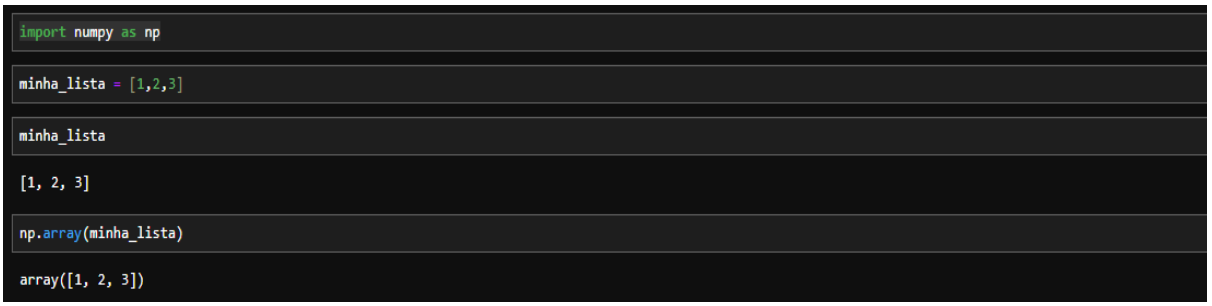
Relatório 4 - Prática: Principais Bibliotecas e Ferramentas Python para Aprendizado de Máquina (I)

Higor Miller Grassi

No primeiro módulo, configuramos o Jupyter Notebook em nossa maquina, ajudando na instalação e como mexer nele, configurando o ambiente de trabalho.

Após tudo isso, é ensinado a biblioteca numpy no python, sendo essencial na linguagem Python para computação científica e análise de dados, fornecendo a criação e manipulação de arrays e matrizes multidimensionais, além de efetuar operações matemáticas entre arrays.

Aqui, importamos a biblioteca no numpy e criamos nossa lista e matriz, onde depois declaramos ambas como um array utilizando o np.array().



Np.arange(), quando colocamos dois números como parâmetro, ele faz um array do primeiro até o segundo número colocado, agora quando colocamos um terceiro elemento, ele realiza o espaçamento correto, funcionando como um intervalo.

Np.zeros() e np.ones(), onde passamos um parâmetro, e o valor da matriz/array é preenchido com respectivos números, e também, o np.eye() que é usada para criar uma matriz identidade.

```
arr = np.zeros((5,5))
```

```
arr
```

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

```
arr_one = np.ones((5,5))
```

```
arr_one
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

```
arr_eye = np.eye(5)
```

```
arr_eye
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

Linspace(), utilizado para gerar um array com números igualmente espaçados, diferentemente do arange, podendo especificar o tamanho do espaçamento, onde no arange é fixo.

```
arr_lins = np.linspace(0,10,5)
```

```
arr_lins
```

```
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

Np.random nos permite criar números aleatórios de 0 a 1, e o np.random.rand colocar o tamanho do array que quisermos

```
arr_rand = np.random.rand(10)
```

```
arr_rand
```

```
array([0.26938695, 0.73878986, 0.40957803, 0.555605  , 0.67102221,
       0.60507205, 0.95748522, 0.16141365, 0.16896027, 0.12459508])
```

```
arr_rand2 = np.random.rand(3,3,3)
```

```
arr_rand2
```

```
array([[[0.15991641, 0.38361013, 0.31245888],
        [0.29008148, 0.82000332, 0.83408326],
        [0.58878547, 0.285967  , 0.61619254]],

       [[0.81998742, 0.29540165, 0.44963071],
        [0.22629245, 0.53972688, 0.80349486],
        [0.85797108, 0.10518407, 0.25432317]],

       [[0.10460124, 0.83579431, 0.242847  ],
        [0.56396913, 0.7278422 , 0.75050924],
        [0.4411314 , 0.38112541, 0.00975124]]])
```

Np.random.randn cria um array com uma distribuição uniforme com desvio padrão entre 0 e 1 e o np.random.randint podemos colocar o valor mínimo e máximo e escolher o tamanho do array.

Na imagem, o randn cria uma matriz de dimensão 3x3x3 e o randint cria um array com 5 números de 0 a 10(onde o 10 não pode ser incluso).

```
arr_rand3 = np.random.randn(3,3,3)
```

```
arr_rand3
```

```
array([[[ 1.75580279, -0.90067796, -1.39218305],
        [-0.52460842,  1.54015485,  1.75439997],
        [-0.43015073, -0.41335097, -0.04368778]],

       [[ 0.67682224, -1.36657187, -0.78215864],
        [-0.99431373, -0.10916139,  1.21606401],
        [-0.42158769,  0.64913665, -0.17749905]],

       [[-1.08760778,  0.24910893, -0.10151757],
        [-1.4036838 ,  1.49281575, -0.69912628],
        [-0.34913655, -2.02337153,  1.07501337]]])
```

```
arr_rand4 = np.random.randint(0,10,5)
```

```
arr_rand4
```

```
array([8, 8, 3, 3, 1])
```

Np.round gera uma array de tamanho escolhido, com números entre 0 e 1, arredondando, por exemplo, o número 0,4 vai para 0 e o 0,6 para 1.

```
arr_round = np.round(np.random.rand(3,3,3))
```

```
arr_round
```

```
array([[[0., 1., 0.],
        [1., 0., 1.],
        [0., 1., 1.]],

       [[1., 0., 0.],
        [1., 0., 0.],
        [1., 1., 1.]],

       [[0., 0., 0.],
        [0., 1., 0.],
        [1., 1., 0.]]])
```

Np.reshape() transforma um array em uma matriz com as dimensões especificadas

```
arr_shape = np.random.rand(20)
```

```
arr_shape
```

```
array([0.61382724, 0.78672926, 0.77250446, 0.66692541, 0.5233253 ,
       0.42272834, 0.55899162, 0.0451963 , 0.12595108, 0.60758466,
       0.5881559 , 0.7167876 , 0.34724693, 0.01581279, 0.8136497 ,
       0.43204589, 0.13530849, 0.21284378, 0.16580637, 0.88557278])
```

```
arr_shape = arr_shape.reshape(4,5)
```

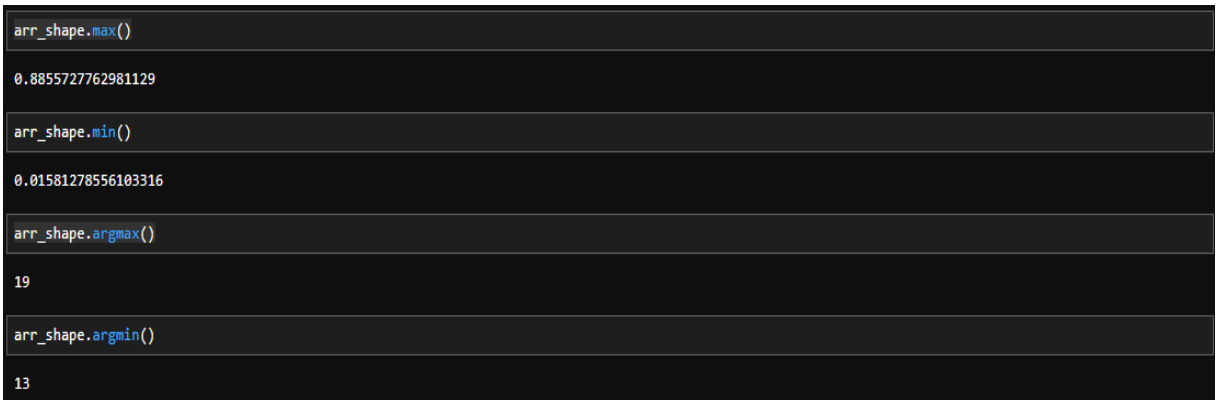
```
arr_shape
```

```
array([[0.61382724, 0.78672926, 0.77250446, 0.66692541, 0.5233253 ],
       [0.42272834, 0.55899162, 0.0451963 , 0.12595108, 0.60758466],
       [0.5881559 , 0.7167876 , 0.34724693, 0.01581279, 0.8136497 ],
       [0.43204589, 0.13530849, 0.21284378, 0.16580637, 0.88557278]])
```

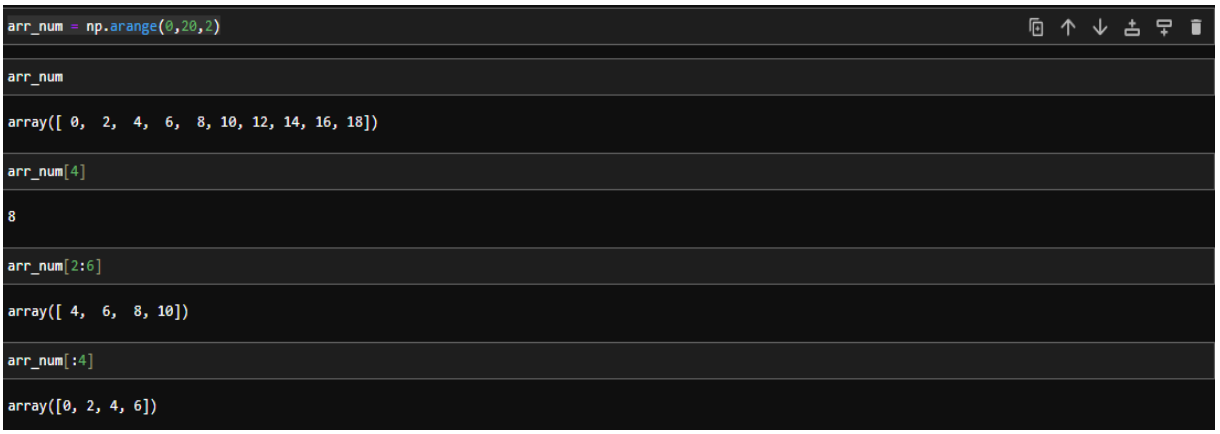
```
arr_shape.shape
```

```
(4, 5)
```

Para descobrir qual o valor máximo/mínimo e em que posição que encontramos os mesmo, utilizamos o .max(), .min(), .argmax() e o .argmin() respectivamente.



Para conseguir acessar informações e posições específicas, utilizamos [:], onde colocamos o valor inicial e final desejados.



Podemos também definir um valor para x posições.

Como o python precisa ser necessariamente mais eficiente e rápido, caso você queira copiar um array, não pode apenas igualar, pois caso precise alterar algum valor da cópia o original também mudará. Basta colocar no final o .copy() para podermos manter da forma que quisermos sem causar alterações indesejadas.

```
arr_num = np.arange(50).reshape(5,10)
```

```
arr_num
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

```
arr_num.shape
```

```
(5, 10)
```

```
arr_num[:3][:]
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])
```

```
arr_num[:3]
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])
```

```
arr_num2 = arr_num[:3].copy()
```

```
arr_num2
```

```
array([[100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100, 100, 100, 100]])
```

```
arr_num2[:] = 5
```

```
arr_num2
```

```
array([[5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
       [5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
       [5, 5, 5, 5, 5, 5, 5, 5, 5, 5]])
```

```
arr_num
```

```
array([[100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [ 30,  31,  32,  33,  34,  35,  36,  37,  38,  39],
       [ 40,  41,  42,  43,  44,  45,  46,  47,  48,  49]])
```

Conseguimos efetuar um booleano de forma prática, para obter resultados dentro de um array.

```
bol = arr_num > 50
```

```
bol
```

```
array([[ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True],
       [False, False, False, False, False, False, False, False, False,
        False],
       [False, False, False, False, False, False, False, False, False,
        False]])
```

```
arr_num[bol]
```

```
array([100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
       100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
       100, 100, 100, 100])
```

Operações matemáticas entre arrays também são possíveis, de forma simples. Porém, pode causar uma mensagem que, quando uma divisão é impossível, ele retorna “nan” e quando infinita retorna uma mensagem “int”.

```
arr_arange = np.arange(0,16)
```

```
arr_arange
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

```
arr_arange + arr_arange
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30])
```

```
arr_arange * arr_arange
```

```
array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 144,
       169, 196, 225])
```

```
arr_arange - arr_arange
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
arr_arange / arr_arange
```

```
C:\Users\Higor\AppData\Local\Temp\ipykernel_44900\1233395723.py:1: RuntimeWarning: invalid value encountered in divide
  arr_arange / arr_arange
array([nan,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
       1.,  1.,  1.])
```

```
1/arr_arange
```

```
C:\Users\Higor\AppData\Local\Temp\ipykernel_44900\3395063807.py:1: RuntimeWarning: divide by zero encountered in divide
  1/arr_arange
array([       inf, 1.        , 0.5       , 0.33333333, 0.25      ,
       0.2       , 0.16666667, 0.14285714, 0.125     , 0.11111111,
       0.1       , 0.09090909, 0.08333333, 0.07692308, 0.07142857,
       0.06666667])
```

Da mesma forma, podemos realizar operações matemáticas com os números dentro do array, como multiplicar, somar,dividir ou subtrair algum número.

```
arr_arange * 10
```

```
array([  0,  10,  20,  30,  40,  50,  60,  70,  80,  90, 100, 110, 120,
       130, 140, 150])
```

```
arr_arange + 10
```

```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25])
```

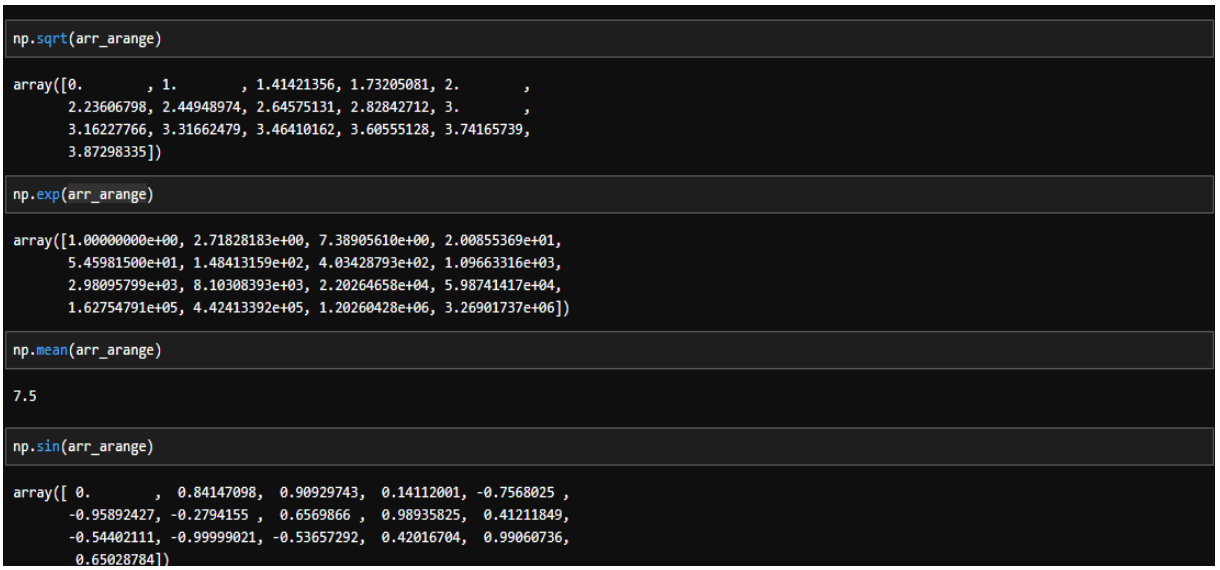
```
arr_arange - 10
```

```
array([-10,  -9,  -8,  -7,  -6,  -5,  -4,  -3,  -2,  -1,   0,   1,   2,
         3,   4,   5])
```

```
arr_arange / 10
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2,
       1.3, 1.4, 1.5])
```

Conseguimos também, descobrir a raiz quadrada, media e o seno dos respectivos números.



```
np.sqrt(arr_arange)

array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ,
       3.16227766, 3.31662479, 3.46410162, 3.60555128, 3.74165739,
       3.87298335])

np.exp(arr_arange)

array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03, 2.20264658e+04, 5.98741417e+04,
       1.62754791e+05, 4.42413392e+05, 1.20260428e+06, 3.26901737e+06])

np.mean(arr_arange)

7.5

np.sin(arr_arange)

array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849,
       -0.54402111, -0.99999021, -0.53657292,  0.42016704,  0.99060736,
        0.65028784])
```

E para descobrir qual o valor máximo do array, temos duas maneiras para chegar nesta resposta.

```
np.max(arr_arange)

15

arr_arange.max()

15
```

Lista de exercício de numpy:

Também aprendemos sobre a biblioteca Pandas, que é usada para a manipulação de dados, comparado ao vídeo como o excel do Python.

No primeiro vídeo, vemos como usar o Series, no começo, importamos a biblioteca do numpy e pandas e criamos uma lista, array e um dicionário e utilizamos o series para igualar duas variáveis.

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

```
labels = ('a', 'b', 'c')
```

```
minha_lista = (5,10,15)
arr1 = np.array([5,10,15])
dic = {'a': 5, 'b': 10, 'c':15}
```

```
pd.Series(data = minha_lista, index=labels)
```

```
a     5
b    10
c    15
dtype: int64
```

Depois, aprendemos a acessar um valor específico dentro da série criada, e também, não precisamos necessariamente colocar “data =”, apenas colocamos na ordem em que queremos o nome da variável e dá certo.

```
series = pd.Series(data = minha_lista, index=labels)
```

```
series["b"]
```

```
10
```

```
pd.Series(arr1, labels)
```

```
a     5
b    10
c    15
dtype: int32
```

Caso queiramos somar duas séries, o python efetua automaticamente variáveis com o mesmo nome, independente da ordem seguida, e caso não houver manipulação matemática, aparece a mensagem NAN.

```
serie1 = pd.Series([1,2,3,4], index=["Mexico", "Brasil", "EUA", "Argentina"])
```

```
serie1
```

```
Mexico       1
Brasil       2
EUA          3
Argentina    4
dtype: int64
```

```
serie2 = pd.Series([1,2,3,4], index=["Brasil", "Mexico", "EUA", "Colombia"])
```

```
serie2
```

```
Brasil      1
Mexico      2
EUA         3
Colombia    4
dtype: int64
```

```
serie1+serie2
```

```
Argentina    NaN
Brasil       3.0
Colombia     NaN
EUA          6.0
Mexico       3.0
dtype: float64
```

No segundo vídeo, nos vimos sobre data frame que de forma resumida seria uma estrutura de dados bidimensional, criamos uma com as linhas A,B,C,D,E e colunas W,X,Y,Z com os valores de forma randômica.

```
dataf = pd.DataFrame(np.random.randn(5,4), index = "A B C D E".split(), columns="W X Y Z".split())
```

```
dataf
```

	W	X	Y	Z
A	0.833029	0.975720	-0.388239	0.783316
B	-0.708954	0.586847	-1.621348	0.677535
C	0.026105	-1.678284	0.333973	-0.532471
D	2.117727	0.197524	2.302987	0.729024
E	-0.863091	0.305632	0.243178	0.864165

Caso pegamos apenas uma coluna, podemos utilizar o comando dataf[“X”], sendo assim, ele é tratado como uma série, também podemos concatenar duas colunas.

```
dataf["X"]
```

```
A    0.975720
B    0.586847
C   -1.678284
D    0.197524
E    0.305632
Name: X, dtype: float64
```

```
type(dataf["X"])
```

```
pandas.core.series.Series
```

```
type(dataf)
```

```
pandas.core.frame.DataFrame
```

```
dataf[["W", "X"]]
```

	W	X
A	0.833029	0.975720
B	-0.708954	0.586847
C	0.026105	-1.678284
D	2.117727	0.197524
E	-0.863091	0.305632

Para criar uma nova coluna, basta colocar “dataf[“new”] = a soma de duas colunas”, porém, para remover, se apenas colocar o dataf.drop(), ela apenas sai temporariamente, se pegarmos os dados do data frame, veremos que não foi removido com sucesso.

```
dataf["new"] = dataf["X"] + dataf["W"]
```

```
dataf.drop("new", axis=1)
```

	W	X	Y	Z
A	0.833029	0.975720	-0.388239	0.783316
B	-0.708954	0.586847	-1.621348	0.677535
C	0.026105	-1.678284	0.333973	-0.532471
D	2.117727	0.197524	2.302987	0.729024
E	-0.863091	0.305632	0.243178	0.864165

Para remover permanentemente, usamos o inplace = true.

```
dataf.drop("new", axis=1, inplace = True)
```

```
dataf
```

	W	X	Y	Z
A	0.833029	0.975720	-0.388239	0.783316
B	-0.708954	0.586847	-1.621348	0.677535
C	0.026105	-1.678284	0.333973	-0.532471
D	2.117727	0.197524	2.302987	0.729024
E	-0.863091	0.305632	0.243178	0.864165

Usando o loc, podemos chegar a números específicos, no exemplo onde pegamos apenas a linha “A”, ela é considerada uma series também.

```
dataf.loc["A","X"]
```

```
0.9757196803436843
```

```
dataf.loc["A"]
```

```
W    0.833029
X    0.975720
Y   -0.388239
Z    0.783316
Name: A, dtype: float64
```

```
dataf.loc[["A", "B"], ["X", "Y"]]
```

	X	Y
A	0.975720	-0.388239
B	0.586847	-1.621348

E para localizações especificando o intervalo, utilizamos o iloc.

```
dataf.iloc[1:3,2:]
```

	Y	Z
B	-1.621348	0.677535
C	0.333973	-0.532471

No terceiro vídeo, visualizamos duas formas de como manipular booleanos, com duas maneiras de impressão do data frame diferentes, onde o primeiro retorna apenas true ou false, e o segundo retorne NAN, nas colunas que primordialmente seriam nulas.

```
bol = dataf > 0
```

```
bol
```

	W	X	Y	Z
A	True	True	False	True
B	False	True	False	True
C	True	False	True	False
D	True	True	True	True
E	False	True	True	True

```
dataf[bol]
```

	W	X	Y	Z
A	0.833029	0.975720	NaN	0.783316
B	NaN	0.586847	NaN	0.677535
C	0.026105	NaN	0.333973	NaN
D	2.117727	0.197524	2.302987	0.729024
E	NaN	0.305632	0.243178	0.864165

Quando queremos por exemplo apenas coluna do X maiores a 0, é impresso apenas as linhas em que esse valor não se encontra, excluindo a linha nula.

```
dataf[dataf["X"]>0]
```

	W	X	Y	Z
A	0.833029	0.975720	-0.388239	0.783316
B	-0.708954	0.586847	-1.621348	0.677535
D	2.117727	0.197524	2.302987	0.729024
E	-0.863091	0.305632	0.243178	0.864165

A Coluna Y, apenas quando o X não é nulo.

```
dataf[dataf["X"]>0]["Y"]
```

```
A   -0.388239
B   -1.621348
D    2.302987
E    0.243178
Name: Y, dtype: float64
```

Para utilizar duas comparações, é utilizado o & para substituir o “and”, e o | para substituir o “or”.

```
dataf[(dataf["X"]>0)&(dataf["Y"]>1)]
```

	W	X	Y	Z
D	2.117727	0.197524	2.302987	0.729024

Para resetar o index, usamos o dataf.reset_index(), lembrando de utilizar o inplace=true para ser de forma definitiva.

```
dataf.reset_index(inplace=True)
```

```
dataf
```

	index	W	X	Y	Z
0	A	0.833029	0.975720	-0.388239	0.783316
1	B	-0.708954	0.586847	-1.621348	0.677535
2	C	0.026105	-1.678284	0.333973	-0.532471
3	D	2.117727	0.197524	2.302987	0.729024
4	E	-0.863091	0.305632	0.243178	0.864165

Desta forma, setamos o estado como uma nova coluna.

```
col = "SP RJ MS PR SC".split()
```

```
dataf["Est"] = col
```

```
dataf
```

	index	W	X	Y	Z	Est
0	A	0.833029	0.975720	-0.388239	0.783316	SP
1	B	-0.708954	0.586847	-1.621348	0.677535	RJ
2	C	0.026105	-1.678284	0.333973	-0.532471	MS
3	D	2.117727	0.197524	2.302987	0.729024	PR
4	E	-0.863091	0.305632	0.243178	0.864165	SC

Para substituir os índices da linha pelo estado, novamente lembrando de utilizar o inplace=true.

```
dataf.set_index("Est", inplace=True)
```

```
dataf
```

Est	index	W	X	Y	Z
SP	A	0.833029	0.975720	-0.388239	0.783316
RJ	B	-0.708954	0.586847	-1.621348	0.677535
MS	C	0.026105	-1.678284	0.333973	-0.532471
PR	D	2.117727	0.197524	2.302987	0.729024
SC	E	-0.863091	0.305632	0.243178	0.864165

Índice de multiníveis permite organizar dados em estruturas mais complexas, sendo muito útil quando você deseja trabalhar com conjuntos de dados que têm mais de uma dimensão de agrupamento, como por exemplo, estado, cidade, bairro e rua.

```
# Níveis de índice
outside = ["G1", "G1", "G1", "G2", "G2", "G2"]
inside = [1, 2, 3, 1, 2, 3]
hier_index = list(zip(outside, inside))
hier_index = pd.MultiIndex.from_tuples(hier_index)
```

```
dataf = pd.DataFrame(np.random.rand(6,2),hier_index, columns=["A", "B"])
```

```
dataf
```

		A	B
G1	1	0.184795	0.967445
	2	0.879504	0.119520
	3	0.690088	0.904004
G2	1	0.604129	0.491408
	2	0.793152	0.177025
	3	0.342030	0.255531

Com o loc, visto anteriormente, conseguimos capturar um índice em especifico.

```
dataf.loc["G1"]
```

	A	B
1	0.184795	0.967445
2	0.879504	0.119520
3	0.690088	0.904004

Conseguimos definir/mudar o nome dos index.

```
dataf.index.names = ["Bairro","Estado"]
```

```
dataf
```

		A	B
Bairro	**Estado**		
G1	**1**	0.184795	0.967445
	2	0.879504	0.119520
	3	0.690088	0.904004
G2	**1**	0.604129	0.491408
	2	0.793152	0.177025
	3	0.342030	0.255531

Para encontrar um valor em específico, assim como o loc, conseguimos utilizar o xs, ele acaba sendo mais vantajoso pois podemos ser mais específico em relação a qual dados nós queremos.

```
dataf.xs("G1")
```

	A	B
Estado		
1	0.184795	0.967445
2	0.879504	0.119520
3	0.690088	0.904004

```
dataf.xs(1,level="Estado")
```

	A	B
Bairro		
G1	0.184795	0.967445
G2	0.604129	0.491408

Com os dados ausentes, é criado um dicionário, onde propositalmente é deixado alguns espaços sem valor.

```
dic = {'A': [1, 2, np.nan], 'B': [8, np.nan, np.nan], 'C': [4, 6, 8]}
```

```
dt = pd.DataFrame(dic)
```

```
dt
```

	A	B	C
0	1.0	8.0	4
1	2.0	NaN	6
2	NaN	NaN	8

Para dropar esses valores, é utilizado o dropna(), porém, com isso, a linha toda é dropada.

```
dt.dropna()
```

	A	B	C
0	1.0	8.0	4

Com o thresh, é utilizado uma condição para que, apenas a linha com dois valores no mínimo que são nulos, seja dropada.

```
dt.dropna(thresh=2)
```

	A	B	C
0	1.0	8.0	4
1	2.0	NaN	6

Para alterarmos o dado em que está nulo, é utilizado o fillna, no primeiro exemplo, é preenchido com a palavra nulo, no segundo com a média da coluna A, e no último exemplo é utilizado o ffill, onde ele pega o último valor da coluna antes do valor nulo, e se propaga aos demais.

```
dt.fillna(value="nulo")
```

	A	B	C
0	1.0	8.0	4
1	2.0	nulo	6
2	nulo	nulo	8

```
dt["A"].fillna(value = dt["A"].mean())
```

```
0    1.0
1    2.0
2    1.5
Name: A, dtype: float64
```

```
dt.fillna(method="ffill")
```

```
C:\Users\Higor\AppData\Local\Temp\ipykernel_18256\910458961.py:1: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill() or obj.bfill() instead.
  dt.fillna(method="ffill")
```

	A	B	C
0	1.0	8.0	4
1	2.0	8.0	6
2	2.0	8.0	8

O group by em com data frame é muito utilizado para afunilar resultados conforme necessário.

É criado um data frame com vendas de uma empresa, com o valor associado a cada funcionário.

```
data = {
    'Empresa': ['Google', 'Google', 'Microsoft', 'Microsoft', 'Facebook', 'Facebook'],
    'Nome': ['Sam', 'Charlie', 'Amy', 'Vanessa', 'Carl', 'Sarah'],
    'Venda': [200, 120, 340, 124, 243, 350]
}
```

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

```
data_group
```

	Empresa	Nome	Venda
0	Google	Sam	200
1	Google	Charlie	120
2	Microsoft	Amy	340
3	Microsoft	Vanessa	124
4	Facebook	Carl	243
5	Facebook	Sarah	350

Utilizamos o group by para saber as vendas de cada empresa e de cada funcionario

```
group = data_group.groupby("Empresa")
```

```
group
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000012CEA0A62D0>
```

```
group.sum()
```

Empresa	Nome	Venda
Facebook	CarlSarah	593
Google	SamCharlie	320
Microsoft	AmyVanessa	464

```
group = data_group.groupby("Nome")
```

```
group.sum()
```

Nome	Empresa	Venda
Amy	Microsoft	340
Carl	Facebook	243
Charlie	Google	120
Sam	Google	200
Sarah	Facebook	350
Vanessa	Microsoft	124

Já o método de sum mostra a soma de cada empresa, qual o total de vendas. E o describe, mostra várias informações, sejam elas a média, o mínimo, máximo e as porcentagem das respectivas vendas.

```
group.sum()
```

Empresa	Nome	Venda
Facebook	CarlSarah	593
Google	SamCharlie	320
Microsoft	AmyVanessa	464

```
group.describe()
```

	Venda							
	count	mean	std	min	25%	50%	75%	max
Empresa								
Facebook	2.0	296.5	75.660426	243.0	269.75	296.5	323.25	350.0
Google	2.0	160.0	56.568542	120.0	140.00	160.0	180.00	200.0
Microsoft	2.0	232.0	152.735065	124.0	178.00	232.0	286.00	340.0

Também é possível pegar o total de vendas de apenas um funcionário, neste caso o Carl.

```
group.sum().loc["Carl"]
```

```
Empresa    Facebook
Venda           243
Name: Carl, dtype: object
```

Criei 3 dataframes diferentes, com o nome, idade, altura e peso

```
df1 = pd.DataFrame({
    'Nome': ['Alice', 'Bruno', 'Carlos', 'Diana'],
    'Idade': [25, 30, 22, 28],
    'Altura': [1.65, 1.75, 1.80, 1.60],
    'Peso': [60, 80, 75, 55]
})
```

```
df1
```

	Nome	Idade	Altura	Peso
0	Alice	25	1.65	60
1	Bruno	30	1.75	80
2	Carlos	22	1.80	75
3	Diana	28	1.60	55

```python
df2 = pd.DataFrame({
    'Nome': ['Eduardo', 'Fernanda', 'Gabriel', 'Helena'],
    'Idade': [45, 32, 29, 24],
    'Altura': [1.85, 1.70, 1.90, 1.65],
    'Peso': [85, 68, 90, 58]
})
```

```python
df2
```

	Nome	Idade	Altura	Peso
0	Eduardo	45	1.85	85
1	Fernanda	32	1.70	68
2	Gabriel	29	1.90	90
3	Helena	24	1.65	58

```python
df3 = pd.DataFrame({
    'Nome': ['Igor', 'Julia', 'Karla', 'Luis'],
    'Idade': [34, 27, 31, 40],
    'Altura': [1.76, 1.68, 1.73, 1.82],
    'Peso': [70, 59, 63, 88]
})
```

```python
df3
```

	Nome	Idade	Altura	Peso
0	Igor	34	1.76	70
1	Julia	27	1.68	59
2	Karla	31	1.73	63
3	Luis	40	1.82	88

Com a concatenação, ficou assim, realizando as unioes com as colunas de mesmo nome.

```python
pd.concat([df1,df2,df3])
```

	Nome	Idade	Altura	Peso
0	Alice	25	1.65	60
1	Bruno	30	1.75	80
2	Carlos	22	1.80	75
3	Diana	28	1.60	55
0	Eduardo	45	1.85	85
1	Fernanda	32	1.70	68
2	Gabriel	29	1.90	90
3	Helena	24	1.65	58
0	Igor	34	1.76	70
1	Julia	27	1.68	59
2	Karla	31	1.73	63
3	Luis	40	1.82	88

```
pd.concat([df1,df2,df3], axis=1)
```

	Nome	Idade	Altura	Peso	Nome	Idade	Altura	Peso	Nome	Idade	Altura	Peso
0	Alice	25	1.65	60	Eduardo	45	1.85	85	Igor	34	1.76	70
1	Bruno	30	1.75	80	Fernanda	32	1.70	68	Julia	27	1.68	59
2	Carlos	22	1.80	75	Gabriel	29	1.90	90	Karla	31	1.73	63
3	Diana	28	1.60	55	Helena	24	1.65	58	Luis	40	1.82	88

Realizamos o merge, ou seja, juntar dois dataframes com chaves ou índices em comum, neste caso o peso de Carlos e Gabriel era 90

```
df_merged = pd.merge(df1, df2, how="inner", on="Peso")
```

```
df_merged
```

	Nome_x	Idade_x	Altura_x	Peso	Nome_y	Idade_y	Altura_y
0	Carlos	22	1.8	90	Gabriel	29	1.9

Para realizar o join, criei um dataframe semelhante ao vídeo, onde quando o index for diferente, é preenchido com o NaN.

```
left = pd.DataFrame({
    "A": ["A0", "A1", "A2"],
    "B": ["B0", "B1", "B2"]
}, index=["K0", "K1", "K2"])

right = pd.DataFrame({
    "C": ["C0", "C1", "C2"],
    "D": ["D0", "D1", "D2"]
}, index=["K0", "K3", "K2"])
```

```
left
```

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

```
right
```

	C	D
K0	C0	D0
K3	C1	D1
K2	C2	D2

```
left.join(right)
```

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

No how = “outer” captura todos os valores existentes nos dataframes, depois, poderia ser utilizado o método para preencher os dados faltantes.

```
left.join(right, how = "outer")
```

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C1	D1

Criei o data frame seguinte com 3 colunas e é mostrado a forma de mostrar o valor de apenas uma coluna com apenas o unique e com a biblioteca numpy, que na teoria, é a mesma coisa.

```
dtf = pd.DataFrame({"col1":(1,2,3,4), "col2":(333,444,777,777), "col3":("acb","def","ghi","jkl")})
```

```
dtf
```

	col1	col2	col3
0	1	333	acb
1	2	444	def
2	3	777	ghi
3	4	777	jkl

```
dtf["col2"].unique()
```

```
array([333, 444, 777], dtype=int64)
```

```
np.unique(dtf["col2"])
```

```
array([333, 444, 777], dtype=int64)
```

O counts conta quantas vezes os números aparecem na coluna.
Já na parte que está embaixo, é passo por duas condições, onde a coluna 1 tem que ser maior ou igual a 2 e a coluna 3 tem q ser igual a 444.

```
dtf["col2"].value_counts()
```

```
col2
777    2
333    1
444    1
Name: count, dtype: int64
```

```
dtf[(dtf["col1"]>=2)&(dtf["col2"]==444)]
```

	col1	col2	col3
1	2	444	def

Na função, passou como x o parâmetro é multiplicado por 4, utilizado logo em seguida para multiplicar a coluna 1.
Na outra parte, é calculado o lambda da coluna 1 também.

```
def vezes4(x):
    return x*4
```

```
dtf["col1"].apply(vezes4)
```

```
0     4
1     8
2    12
3    16
Name: col1, dtype: int64
```

```
dtf["col1"].apply(lambda x: x*x)
```

```
0     1
1     4
2     9
3    16
Name: col1, dtype: int64
```

O dft.columns mostra quais colunas temos no dataframe, e o sort organiza em ordem crescente os valores da coluna 2.

```
dtf.columns
```

```
Index(['col2', 'col3'], dtype='object')
```

```
dtf.sort_values(by="col2",inplace=True)
```

```
dtf
```

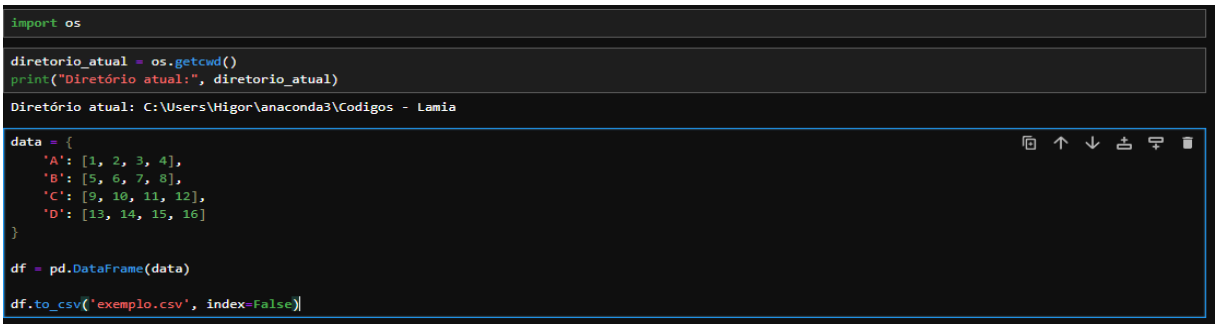
	col1	col2	col3
0	1	333	acb
1	2	444	def
2	3	777	ghi
3	4	777	jkl

De forma booleana, os isnull mostra quais valores é nulo(false), como neste caso não continha nenhum, deu tudo false.

```
dtf.isnull()
```

	col1	col2	col3
0	False	False	False
1	False	False	False
2	False	False	False
3	False	False	False

Para conseguir salvar um arquivo .csv; que seria um formato de arquivo usado para armazenar dados tabulares, onde cada linha do arquivo representa um registro e as colunas são separadas por vírgulas; eu importei a biblioteca os e utilizei o getcwd para descobrir com precisão onde está meu diretório para eu poder adicionar um arquivo lá sem problemas.



```
import os

diretorio_atual = os.getcwd()
print("Diretório atual:", diretorio_atual)
```

```
Diretório atual: C:\Users\Higor\anaconda3\Codigos - Lamia
```

```
data = {
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8],
    'C': [9, 10, 11, 12],
    'D': [13, 14, 15, 16]
}

df = pd.DataFrame(data)

df.to_csv('exemplo.csv', index=False)
```

Criei o dataframe com os respectivos valores usando o “;” como separação e o “.” como separação de decimais.

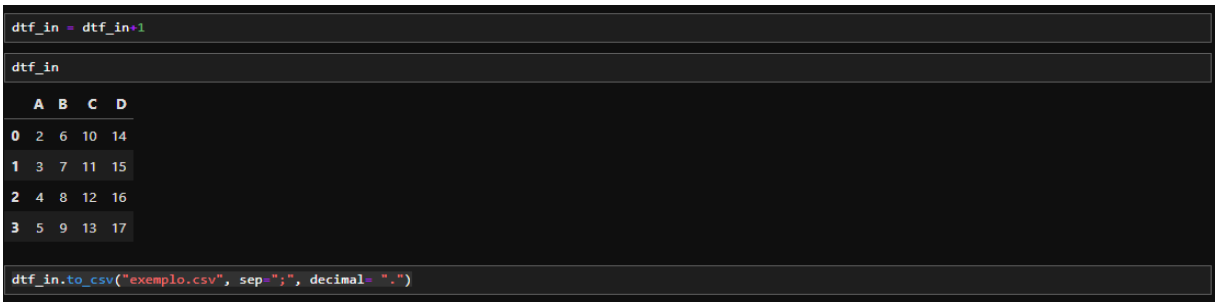
```
dtf_in = pd.read_csv("exemplo.csv", sep=";", decimal= ".")
```

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

```
dtf_in
```

	A	B	C	D
0	1	5	9	13
1	2	6	10	14
2	3	7	11	15
3	4	8	12	16

Somei um a todos os valores dentro do dataframe e salvei no arquivo de forma correta.



```
dtf_in = dtf_in+1
```

```
dtf_in
```

	A	B	C	D
0	2	6	10	14
1	3	7	11	15
2	4	8	12	16
3	5	9	13	17

```
dtf_in.to_csv("exemplo.csv", sep=";", decimal= ".")
```

E nesses últimos dois exemplos, criei um arquivo no formato de excel(xlsx) , onde o sheet_name é utilizado ao ler/escrever arquivos xlsx especificando qual planilha do arquivo deve ser lida/escrita. E por último puxei os dados de um site(html) apenas copiando a sua url.

```
dtf_in.to_excel("exemplo.xlsx", sheet_name="shoot1")
```

```
dtf = pd.read_html("https://www.fdic.gov/bank-failures/failed-bank-list")
```

Exercícios Práticos disponíveis em: GitHub

Conclusão: Nestas seções, aprendemos a manipular de forma prática as bibliotecas numpy fundamental para a computação científica em Python, fornecendo suporte para arrays, e também a biblioteca pandas que nos oferece estruturas de dados para ajudar na manipulação de data frames(semelhante a planilha do excel), aprendendo a mexer com linhas e colunas de forma simples para descobrir valores específicos,, há também o conhecimento de como fazemos para puxar arquivos do csv, excel e html também.