```
campBrasileiros = ['Palmeiras', 'Santos', 'Flamengo', 'Corinthians'
                   'Cruzeiro', 'Fluminense', 'Vasco da Gama', 'Inte
                    'Bahia', 'Botafogo', 'Grêmio', 'Athletico-PR',
títulos = [12, 8, 8, 7, 6, 4, 4, 4, 3, 3, 2, 2, 2, 1, 1, 1, 1]

for time, titulo in zip(campBrasileiros, títulos):
    dicBrasileirao[time] = titulo

times = ['Flamengo', 'Corinthians', 'São Paulo', 'Palmeiras', 'Vasc
         'Grêmio', 'Atlético-MG', 'Bahia', 'Internacional', 'Flumin

torcida = [46953599, 30444799, 22225800, 20225600, 13292800, 130784
           7718400, 7504000, 7289600, 6646400, 4288000, 4073600]

for time, torc in zip(times, torcida):
    torcidas[time] = torc

times = pd.DataFrame([torcidas, dicBrasileirao], index=['Torcida',
times
```

Podemos utilizar também dicionários de `Series`.

In [ ]:
```
campBrasileiros = ['Palmeiras', 'Santos', 'Flamengo', 'Corinthians'
                   'Cruzeiro', 'Fluminense', 'Vasco da Gama', 'Inte
                    'Bahia', 'Botafogo', 'Grêmio', 'Athletico-PR',
títulos = [12, 8, 8, 7, 6, 4, 4, 4, 3, 3, 2, 2, 2, 1, 1, 1, 1]

for time, titulo in zip(campBrasileiros, títulos):
    dicBrasileirao[time] = titulo

timTitulos = pd.Series(dicBrasileirao)

times = ['Flamengo', 'Corinthians', 'São Paulo', 'Palmeiras', 'Vasc
         'Grêmio', 'Atlético-MG', 'Bahia', 'Internacional', 'Flumin

torcida = [46953599, 30444799, 22225800, 20225600, 13292800, 130784
           7718400, 7504000, 7289600, 6646400, 4288000, 4073600]

for time, torc in zip(times, torcida):
    torcidas[time] = torc

timTorcida = pd.Series(torcidas)

times = pd.DataFrame({'Títulos':timTitulos, 'Torcida':timTorcida})
times
```

E, pensando num `DataFrame` como um generalização de um *ndarray* **NumPy** bidimensional, podemos construir um `DataFrame` usando uma *ndarray*.

In [ ]:
```
títulos = [12, 8, 8, 7]
torcida = [20225600, 6646400, 46953599, 30444799]
tittor = np.array([títulos, torcida])
print(tittor.T)
tittor = pd.DataFrame(tittor.T, columns=['Títulos', 'Torcida'],
                      index=['Palmeiras', 'Santos', 'Flamengo', 'Co

print(tittor)
```

Podemos ainda utilizar um *ndarray* estruturado, como o que vimos na aula anterior para construir um `DataFRame`

In [ ]:
```
alunos = ['nomeAluno01', 'nomeAluno02', 'nomeAluno03', 'nomeAluno04
matrícula = np.random.randint(0, 1000, 4)
prova_1 = np.random.uniform(0, 10, 4)

data = np.zeros(4, dtype={'names':('nome', 'nMatricula', 'prova_1')
                          'formats':('U50', 'i4', 'f4')})

data['nome'] = alunos
data['nMatricula'] = matrícula
data['prova_1'] = prova_1
print(data)
```

In [ ]:
```
dataDF = pd.DataFrame(data)
print(dataDF)
```

Pandas `Index`

Vimos aqui que ambos os objetos `Series` e `DataFrame` contêm um **índice** explícito que permite referenciar e modificar dados. Trata-se de um objeto `Index` que pode ser pensado como um *matriz imutável* ou como um *conjunto ordenado*. Vamos entender algumas das operações que podem ser feitas com objetos da classe `Index` .

Vamos começar criando um `Index` mais clássico.

In [ ]:
```
índice = pd.Index([i for i in range(5)])
índice
```

O `Index` funciona em, alguns contextos, como um array. Por exemplo, podemos usar a notação de indexação padrão do **Python** para recuperar valores ou *slicings*.

In [ ]:
```
print(índice[0])
print(índice[-1])
print(índice[1:3])
```

Os objetos da classe `Index` também tem muitos dos atributos presentes nos *ndarrays* **NumPy**.

In [ ]:
```
print(índice.size, índice.shape, índice.ndim, índice.dtype)
```

Entretanto os objetos `Index` são imutáveis, ou seja, eles não podem ser modificados pelos meios normais.

In [ ]:
```
try:
    índice[0] = 1
except Exception as e:
    print(e)
```

Indexando objetos Pandas

Já abordamos os principais objetos deo **Pandas** e suas principais características. Eles guardam uma relação estreita com os *ndarrays* de **NumPy** dos quais vimos as diversas formas de acessar e modificar.

Seja utilizando indexação direta, *slicing*, mascaramento ou alguma combinação das opções anteriores, os *ndarrays* podem ser manipulados de forma muito eficiente.

Mas como acessar os objetos **Pandas**? Os padrões utilizados no **Pandas** parecerão muito familiares para quem domina os objetos **NumPy**, ainda que existam algumas peculiaridades a serem observadas.

Acessando `Series`

Como já vimos, o objeto `Series`, da mesma forma que um dicionário, fornece um mapeamento de uma coleção de chaves para uma coleção de valores.

In [ ]:
```
print(lCampBrasileiros)
dCampBrasileiros = {}
for time, títulos, anos in lCampBrasileiros:
   for ano in anos:
       dCampBrasileiros[str(ano)] = time
print(dCampBrasileiros)
sCampBrasileiros = pd.Series(dCampBrasileiros)
print(sCampBrasileiros)
```

Podemos acessar então utilizando as chaves.

In [ ]:
```
print("O campeão de 1987 foi o", sCampBrasileiros['1987'])
```

Podemos pesquisar por uma chave para saber se ela faz parte do objeto.

In [ ]:
```
'2019' in sCampBrasileiros
#1030 in sCampBrasileiros
```

Temos então um índice de chaves.

In [ ]:
```
sCampBrasileiros.keys().sort_values()
```

In [ ]:
```
list(sCampBrasileiros.items())
```

Podemos então modificar os itens acessando via chave. Mas ainda, objetos `Series` podem ser incrementados da mesma forma que um dicionário: atribuindo uma nova chave com um valor.

In [ ]:
```
sCampBrasileiros['2024'] = 'Vasco da Gama'
print(sCampBrasileiros[-5:])
```

A classe `Series` utiliza então uma interface semelhante à de um dicionário. Entretanto ela fornece também a possibilidade de selecionar itens no estilo array, por meio dos mesmos mecanismos básicos dos *ndarrays* **NumPy**.

In [ ]:
```
print(sCampBrasileiros[:'1972']) # Dúvida aqui...
```

Ainda que tenhamos definidos os índices explicitamente como uma lista de strings, os objetos `Series` possuem um índice implícito inteiro.

In [ ]:
```
print(sCampBrasileiros[:3])
```

Podemos também utilizar máscaras.

In [ ]:
```
sCampBrasileiros[(sCampBrasileiros != 'Flamengo') &
                 (sCampBrasileiros!= 'Corinthians') &
                 (sCampBrasileiros != 'Palmeiras') &
                 (sCampBrasileiros != 'Santos')]
```

In [ ]:
```
sCampBrasileiros[['2020', '2021', '2022', '2023', '2024']]
```

Repare que, no *slicing* com o índice explícito o índice final é incluído na fatia, enquanto ao fatiar com um índice implícito o índice final é excluído da fatia, como esperado.

Reparem que, apesar dos anos serem números inteiros, a lista de índices foi criada com uma lista de strings.

O problema é que, se sua `Series` tiver um índice inteiro explícito, uma operação de indexação como `sCampBrasileiros[1960]` usará os índices explícitos, enquanto uma operação de *slicing* como `sCampBrasileiros[:1960]` usará o índice implícito no estilo **Python**.

In [ ]:
```
alunos = pd.Series(['nomeAluno01', 'nomeAluno02', 'nomeAluno03', 'n
                    index=[1, 2, 3, 4])

print(alunos)
```

In [ ]:
```
print(alunos[2])
```

In [ ]:
```
print(alunos.loc[:2])
```

Devido a essa confusão potencial no caso de índices inteiros, o **Pandas** fornece alguns atributos especiais que expõem o esquemas de indexação.

Primeiro, o atributo `loc` permite indexação e *slicing* que sempre faz referência ao índice explícito.

In [ ]:
```
alunos.loc[2]
```

In [ ]:
```
alunos.loc[:2]
```

O atributo `iloc` permite indexação e *slicing* que sempre faz referência ao índice implícito no estilo **Python**:

In [ ]:
```
alunos.iloc[2]
```

In [ ]:
```
alunos.iloc[:2]
```

Um princípio orientador do código **Python** é que “explícito é melhor que implícito”. A natureza explícita de `loc` e `iloc` os torna muito úteis na manutenção de código limpo e legível, especialmente no caso de índices inteiros.

Acessando `DataFrame`

Como vimoa anteriormente, um `DataFrame` pode ser tratado de muitas maneiras: como um array bidimensional ou estruturado, e de outras maneiras como um dicionário de estruturas `Series` compartilhando o mesmo índice.

A primeira analogia que consideraremos é o DataFrame como um dicionário de objetos `Series`.

In [ ]:
```
times
```

As `Series` individuais que compõem as colunas do `DataFrame` podem ser acessadas por meio de indexação no estilo de dicionário.

In [ ]:
```
times['Títulos']
```

De forma análoga, podemos usar acesso no estilo de atributo com o nome das coluna.

In [ ]:
```
times.Torcida
```

In [ ]:
```
times.Torcida is times['Torcida']
```

Tal como acontece com os `Series` a sintaxe estilo dicionário também pode ser usada para modificar o `DataFrame` ou adicionar uma nova coluna.

In [ ]:
```
times['TitulosPorTorcedor'] = times.Títulos / times.Torcida
times
```

O `DataFrame` também pode sere tratado como um array bidimensional aprimorado.

In [ ]:
```
times.values
```

Pensando o `DataFrame` como uma matriz bidimensional, podemos aplicar diversas formas de manipular o *ndarray* ao `DataFrame` .

In [ ]:
```
times.T
```

Quando se trata de indexação de objetos `DataFrame` , entretanto, fica claro que a indexação de colunas no estilo de dicionário impede nossa capacidade de simplesmente tratá-la como uma matriz **NumPy**.

In [ ]:
```
times.values[7]
```

Na realidade, quando passamos um único índice, estamos nos referindo a uma coluna.

In [ ]:
```
times['Títulos']
```

Portanto, para indexação em estilo array, precisamos utilizar outros recursos: os indexadores `loc`, `iloc` e `ix`.

Usando o indexador `iloc`, podemos indexar o array subjacente como se fosse um array NumPy simples (usando o índice implícito no estilo Python), mas o índice `DataFrame` e os rótulos das colunas são mantidos no resultado.

In [ ]:
```
times.iloc[7]
```

Da mesma forma, usando o indexador `loc`, podemos indexar os dados subjacentes em um estilo semelhante a um array, mas usando o índice explícito e os nomes das colunas.

In [ ]:
```
times.loc[:'Bahia', :'Torcida']
```

Operando com dados em Pandas

Uma das peças essenciais do **NumPy** é a capacidade de realizar operações rápidas entre elementos, tanto com aritmética básica (adição, subtração, multiplicação, etc.) quanto com operações mais sofisticadas (funções trigonométricas, funções exponenciais e logarítmicas, etc.).

O **Pandas** herda grande parte dessa funcionalidade do **NumPy**. No entanto, o **Pandas** inclui algumas alterações úteis:

- Para operações unárias como negação e funções trigonométricas as *ufuncs* preservarão rótulos de índice e coluna na saída.
- Para operações binárias como adição e multiplicação, o Pandas alinhará automaticamente os índices ao passar os objetos para o *ufunc*.

Vamos tentar entender como isto melhora o processamento em relação a os *ndarrays* de **NumPy**.

Como o **Pandas** foi projetado para funcionar com **NumPy**, qualquer *ufunc* **NumPy** funcionará em objetos **Pandas**, sejam `Series` ou `DataFrame`.

In [ ]:
```
pSerie = pd.Series(np.random.uniform(0, np.pi, 10))
pSerie
```

In [ ]:
```
pDataFrame = pd.DataFrame(np.random.randint(0, 256, (5, 5)),
                          columns=['alpha', 'beta', 'gamma', 'delta', 'gamm
pDataFrame
```

Se aplicarmos uma **NumPy** *ufunc* em qualquer um desses objetos, o resultado será outro objeto **Pandas** com os mesmos índices.

In [ ]:
```
np.log1p(pDataFrame)
```

In [ ]:
```
np.sin(pSerie)
```

UFuncs: alinhamento do índice

Para operações binárias em dois objetos `Series` ou `DataFrame` , o **Pandas** alinhará os índices no processo de execução da operação.

Imaginemos que temos acesso a um conjunto de dados simples contendo a altura em polegadas e o peso em libras de 25.000 humanos diferentes de 18 anos de idade. Este conjunto de dados pode ser usado, por exemplo, para construir um modelo que pode prever as alturas ou pesos de um ser humano.

Os dados utilizados neste exemplo estão disponíveis na plataforma Kaggle (https://www.kaggle.com/datasets/burnoutminer/heights-and-weights-dataset?resource=download). Inicialmente vamos trabalhar um pequeno subconjunto destes dados.

In [ ]:
```
dados = [(65.78331,112.9925), (71.51521,136.4873), (69.39874,153.02
 (68.69784,123.3024), (69.80204,141.4947), (70.01472,136.4623), (67

dataSet = pd.DataFrame(dados, columns=['altura', 'peso'])
dataSet
```

Com este `DataFrame` podemos calcular a relação peso altura de cada um dos indivíduos cadastrados.

In [ ]:
```
dataSet["altura/peso"] = dataSet.altura / dataSet.peso
dataSet
```

Entretanto, se o mesmo conjunto de dados estivesse incompleto.

In [ ]:
```
#dados = [(65.78331,112.9925), (71.51521,136.4873), (69.39874,153.0
# (68.69784,123.3024), (69.80204,141.4947), (70.01472,136.4623), (6

pesoSer = pd.Series([112.9925, 153.0269, 142.3354, 123.3024, 141.49
                     index = ["p01", "p03", "p04", "p06", "p07", "p0
alturaSer = pd.Series([65.78331, 71.51521, 68.2166, 67.78781,
                       68.69784, 69.80204, 70.01472,66.78236],
                        index = ["p01", "p02", "p04", "p05", "p06",
print(pesoSer)
print(alturaSer)
```

In [ ]:
```
dataSet = pd.DataFrame({"peso":pesoSer, "altura":alturaSer})
dataSet
```

Tanto conseguimos calcular a relação entre altura e peso a partir dos objetos `Series` ...

In [ ]:
```
alturaPesoSer = dataSet.altura / dataSet.peso
alturaPesoSer
```

quanto a partir das colunas do `DataFrame` .

In [ ]:
```
dataSet["altura/peso"] = dataSet.altura / dataSet.peso
dataSet
```

Qualquer item para o qual um ou outro objeto não tenha uma entrada é marcado com `NaN` , ou "Não é um número", que é como o **Pandas** marca os dados ausentes.

Esta correspondência de índice é implementada desta forma para qualquer uma das expressões aritméticas integradas do Python; quaisquer valores ausentes são preenchidos com NaN por padrão.

Veja outros exemplos.

In [ ]:
```
img = np.random.randint(0, 256, (5, 5), dtype=np.uint8)
print(img)
print(img.sum())
```

In [ ]:
```
img = pd.DataFrame(img, columns=['C01', 'C02', 'C03', 'C04', 'C05']
                   index=['L01', 'L02', 'L03', 'L04', 'L05'])
print(img)
print(img.sum())
```

In [ ]:
```
masc = np.ones((3, 3), dtype=np.uint8)
masc = pd.DataFrame(masc, columns=['C02', 'C03', 'C04'],
                    index=['L02', 'L03', 'L04'])

print(masc)
```

In [ ]:
```
imgMasc = img * masc
print(imgMasc)
print(imgMasc.sum().sum())
```

Podemos usar ainda o método aritmético do objeto associado e passar o `fill_value` desejado para ser usado no lugar das entradas ausentes.

In [ ]:
```
imgMasc = img.mul(masc, fill_value=0)
print(imgMasc)
```

Pandas, da mesma forma que **NumPy**, fornece métodos específicos que implementam a sobrecarga dos operadores aritméticos tradicionais, expandindo suas possibilidades de uso.

Operador	Método Pandas
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Podemos utilizar estes operadores também para manipular objetos `Series` e `DataFrame` juntos.

In [ ]:
```
min = imgMasc['L02':'L04'].min()
min = min['C02':'C04'].min()
print(min)
min = pd.Series(min, index=['C02', 'C03', 'C04'])
print(min)
```

In [ ]:
```
imgNorm = imgMasc - min
print(imgNorm)
```

De acordo com as regras de *broadcasting* do **NumPy**, que abordamos na aula anterior, a subtração entre uma matriz bidimensional e um array unidimensional é aplicada por linhas.

No Pandas, a convenção funciona de forma semelhante em linhas por padrão:

In [ ]:
```
#npyImgNorm = imgMasc.values - min.values    # Numpy array
npyImgNorm = imgMasc.values[:,1:4] - min.values   # Numpy array
print(npyImgNorm)
```

Repare que neste como em outros exemplos que envolvam operações entre `DataFrame e Series` os índices são alinhados automaticamente.

Entretanto, ee preferir operar em colunas, você pode usar os métodos do objeto, mencionados anteriormente, especificando a palavra-chave `axis` .

In [ ]:
```
min = pd.Series(16, index=['L02', 'L03', 'L04'])
imgNorm = imgMasc.subtract(min, axis=0)
print(imgNorm)
```