Programação II

Computação numérica com arrays (numpy)

Hugo Pacheco

DCC/FCUP
21/22

Operações matemáticas

- Podemos utilizar listas e listas de listas para representar vetores e matrizes
- No entanto, as operações habituais em matemática vetorial não estão disponíveis
- Temos de fazer a operação elemento a elemento
- E.g. somar ou multiplicar vetor por constante

```
>>> xs = [1,2,3]
>>> xs + 3
TypeError: can only concatenate list (not "int") to list
>>> xs * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> [ x * 3 for x in xs ]
[3, 6, 9]
```

Operações matemáticas

- Podemos utilizar listas e listas de listas para representar vetores e matrizes
- No entanto, <u>não são necessariamente eficientes</u>
  - espaço ocupado proporcional às dimensões
  - listas têm tipo/tamanho variável, matrizes não

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

```
matriz = [[0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0],
          [0, 2, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 3, 0]]
print(matriz[1][1])
```

Operações matemáticas

- Podemos utilizar dicionários para representar matrizes
- No entanto, as operações habituais em matemática vetorial não estão disponíveis
- No entanto, não são necessariamente eficientes
  - usa menos espaço, mas acessos mais lentos
  - dicionários também têm tipo/tamanho variável

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

```
matriz = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
print(matriz.get((1, 3), 0))

>>> matriz * 3
TypeError: unsupported operand type(s) for *: 'dict' and 'int'
>>> { xy : v*3 for xy,v in matriz.items() }
{(0, 3): 3, (2, 1): 6, (4, 3): 9}
```

NumPy

- O módulo *numpy* fornece implementações eficientes de arrays multi-dimensionais (vetores, matrizes, etc) e suporta operações matemáticas
  - dimensões fixas
  - todos os elementos têm o mesmo tipo

- E.g., para multiplicar por uma constante

```
>>> import numpy as np
>>> a = np.array([2, 3, 8])
>>> a.dtype
int64
>>> b = 2.1 * a
>>> b
[ 4.2, 6.3, 16.8]
>>> b.dtype
float64
```

NumPy (shape)

- E.g., para multiplicar vetores ponto-a-ponto

```
>>> a = np.array([2, 3, 8])
>>> a * a
[ 4, 9, 64]
```

- Também podemos redimensionar arrays

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> a.shape
(2, 3)
>>> a.reshape(3,2)
[[1 2]
 [3 4]
 [5 6]]
>>> a.reshape(4,4)
ValueError: cannot reshape array of
size 6 into shape (4,4)
```

NumPy (criação)

- Podemos criar arrays passando listas ao construtor *array*

- Ou com valores por defeito

```
>>> a = np.zeros((2, 2))
>>> a
[[0. 0.]
 [0. 0.]]
>>> a.dtype
float64
>>> np.ones((2, 2),'uint16')
[[1 1]
 [1 1]]
```

```
>>> np.empty((2, 2),'uint16')
[[    0     0]
 [    0 16368]]
>>> np.full((2, 2), 3.5)
[[3.5 3.5]
 [3.5 3.5]]
```

NumPy (criação)

- Ou utilizando *arange* como *range* para sequências de inteiros
- Ou utilizando *linspace* para sequências de números não inteiros
  - recebe limites inferior e superior do intervalo (inclusivé) e número de elementos a gerar
  - gera um array de elementos equidistantes

```
>>> np.arange(4)
[0 1 2 3]
>>> np.arange(1,5).reshape(2,2)
[[1 2]
 [3 4]]
```

```
>>> np.linspace(0, math.pi, 5)
[0. 0.78539816 1.57079633
2.35619449 3.14159265]
```

NumPy (indexação)

- Podemos projetar arrays a várias dimensões
- Cuidado: **projeções são vistas, não cópias!**

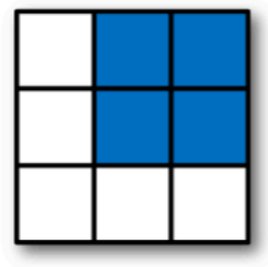
```
>>> a = np.array([2, 3, 8])
>>> a[2]
8
>>> b = np.array([
[2, 3, 8],
[4, 5, 6],
])
>>> c = b[1]
>>> c
array([4, 5, 6])
```

```
>>> b[1][2]
6
>>> b[1, 2]
6
>>> c[1] = 9
>>> b
[[2 3 8]
 [4 9 6]]
```

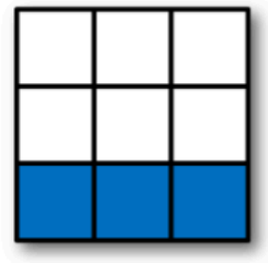
NumPy (slices)

- Podemos utilizar a notação familiar de slices
- Cuidado : **também são vistas!**

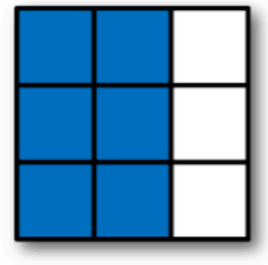
```
>>> vec = np.array([2, 3, 8])
>>> arr = np.array([[2, 3, 8],
                    [4, 5, 6]])
>>> vec[1:3] = 5
>>> vec
[2 5 5]
>>> arr[1, :2]
[4 5]
>>> arr[:2,1]
[3 5]
>>> arr[:, :2]
>[[2 3]
> [4 5]]
```



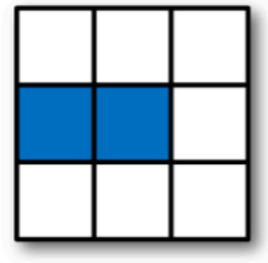
Expression	Shape
arr[:2, 1:]	(2, 2)



arr[2]	(3,)
arr[2, :]	(3,)
arr[2:, :]	(1, 3)



arr[:, :2]	(3, 2)



arr[1, :2]	(2,)
arr[1:2, :2]	(1, 2)

NumPy (máscaras)

- Máscara = vetor de booleanos com o tamanho do array
- Pode ser criada aplicando uma operação ao array
- Pode ser usada como índice

```
>>> a = np.array([1,2,3,4])
>>> mask = (a >= 2) & (a < 4)
>>> mask
[False True True False]
>>> a[mask]
[2 3]
>>> a[mask] = 0
>>> a
[1 0 0 4]
```

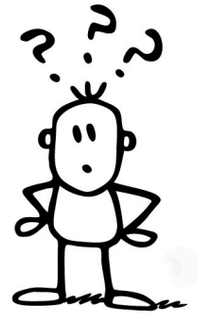
NumPy (broadcasting)

- Já vimos que podemos aplicar operações entre um array e uma constante, ou entre arrays com shapes iguais

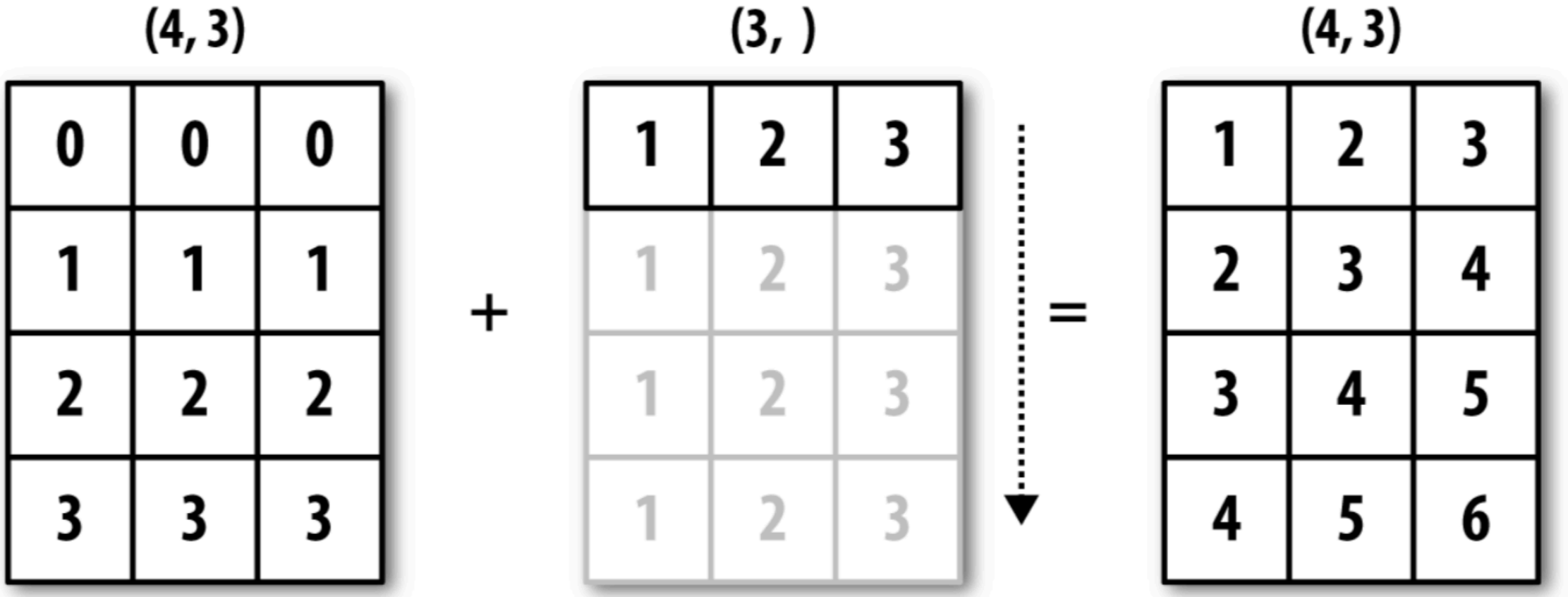
```
>>> a =
np.array([1,2,3])
>>> a > 2
[False False  True]
>>> b =
np.array([[1,2,3],
[4,5,6]])
>>> b * 3
[[ 3  6  9]
 [12 15 18]]
```

```
>>> a =
np.array([[1,2,3],
[4,5,6]])
>>> b =
np.array([[1,1,1],
[2,2,2]])
>>> a * b
[[ 1  2  3]
 [ 8 10 12]]
```

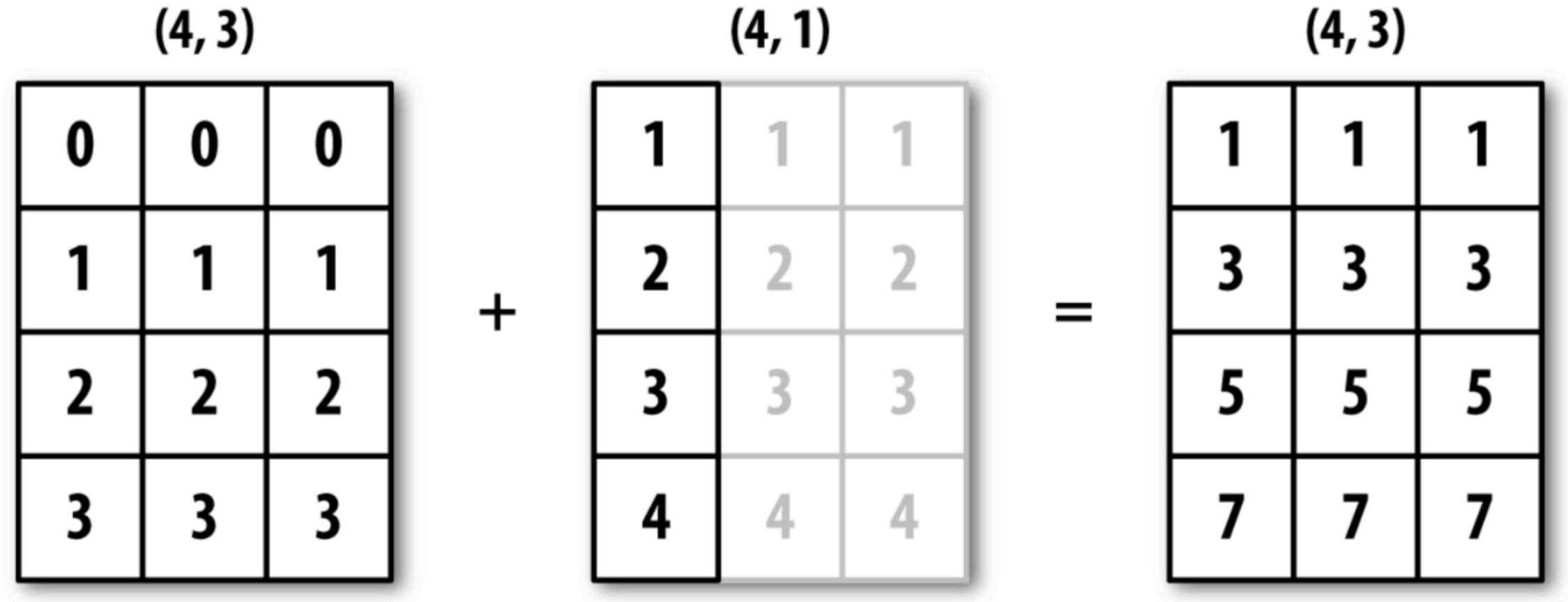
- Generaliza para arrays de shapes diferentes, desde que, expandindo as dimensões em falta, tenham a mesma shape



NumPy (broadcasting)

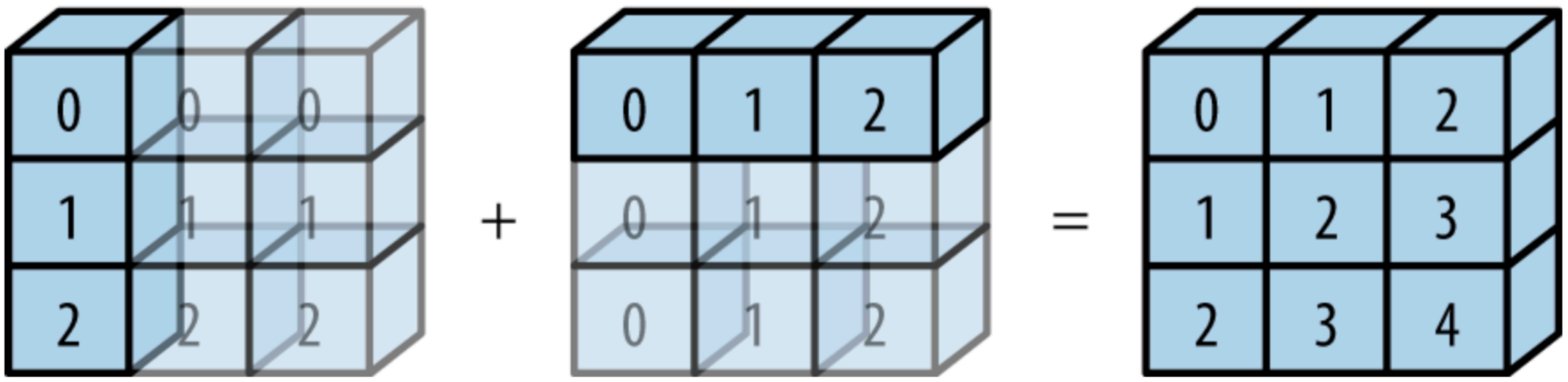


NumPy (broadcasting)



NumPy (broadcasting)

```
np.arange(3)
.reshape(3,1) + np.arange(3)
```



NumPy (funções universais)

- Além de operadores binários (+, -, *, /, etc), o *numpy* também redefine funções matemáticas (*sin*, *cos*, *sqrt*, etc)

- Operam elemento a elemento

- Lista completa

```
>>> a = np.array([1,2,3,4])
>>> np.sqrt(a)
[1. 1.41421356 1.73205081 2. ]
>>> np.exp(a)
[2.71828183 7.3890561 20.08553692 54.59815003]
>>> b = np.array([4,3,2,1])
>>> np.maximum(a,b)
[4 3 3 4]
```

NumPy (vectorize)

- A função a função *vectorize, analogamente* à função *map* para sequências, permite converter uma função genérica numa função universal *numpy* que pode ser aplicada a cada elemento do array

- E.g., para interpretar cada elemento como um índice e substituí-lo pelo valor correspondente numa lista

```
assocs = [10,20,30,40,50,60]
a = np.array([[4,2,1,3] \
              ,[5,0,1,2]])

# substitui cada elemento pelo seu valor na lista
def assoc(i) : return (assocs[i])
npassoc = np.vectorize(assoc)
print(npassoc(a))
```

```
[[50 30 20 40]
 [60 10 20 30]]
```

NumPy (agregação)

- Algumas funções *numpy* agregam os elementos de um array, analogamente a funções de agregação para sequências

```
>>> a = np.array([[1,2,3],[4,5,6]])
# soma
>>> np.sum(a)
21
# mínimo
>>> np.min(a)
1
# máximo
>>> np.max(a)
6
# desvio padrão
>>> np.std(a)
1.707825127659933
```

NumPy (apply)

- Também podemos aplicar funções ao longo de uma dimensão específica de um array multidimensional.
  - Dimensões são índices 0,1,…
  - As funções operam sobre as projeções do array (arrays com menos uma dimensão) ao longo da dimensão escolhida

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

# soma de cada linha na vertical
>>> np.apply_along_axis(np.sum,0,a))
[5 7 9]

# soma de cada linha na horizontal
>>> np.apply_along_axis(np.sum,1,a))
[ 6 15]
```

NumPy (álgebra linear)

- O *numpy* também oferece operações de álgebra linear
- E.g., produto de matrizes

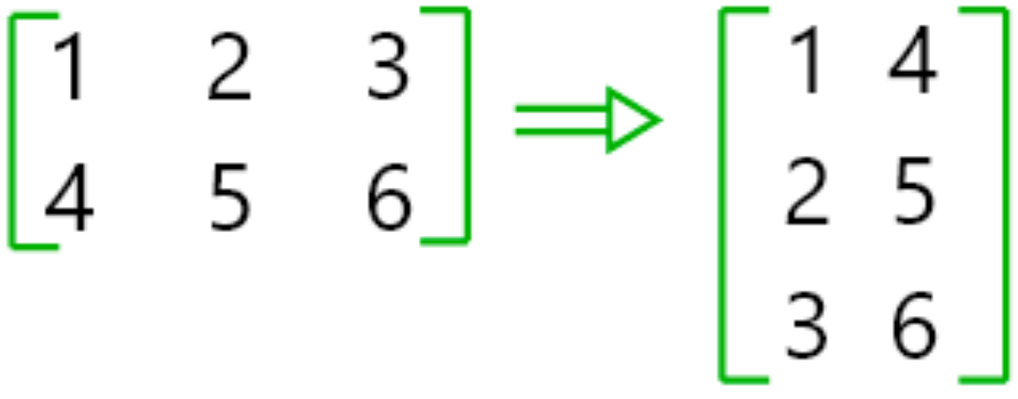
```
>>> A = np.array([[1,2,3,4],[5,6,7,8]])
>>> B = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
>>> A @ B
[[ 70  80  90]
 [158 184 210]]
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} = \begin{pmatrix} 70 & 80 & 90 \\ 158 & 184 & 210 \end{pmatrix}$$

NumPy (álgebra linear)

- E.g., transposta de matrizes

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



Exemplo NumPy (CSV)

- Podemos ler facilmente um ficheiro CSV para uma matriz *numpy*
- E.g., índice de secura mensal para o Porto publicado pelo IPMA aqui
    - Utilizado na última aula
- Contar os dias de chuva (a julgar pelo valor médio) e listar esses mesmos dias

```
data = np.genfromtxt('mpdsi-1312-porto.csv',delimiter=',',skip_header=1)
# coluna 4 = médias de índice de secura
medias = data[:,4]
# dias chuvosos têm índice >= 1
chuvosos = medias >= 1
num_dias_chuvosos = (medias[chuvosos]).size
print(num_dias_chuvosos)

dates = np.genfromtxt('mpdsi-1312-
porto.csv',delimiter=',',dtype='datetime64',usecols=0,skip_header=1)
dias_chuvosos = dates[chuvosos]
print(dias_chuvosos)
```

Exemplo NumPy (Excel)

- Podemos ler facilmente uma folha de um ficheiro Excel para uma matriz *numpy* (via *pandas*)
- E.g., precipitação durante os últimos ~90 anos publicado pelo IPMA aqui e aqui
- Anos mais e menos chuvoso de que há registo
- Médias de precipitação nos Séculos XX e XXI

```
import pandas as pd

dados = pd.read_excel('PT100-tx-tn-prec.xlsx',sheet_name=3).to_numpy()
anual = dados[:89,[0,17]]

min_prec = anual[:,1].min()
max_prec = anual[:,1].max()

xx = anual[anual[:,0] < 2000]
xx_prec = np.mean(xx[:,1])

xxi = anual[anual[:,0] >= 2000]
xxi_prec = np.mean(xxi[:,1])
```