

# Introdução ao Numpy

---

Patrícia Novais



- » O que é Numpy e onde é utilizado
- » Arrays
- » Índices
- » Fatiamentos e seleções
- » Operações com arrays
- » Broadcasting
- » Funções universais
- » Métodos matemáticos

# O que é Numpy?

- » Numpy (*Numerical Python*) é uma biblioteca Python que permite trabalhar computação científica, com o uso de arrays e matrizes multidimensionais.
- » Inspirada no Matlab
- » Cálculos computacionais muito mais eficientes
- » Funções nativas que facilitam e agilizam a manipulação de dados.
- » O Numpy é *core* de diversas outras bibliotecas dentro do Python, como as bibliotecas Pandas e Scipy.



- » Numpy não é uma função built-in do Python, o que significa que ela não é nativa da linguagem e precisa ser carregada/importada antes de utilizarmos.
- » Em Python é muito comum o uso de *alias* para a chamada de funções de bibliotecas não nativas.
- » O Numpy, em geral, é chamado com o *alias* *np*.

```
import numpy as np
```

## Arrays: o que são?

- » A principal estrutura do Numpy é o array, uma estrutura que armazena diversos dados em N dimensões.
- » É similar ao conceito de listas.
- » Todos os dados precisam ser de mesmo tipo.
- » Existem diversas maneiras de criar arrays no Numpy. Dentre elas, podemos citar as funções `np.array()`, `np.ones()`, `np.full()`.

» A maneira mais simples e comum de declarar um array é usando a função `np.array()` onde usamos uma lista de valores como argumento.

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

array unidimensional

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

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

array bidimensional

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

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

array tridimensional

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

## Arrays: outros tipos

» Podemos ainda criar arrays específicos como, por exemplo, com 1 ou 0 com as funções `np.ones()` e `np.zeros()` em que os argumentos são as dimensões.

```
one = np.ones([2,2])  
one
```

array bidimensional, com valores  
1s.

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

```
zeros = np.zeros([2,2,2])  
zeros
```

array tridimensional, com valores  
0s.

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

*Table 4-1. Array creation functions*

Function	Description
<code>array</code>	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
<code>asarray</code>	Convert input to ndarray, but do not copy if the input is already an ndarray
<code>arange</code>	Like the built-in <code>range</code> but returns an ndarray instead of a list
<code>ones</code> , <code>ones_like</code>	Produce an array of all 1s with the given shape and dtype; <code>ones_like</code> takes another array and produces a ones array of the same shape and dtype
<code>zeros</code> , <code>zeros_like</code>	Like <code>ones</code> and <code>ones_like</code> but producing arrays of 0s instead
<code>empty</code> , <code>empty_like</code>	Create new arrays by allocating new memory, but do not populate with any values like <code>ones</code> and <code>zeros</code>
<code>full</code> , <code>full_like</code>	Produce an array of the given shape and dtype with all values set to the indicated “fill value” <code>full_like</code> takes another array and produces a filled array of the same shape and dtype
<code>eye</code> , <code>identity</code>	Create a square $N \times N$ identity matrix (1s on the diagonal and 0s elsewhere)



» Assim como as listas, podemos acessar informações dos arrays através dos índices dos elementos.

» Array unidimensional:

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

```
data[2]
```

```
2
```

```
data[2:4]
```

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

» Array com mais de uma dimensão:

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

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

```
vet[1,:]
```

```
array([4, 5, 6])
```

```
vet[:,2]
```

```
array([3, 6, 9])
```

```
vet[1,1]
```

```
5
```

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

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

```
arr.shape
```

```
(2, 2, 2)
```

```
arr[0,0,1]
```

```
2
```

```
arr[1,1,0]
```

```
7
```

		5	6
1	2		8
3	4		

- » É preciso muito cuidado ao fatiar um array do Numpy.
- » Qualquer alteração no fatiamento irá refletir no próprio array original.

```
data = np.arange(10)
```

```
data
```

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

```
data[0:5]
```

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

```
data[0:5] = -99
```

```
data
```

```
array([-99, -99, -99, -99, -99, 5, 6, 7, 8, 9])
```

```
data_1 = data[0:5]
```

```
data_1
```

```
array([-99, -99, -99, -99, -99])
```

```
data_1[2:4] = 5555
```

```
data
```

```
array([-99, -99, 5555, 5555, -99, 5, 6, 7, 8, 9])
```

- » O fatiamento do Numpy gera uma visualização do array original.
- » Isso porque o Numpy foi criado para trabalhar com muitos dados e fazer diversas cópias de um array pode sair computacionalmente caro.
- » Sempre que precisar copiar uma fatia de um array, use a função `copy()`.

```
data_c = data[5:].copy()
```

```
data_c
```

```
array([5, 6, 7, 8, 9])
```

```
data_c[0:3] = -99
```

```
data_c
```

```
array([-99, -99, -99,  8,  9])
```

```
data
```

```
array([ -99,  -99, 5555, 5555, -99,  5,  6,  7,  8,  9])
```

» Podemos fatiar um array de acordo com uma condição.

```
data = np.arange(10)
```

```
data[data > 5]
```

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

```
data[data == 4]
```

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

```
data[data != 4]
```

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

1. Abra o Jupyter lab
2. Crie um array *data* com a seguinte lista: [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
3. Selecione apenas os dados diferentes de 4,6,8,10. Aloque em um array chamado *data\_2*.
4. Crie um array *dim2* que tenha 2 dimensões e a lista [1,2,3,4,5] na primeira linha e a lista [6,7,8,9,10] na 2a linha.
5. Crie um array chamado *dim2\_2* que seja uma fatia de *dim2* e que contenha apenas os valores 3,4,8,9.

## Arrays: funções úteis

- » Há várias funções que podem ser aplicadas aos arrays e que são muito úteis.
- » *shape*: informa as dimensões de um array.

```
data1 = np.array([[1,2,3],[4,5,6],[7,8,9]])  
data2 = np.array([[1,2,3,4],[4,5,6,7],[7,8,9,10]])  
data3 = np.array([[1],[2],[3],[4],[5]])  
data4 = np.array([1,2,3,4,5])
```

```
data1.shape
```

```
(3, 3)
```

```
data2.shape
```

```
(3, 4)
```

```
data3.shape
```

```
(5, 1)
```

```
data4.shape
```

```
(5,)
```

» *len*: determina o comprimento da 1a dimensão de um array.

```
len(data1)
```

3

```
len(data2)
```

3

```
len(data3)
```

5

```
len(data4)
```

5

Ambos possuem apenas 1  
dimensão e de mesmo  
comprimento!



# Arrays: funções úteis

» *ndim*: determina o número de dimensões de um array.0

```
data1.ndim
```

```
2
```

```
data2.ndim
```

```
2
```

```
data3.ndim
```

```
2
```

```
data4.ndim
```

```
1
```

```
zeros.ndim
```

```
3
```

**data3 possui uma coluna  
e 5 linhas, por isso 2  
dimensões!**

» *size*: retorna a quantidade de elementos presente em um array.

```
data1.size
```

```
9
```

```
data2.size
```

```
12
```

```
data3.size
```

```
5
```

```
data4.size
```

```
5
```

» A função *reshape* permite remodelar um array sem alterar os elementos que o compõe.

data2

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

data2.reshape(4,3)

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

data3

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

data3.reshape(1,5)

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

» A função *flatten* transforma um array multidimensional em um unidimensional.

```
data1.flatten()
```

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

```
data2.flatten()
```

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

```
data3.flatten()
```

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

```
data4.flatten()
```

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

» Para transpor um array, podemos usar a função *np.transpose()*.

data1

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

data1.transpose()

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

data2

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

data2.transpose()

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

1. Abra o Jupyter lab
2. Crie um array *data* com a seguinte lista: [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
3. Verifique a dimensão, a quantidade de elementos e a forma desse array.
4. Crie um novo array chamado *data2*, com 4x4 elementos, aplicando a função *reshape* no array *data*.
5. Verifique a dimensão, a quantidade de elementos e a forma desse array.
6. Crie um novo array chamado *data\_t*, que seja a transposição do array *data2*.
7. Use a função *np.flatten()* e crie o array *data\_flat*.
8. Compare o array *data* com o array *data\_flat*.

» Podemos calcular as operações aritméticas básicas dentro de um array (soma, subtração, multiplicação, divisão...)

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

```
a
```

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

```
a*2
```

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

```
a + 3
```

```
array([[4, 5],  
       [6, 7]])
```

```
a - 2
```

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

```
a/5
```

```
array([[0.2, 0.4],  
       [0.6, 0.8]])
```

```
a ** 2
```

```
array([[ 1,  4],  
       [ 9, 16]], dtype=int32)
```

» Quando temos dois arrays de mesma forma (*shape*), também podemos realizar algumas operações e comparações.

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

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

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

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

```
a + b
```

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

```
a - b
```

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

```
a > b
```

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

```
a == b
```

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

```
a * b
```

```
array([[ 1,  4],  
       [ 9, 16]])
```



» *Broadcasting* é um método usado pelo Numpy para realizar operações entre arrays de tamanhos diferentes.

```
b = np.array([2,3])
```

```
b
```

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

```
b * 3
```

escalar e 1 dimensional

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

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

```
a
```

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

```
a * 3
```

escalar e 2 dimensões

```
array([[ 3,  6],  
       [ 9, 12]])
```

```
a * b
```

1 dimensão e 2 dimensões

```
array([[ 2,  6],  
       [ 6, 12]])
```

1. Abra o Jupyter lab
2. Crie 5 arrays **a**, **b**, **c**, **d** e **e** com as seguintes listas:
  - a. [7]
  - b. [[4,6,8],[3,5,7]]
  - c. [3,3,3]
  - d. [[3,2,1],[1,2,3]]
  - e. [5,10]
3. Faça as seguintes operações:
  - a.  $a * b$
  - b.  $d - b$
  - c.  $c + b$
  - d.  $c + e$
4. O que aconteceu no item **d**) do último exercício? Explique.

» Funções universais são funções matemáticas que são aplicadas a todos os elementos de um array.

» Consulte as funções universais na documentação do Numpy:

<https://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs>

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

raíz quadrada

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

```
np.sqrt(a)
```

exponencial

```
array([[1.          , 1.41421356],  
       [1.73205081, 2.          ]])
```

```
np.exp(a)
```

```
array([[ 2.71828183,  7.3890561 ],  
       [20.08553692, 54.59815003]])
```

```
np.sin(a)
```

seno

```
array([[ 0.84147098,  0.90929743],  
       [ 0.14112001, -0.7568025 ]])
```

```
np.negative(a)
```

negativo

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

- » O Numpy possibilita ainda extrair informações matemáticas e estatísticas dos arrays.
- » Podemos calcular a média, mediana, soma ou ainda extrair os máximos e mínimos de arrays.

```
a = np.array([[15,22],[7,12]])  
a
```

```
array([[15, 22],  
       [ 7, 12]])
```

```
a.mean()
```

```
14.0
```

média

```
a.min()
```

```
7
```

mínimo

```
a.max()
```

```
22
```

máximo

```
a.sum()
```

```
56
```

soma

```
a.cumsum()
```

```
array([15, 37, 44, 56], dtype=int32)
```

soma cumulativa

1. Abra o Jupyter lab
2. Crie 2 arrays **a** e **b** com as seguintes listas:
  - a. `[[4,6,8],[3,5,7]]`
  - b. `[[11,13,17],[23,29,31]]`
3. Use a função `cbrt()` e calcule a raiz cúbica dos arrays **a** e **b**.
4. Calcule a soma cumulativa de **a**.
5. Calcule a média de **a** e **b**.
6. Aplique a função `np.negative()` no array **b** e soma com **b**. O que aconteceu?

- » Para concatenar arrays, usamos a função *np.concat()*.
- » Para arrays unidimensionais:

```
a = np.array([1,2])  
a
```

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

```
b = np.array([3,4,5,6])  
b
```

```
array([3., 4., 5., 6.])
```

```
c = np.array([7,8,9])  
c
```

```
array([7., 8., 9.])
```

```
np.concatenate((a, b, c))
```

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

» Arrays com mais dimensões:

Por padrão, a concatenação acontece na 1ª dimensão.

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

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

```
b = np.array([[5, 6], [7, 8]])  
b
```

```
array([[5., 6.],  
       [7., 8.]])
```

```
np.concatenate((a,b))
```

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

» Para especificar o eixo da concatenação, usamos o argumento *axis*.

```
np.concatenate((a,b), axis=0)
```

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

```
np.concatenate((a,b), axis=1)
```

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



» Para especificar o eixo da concatenação, usamos o argumento *axis*.

```
a = np.array([[[5, 6], [7,8]], [[5, 6], [7,8]]], float)
a
```

```
array([[[5., 6.],
        [7., 8.]],
       [[5., 6.],
        [7., 8.]])
```

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

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

```
np.concatenate((b,c), axis=0)
```

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

```
np.concatenate((b,c), axis=1)
```

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

```
np.concatenate((b,c), axis=2)
```

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

1. Abra o Jupyter lab
2. Crie 2 arrays **arrays** com as seguintes listas:
  - a. `[[1,2,3],[4,5,6]]`
  - b. `[[2,4,6,8],[10,12,14,16]]`
  - c. `[[11,13,17],[23,29,31]]`
  - d. `[[13,14,5],[19,21,23]]`
3. Concatene **a** e **b**. O que aconteceu?
4. Concatene **b** e **c**, tanto usando o eixo 0 (linha) quanto o eixo 1 (coluna).
5. Concatene **a** e **d**, tanto usando o eixo 0 (linha) quanto o eixo 1 (coluna).