```
import numpy as np
```

Índice

- Seção 1 - CONCEITOS BÁSICOS EM NUMPY
  - 1.1. - O que é o NumPy?
  - 1.2. - O que são arrays?

CONCEITOS BÁSICOS EM NUMPY

1.1. O que é o NumPy?

O NumPy (Numerical Python) é uma biblioteca fundamental para computação científica em Python. Ele fornece suporte para arrays e matrizes multidimensionais, juntamente com uma ampla coleção de funções matemáticas, lógicas e estatísticas para operar eficientemente sobre esses arrays.

Criada em 2005 como sucessora do Numeric, o NumPy se tornou uma ferramenta indispensável para qualquer trabalho que exija cálculos numéricos, seja em áreas como ciência de dados, machine learning ou visualização de dados.

A grande diferença do NumPy em comparação com outras estruturas em Python está em sua capacidade de realizar cálculos eficientes, aproveitando ao máximo o processamento vetorial e matricial, o que resulta em uma grande melhoria de performance.

1.2. O que são arrays?

No NumPy, **os arrays são estruturas de dados homogêneas que armazenam elementos de um mesmo tipo.** Eles podem ser **unidimensionais (vetores)** ou **multidimensionais (matrizes ou tensores)**.

Escalar: um elemento único, adimensional. Pode ser um inteiro, float, hexadecimal, caractere, e vários outros tipos de dado.

Ex.: `np.array(42)`

Array unidimensional: uma lista de escalares onde podemos identificar cada um deles pela sua posição, ou índice, na lista. É preciso ressaltar que todos os elementos escalares aqui possuem o mesmo tipo, e que se não for especificado durante a definição do array, o Numpy trabalhará uma rotina para determinar o tipo que garanta a característica homogênea.

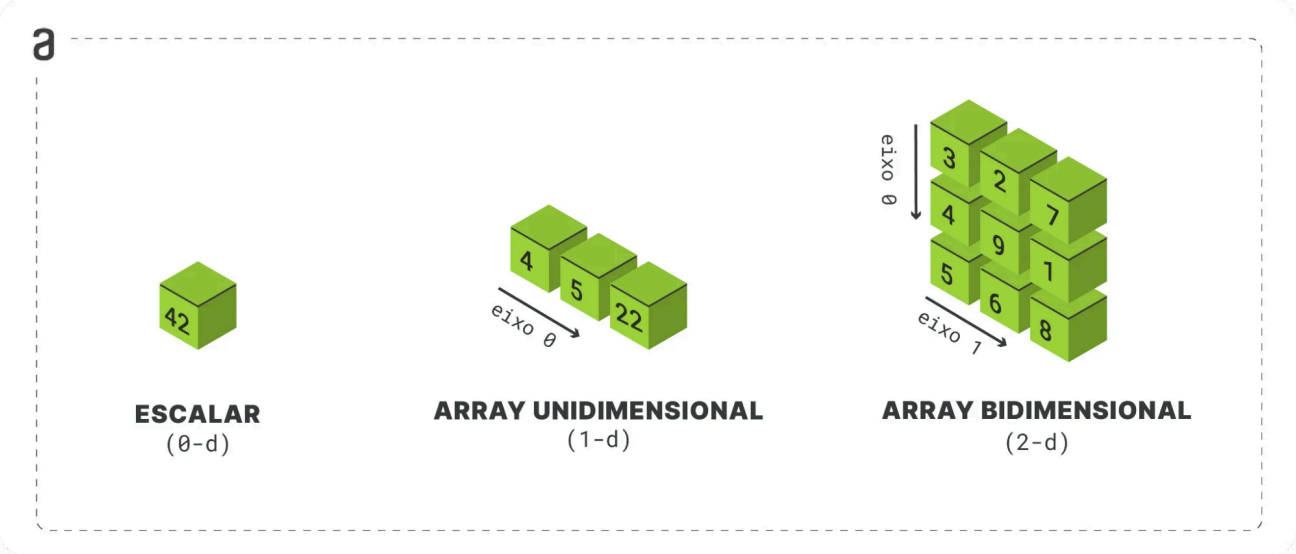
Ex.: `np.array([4, 5, 22, 20])`

Array bidimensional: uma lista de arrays unidimensionais, com o formato de uma matriz (tabela), onde precisamos especificar uma posição de linha e uma posição de coluna para localizar um elemento escalar.

Ex.: `np.array([[3, 2, 7], [4, 9, 1], [5, 6, 8]])`

Diferentemente das listas do Python, que podem conter elementos de diferentes tipos, os arrays NumPy são otimizados para executar operações matemáticas de forma muito mais rápida.

Além disso, os arrays NumPy permitem operações vetoriais, que processam múltiplos elementos ao mesmo tempo, aumentando a eficiência computacional em comparação com as listas Python, que precisam ser iteradas manualmente.




```
# np.loadtxt('apples_ts.csv', delimiter=',', usecols=np.arange(1, 88, 1))
```

Criando Arrays Numpy

```
# Array criado a partir de uma lista python
arr1 = np.array([10, 21, 32, 43, 48, 15, 76, 57, 89])

arr1
```

```
array([10, 21, 32, 43, 48, 15, 76, 57, 89])
```

```
# Criando array Bidimensional
arr2 = np.array([[3, 2, 7], [4, 9, 1], [5, 6, 8]])
#                Coluna 1    Coluna 2   Coluna 3

arr2
```

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

Outras maneiras de criar arrays

```
# Zeros, Uns, Range e Aleatórios
np.zeros((2, 3))      # Matriz 2x3 com zeros
```

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

```
np.ones((3, 3))       # Matriz 3x3 com uns
```

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

```
np.arange(0, 10, 2)   # [0 2 4 6 8]
```

```
array([0, 2, 4, 6, 8])
```

```
np.random.rand(2, 3)  # Matriz aleatória 2x3
```

```
array([[0.75289799, 0.72013166, 0.96843239],
       [0.27626626, 0.90778011, 0.19951053]])
```

Anteriormente foi falado que um array são estruturas de dados homogêneas, ou seja, o array será de apenas um tipo. E podemos verificar com os exemplos abaixo, onde será criado três arrays:

- O primeiro com numeros inteiros;
- O segundo com numeros inteiros e um elemento de ponto flutuante;
- E por último, com números inteiros e um elemento de string.

E com isso iremos ver que mesmo que sejá criado um array com tipos diferentes, ao ser armazenado na memória, ele será de apenas um único tipo.

```
a1 = np.array([1, 2, 3, 4, 5]) # Tipo int
a2 = np.array([1, 2, 3, 4.5, 5]) # Tipo float
a3 = np.array([1, 2, 3, '4', 5]) # Tipo string
```

```
# Qual o tipo de dado do array 'a1'?
a1.dtype # int64
```

```
dtype('int64')
```

```
# Qual o tipo de dado do array 'a2'?
a2.dtype # float64
```

```
dtype('float64')
```

```
# E por último, qual o tipo de dado do array 'a2'?
a3.dtype # <U21
```

```
dtype('<U21')
```

Com isso podemos perceber que, mesmo que contenha apenas um tipo de dado diferente no meu array, ele será desse tipo. Pelo motivo de manter a integridade dos dados e preservar a facilidade. Pois é muito mais fácil transformar todos os outros valores em um ponto flutuante, por exemplo, do que fazer o inverso.

Ou até mesmo transformar todos os outros valores em String do que o inverso (já que nem toda String pode se tornar um int)

```
a2
```

```
array([1. , 2. , 3. , 4.5, 5. ])
```

```
a3
```

```
array(['1', '2', '3', '4', '5'], dtype='<U21')
```

Indexação em Arrays NumPy

- **O que é Indexação?**

Indexar significa acessar um elemento específico do array, informando a sua posição. É semelhante às listas do Python, mas o NumPy também aceita indexação em múltiplas dimensões (matrizes, por exemplo).

- **O que é Fatiamento (Slicing)?**

Fatiar significa pegar um subconjunto do array — uma “fatia” dele — usando a notação inicio:fim:passo.

```
print(arr1)
```

```
[10 21 32 43 48 15 76 57 89]
```

```
# Imprimindo um elemento específico no array
arr1[4] # Quero o elemento do indice 4
```

```
np.int64(48)
```

```
# Fatiamento
arr1[1:4]
```

```
array([21, 32, 43])
```

```
arr1[:2]
```

```
array([10, 21])
```

Indexação em Arrays 2D

Em matrizes (ou seja, arrays 2D), você precisa passar dois índices:

```
print(arr2)
```

```
[[3 2 7]
 [4 9 1]
 [5 6 8]]
```

```
arr2[0, 0]
```

```
np.int64(3)
```

```
arr2[1, 2]  # 1 (segunda linha, terceira coluna)
```

```
np.int64(1)
```

Fatiamento em Arrays 2D

```
arr2
```

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

```
# todas as linhas, apenas coluna 1
arr2[:, 1]    # [2, 9, 6]
```

```
array([2, 9, 6])
```

```
# apenas linha 0, todas as colunas
arr2[0, :]    # [3, 2, 7]
```

```
array([3, 2, 7])
```

```
# linhas 0 e 1, colunas 1 e 2
arr2[0:2, 1:3]  # [[2, 7]
                #  [9, 1]]
```

```
array([[2, 7],
       [9, 1]])
```

Fatiamento com condições

```
a_con = np.arange(16)

a_con
```

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

```
# Buscando os valores maiores que 10
a_con[a_con>10]
```

```
array([11, 12, 13, 14, 15])
```

```
# Buscando os valores iguais a 10
a_con[a_con==10]
```

```
array([10])
```

```
# Buscando os valores diferentes de 10
a_con[a_con!=10]
```

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

```
# Buscando os valores maiores que 5 E(&) menores que 11
# (Pode-se utilizar também o OU(|) ou o NÃO(~))
a_con[(a_con>5) & (a_con < 11)]
```

```
array([ 6,  7,  8,  9, 10])
```

Índices booleanos

O NumPy também permite selecionar elementos usando condições:

```
arr1[arr1 > 40] # Valores no array maior que 40
```

```
array([43, 48, 76, 57, 89])
```

Alterando elementos de um array

```
# Alterando um elemento do array
arr1[0] = 100
```

```
arr1
```

```
array([100,  21,  32,  43,  48,  15,  76,  57,  89])
```

Funções NumPy

No geral, as funções do NumPy servem para operações vetorizadas (ou seja, operam sobre arrays inteiros de uma vez), sem precisar de loops explícitos, o que as torna muito rápidas.

1. Algumas funções:

```
arr3 = np.array([[1, 2, 3], [4, 5, 6]])

arr3
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
arr3.shape # (2, 3) -> 2 Linhas, 3 Colunas
```

```
(2, 3)
```

```
arr3.ndim # Número de dimensões (eixos)
```

```
2
```

```
arr3.size # Número total de elementos
```

```
6
```

```
arr3.itemsize # Quantidade de Bits no array
```

```
8
```

```
arr3.nbytes # Quantidade de bytes no array
```

```
48
```

```
arr3.dtype # Tipo dos dados (int64)
```

```
dtype('int64')
```

---

2. Funções de álgebra linear

```
m = np.array([[1, 2],
              [3, 4]])

m
```

```
array([[1, 2],
       [3, 4]])
```

```
np.transpose(m)           # transposta
```

```
array([[1, 3],
       [2, 4]])
```

```
np.linalg.inv(m)          # inversa
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

```
np.dot(m, m)              # produto matricial
```

```
array([[ 7, 10],
       [15, 22]])
```

```
np.linalg.det(m)          # determinante
```

```
np.float64(-2.0000000000000004)
```

---

3. Funções Estatísticas (Matemática Básica)

São usadas para calcular métricas estatísticas (*media, mediana, moda, desvio padrão, etc*) diretamente em arrays de forma rápida e vetorizada;

sum()

OBS: A questão do `axis=1 ou axis=0` valem para qualquer função do numpy

```
# Função sum()
b = np.arange(6).reshape(2, 3)
b
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
b.sum()
```

```
np.int64(15)
```

```
# Somando apenas os valores da linha
# para somar apenas as linhas: axis=1
# Para somar apenas as colunas: axis=0
b.sum(axis=1)
```

```
array([ 3, 12])
```

```
# Quero a soma apenas da primeira linha
b[0].sum()
```

```
np.int64(3)
```

mean()

```
# 1. mean() -> Calcula a média aritmética dos elementos:
a = np.array([1, 2, 3, 4, 5])
np.mean(a)
```

```
np.float64(3.0)
```

Em matrizes, você pode aplicar ao longo de um eixo específico:

```
a_matriz = np.array([ [1,2], [3, 4] ])

a_matriz
```

```
array([[1, 2],
       [3, 4]])
```

```
np.mean(a_matriz, axis=0) # Média das colunas
```

```
array([2., 3.])
```

```
np.mean(a_matriz, axis=1) # Média das linhas
```

```
array([1.5, 3.5])
```

median()

- Calcula a mediana, ou seja, o valor do meio de um conjunto ordenado;

- Se houver número par de elementos, faz a média dos dois valores centrais.

```
b = np.array([1, 3, 5, 7])
np.median(b) # Vai somar 3 e 5 e dar sua média (3 + 5 = 8 | 8/2 = 4)
```

```
np.float64(4.0)
```

std() - Desvio Padrão

- Desvio padrão mede o quanto os valores se afastam da média;
- Quanto maior o desvio padrão, maior a dispersão dos dados;
- O desvio padrão é a raiz quadrada da variância;
- Ele serve para trazer a dispersão para a mesma unidade de medida dos dados originais, porque, a VARIÂNCIA, como usa o quadrado das diferenças, fica "em unidades ao quadrado".

```
c = np.array([2, 4, 4, 4, 6, 8])
np.std(c)
```

```
np.float64(2.0)
```

var() - Variância

- A variância mede o grau de dispersão dos valores em relação à média, em outras palavras, ela mostra o quanto os valores se afastam da média;
- Se a variância **for pequena**, significa que os dados estão **bem próximos** da média;
- Por outro lado, se ela **for grande**, significa que os dados estão **bem espalhados**.

OBS: Variância é o quadrado do desvio padrão

```
c
```

```
array([2, 4, 4, 4, 6, 8])
```

```
np.var(c) # ~4 ("unidade ao quadrado")
```

```
np.float64(3.5555555555555554)
```

percentile()

- Retorna o percentil desejado - muito útil para entender a distribuição dos dados;
- Em análise de dados, isso é muito utilizado para:
  - identificar valores muito altos/baixos;
  - detectar outliers;
  - resumir a dispersão de dados.

```
d = np.array([10, 20, 30, 40, 50])

np.percentile(d, 25) # 20 -> o 25 representa a porcentagem | 25% dos valores estão abaixo de 20
```

```
np.float64(20.0)
```

```
np.percentile(d, 50)  # 30 -> o 50 representa a porcentagem (nesse caso será utilizado a mediana) | 50% dos valores estão abaixo de 30
```

```
np.float64(30.0)
```

```
np.percentile(d, 75)  # 40 -> o 75 representa a porcentagem | 75% dos valores estão abaixo de 40
```

```
np.float64(40.0)
```

Nesse caso do percentil (e também do quantil), a informação que vai após o array vai definir a porcentagem dos dados.

Por exemplo: *Eu quero saber a porcentagem dos dados abaixo de 75%*

quantile()

- O quantil é um conceito muito parecido com o percentil, mas **expresso em frações**;
- Vai de 0 a 1:
  - Exemplo: 0,25 -> 25% dos dados | 0,50 -> 50% dos dados

```
np.quantile(d, 0.25) # valor onde contém 25% dos dados
```

```
np.float64(20.0)
```

Exemplo:

Imagine um array de salários: `[1000, 1500, 2000, 2500, 3000, 3500, 4000]`. Me diga: *onde 75% das pessoas ganham menos?*

```
sal = np.array([1000, 1500, 2000, 2500, 3000, 3500, 4000])

np.quantile(sal, 0.75)
```

```
np.float64(3250.0)
```

max() e min()

- Encontra o maior e o menor, respectivamente, dos elementos.

```
d
```

```
array([10, 20, 30, 40, 50])
```

```
np.max(d)
```

```
np.int64(50)
```

```
np.min(d)
```

```
np.int64(10)
```

argmax() e argmin()

- Retorna o **índice** do maior e menor elemento;

```
d
```

```
array([10, 20, 30, 40, 50])
```

```
np.argmax(d)
```

```
np.int64(4)
```

```
np.argmin(d)
```

```
np.int64(0)
```

---

4. Funções de criação de Arrays

Para gerar rapidamente dados de teste:

```
np.zeros((2,3))      # matriz 2x3 de zeros
```

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

```
np.ones((4,4))       # matriz 4x4 de uns
```

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

```
np.eye(3)            # matriz identidade 3x3
```

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

```
np.arange(0, 10, 2)  # array de 0 a 10, passo 2
```

```
array([0, 2, 4, 6, 8])
```

```
np.linspace(0, 1, 5) # 5 valores igualmente espaçados entre 0 e 1
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

---

5. Funções de manipulação:

Para mudar a forma do array, reorganizar dados, etc.:

```
a
```

```
array([1, 2, 3, 4])
```

```
a.reshape((2, 2))   # muda o formato
```

```
array([[1, 2],
       [3, 4]])
```

```
a.flatten()         # “achata” (vira 1D)
```

```
array([1, 2, 3, 4])
```

```
np.concatenate([a, a])  # concatena arrays
```

```
array([1, 2, 3, 4, 1, 2, 3, 4])
```

Broadcasting

Permite fazer operações entre arrays de formas diferentes, desde que compatíveis.

```
a = np.array([1, 2, 3])
b = 2
a + b   # [3 4 5] — soma 2 a cada elemento
```

```
array([3, 4, 5])
```

```
arr2
```

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

```
# Vamos somar o 'arr2' com a variável 'b'
arr2 + b
```

```
array([[ 5,  4,  9],
       [ 6, 11,  3],
       [ 7,  8, 10]])
```

```
arr2 * b
```

```
array([[ 6,  4, 14],
       [ 8, 18,  2],
       [10, 12, 16]])
```

```
arr2 / b
```

```
array([[1.5, 1. , 3.5],
       [2. , 4.5, 0.5],
       [2.5, 3. , 4. ]])
```

```
arr2 - b
```

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

Manipulação de Arquivos com NumPy

Análise Estatísica Básica com NumPy

```
# Criando um array
arr5 = np.array([15, 23, 63, 94, 75])
```

Em Estatística, a média é uma medida de tendência central que representa o valor central de um conjunto de dados. É calculada somando-se todos os valores do conjunto de dados e dividindo-se pelo número de observações.

```
# Média
np.mean(arr5)
```

```
np.float64(54.0)
```

O desvio padrão é uma medida estatística de dispersão que indica o quanto os valores de um conjunto de dados se afastam da média. Ele é calculado como a raiz quadrada da variância, que é a média dos quadrados das diferenças entre cada valor e a média.

O desvio padrão é uma medida útil porque permite avaliar a variabilidade dos dados em torno da média. Se os valores estiverem próximos da média, o desvio padrão será baixo, indicando que os dados têm pouca variabilidade. Por outro lado, se os valores estiverem muito distantes da média, o desvio padrão será alto, indicando que os dados têm alta variabilidade.

O desvio padrão é amplamente utilizado em Análise e Ciência de Dados, para avaliar a consistência dos dados e comparar conjuntos de dados diferentes. É importante notar que o desvio padrão pode ser influenciado por valores extremos (outliers) e pode ser afetado por diferentes distribuições de dados.

```
# Desvio Padrão (Standard Deviation)
np.std(arr5)
```

```
np.float64(30.34468652004828)
```

A variância é uma medida estatística que quantifica a dispersão dos valores em um conjunto de dados em relação à média. Ela é calculada como a média dos quadrados das diferenças entre cada valor e a média.

A fórmula para o cálculo da variância é:

var = 1/n * Σ(xi - x̄)^2

Onde:

var é a variância n é o número de observações Σ é o somatório xi é o i-ésimo valor no conjunto de dados x̄ é a média dos valores A variância é uma medida útil para avaliar a

variabilidade dos dados em torno da média. Se a variância for baixa, isso indica que os valores estão próximos da média e têm pouca variabilidade. Por outro lado, se a variância for alta, isso indica que os valores estão distantes da média e têm alta variabilidade.

```
# Variância
np.var(arr5)
```

```
np.float64(920.8)
```

Operações matemáticas com Arrays NumPy

```
arr6 = np.arange(1, 10)

arr6
```

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

```
# Soma dos elementos do array
np.sum(arr6)
```

```
np.int64(45)
```

```
# Retorna o produtos dos elementos
np.prod(arr6)
```

```
np.int64(362880)
```

```
# Soma acumulada dos elementos
np.cumsum(arr6)
```

```
array([ 1,  3,  6, 10, 15, 21, 28, 36, 45])
```

```
a1 = np.array([3, 2, 1])
a2 = np.array([1, 2, 3])

# Soma de dois arrays
arr7 = np.add(a1, a2)

arr7
```

```
array([4, 4, 4])
```

Multiplicação de Matrizes

Para multiplicar duas matrizes NumPy, podemos usar a função dot() ou o operador @.

Ambos os métodos executam a multiplicação matricial. É importante lembrar que, **para que a multiplicação de matrizes possa ser executada, o número de colunas da**

primeira matriz deve ser igual ao número de linhas da segunda matriz.

Há várias formas de multiplicar elementos de matrizes NumPy. A função dot() é um método bastante utilizado

```
arr8 = np.array([ [1, 2], [3, 4] ])
arr9 = np.array([ [5, 6], [0, 7] ])
```

```
arr8
```

```
array([[1, 2],
       [3, 4]])
```

```
arr9
```

```
array([[5, 6],
       [0, 7]])
```

```
# Multiplicar as duas matrizes usando a função .dot
arr10 = np.dot(arr8, arr9)

arr10
```

```
array([[ 5, 20],
       [15, 46]])
```


```
# Multiplicar as duas matrizes usando o @
arr11 = arr8 @ arr9

arr11
```

```
array([[ 5, 20],
       [15, 46]])
```

Slicing (Fatiamento) de Arrays NumPy

```
# Criando uma matriz com .diag:
arr12 = np.diag(np.arange(3))

arr12
```

```
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
```

```
arr12[1, 1] # Linha 1, Coluna 1
```

```
np.int64(1)
```

```
arr12[2, 2] # Linha 2, Coluna 2
```

```
np.int64(2)
```

```
# Quero apenas uma única linha:
arr12[1]
```

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

```
# Quero apenas uma única coluna:
arr12[:, 2]
```

```
array([0, 0, 2])
```

```
# Somando um valor a cada elemento do array
np.array([1, 2, 3]) + 1.5
```

```
array([2.5, 3.5, 4.5])
```

Método flatten()

O método flatten() com NumPy é usado para **criar uma cópia unidimensional (ou "achatada") de um array multidimensional**.

Isso significa que o método cria um novo array unidimensional, que contém todos os elementos do array multidimensional original, mas que está organizado em uma única linha. A ordem dos elementos no novo array unidimensional segue a ordem dos elementos no array multidimensional original.

```
arr13 = np.array([ [1, 2, 3, 4], [5, 6, 7, 8] ])

arr13
```

```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

```
# "Achatando" a matriz
arr14 = arr13.flatten()

arr14
```

```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Referências

O conteudo abaixo é um resumo de diferentes cursos estudados, são eles:

- Fundamentos de Linguagem Python para Análise de dados - DSA;
- Ciência de Dados Impressionador;
-
