centro universitário FEI

# CC8210 – NCA210 Programação Avançada I

Prof. Reinaldo A. C. Bianchi
Prof. Isaac Jesus da Silva
Prof. Danilo H. Perico

# A biblioteca matemática NumPy

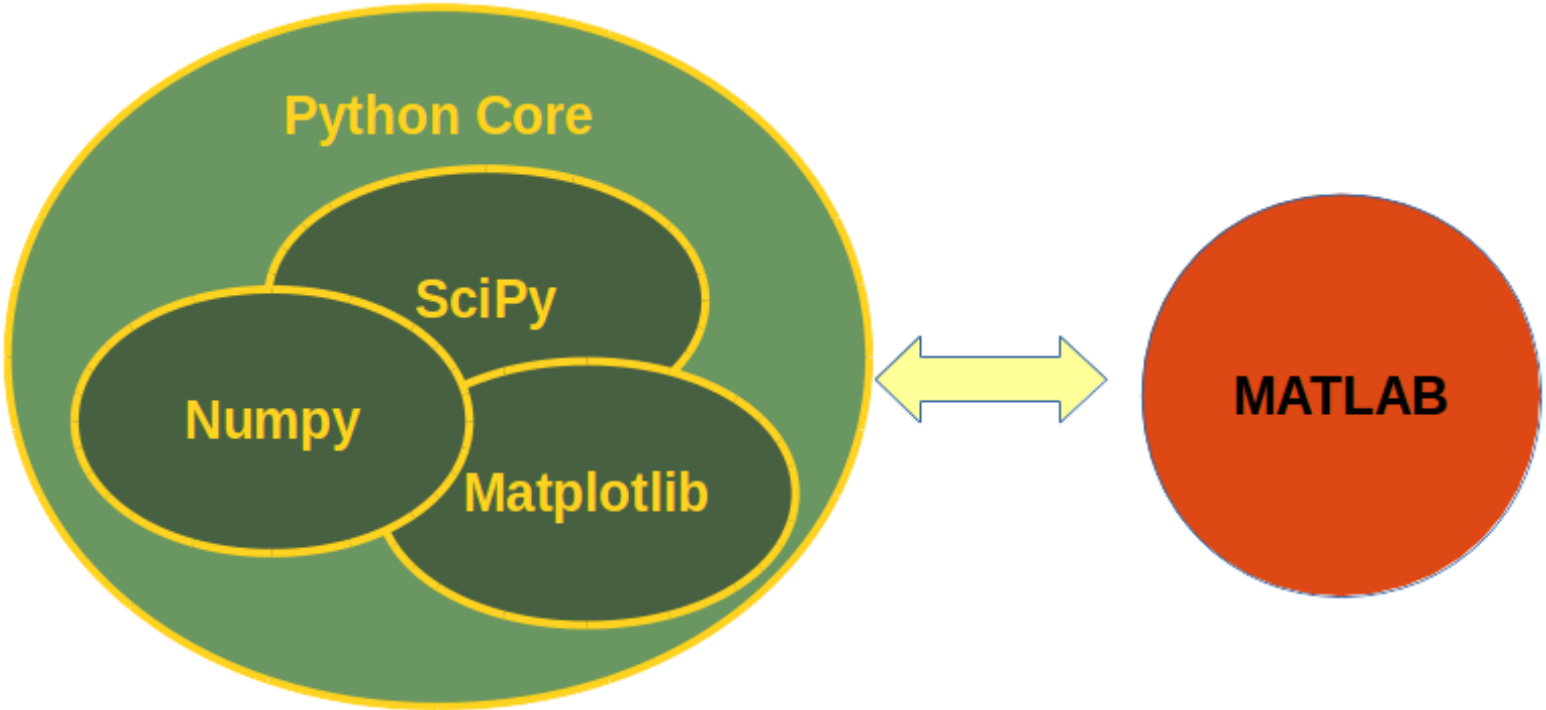
# NumPy


- NumPy é um pacote para a linguagem Python que suporta arrays e matrizes multidimensionais, possuindo uma larga coleção de funções matemáticas para trabalhar com estas estruturas.
- Fast and versatile, the NumPy vectorization, indexing, and broadcasting concepts are the de-facto standards of array computing today.
- NumPy foi criado em 2005 por Travis Oliphant.
- É um projeto comunitário, multiplataforma, de código livre.

# NumPy é utilizado em... qualquer tarefa matemática

- NumPy é bastante útil para executar várias tarefas matemáticas como integração numérica, diferenciação, interpolação, extrapolação e muitas outras.
- O NumPy possui também funções incorporadas para álgebra linear, transformadas de fourrier e geração de números aleatórios.
- Quando usada em conjunto com SciPy e Matplotlib, pode substituir programas como o Matlab para o tratamento de tarefas matemáticas.

# NumPy é utilizado em... qualquer tarefa matemática






https://medium.com/ensina-ai/entendendo-a-biblioteca-numpy-4858fde63355

# NumPy é utilizado por... Processamento de Imagem e Computação Gráfica

- Imagens no computador são representadas como Arrays Multidimensionais de números.
- NumPy torna-se a escolha mais natural para o mesmo.
- O NumPy, na verdade, fornece algumas excelentes funções de biblioteca para rápida manipulação de imagens.
- Alguns exemplos são o espelhamento de uma imagem, a rotação de uma imagem por um determinado ângulo etc.

# NumPy é utilizado por... Machine Learning

- Ao escrever algoritmos de Machine Learning, supõe-se que se realize vários cálculos numéricos em Array.
  - Por exemplo, multiplicação de Arrays, transposição, adição, etc.
- O NumPy fornece uma excelente biblioteca para cálculos fáceis (em termos de escrita de código) e rápidos (em termos de velocidade).
- Os Arrays NumPy são usados para armazenar os dados de treinamento, bem como os parâmetros dos modelos de ML.

# Por que usar NumPy

- Em Python usamos listas para tratar vetores e matrizes, mas elas são lentas para processar.
- O NumPy fornece um objeto array que é até 50x mais rápido que o Python tradicional.
    - O objeto array em NumPy fornece funções de suporte que tornam o trabalho com vetores e matrizes muito fácil.
- Os arrays são muito frequentemente usados em ciência de dados, onde velocidade e recursos são muito importantes.

# Por que usar NumPy

- Os arrays NumPy são armazenados em um lugar contínuo na memória - ao contrário das listas, para que os processos possam manipulá-los de forma muito eficiente.
    - Esse comportamento é chamado de localidade de referência em ciência da computação.
- Esta é a principal razão pela qual o NumPy é mais rápido do que as listas.
- Também é otimizado para trabalhar com as mais recentes arquiteturas de CPU.

# Instalando o NumPy

- If you use conda, you can install it with:

```
conda install numpy
```

- If you use pip, you can install it with:

```
pip install numpy
```

- If you use Linux, you can isnatll via terminal with:

```
sudo apt-get install python-numpy
```

- Já vem pre-instalado no Anaconda.

# Usando o NumPy

- Para utilizar a biblioteca NumPy em um programa, é necessário importa-la:

```
import numpy
```

- Ou:

```
import numpy as np
```

- Imprima a versão instalada:

```
print(np.__version__)
```

# Criando arrays em NumPy

- O objeto array em NumPy é chamado `ndarray`.
- Podemos criar um objeto ndarray usando a função `array()`:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

# Criando arrays em NumPy

| Command | NumPy Array |
| --- | --- |
| `np.array([1,2,3])` |  |

| |
| --- |
| 1 |
| 2 |
| 3 |

# Criando arrays em NumPy

- Para criar um array, podemos passar uma lista, tupla ou qualquer objeto semelhante a matriz no método array() e ele será convertido em um objeto do tipo ndarray.

```
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

```
[1 2 3 4 5]
```

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

```
[[1 2]
 [3 4]]
```

# Criando arrays em NumPy

Criar um ndarray a partir de sequências aninhadas irregulares.

```
arr = np.array([[1, 2], [3, 4, 5]])
print(arr)
```

```
[list([1, 2]) list([3, 4, 5])]
```

# Dimensão em arrays

- 0-D: Representa um número escalar

```
arr = np.array(42)
print(arr)
```

```
42
```

- 1-D: Representa um vetor

```
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

```
[1 2 3 4 5]
```

# Dimensão em arrays

- 2-D:
  - Um array que tem arrays 1D como seus elementos é chamado de array 2D.
  - Estes são frequentemente usados para representar matrizes.
  - NumPy tem um subconjunto inteiro dedicado às operações de matriz.

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr)
```

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

# Dimensão em arrays

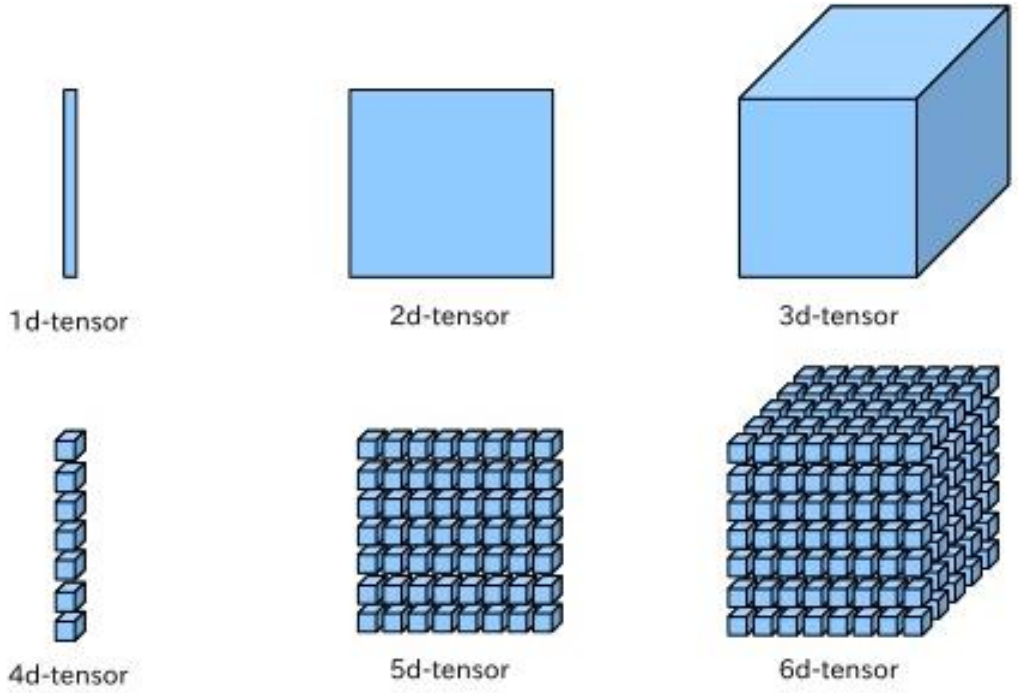
- 3-D:
  - Um array que tem arrays 2D como seus elementos é chamado de array 3D.
  - Estes são frequentemente usados para representar um tensor de 3ª ordem.

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

```
[[[1 2 3]
  [4 5 6]]

 [[1 2 3]
  [4 5 6]]]
```

# Dimensões superiores: Tensores




https://medium.com/datadriveninvestor/from-scalar-to-tensor-fundamental-mathematics-for-machine-learning-with-intuitive-examples-part-163727dfea8d

# Descobrindo a dimensão de um array

- O NumPy Arrays fornece o atributo **`ndim`** que retorna um inteiro que nos diz quantas dimensões o array tem.

```
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2], [3, 4]], [[5, 6], [7 ,8]]])
e = np.array([[[[1, 2], [3, 4]], [[5, 6], [7 ,8]]], [[[1, 2], [3, 4]], [[5, 6], [7 ,8]]]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
print(e.ndim)
```

```
0
1
2
3
4
```

# Índices dos arrays

- Elementos dos arrays são acessados iguais as listas.

```
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

```
1
```

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(arr[0][1])
```

```
2
```

```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0][1][2])
```

```
6
```

# Índices dos arrays

- Mas arrays de numpy também são acessados utilizando colchetes, com as dimensões separadas por vírgulas.

```
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

```
1
```

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(arr[0, 1])
```

```
2
```

```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```

```
6
```

# Índices negativos

- Igual às listas, os índices negativos indexam os elementos a partir do final do array:

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Último elemento da segunda lista: ', arr[1, -1])
```

```
Último elemento da segunda lista:  10
```

# Fatiamento de arrays

- Fatiar em Python significa pegar elementos de um dado índice até outro dado índice.
- Para fatiar usamos os índices como:
  - `[ start : end ]`
  - Se não definirmos o start, se considera iniciado em 0.
  - Se não passarmos o end, se fatia até o final da dimensão.
- Também podemos definir o passo, assim:
  - `[ start : end : step ]`

# Fatiamento de arrays

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

```
[2 3 4 5]
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
```

```
[5 6 7]
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:4])
```

```
[1 2 3 4]
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

```
[2 4]
```

# Fatiamento de arrays de mais dimensões

```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
```

```
[7 8 9]
```

```
print(arr[0:2, 2])
```

```
[3 8]
```

```
print(arr[0:2, 1:4])
```

```
[[2 3 4]
 [7 8 9]]
```

# Fatiamento de arrays: retirando uma linha ou coluna

```
arr = np.array([[1, 2, 3], [6, 7, 8], [7, 8, 9]])
```

- Capturando uma linha:
  - `print(arr[1, :])`

```
print(arr[1, :])
```

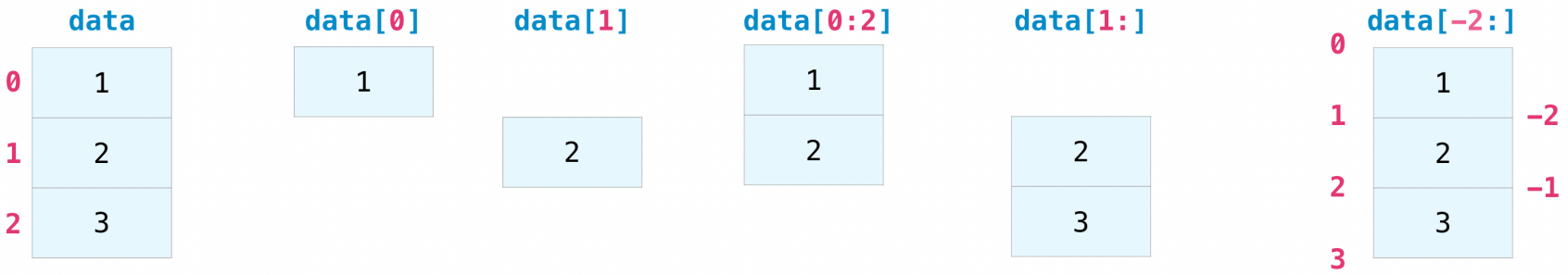
```
[6 7 8]
```

- Capturando uma coluna:
  - `print(arr[:, 1])`

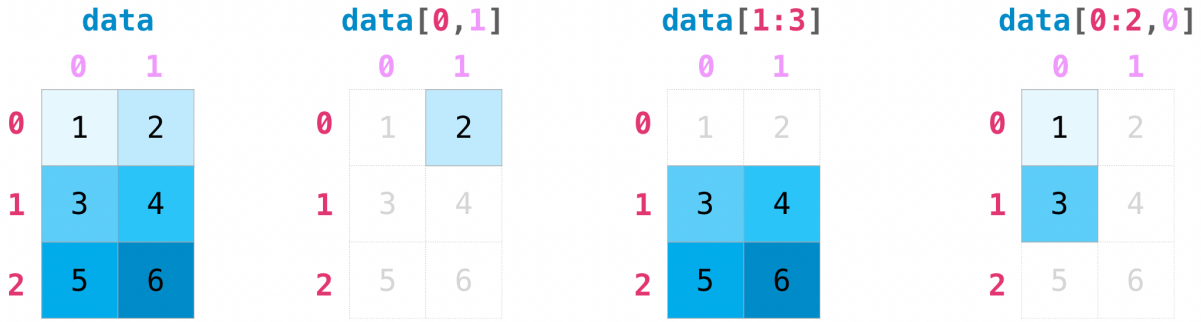
```
print(arr[:, 1])
```

```
[2 7 8]
```

# Fatiamento - Slicing



# Fatiamento - Slicing



# Tipos de dados do NumPy

- O NumPy define mais tipos de dados que os existentes em Python:
  - i  - integer
  - b - boolean
  - u - unsigned integer
  - f  - float
  - c - complex float
  - m - timedelta
  - M - datetime
  - O - object
  - S - string
  - U - unicode string
  - V - fixed chunk of memory for other type ( void )
- Para descobrir o tipo de dado em NumPy use `dtype()`

# Tipos de dados do NumPy

```
arr = np.array([1, 2, 3, 4])
print(arr.dtype)
```

```
int32
```

```
arr = np.array([1.0, 2.0, 3.0, 4.0])
print(arr.dtype)
```

```
float64
```

```
arr = np.array(['apple', 'banana', 'cherry'])
print(arr.dtype)
```

```
<U6
```

# Criando um array com um tipo de dado definido

- A função `array()` usada para criar arrays pode ter um argumento opcional - `dtype` - que nos permite definir o tipo de dados esperado dos elementos de matriz:

```
arr = np.array([1, 2, 3, 4], dtype='S')
```

- Foi criado um array de strings com os números.

# Tipos de dados do NumPy

- Os tipos i, u, f, S and U permitem definir o tamanho, em bytes.
  - i - integer
  - u - unsigned integer
  - f - float
  - S - string
  - U - unicode string
- Se define com 2, 4 ou 8 bytes para números
- Para strings, se define a quantidade de caracteres.

```
arr = np.array([1, 2, 3, 4], dtype='S5')
print(arr.dtype)
```

```
|S5
```

# Copiando um array

- Para copiar um array é necessário utilizar a função `copy()`:

```
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42
print(arr)
print(x)
```

```
[42  2  3  4  5]
[1 2 3 4 5]
```

# Copiando um array

- Quando não usamos a função `copy()`, criamos apenas uma referência para o array original:

```
arr = np.array([1, 2, 3, 4, 5])
x = arr
arr[0] = 42
print(arr)
print(x)
```

```
[42  2  3  4  5]
[42  2  3  4  5]
```

# O Shape de um array

- O shape, ou forma, de um array, é dado pelo número de elementos em cada dimensão.
- Os arrays NumPy têm um atributo chamado shape que retorna uma tupla com cada índice tendo o número de elementos correspondentes em cada dimensão:

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

```
(2, 3)
```

# Reshaping arrays

- Reshape significa mudar a forma de um array.
- Reshaping podemos adicionar ou remover dimensões ou alterar o número de elementos em cada dimensão.
- Muito usado em ciência de dados e ML.
- Utiliza-se o método `reshape(newshape)`
  - `newshape` define a nova forma do array.
  - Pode ter qualquer forma, desde que o número de elementos do novo array seja igual ao original.

# Reshaping um array

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(4, 2)
print(arr)
print(newarr)
```

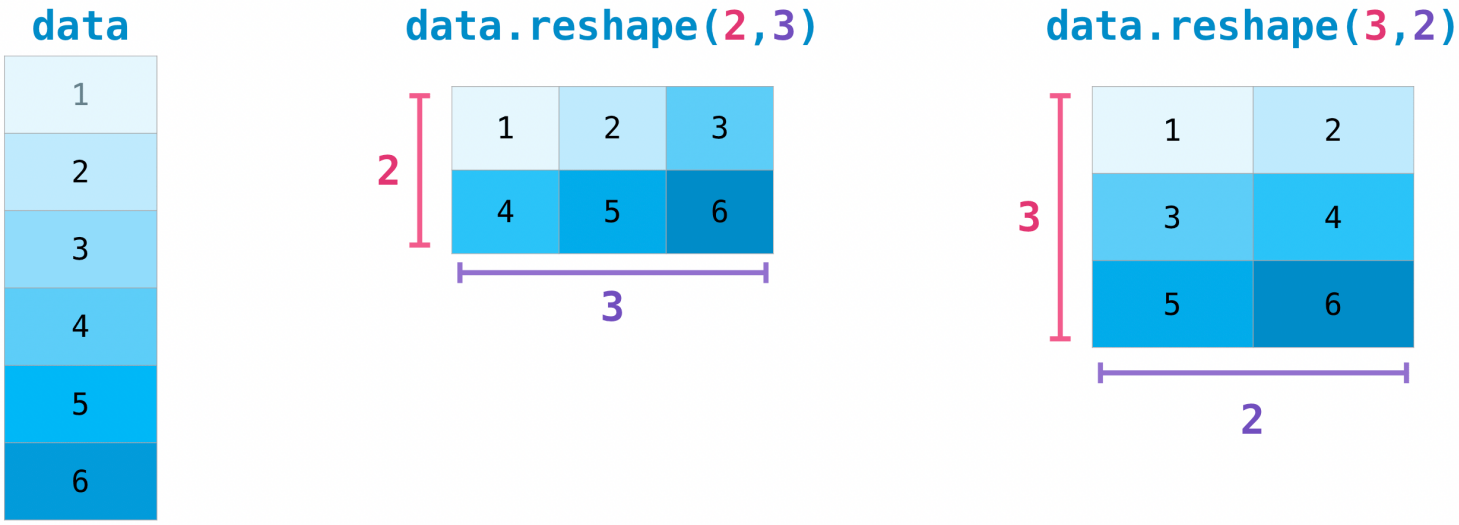
```
[1 2 3 4 5 6 7 8]
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

# Reshaping um array

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 4)
print(newarr)
```

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

# Reshaping um array



# Reshaping um array

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, 2)
print(newarr)
```

```
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

# Reshaping um array

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
newarr2 = arr.reshape(8)
print(arr)
print(newarr2)
```

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

# Também conhecido como Flattening

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, 2)
newarr2 = arr.reshape(-1)
print(newarr2)
```

Saída:
[1 2 3 4 5 6 7 8]

- Flattening:
  - Reduz qualquer array multidimensional à uma dimensão

# Iterando nos arrays NumPy

- Iterar em um array NumPy é semelhante a iterar em listas.
- Utiliza-se um comando de repetição como o `for`:

```
arr = np.array([1, 2, 3])
for x in arr:
      print(x)
```

Saída:
1
2
3

# Iterando nos arrays NumPy – 2D

- Iterar em um array NumPy é semelhante a iterar em listas.
- Utiliza-se um comando de repetição como o `for`:

```
arr = np.array([[1, 2, 3],[4, 5, 6]])
for x in arr:
    print(x)
```

Saída:
[1 2 3]
[4 5 6]

# Iterando nos arrays NumPy – 2D

- Iterar em um array NumPy é semelhante a iterar em listas.
- Utiliza-se um comando de repetição como o `for`:

```
arr = np.array([[1, 2, 3],[4, 5, 6]])
for x in arr:
  for y in x:
    print(y)
```

Saída:
1
2
3
4
5 ... 6

# Iterando nos arrays NumPy – 3D

- Iterar em um array NumPy é semelhante a iterar em listas.
- Utiliza-se um comando de repetição como o `for`:

```
arr = np.array([[[1,2],[3,4]],[[5, 6],[7,8]]])
for x in arr:
   for y in x:
      for z in y:
         print(z)
```

Saída:
1
2
3
4
5... 6 ... 7 ... 8

# Busca em arrays

- Para pesquisar um array, use o método where():

```
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

Saída:

(array([3, 5, 6],)

# Ordenando arrays

- Ordenar significa colocar elementos em uma sequência ordenada.
- Sequência ordenada é qualquer sequência que tem uma ordem correspondente a elementos, como numérico ou alfabético, ascendente ou descendente.
- O objeto array em NumPy tem uma função chamada `sort()`, que classificará um array especificado.

# Ordenando arrays

```
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```

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

```
arr = np.array(['banana', 'cherry', 'apple'])
print(np.sort(arr))
```

```
['apple' 'banana' 'cherry']
```

```
arr = np.array([[3, 2, 4], [5, 0, 1]])
print(np.sort(arr))
```

```
[[2 3 4]
 [0 1 5]]
```

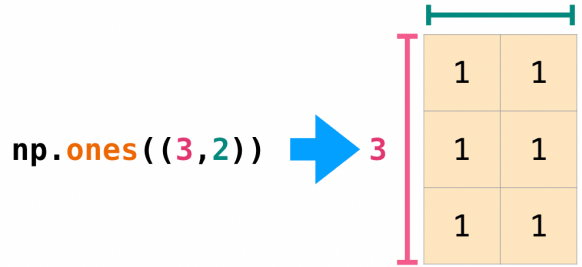
# Funções Básicas

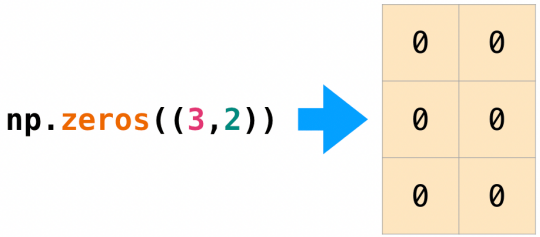
# Realizando operações entre arrays

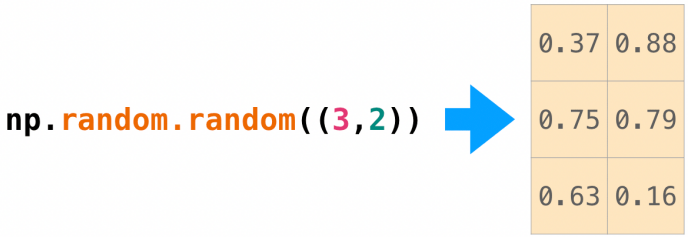
- O NumPy define diversas operações matemáticas entre martrizes, otimizadas para velocidade de processamento.
- Todas as operações básicas são definidas:
  - Soma, subtração, multiplicação e divisão escalar de vetores
  - Multiplicação vetorial
  - Multiplicação de matrizes
  - ...

# Criando matrizes automaticamente

```
np.ones((3, 2))
np.zeros((3, 2))
np.random.rand(3, 2)
```

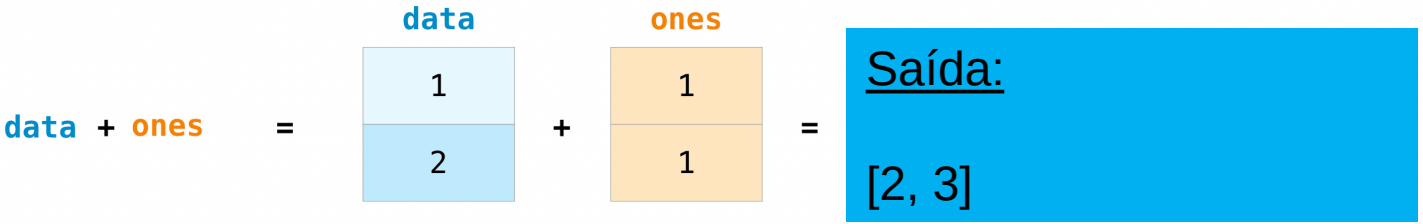


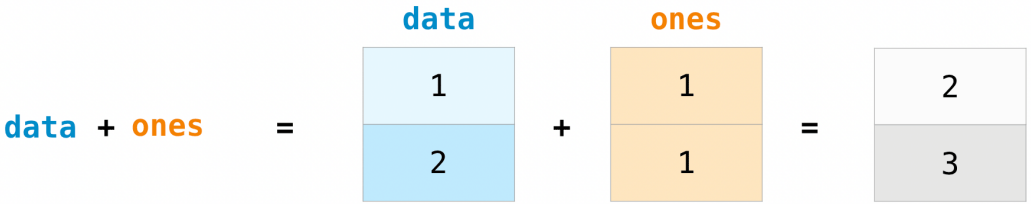
# Soma escalar

```
data = np.array([1, 2])
ones = np.ones(2, dtype=int)
data + ones
```



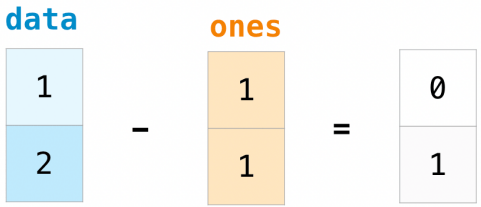
# Soma escalar

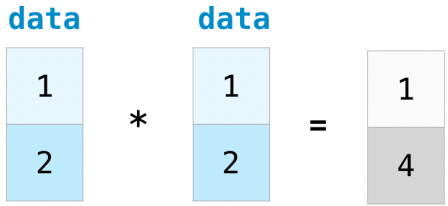
```
data = np.array([1, 2])
ones = np.ones(2, dtype=int)
data + ones
```

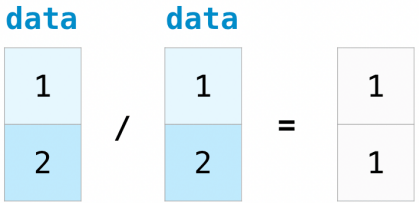


# Subtração, multiplicação, divisão escalar

```
data - ones
data * data
data / data
```

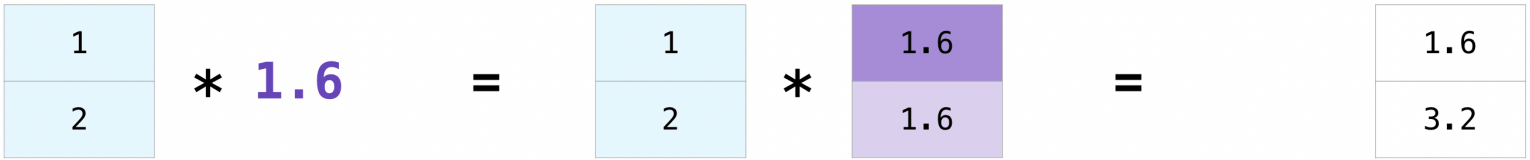


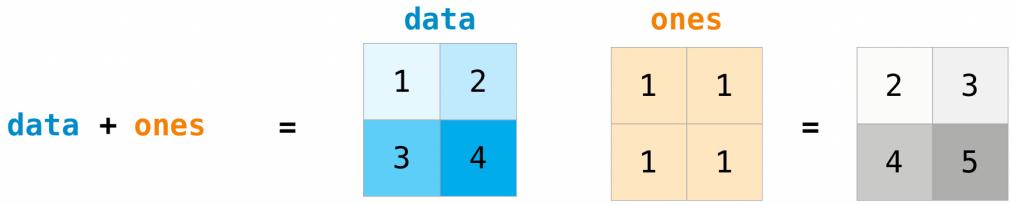
# Broadcast, ou multiplicação por um escalar

```
data = np.array([1.0, 2.0])
data * 1.6
```



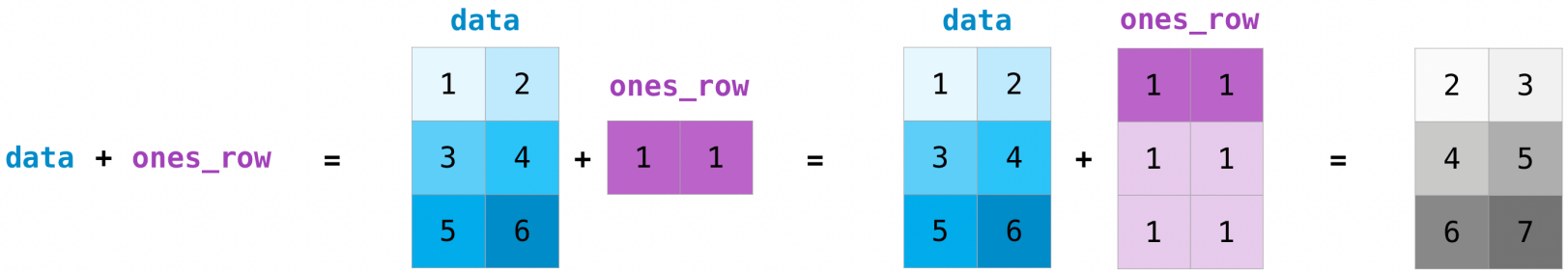
# Soma de matrizes

```
data = np.array([[1, 2], [3, 4]])
ones = np.array([[1, 1], [1, 1]])
data + ones
```



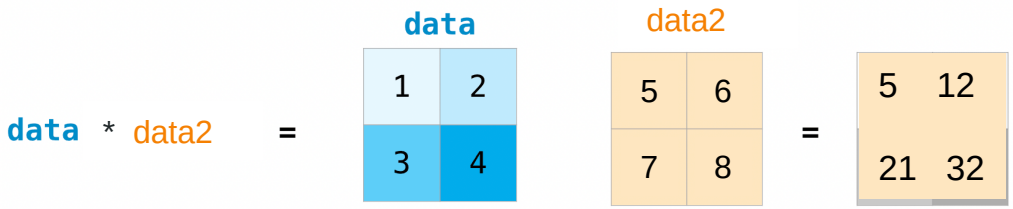
# Soma de matrizes de diferentes tamanhos

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



# Multiplicação de matrizes

```
data = np.array([[1, 2], [3, 4]])
data2 = np.array([[5, 6], [7, 8]])
data * data2
```



# Produto escalar

- O produto escalar é uma operação definida entre dois vetores que fornece um número real (também chamado "escalar") como resultado.
  - É o produto interno padrão do espaço euclidiano.
- Algebricamente, o produto escalar de dois vetores é formado pela multiplicação de seus componentes correspondentes e pela soma dos produtos resultantes.
  - Geometricamente, é o produto das magnitudes euclidianas dos dois vetores e o cosseno do ângulo entre eles.

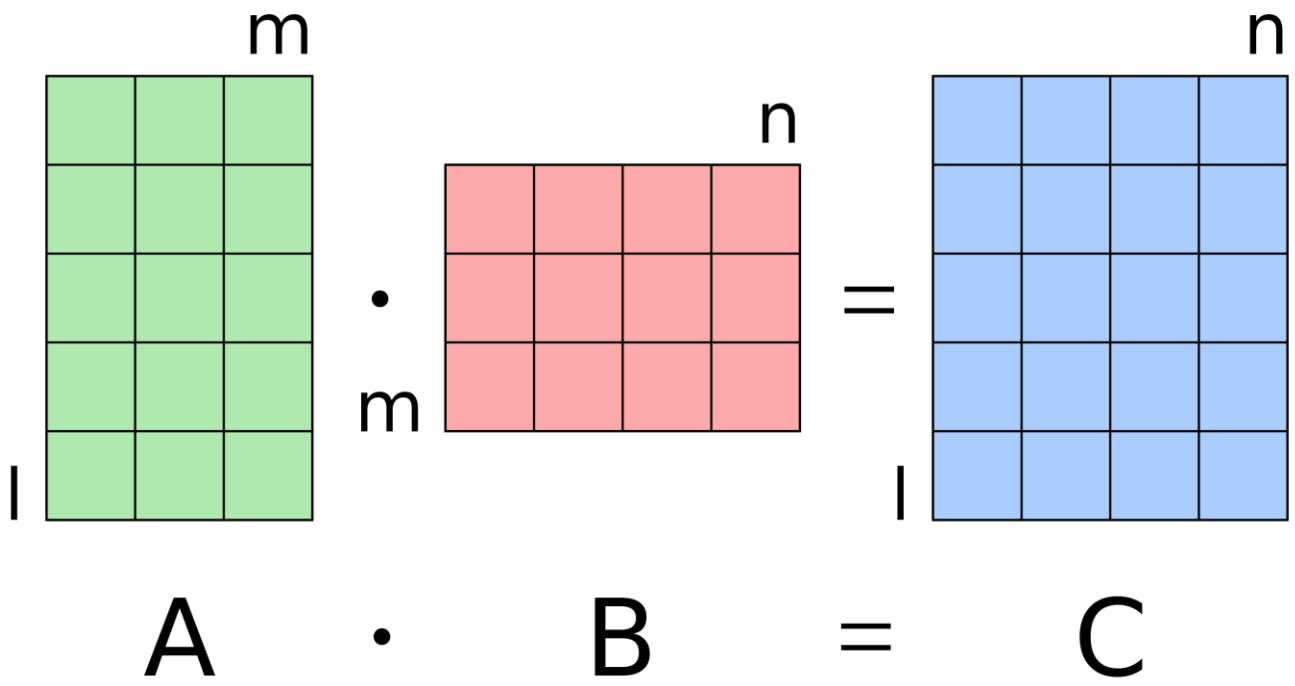
# Produto escalar

$$\begin{bmatrix} a & b \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = a.x + b.y$$

$$\begin{bmatrix} a & b & c \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = a.x + b.y + c.z$$

$$\begin{bmatrix} a & b & c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ t \end{bmatrix} = a.x + b.y + c.z + d.t$$

# Multiplicação de matriz




Número de Colunas de A = Número de Linhas de B

# Produto escalar em NumPy

```
data = np.array([1, 2])
data2 = np.array([3, 4])
x = data.dot(data2)
print(x)
```

Saída:

11

# Produto escalar em NumPy

```
data = np.array([1, 2, 3])
data2 = np.array([4, 5, 6])
x = data.dot(data2)
print(x)
```

Saída:

32

# Multiplicação de matriz em NumPy

```
data = np.array([[1, 2], [3, 4]])
data2 = np.array([[5, 6], [7, 8]])
x = data.dot(data2)
print(x)
```

Saída:

[[19 22]
 [43 50]]

# Multiplicação de matriz em NumPy

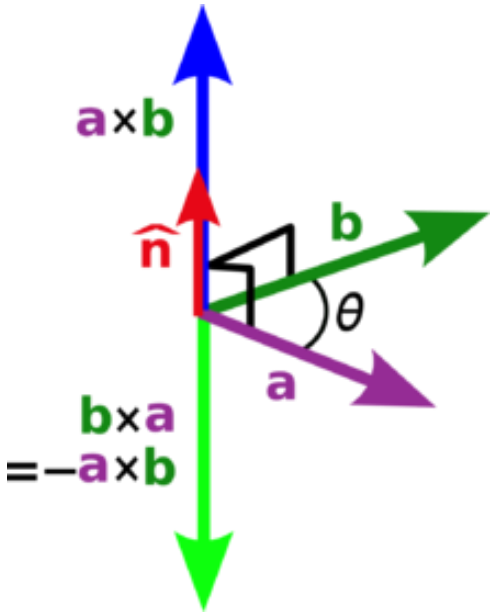
```
data  = np.array([[1,2,3],[4,5,6],[7,8,9]])
data2 = np.array([[1,2,3],[4,5,6],[7,8,9]])
x = data.dot(data2)
print(x)
```

Saída:

```
[[ 30  36  42]
 [ 66  81  96]
 [102 126 150]]
```

# Produto Vetorial

- O produto vetorial é uma operação sobre dois vetores em um espaço tridimensional e é denotado por ×.
  - Dados dois vetores independentes linearmente a e b, o produto vetorial a × b é um vetor perpendicular ao vetor a e ao vetor b e é a normal do plano contendo os dois vetores.
- Seu resultado difere do produto escalar por ser também um vetor, ao invés de um escalar.



# Produto Vetorial em NumPy

- Em NumPy, o produto vetorial, também chamado cross, é escrito como: `cross()`
  - The cross product of a and b R3 is a vector perpendicular to both a and b.
  - If a and b are arrays of vectors, the vectors are defined by the last axis of a and b by default, and these axes can have dimensions 2 or 3.
    - In cases where both input vectors have dimension 2, the z-component of the cross product is returned.
    - Where the dimension of either a or b is 2, the third component of the input vector is assumed to be zero and the cross product calculated accordingly.

# Produto vetorial em NumPy

```
data = np.array([1, 2, 3])
data2 = np.array([4, 5, 6])
x = np.cross(data, data2)
print(x)
```

Saída:

[-3  6 -3]

# Produto vetorial em NumPy

```
data  = np.array([[1,2,3],[4,5,6],[1,0,0]])
data2 = np.array([[4,5,6],[4,5,6],[0,1,0]])
x = np.cross(data, data2)
print(x)
```

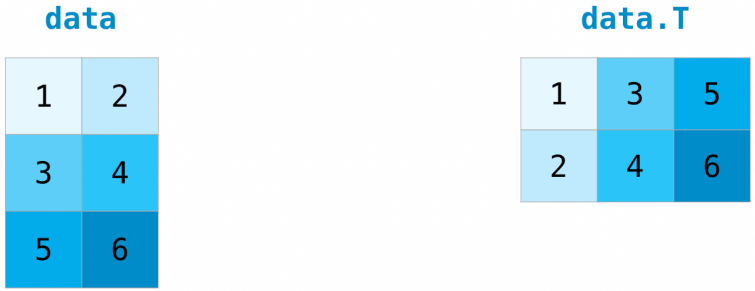
Saída:

```
[[-3  6 -3]
 [ 0  0  0]
 [ 0  0  1]]
```

# Finalmentes...

# Matriz transposta

`arr.transpose()` ou `arr.T()`



# Inversão de matriz

- A matriz inversa de uma matriz é tal que se for multiplicada pela matriz original, resulta em matriz identidade.

- Usamos a função `numpy.linalg.inv()` para calcular a inversa de uma matriz.

# Inversão de matriz

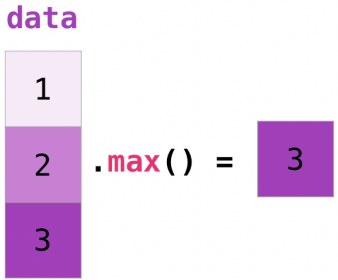
```
x = np.array([[1,2],[3,4]])
y = np.linalg.inv(x)
print (x)
print (y)
print (np.dot(x,y))
```

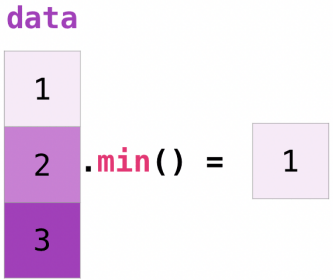
Saída:

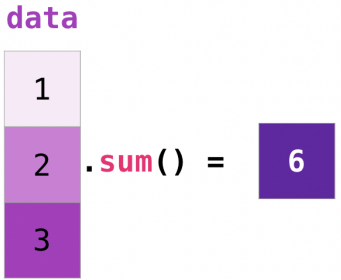
```
[[1 2]
 [3 4]]
[[-2.   1. ]
 [ 1.5 -0.5]]
[[1.0e+00 1.1e-16]
 [0.0e+00 1.0e+00]]
```

# Máximo, mínimo, soma

```
data.max()
data.min()
data.sum()
```

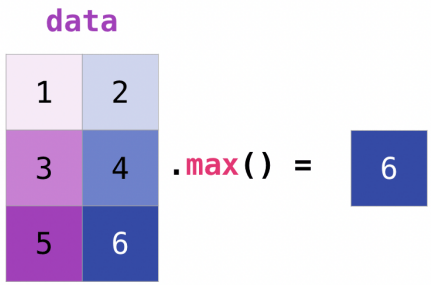


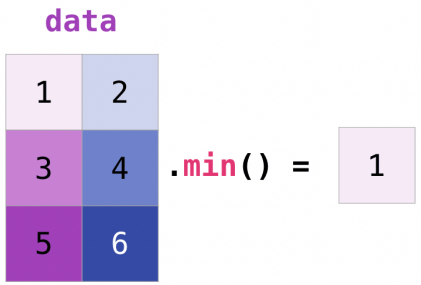



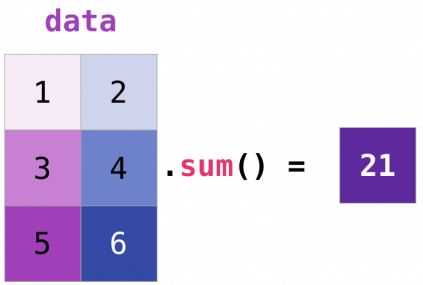



# Máximo, mínimo, soma também para matrizes

```
data.max()
data.min()
data.sum()
```



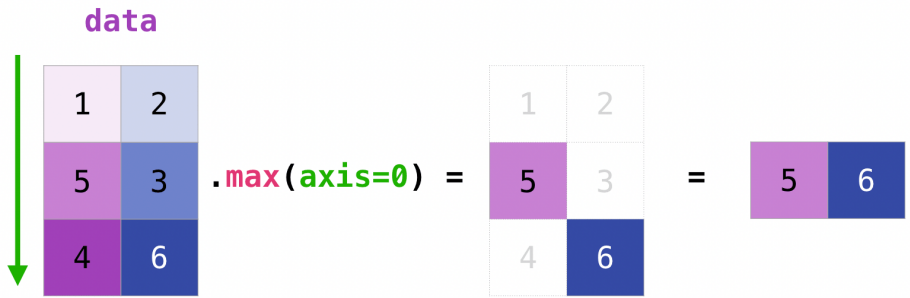



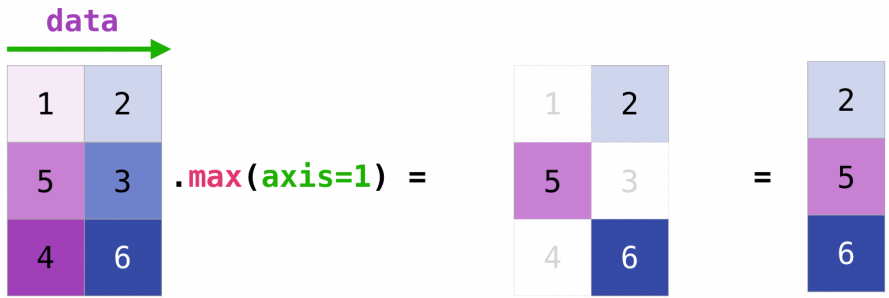



# Máximo, mínimo, soma por linha ou coluna

```
data.max(axis=0)
```

```
data.max(axis=1)
```



# Gerando números aleatórios com NumPy

- NumPy possui o módulo `random` para a geração de números aleatórios.
- Ele possui duas funções básicas:
  - `rand()`: retorna um número real entre 0 e 1.
  - `randint(n)`: retorna um número real entre 0 e n.

# Gerando números aleatórios com NumPy

```
import numpy as np
print (np.random.rand())
print (np.random.randint(100))
```

Saída:

0.10183291451684662
55

# Gerando números aleatórios com NumPy

- NumPy possui diversos algoritmos conhecidos para a geração de números aleatórios...

### Accessing the BitGenerator

| | |
|---|---|
| bit_generator | Gets the bit generator instance used by the generator |

### Simple random data

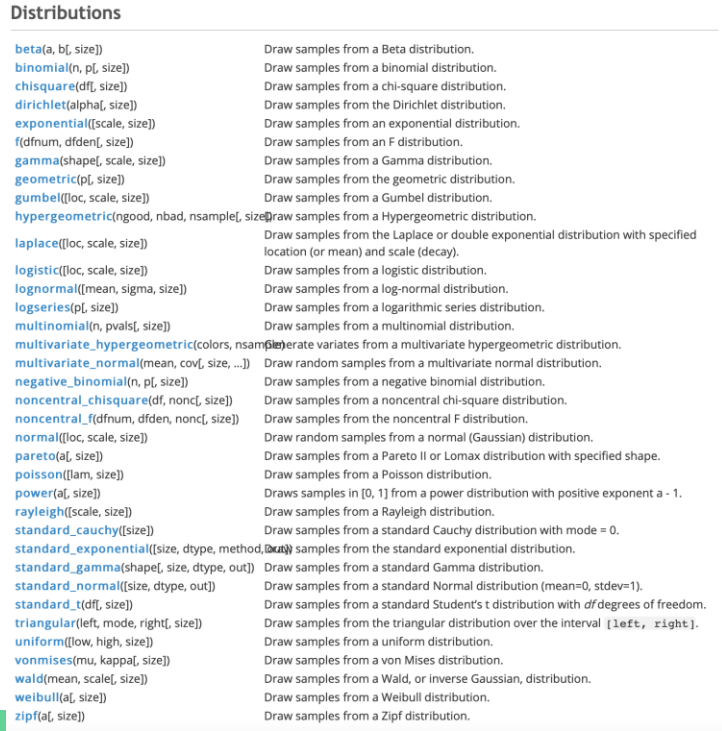
| | |
|---|---|
| integers(low[, high, size, dtype, endpoint]) | Return random integers from *low* (inclusive) to *high* (exclusive), or if endpoint=True, *low* (inclusive) to *high* (inclusive). |
| random([size, dtype, out]) | Return random floats in the half-open interval [0.0, 1.0). |
| choice(a[, size, replace, p, axis, shuffle]) | Generates a random sample from a given 1-D array |
| bytes(length) | Return random bytes. |

### Permutations

| | |
|---|---|
| shuffle(x[, axis]) | Modify a sequence in-place by shuffling its contents. |
| permutation(x[, axis]) | Randomly permute a sequence, or return a permuted range. |

# Gerando números aleatórios com NumPy

- NumPy possui diversos tipos de distribuições probabilísticas...



**Distributions**

| | |
|---|---|
| beta(a, b[, size]) | Draw samples from a Beta distribution. |
| binomial(n, p[, size]) | Draw samples from a binomial distribution. |
| chisquare(df[, size]) | Draw samples from a chi-square distribution. |
| dirichlet(alpha[, size]) | Draw samples from the Dirichlet distribution. |
| exponential([scale, size]) | Draw samples from an exponential distribution. |
| f(dfnum, dfden[, size]) | Draw samples from an F distribution. |
| gamma(shape[, scale, size]) | Draw samples from a Gamma distribution. |
| geometric(p[, size]) | Draw samples from the geometric distribution. |
| gumbel([loc, scale, size]) | Draw samples from a Gumbel distribution. |
| hypergeometric(ngood, nbad, nsample[, size]) | Draw samples from a Hypergeometric distribution. |
| laplace([loc, scale, size]) | Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay). |
| logistic([loc, scale, size]) | Draw samples from a logistic distribution. |
| lognormal([mean, sigma, size]) | Draw samples from a log-normal distribution. |
| logseries(p[, size]) | Draw samples from a logarithmic series distribution. |
| multinomial(n, pvals[, size]) | Draw samples from a multinomial distribution. |
| multivariate_hypergeometric(colors, nsample[, size]) | Generate variates from a multivariate hypergeometric distribution. |
| multivariate_normal(mean, cov[, size, ...]) | Draw random samples from a multivariate normal distribution. |
| negative_binomial(n, p[, size]) | Draw samples from a negative binomial distribution. |
| noncentral_chisquare(df, nonc[, size]) | Draw samples from a noncentral chi-square distribution. |
| noncentral_f(dfnum, dfden, nonc[, size]) | Draw samples from the noncentral F distribution. |
| normal([loc, scale, size]) | Draw random samples from a normal (Gaussian) distribution. |
| pareto(a[, size]) | Draw samples from a Pareto II or Lomax distribution with specified shape. |
| poisson([lam, size]) | Draw samples from a Poisson distribution. |
| power(a[, size]) | Draws samples in [0, 1] from a power distribution with positive exponent a - 1. |
| rayleigh([scale, size]) | Draw samples from a Rayleigh distribution. |
| standard_cauchy([size]) | Draw samples from a standard Cauchy distribution with mode = 0. |
| standard_exponential([size, dtype, method, out]) | Draw samples from the standard exponential distribution. |
| standard_gamma(shape[, size, dtype, out]) | Draw samples from a standard Gamma distribution. |
| standard_normal([size, dtype, out]) | Draw samples from a standard Normal distribution (mean=0, stdev=1). |
| standard_t(df[, size]) | Draw samples from a standard Student's t distribution with *df* degrees of freedom. |
| triangular(left, mode, right[, size]) | Draw samples from the triangular distribution over the interval `[left, right]`. |
| uniform([low, high, size]) | Draw samples from a uniform distribution. |
| vonmises(mu, kappa[, size]) | Draw samples from a von Mises distribution. |
| wald(mean, scale[, size]) | Draw samples from a Wald, or inverse Gaussian, distribution. |
| weibull(a[, size]) | Draw samples from a Weibull distribution. |
| zipf(a[, size]) | Draw samples from a Zipf distribution. |

# Conclusão

# Conclusão

- NumPy provém uma maneira simples tratar vetores, matrizes e dados em geral.
- NumPy resolve quase toda matemática...
- NumPy possui muitas outras funções prontas...
  - https://numpy.org/doc/stable/reference/routines.html

# Mathematical functions

## Trigonometric functions

| | |
|---|---|
| sin(x, /[, out, where, casting, order, ...]) | Trigonometric sine, element-wise. |
| cos(x, /[, out, where, casting, order, ...]) | Cosine element-wise. |
| tan(x, /[, out, where, casting, order, ...]) | Compute tangent element-wise. |
| arcsin(x, /[, out, where, casting, order, ...]) | Inverse sine, element-wise. |
| arccos(x, /[, out, where, casting, order, ...]) | Trigonometric inverse cosine, element-wise. |
| arctan(x, /[, out, where, casting, order, ...]) | Trigonometric inverse tangent, element-wise. |
| hypot(x1, x2, /[, out, where, casting, ...]) | Given the "legs" of a right triangle, return its hypotenuse. |
| arctan2(x1, x2, /[, out, where, casting, ...]) | Element-wise arc tangent of `x1/x2` choosing the quadrant correctly. |
| degrees(x, /[, out, where, casting, order, ...]) | Convert angles from radians to degrees. |
| radians(x, /[, out, where, casting, order, ...]) | Convert angles from degrees to radians. |
| unwrap(p[, discont, axis]) | Unwrap by changing deltas between values to 2*pi complement. |
| deg2rad(x, /[, out, where, casting, order, ...]) | Convert angles from degrees to radians. |
| rad2deg(x, /[, out, where, casting, order, ...]) | Convert angles from radians to degrees. |

## Hyperbolic functions

| | |
|---|---|
| sinh(x, /[, out, where, casting, order, ...]) | Hyperbolic sine, element-wise. |
| cosh(x, /[, out, where, casting, order, ...]) | Hyperbolic cosine, element-wise. |
| tanh(x, /[, out, where, casting, order, ...]) | Compute hyperbolic tangent element-wise. |
| arcsinh(x, /[, out, where, casting, order, ...]) | Inverse hyperbolic sine element-wise. |
| arccosh(x, /[, out, where, casting, order, ...]) | Inverse hyperbolic cosine, element-wise. |
| arctanh(x, /[, out, where, casting, order, ...]) | Inverse hyperbolic tangent element-wise. |

## Rounding

| | |
|---|---|
| around(a[, decimals, out]) | Evenly round to the given number of decimals. |
| round_(a[, decimals, out]) | Round an array to the given number of decimals. |
| rint(x, /[, out, where, casting, order, ...]) | Round elements of the array to the nearest integer. |
| fix(x[, out]) | Round to nearest integer towards zero. |
| floor(x, /[, out, where, casting, order, ...]) | Return the floor of the input, element-wise. |
| ceil(x, /[, out, where, casting, order, ...]) | Return the ceiling of the input, element-wise. |
| trunc(x, /[, out, where, casting, order, ...]) | Return the truncated value of the input, element-wise. |

# Statistics

## Order statistics

| | |
|---|---|
| amin(a[, axis, out, keepdims, initial, where]) | Return the minimum of an array or minimum along an axis. |
| amax(a[, axis, out, keepdims, initial, where]) | Return the maximum of an array or maximum along an axis. |
| nanmin(a[, axis, out, keepdims]) | Return minimum of an array or minimum along an axis, ignoring any NaNs. |
| nanmax(a[, axis, out, keepdims]) | Return the maximum of an array or maximum along an axis, ignoring any NaNs. |
| ptp(a[, axis, out, keepdims]) | Range of values (maximum - minimum) along an axis. |
| percentile(a, q[, axis, out, ...]) | Compute the q-th percentile of the data along the specified axis. |
| nanpercentile(a, q[, axis, out, ...]) | Compute the qth percentile of the data along the specified axis, while ignoring nan values. |
| quantile(a, q[, axis, out, overwrite_input, ...]) | Compute the q-th quantile of the data along the specified axis. |
| nanquantile(a, q[, axis, out, ...]) | Compute the qth quantile of the data along the specified axis, while ignoring nan values. |

## Averages and variances

| | |
|---|---|
| median(a[, axis, out, overwrite_input, keepdims]) | Compute the median along the specified axis. |
| average(a[, axis, weights, returned]) | Compute the weighted average along the specified axis. |
| mean(a[, axis, dtype, out, keepdims]) | Compute the arithmetic mean along the specified axis. |
| std(a[, axis, dtype, out, ddof, keepdims]) | Compute the standard deviation along the specified axis. |
| var(a[, axis, dtype, out, ddof, keepdims]) | Compute the variance along the specified axis. |
| nanmedian(a[, axis, out, overwrite_input, ...]) | Compute the median along the specified axis, while ignoring NaNs. |
| nanmean(a[, axis, dtype, out, keepdims]) | Compute the arithmetic mean along the specified axis, ignoring NaNs. |
| nanstd(a[, axis, dtype, out, ddof, keepdims]) | Compute the standard deviation along the specified axis, while ignoring NaNs. |
| nanvar(a[, axis, dtype, out, ddof, keepdims]) | Compute the variance along the specified axis, while ignoring NaNs. |

## Correlating

| | |
|---|---|
| corrcoef(x[, y, rowvar, bias, ddof]) | Return Pearson product-moment correlation coefficients. |
| correlate(a, v[, mode]) | Cross-correlation of two 1-dimensional sequences. |
| cov(m[, y, rowvar, bias, ddof, fweights, ...]) | Estimate a covariance matrix, given data and weights. |

## Histograms

| | |
|---|---|
| histogram(a[, bins, range, normed, weights, ...]) | Compute the histogram of a set of data. |
| histogram2d(x, y[, bins, range, normed, ...]) | Compute the bi-dimensional histogram of two data samples. |
| histogramdd(sample[, bins, range, normed, ...]) | Compute the multidimensional histogram of some data. |
| bincount(x[, weights, minlength]) | Count number of occurrences of each value in array of non-negative ints. |
| histogram_bin_edges(a[, bins, range, weights]) | Function to calculate only the edges of the bins used by the histogram function. |
| digitize(x, bins[, right]) | Return the indices of the bins to which each value in input array belongs. |

## Decompositions

| | |
|---|---|
| linalg.cholesky(a) | Cholesky decomposition. |
| linalg.qr(a[, mode]) | Compute the qr factorization of a matrix. |
| linalg.svd(a[, full_matrices, compute_uv]) | Singular Value Decomposition. |

## Matrix eigenvalues

| | |
|---|---|
| linalg.eig(a) | Compute the eigenvalues and right eigenvectors of a square array. |
| linalg.eigh(a[, UPLO]) | Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix. |
| linalg.eigvals(a) | Compute the eigenvalues of a general matrix. |
| linalg.eigvalsh(a[, UPLO]) | Compute the eigenvalues of a complex Hermitian or real symmetric matrix. |

## Norms and other numbers

| | |
|---|---|
| linalg.norm(x[, ord, axis, keepdims]) | Matrix or vector norm. |
| linalg.cond(x[, p]) | Compute the condition number of a matrix. |
| linalg.det(a) | Compute the determinant of an array. |
| linalg.matrix_rank(M[, tol, hermitian]) | Return matrix rank of array using SVD method |
| linalg.slogdet(a) | Compute the sign and (natural) logarithm of the determinant of an array. |
| trace(a[, offset, axis1, axis2, dtype, out]) | Return the sum along diagonals of the array. |

## Solving equations and inverting matrices

| | |
|---|---|
| linalg.solve(a, b) | Solve a linear matrix equation, or system of linear scalar equations. |
| linalg.tensorsolve(a, b[, axes]) | Solve the tensor equation `a x = b` for x. |
| linalg.lstsq(a, b[, rcond]) | Return the least-squares solution to a linear matrix equation. |
| linalg.inv(a) | Compute the (multiplicative) inverse of a matrix. |
| linalg.pinv(a[, rcond, hermitian]) | Compute the (Moore-Penrose) pseudo-inverse of a matrix. |
| linalg.tensorinv(a[, ind]) | Compute the 'inverse' of an N-dimensional array. |

# Discrete Fourier Transform (numpy.fft)

## Standard FFTs

| | |
|---|---|
| fft(a[, n, axis, norm]) | Compute the one-dimensional discrete Fourier Transform. |
| ifft(a[, n, axis, norm]) | Compute the one-dimensional inverse discrete Fourier Transform. |
| fft2(a[, s, axes, norm]) | Compute the 2-dimensional discrete Fourier Transform |
| ifft2(a[, s, axes, norm]) | Compute the 2-dimensional inverse discrete Fourier Transform. |
| fftn(a[, s, axes, norm]) | Compute the N-dimensional discrete Fourier Transform. |
| ifftn(a[, s, axes, norm]) | Compute the N-dimensional inverse discrete Fourier Transform. |

## Real FFTs

| | |
|---|---|
| rfft(a[, n, axis, norm]) | Compute the one-dimensional discrete Fourier Transform for real input. |
| irfft(a[, n, axis, norm]) | Compute the inverse of the n-point DFT for real input. |
| rfft2(a[, s, axes, norm]) | Compute the 2-dimensional FFT of a real array. |
| irfft2(a[, s, axes, norm]) | Compute the 2-dimensional inverse FFT of a real array. |
| rfftn(a[, s, axes, norm]) | Compute the N-dimensional discrete Fourier Transform for real input. |
| irfftn(a[, s, axes, norm]) | Compute the inverse of the N-dimensional FFT of real input. |

## Hermitian FFTs

| | |
|---|---|
| hfft(a[, n, axis, norm]) | Compute the FFT of a signal that has Hermitian symmetry, i.e., a real spectrum. |
| ihfft(a[, n, axis, norm]) | Compute the inverse FFT of a signal that has Hermitian symmetry. |

## Helper routines

| | |
|---|---|
| fftfreq(n[, d]) | Return the Discrete Fourier Transform sample frequencies. |
| rfftfreq(n[, d]) | Return the Discrete Fourier Transform sample frequencies (for usage with rfft, irfft). |
| fftshift(x[, axes]) | Shift the zero-frequency component to the center of the spectrum. |
| ifftshift(x[, axes]) | The inverse of fftshift. |

# Financial functions

## Simple financial functions

| | |
|---|---|
| **fv**(rate, nper, pmt, pv[, when]) | Compute the future value. |
| **pv**(rate, nper, pmt[, fv, when]) | Compute the present value. |
| **npv**(rate, values) | Returns the NPV (Net Present Value) of a cash flow series. |
| **pmt**(rate, nper, pv[, fv, when]) | Compute the payment against loan principal plus interest. |
| **ppmt**(rate, per, nper, pv[, fv, when]) | Compute the payment against loan principal. |
| **ipmt**(rate, per, nper, pv[, fv, when]) | Compute the interest portion of a payment. |
| **irr**(values) | Return the Internal Rate of Return (IRR). |
| **mirr**(values, finance_rate, reinvest_rate) | Modified internal rate of return. |
| **nper**(rate, pmt, pv[, fv, when]) | Compute the number of periodic payments. |
| **rate**(nper, pmt, pv, fv[, when, guess, tol, ...]) | Compute the rate of interest per period. |

# Conclusão

- NumPy provém uma maneira simples tratar vetores, matrizes e dados em geral.
- NumPy resolve quase toda matemática...
- NumPy possui muitas outras funções prontas...

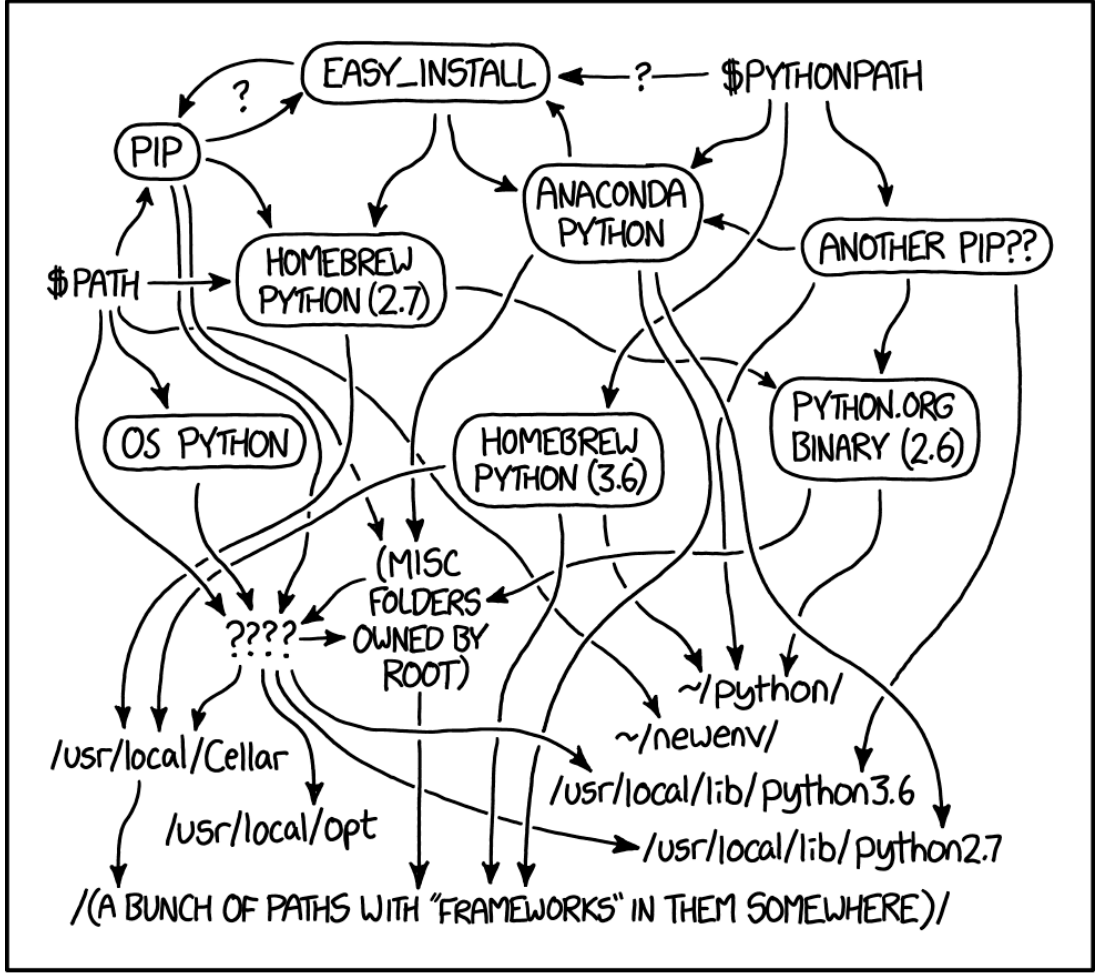
- Em Python, tudo é muito simples.

# Sites usados

- https://numpy.org
- https://numpy.org/doc/stable/user/absolute_beginners.html
- https://medium.com/ensina-ai/entendendo-a-biblioteca-numpy-4858fde63355
- https://www.w3schools.com/python/numpy_intro.asp
- https://www.tutorialspoint.com/numpy/numpy_inv.htm

# Fim

xkcd.com/1987/




MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.