

# PACKAGES PARA A MACHINE LEARNING

# Estendendo as capacidades do Python na Data Science e ML

Seguem-se algumas bibliotecas importantes do Python, já presentes no ambiente Anaconda, que tornarão a nossa tarefa de desenvolvimento de projetos de ML bem mais fácil e produtiva

- NumPy (Documentação: <https://numpy.org/doc/>)
  - *para arrays multidimensionais, álgebra linear e outras operações de computação matemática de alto nível*
- Pandas (Documentação: <https://pandas.pydata.org/docs/>)
  - *para representação e tratamento dos dados na forma de tabelas (dataframes)*
- Matplotlib e Seaborn (Documentação: <https://matplotlib.org/contents.html> e <https://seaborn.pydata.org/>)
  - *para visualização gráfica dos dados*
- Scikit-learn ([https://scikit-learn.org/stable/user\\_guide.html](https://scikit-learn.org/stable/user_guide.html))
  - *para a ML (disponibiliza-nos os principais algoritmos de ML)*

# Uma breve visão do NumPy

- Em Python, usamos uma lista para facilmente colecionar elementos
  - *uma lista é um conceito algo equivalente aos arrays das outras linguagens, ainda que nos proporcione outra flexibilidade*
    - por exemplo, a lista permite-nos colecionar um conjunto heterogéneo de elementos (elementos de tipos diferentes)

```
lista=[‘quarta’, 28, ‘outubro’, 2020]
```
  - *Mas esta característica interessante das listas também se pode revelar uma desvantagem quando estão em causa grandes quantidades de dados*
    - que é tipicamente o que se passa com os datasets tratados em ML
- Para tornar as listas estruturas flexíveis, o Python assegura o acesso e manipulação dos seus elementos com operações computacionalmente pouco eficientes
- o NumPy ajuda a ultrapassar esta dificuldade,
  - *ao suportar arrays multidimensionais*
  - *e todo um conjunto de operações e funções matemáticas de grande utilidade, que nos permitem lidar de forma eficiente com os grandes volumes de dados representados nessas estruturas.*
  - *Porém, contrariamente às listas, nos arrays do NumPy os elementos têm quer ser todos do mesmo tipo (coleções homogéneas)*

# Uma breve visão do NumPy

O package NumPy fornece-nos então uma maneira eficiente de guardar e manipular arrays multidimensionais em Python

Como estudaremos mais adiante, a principal biblioteca do Python para a ML – a scikit-learn – usa arrays NumPy bidimensionais para representação dos *dados* processados pelos seus algoritmos

Principais recursos do NumPy:

- Fornece uma estrutura *ndarray*, (*n-dimensional array*) que permite o armazenamento e a manipulação eficientes de vetores, matrizes e outras estruturas de ordem superior.
- Fornece uma sintaxe legível e eficiente para operar com essas estruturas de dados, desde aritmética elementar simples até operações algébricas lineares mais complexas.

Nos casos mais simples, os arrays do NumPy parecem-se bastante com as listas do Python

- Por exemplo, segue-se um array NumPy contendo o intervalo de números de 0 a 15 (equivalente à função do Python `range()`):

```
import numpy as np  
a=np.arange(16)  
a
```

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

# Uma breve visão do NumPy

- Os arrays do NumPy tornam eficiente, quer o armazenamento dos dados, quer as operações elemento a elemento.
  - *Por exemplo, para elevar ao quadrado cada elemento do array, basta aplicar diretamente ao array o operador \*\*:*

```
a**2
```

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

- Operação que resultaria bem mais emaranhada caso fosse realizada ao estilo do Python, usando listas por compreensão:

```
[x**2 for x in range(16)]
```

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

- Contrariamente às listas do Python (que estão limitadas a uma dimensão), os arrays NumPy podem ser multidimensionais
  - *Por exemplo, alteremos a forma do array a, de unidimensional para bidimensional 4x4:*

```
m=a.reshape(4,4)
```

```
m
```

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

# Uma breve visão do NumPy

- Um array bidimensional representa uma matriz, e o NumPy sabe como realizar de forma eficiente as operações típicas de matrizes
  - *Por exemplo, a transposta duma matriz pode ser facilmente obtida invocando o operador T (o mesmo que invocar o método transpose()):*

```
m.T
```

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

- *e o produto duma matriz por um vetor pode ser realizada com np.dot():*

```
np.dot(m, [3,2,1,0])
```

```
array([ 4, 28, 52, 76])
```

- *e até operações mais sofisticadas como é o caso da decomposição de uma matriz nos seus valores próprios:*

```
np.linalg.eigvals(m)
```

```
array([ 3.24642492e+01, -2.46424920e+00,  1.92979794e-15, -4.09576009e-16])
```

- As operações de álgebra linear sustentam grande parte das análises de dados que se fazem atualmente na ML e no *data mining*.

# Criação de arrays NumPy

- Com a função `numpy.array()`, conseguimos criar um qualquer array multidimensional, usando uma estrutura de listas para definir a sua forma e conteúdo

```
linha0=['00', '01', '02', '03']; linha1=['10', '11', '12', '13']
linhas=[linha0, linha1]
y=np.array(linhas)
print(y)

[['00' '01' '02' '03']
 ['10' '11' '12' '13']]
```

- Para além da função `numpy.array()`, e da função `numpy.arange()` que nos permite apenas criar arrays NumPy de uma só dimensão, existem outras funções para criar arrays preenchidos já com valores:

Função	Array criado
<code>numpy.zeros()</code>	array de 0's
<code>numpy.ones()</code>	array de 1's
<code>numpy.full()</code>	array preenchido com um valor específico
<code>numpy.random.random()</code>	array de valores aleatórios uniformemente distribuídos entre 0 e 1
<code>numpy.eye()</code>	matriz identidade
<code>numpy.diag()</code>	matriz diagonal

# numpy.zeros() / numpy.ones()

► ML

```
a=np.zeros((3,10))      #Matriz de zeros  
print('nº linhas, nº colunas =', a.shape)  
print('nº de dimensões =', a.ndim)  
a
```

(nº linhas, nº colunas) = (3, 10)

nº de dimensões = 2

```
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., 0., 0., 0., 0., 0.]])
```

► ML

```
a=np.ones((2,6))      #Matriz de uns  
print('nº linhas, nº colunas =', a.shape)  
print('nº de dimensões =', a.ndim)  
a
```

(nº linhas, nº colunas) = (2, 6)

nº de dimensões = 2

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

# numpy.full() / numpy.random.random()

► ML

```
a=np.full((3,4), 5)      #Matriz de cinco  
print('nº linhas, nº colunas =', a.shape)  
print('nº de dimensões =', a.ndim)  
a
```

```
(nº linhas, nº colunas) = (3, 4)  
nº de dimensões = 2
```

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

► ML

```
a=np.random.random((2,5))      #Matriz de valores aleatórios entre 0 e 1  
print('nº linhas, nº colunas =', a.shape)  
print('nº de dimensões =', a.ndim)  
a
```

```
(nº linhas, nº colunas) = (2, 5)  
nº de dimensões = 2
```

```
array([[0.62132656, 0.20543814, 0.99216038, 0.89813845, 0.60342637],  
       [0.32339668, 0.94417811, 0.42589372, 0.89772876, 0.76331926]])
```

# numpy.eye() / numpy.diag()

▶ M↓

```
a=np.eye(3)      #Matriz identidade
print('nº linhas, nº colunas =', a.shape)
print('nº de dimensões =', a.ndim)
a
```

```
(nº linhas, nº colunas) = (3, 3)
nº de dimensões = 2
```

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

▶ M↓

```
a=np.diag([10,20,30])      #Matriz diagonal
print('nº linhas, nº colunas =', a.shape)
print('nº de dimensões =', a.ndim)
a
```

```
(nº linhas, nº colunas) = (3, 3)
nº de dimensões = 2
```

```
array([[10,  0,  0],
       [ 0,  20,  0],
       [ 0,  0,  30]])
```

# Indexação de arrays NumPy

- O acesso aos elementos de um array NumPy faz-se usando indexação, tal como o fazemos nas outras linguagens

```
a0=np.arange(0,100,10)
a0
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])

a1=a0.reshape((2,5))
a1
array([[ 0, 10, 20, 30, 40],
       [50, 60, 70, 80, 90]])

a0[5]    50
a1[1,0]  50
```

- Indexação booleana

- Adicionalmente, o NumPy permite que se use como índices os valores lógicos *True* ou *False* em todas as posições do array, para sinalizar os elementos que, respetivamente, serão ou não selecionados

```
a0[[False,True,False,False,False,True,False,False,False]] array([10, 50])
```

- E como o uso dum array numa condição, converte os seus valores em valores lógicos

```
a0>40  array([False, False, False, False, False, True, True, True, True, True])
```

a seleção dos elementos de acordo com essa condição resulta simplificada

```
a0[a0>40]  array([50, 60, 70, 80, 90])
```

- Trata-se de uma forma de indexação bastante útil. Veja-se o exemplo seguinte

```
a2=np.arange(15)  array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

```
a2[a2 % 2 != 0] #escolher apenas os ímpares  array([ 1, 3, 5, 7, 9, 11, 13])
```

# Slicing de arrays (fatiamento)<sup>1</sup>

- O *slicing* dum array NumPy faz-se de forma semelhante ao *slicing* das listas do Python, mas aplicado às várias dimensões

```
a=np.arange(60) # [0, 1, 2, 3, 4, 5, 6, .., 58, 59]  
  
a=a[a%10<=5] # [0, 1, 2, 3, 4, 5, 10, 11, .., 54, 55]  
  
a=a.reshape((6,6))  
  
>>> a[0, 3:5]  
array([3, 4])  
  
>>> a[4:, 4:]  
array([[44, 45],  
       [54, 55]])  
  
>>> a[:, 2]  
a([2, 12, 22, 32, 42, 52])  
  
>>> a[2::2, ::2]  
array([[20, 22, 24],  
       [40, 42, 44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

<sup>1</sup> Conteúdo adaptado de “Scipy Lecture Notes”, DOI 10.5281/zenodo.594102, 2020 (com licenciamento CC BY 4.0)

# Diferentes formas de indexação<sup>1</sup>

Três formas distintas de indexar um array NumPy

```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]  
array([1, 12, 23, 34, 45])  
  
>>> a[3:, [0,2,5]]  
array([[30, 32, 35],  
       [40, 42, 45],  
       [50, 52, 55]])  
  
>>> mask = np.array([1,0,1,0,0,1], dtype=bool)  
>>> a[mask, 2]  
array([2, 22, 52])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

# Operações matemáticas

- É possível realizar, de uma forma simples, operações matemáticas sobre arrays NumPy
  - *incluindo as operações aritméticas básicas, elemento a elemento*

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

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

```
soma=x+y
```

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

```
dif=x-y
```

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

```
y=x+6    #broadcasting  
y         #(operação de difusão)
```

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

```
prod=x*y
```

```
array([[ 0,  7, 16],  
       [27, 40, 55]])
```

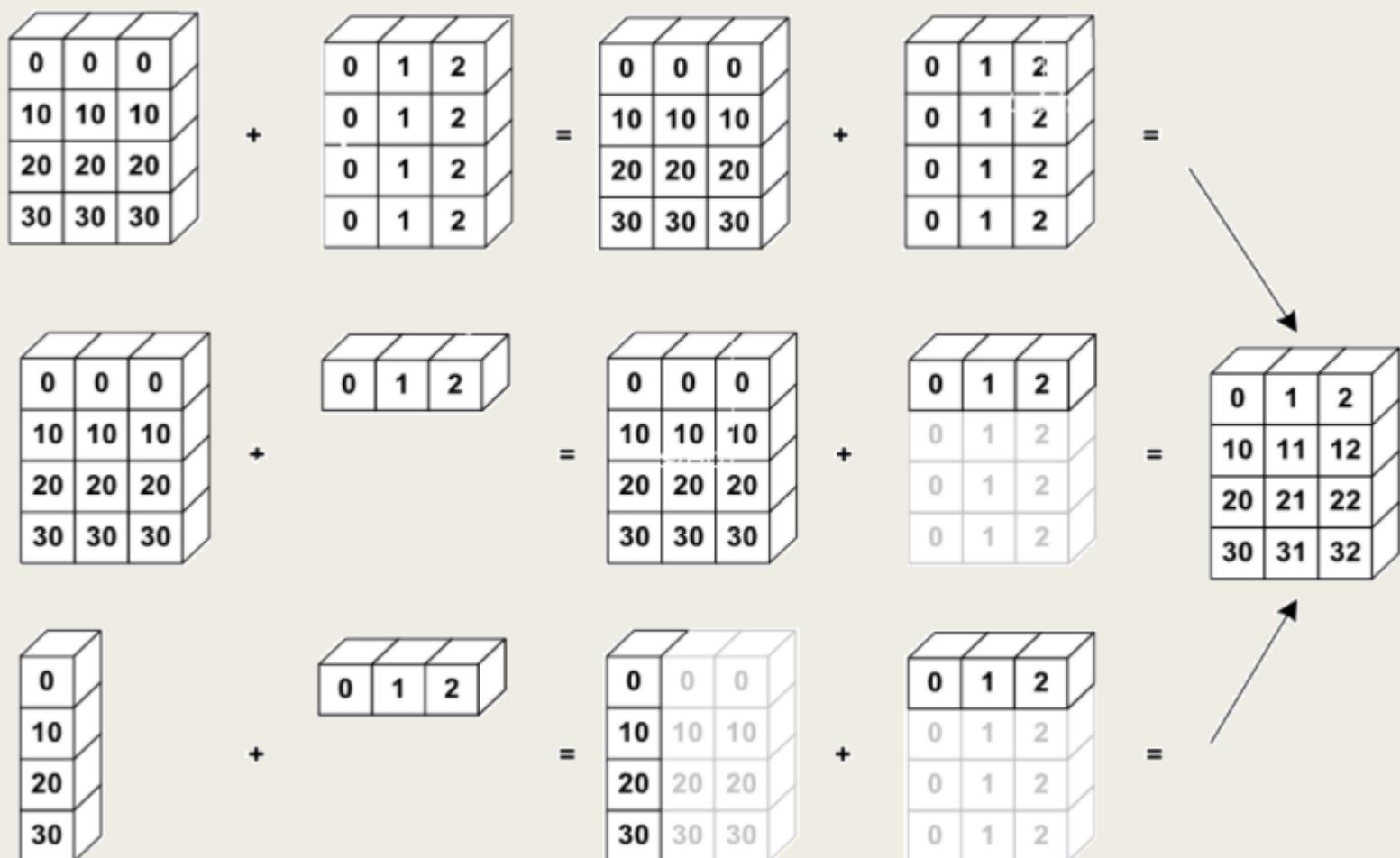
```
div=x/y
```

```
array([[0.      , 0.143, 0.25  ],  
       [0.333, 0.4    0.455]])
```

- As mesmas operações podem ser realizadas através de funções:
  - np.add(x,y), np.subtract(x,y), np.multiply(x,y), np.divide(x,y)
- Para se realizar o produto interno entre dois vetores, representados por 2 arrays NumPy unidimensionais, usa-se a função np.dot(), resultando um valor escalar
  - exemplo: v1=np.array([1,2,3]); v2=np.array([3,2,1]); np.dot(v1,v2) 10
  - Quando usada em arrays bidimensionais, é equivalente à multiplicação de matrizes.

# Broadcasting<sup>1</sup>

- Perante dois arrays de dimensões diferentes envolvidos numa operação aritmética, o NumPy tenta converter esses arrays para a mesma dimensão, replicando o padrão de valores existentes nas novas dimensões criadas
  - Essa replicação de valores é designada ‘broadcasting’ (difusão)



# Matrizes NumPy

- O NumPy fornece-nos um tipo especial de array: o tipo `matrix`
  - *Trata-se de uma subclasse de `ndarray`*
  - *É, portanto, um array mas, necessariamente, bidimensional*
  - *A criação de uma matriz faz-se de forma similar à criação de arrays*

```
m1=np.matrix([[0,1,2],[3,4,5]])  
m1
```

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

- *É também possível converter um array num matriz, com a função `asmatrix()`*

```
a2=np.array([[5,4,3],[2,1,0]])  
m2=np.asmatrix(a2)  
m2
```

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

e uma matriz num array, com a função inversa `asarray()`.

- Outra importante diferença entre arrays e matrizes é quando se multiplicam esses tipos de objetos (com o operador `*`)
  - A multiplicação de arrays é realizada elemento a elemento, como se viu antes
  - Já a multiplicação de matrizes é equivalente à função `np.dot()`, ou seja, implementa o comportamento da multiplicação de matrizes da álgebra linear

<code>print(m1)</code>	<code>print(m2.T)</code>	<code>m1*m2.T</code>	<code>np.asarray(m1)*np.asarray(m2)</code>
<code>[[0 1 2]  [3 4 5]]</code>	<code>[[5 2]  [4 1]  [3 0]]</code>	<code>matrix([[10, 1],           [46, 10]])</code>	<code>array([[0, 4, 6],           [6, 4, 0]])</code>

# Funções de agregação

- Várias funções de agregação podem ser aplicadas aos arrays NumPy (e às matrizes)
  - à *totalidade dos valores ()*, *coluna a coluna (axis=0)*, ou *linha a linha (axis=1)*

```
print(a2)
```

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

```
print(a2.sum(), a2.sum(axis=0), a2.sum(axis=1), sep=' | ')
```

```
15 | [7 5 3] | [12 3]
```

```
print(a2.mean(), a2.mean(axis=0), a2.mean(axis=1), sep=' | ')
```

```
2.5 | [3.5 2.5 1.5] | [4. 1.]
```

```
print(a2.max(), a2.max(axis=0), a2.max(axis=1), sep=' | ')
```

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

```
print(a2.argmax(), a2.argmax(axis=0), a2.argmax(axis=1), sep=' | ')
```

```
0 | [0 0 0] | [0 0]
```

```
print(a2.min(), a2.min(axis=0), a2.min(axis=1), sep=' | ')
```

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

```
print(a2.argmin(), a2.argmin(axis=0), a2.argmin(axis=1), sep=' | ')
```

```
5 | [1 1 1] | [2 2]
```

- Outras funções de agregação do NumPy:
  - *average()*, *product()*, *median()*, *std()*, *var()*, *cumprod()*, *cumsum()*, *corrcoef()*.

# Ordenação de Arrays NumPy

O NumPy disponibiliza-nos funções para ordenação eficiente de arrays

Comecemos por criar um array com o nome de 5 alunos e outro com as suas notas (geradas aleatoriamente, mas positivas...)

- Facilmente conseguimos ordenar as notas com a função NumPy.sort()
- A função sort() não modifica o array original, como se pode constatar
- Outra função muito interessante, é a NumPy.argsort(), uma vez que nos devolve os índices (posições) que permitem ordenar o array
- Se invertermos a ordem desses índices,

*facilmente conseguimos ordenar os alunos pelo valor da sua nota, começando pelas maiores.*

```
nomes=np.array(['Ana', 'Rui', 'Ze', 'To', 'Rita'])  
notas=np.random.randint(10,21,5)  
print(notas)
```

[18 13 11 19 16]  
notas\_ord=np.sort(notas) #[11 13 16 18 19]

```
print(notas)  
[18 13 11 19 16]
```

indices\_ord=np.argsort(notas) #[2 1 4 0 3]

indices\_ord\_inv=indices\_ord[::-1] #[3, 0, 4, 1, 2]

```
print('Alunos ordenados pela nota: ', nomes[indices_ord_inv])  
print('Respetivas notas: ', notas[indices_ord_inv])
```

Alunos ordenados pela nota: ['To' 'Ana' 'Rita' 'Rui' 'Ze']  
Respetivas notas: [19 18 16 13 11]

# Redimensionamento

- Já conhecemos um método que devolve um array NumPy redimensionado

```
x=np.array([10,11,12,13,14,15]); y1=x.reshape(2,3); print(y1) [[10 11 12] [13 14 15]]
```

- e com o parâmetro `-1` podemos deixar que seja o método a definir uma das dimensões, de forma a bater certo com a quantidade de elementos

<pre>y2=x.reshape(2,-1); print(y2)</pre>	<pre>y3=x.reshape(-1,3); print(y3)</pre>
<pre>[[10 11 12] [13 14 15]]</pre>	<pre>[[10 11 12] [13 14 15]]</pre>

- Mas se o objetivo for alterar o próprio array, então podemos redimensioná-lo, atribuindo um tuplo à sua propriedade ‘shape’

```
x.shape=2,-1; print(x) [[10 11 12] [13 14 15]]
```

- e voltar a colocá-lo na sua forma unidimensional alterando a mesma propriedade

```
x.shape=-1; print("x={} ndim={} shape={}".format(x, x.ndim, x.shape))  
x=[10 11 12 13 14 15] ndim=1 shape=(6,)
```

# Cópia de Arrays NumPy

- **Cópia por referência (reference copy)**

- *Tal como com os outros objetos do Python, quando se atribui um Array NumPy a outro, apenas a referência é copiada*

- assim, qualquer alteração num, afeta o outro (na verdade só temos um array)

```
x=np.array([1,2,3,4]); y=x; y[1]=0; y.shape=(2,2); print(x,y) [[1 0] [3 4]] [[1 0] [3 4]]
```

- **Cópia superficial (shallow copy)**

- *Os arrays NumPy têm, no entanto, o método view(), que cria um novo array, mas que partilha os mesmos dados com o original*

- passa é a ter a sua própria forma independente da do array original

```
x=np.array([1,2,3,4]); y=x.view(); y.shape=(2,2); y[1,1]=0; print(x,y)
```

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

- *É este tipo de cópia que também acontece com o slicing e o redimensionamento*

```
x=np.array([1,2,3,4]); y=x[2:]; y[1]=0; print(x,y) [1 2 3 0] [3 0]
```

```
x=np.array([1,2,3,4]); y=x.reshape(2,2); y[1,1]=0; print(x,y) [1 2 3 0] [[1 2] [3 0]]
```

- **Cópia em profundidade (deep copy)**

- *O método copy() é aquele que permite uma duplicação quer do array quer dos elementos contidos nele, desvinculando completamente o array criado do que lhe deu origem*

```
x=np.array([1,2,3,4]); y=x.copy(); y.shape=(2,2); y[1,1]=0; print(x,y)
```

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

# NumPy, um *package* de suporte

- O NumPy, para além de ser o principal *package* de computação numérica do Python, dá suporte a muitos outros *packages* da Data Science, como é, precisamente, o caso daqueles com que também vamos trabalhar: Pandas, Matplotlib, Seaborn e Scikit-learn.