Programação I / Introdução à Programação

Capítulo 6, "NumPy"

João Pedro Pedroso

2024/2025

Aulas passadas:

- programação orientada a objetos
- exceções
  - *"tratar"* um erro de execução com `try ... except ...`
  - *"lançar"* erros com `raise`

Hoje:

- Computação numérica com NumPy

Python e operações matemáticas

- Podemos utilizar listas do Python para representar vetores e matrizes
- No entanto, as operações habituais em matemática vetorial não estão disponíveis
- Exemplo: multiplicar vetor por constante

```
>>> a = [2, 3, 8]
>>> 2.1 * a
TypeError: can't multiply sequence by non-int of type 'float'
```

Python e operações matemáticas

- Podemos utilizar listas do Python para representar vetores e matrizes
- No entanto, as operações habituais em matemática vetorial não estão disponíveis
- Exemplo: multiplicar vetor por constante

```
>>> a = [2, 3, 8]
>>> 2.1 * a
TypeError: can't multiply sequence by non-int of type 'float'
```

  - Para obtermos o resultado pretendido, temos de fazer a multiplicação elemento a elemento:

```
values = [2, 3, 8]
result = []
for x in values:
    result.append(2.1 * x)
```

  - Com o módulo NumPy podemos fazer:

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

Notas

- É frequente abreviar `numpy` para `np`

```
>>> import numpy as np
```

- `np.array` tem como argumento uma lista

```
>>> a = np.array([2, 3, 8])
>>> a
array([2, 3, 8])
```

- o vetor (*array*) a tem elementos inteiros, mas o resultado da multiplicação por um `float` tem elementos `float`

```
>>> 2.1 * a
array([ 4.2, 6.3, 16.8])
```

- a operação * com *arrays* produz a multiplicação elemento a elemento (não é o produto vetorial/matricial):

```
>>> a * a
array([ 4, 9, 64])
```

Dimensões em *arrays*: `shape`

- O atributo `shape` contém um tuplo com o número de elementos em cada dimensão:

```
>>> import numpy as np
>>> a = np.array([2, 3, 8])
>>> a.shape
(3,)
```

- Podemos ter vetores, matrizes, ou *arrays* de dimensões superiores:

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

- Em `NumPy`, dimensões são chamadas *eixos* / *axes*

Arrays: shape

```
>>> x = np.array(["A","E","I","O","U"])
>>> x
array(['A', 'E', 'I', 'O', 'U'], dtype='<U1')
>>> x.shape
(5,)
>>> x = np.array([["A","E","I","O","U"]])
>>> x
array([['A', 'E', 'I', 'O', 'U']], dtype='<U1')
>>> x.shape
(1, 5)
>>> x = np.array([["A"],["E"],["I"],["O"],["U"]])
>>> x
array([['A'],
       ['E'],
       ['I'],
       ['O'],
       ['U']], dtype='<U1')
>>> x.shape
(5, 1)
>>> x = np.array([["A","E","I","O","U"],["1","2","3","4","5"]])
>>> x
array([['A', 'E', 'I', 'O', 'U'],
       ['1', '2', '3', '4', '5']], dtype='<U1')
>>> x.shape
(2, 5)
```

Criar *arrays* com dadas dimensões

- Com zeros

```
>>> np.zeros((2, 2))
array([[0., 0.],
       [0., 0.]])
```

- Com uns

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

- Sem inicializar o conteúdo

```
>>> np.empty((2, 3), 'uint16')
array([[    0,     0,     0],
       [40960, 61235, 21401]], dtype=uint16)
```

- Inicializar com valor dado

```
>>> np.full((2, 2), 17)
array([[17, 17],
       [17, 17]])
```

Fatias (*slices*) em diferentes dimensões

- 1D:

```
>>> a = np.array([2, 3, 8])
>>> a[2]
8
>>> a[1:]
np.array([3, 8])
```

- 2D:

```
>>> b = np.array([
    [2, 3, 8],
    [4, 5, 6],
    ])
>>> b[1]
array([4, 5, 6])
>>> b[1][2]
6
>>> b[1, 2]    # abreviatura da notação anterior:
6
>>> b[:, 1]    # seleção do índice 1 na segunda dimensão (uma coluna)
array([3, 5])
```

Notas

Em NumPy, as fatias são uma vista de *arrays*

- não são cópias
- referem-se sempre a dados numa outra matriz

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

>>> y = x[::2]
>>> y
array([0, 2, 4])

>>> x[0] = 3 # changing x changes y as well, since y is a view on x
>>> y
array([3, 2, 4])
```

Máscaras

- Permitem selecionar facilmente partes de *arrays*
- Exemplo: todos os elementos acima de `cutoff`

```
>>> a = np.array([230, 10, 284, 39, 76])
>>> cutoff = 200
>>> a > cutoff
np.array([True, False, True, False, False])
```

- Podemos agora alterar para zero todos índices onde teste era positivo:

```
>>> a = np.array([230, 10, 284, 39, 76])
>>> cutoff = 200
>>> a[a > cutoff] = 0
>>> a
np.array([0, 10, 0, 39, 76])
```

- Como fazer a mesma coisa com ciclos?

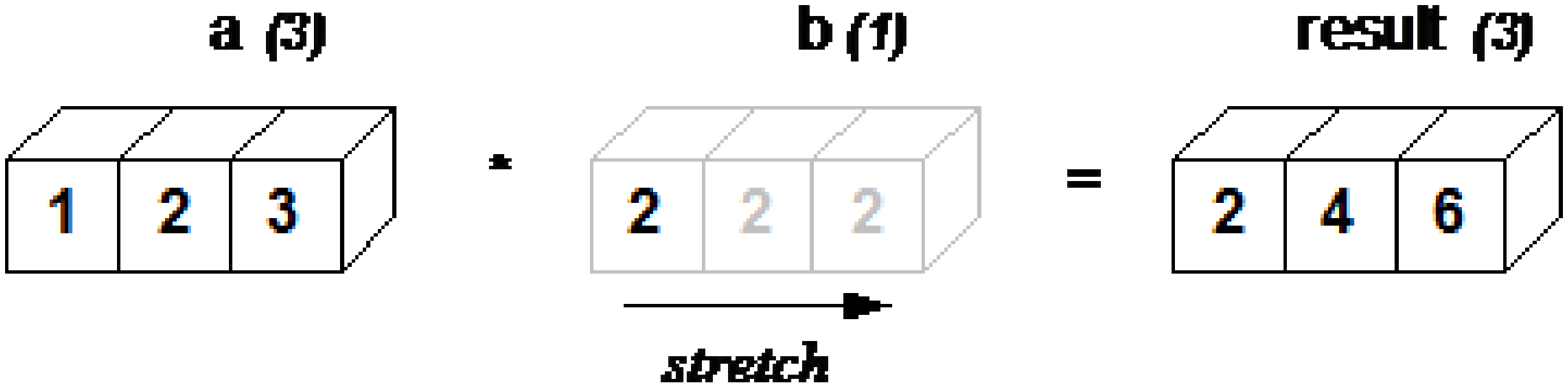
Broadcasting

- Broadcasting acontece quando se faz operações entre *arrays* de dimensões diferentes

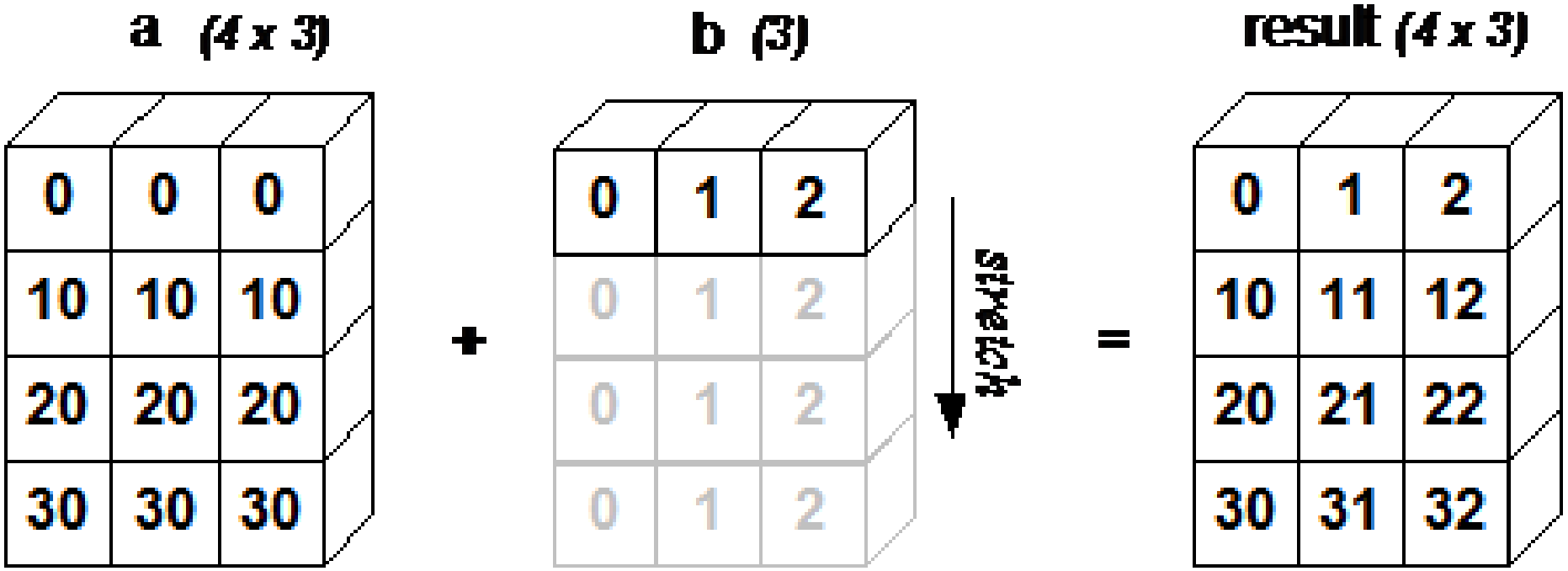
```
>>> a = np.array([
    [0, 1],
    [2, 3],
    [4, 5],
    ])
>>> b = np.array([10, 100])
>>> a * b
array([[ 0, 100],
       [ 20, 300],
       [ 40, 500]])
>>> a + b
array([[ 10, 101],
       [ 12, 103],
       [ 14, 105]])
```

- Neste exemplo: `b` é expandido numa segunda dimensão, repetindo os seus elementos
- Regras:
  - só podem ser expandidas dimensões de tamanho 1
    - `b` é 1D, *shape* $\rightarrow$ (2,)
    - `a` é 2D
    - *broadcasting*: é adicionada a `b` uma dimensão $\rightarrow$ passa a ser (1,2)
    - nova dimensão tem tamanho 1
    - é expandida para ficar (3,2) $\rightarrow$ fica igual a `a`
  - dimensões são comparadas da última para a primeira
    - quando uma não é igual, é expandida
    - mas apenas dimenões de tamanho 1 podem ser expandidas...

Broadcasting



Broadcasting



Broadcasting

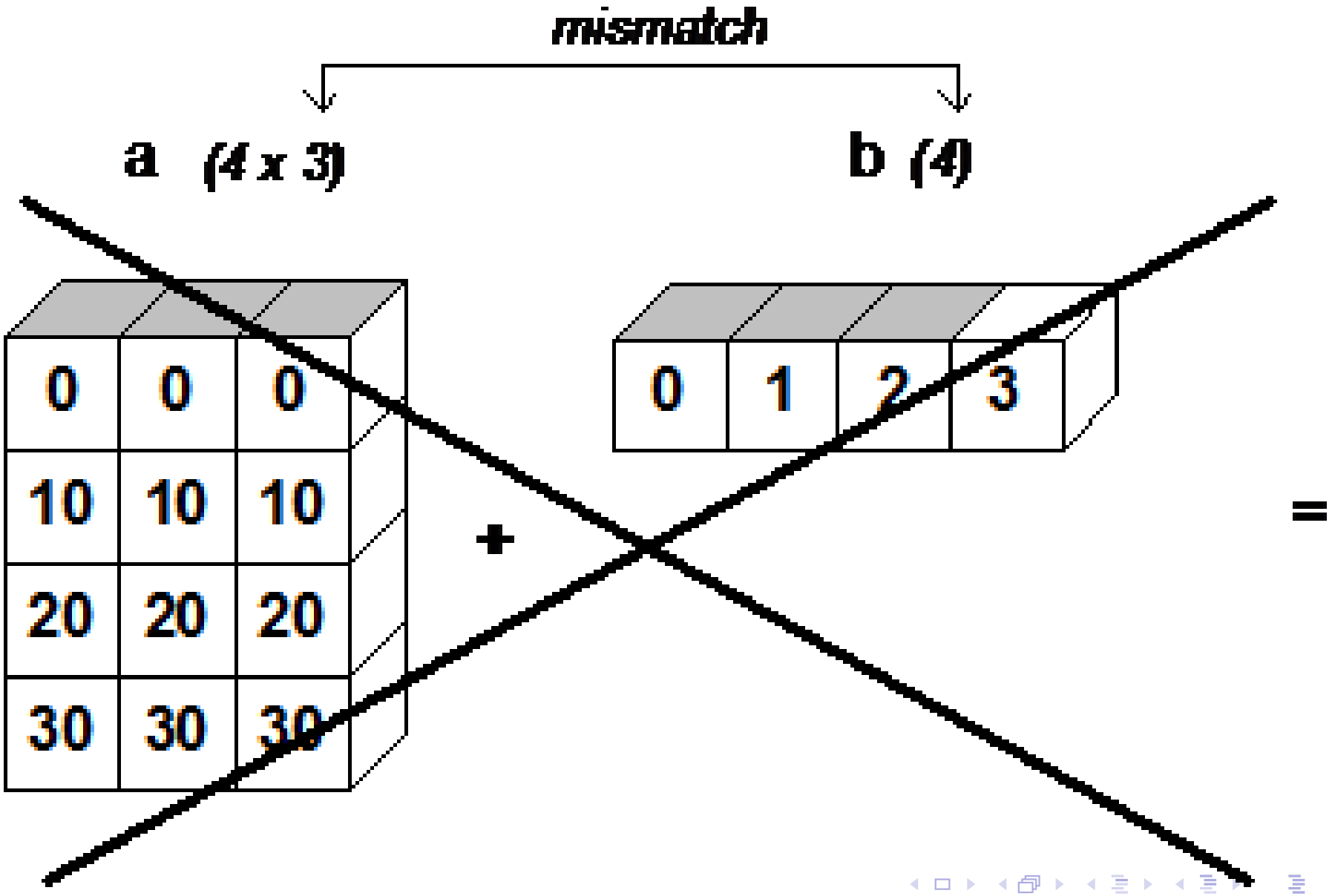
- dimensões incompatíveis → erro

```
>>> b = np.array([10, 100])
>>> c = np.array([
    [0,1,2],
    [3,4,5]
    ])
>>> c * b
ValueError: operands could not be
broadcast together with shapes
(2,3) (2,)
```

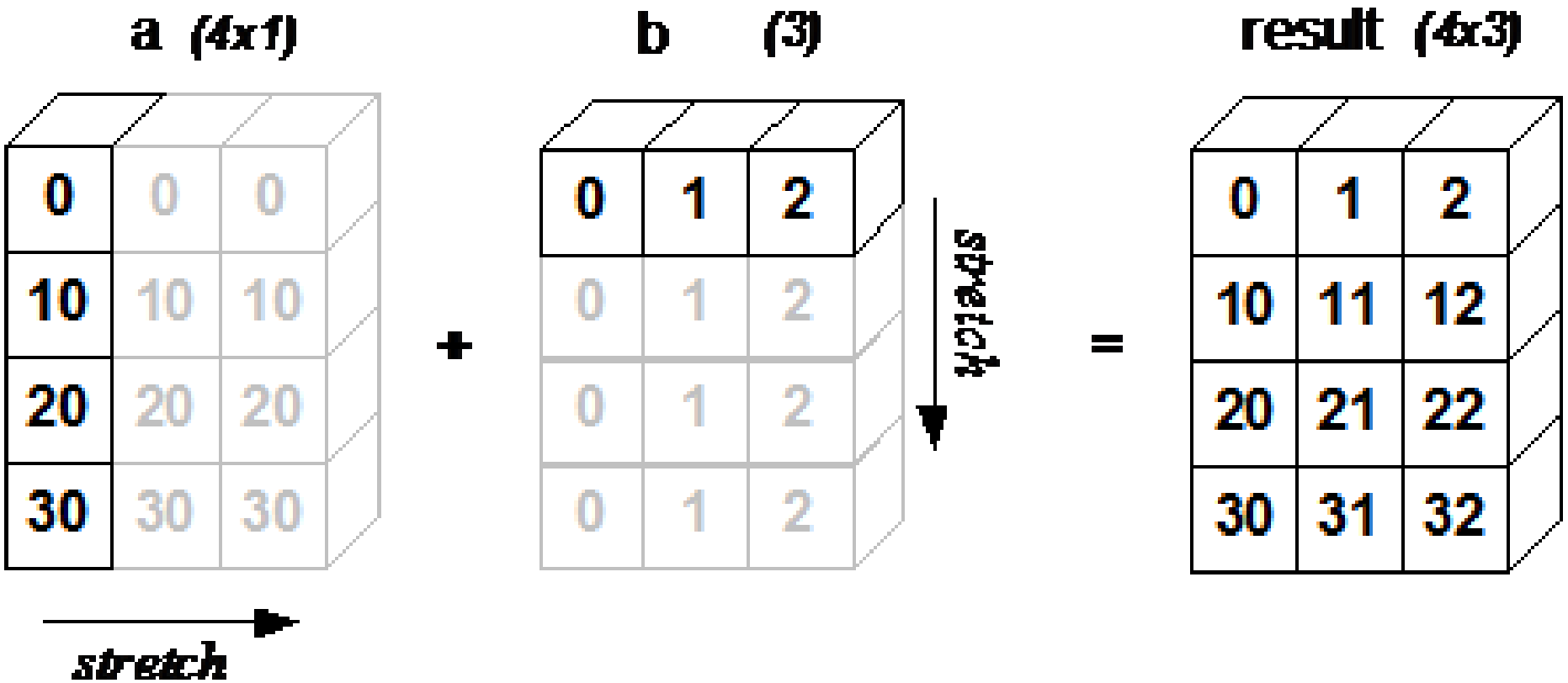
- podemos dar mais uma dimensão a b:

```
>>> b[:, None]  # vetor coluna
array([[ 10],
       [100]])
>>> b.reshape(2,1)  # vetor coluna
array([[ 10],
       [100]])
>>> c * b.reshape(2,1)
array([[  0,  10,  20],
       [300, 400, 500]])
```

Broadcasting



Broadcasting



Tipos dos elementos: dtype

- O tipo de cada elemento pode ser, e.g., inteiro ou float
  - `int` → inteiro com 64 bit
  - `uint8` → inteiro positivo com 8 bits
  - <> → indicam bits mais significativos estão à esquerda ou à direita
- Pode ser dado explicitamente:

```
>>> a = np.array([10, 100])
>>> a
array([ 10, 100])
>>> b = np.array([10, 100], dtype='float')
>>> b
array([ 10., 100.])
```

- Pode ganhar velocidade e espaço, perdendo precisão:

```
>>> s = np.array([10, 100, 1000], dtype='uint8')
>>> s
array([ 10, 100, 232], dtype=uint8)
```

Arrays com gamas de valores igualmente separados

- Em Python: `range()`
  - `range(n)` → [0, 1, ... n-1]
  - `range(m,n)` → [m, m+1, ..., n-1]
  - `range(m,n,k)` → `r[i] = m + k*i`, onde `i >= 0` e
    - `r[i] < n` para `k>0`
    - `r[i] > n` para `k<0`
- Em NumPy, há várias alternativas:
  - `arange`
    - análogo a `range`, mas retorna *array*
    - argumentos podem ser `float`
    - problema: precisão finita pode tornar difícil saber *quantos elementos são gerados*
  - `linspace` → argumentos:
    - limites inferior e superior do intervalo (inclusive)
    - número de elementos a gerar

arange

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )  # accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
>>> np.arange(1.,-1e-20,-1/3)  # -> problems with finite precision
array([1.        , 0.66666667, 0.33333333]) # should include 0
```

linspace

```
>>> np.linspace(0, 2, 9)  # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> np.linspace(1.,0,4)  # -> solves problems with finite precision
array([1.        , 0.66666667, 0.33333333, 0.        ])
>>> x = np.linspace(0, 2*pi, 100)  # useful to evaluate function at
>>> f = np.sin(x)                  # lots of points (eg, for plotting)
```

Funções "universais" (*Universal Functions*)

- NumPy redefine funções matemáticas, como `sin cos exp ...`
  - em NumPy: "universal functions" (ufunc)
- Operam elemento a elemento, em *arrays*
- Produzem *arrays*

```
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([ 1.        ,  2.71828183,  7.3890561 ])
>>> np.sqrt(B)
array([ 0.        ,  1.        ,  1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)  # (o mesmo que B + C)
array([ 2.,  0.,  6.])
```

Arrays com valores aleatórios

- NumPy pode gerar valores aleatórios diretamente em *arrays*:

```
>>> np.random.randint(10, 20, size=10)
array([13, 15, 13, 15, 17, 19, 17, 15, 17, 11])
>>> np.random.randint(low=10, high=20, size=(2, 10))
array([[19, 10, 10, 15, 12, 17, 19, 17, 17, 17],
       [18, 17, 16, 11, 18, 11, 16, 16, 11, 19]])
```

- Outras funções:
  - `np.random.random(size=None)` → distribuição uniforme em [0,1[
  - `np.random.normal(mu=0,stddev=1,size=None)` → distribuição gaussiana
  - `np.random.choice(a,size=None,replace=True)` → amostragem do *array* `a`
    - `replace=True` → com reposição
    - `replace=False` → sem reposição

Indexação: *arrays* unidimensionais

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -1000    # set every 2nd element to -1000
>>> a
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,
         729])
>>> for i in a:
...     print(i**(1/3.))
...
__main__:2: RuntimeWarning: invalid value encountered in power
nan
1.0
nan
3.0
nan
4.999999999999999
5.999999999999999
6.999999999999999
7.999999999999999
```

Indexação: *arrays* multidimensionais

```
>>> def f(i,j):
...     return 10*i+j
...
>>> b = np.fromfunction(f,(2,3),dtype=int) # build array using f
>>> b
array([[ 0,  1,  2],
       [10, 11, 12]])
>>> for i in range(len(b)):
...     for j in range(len(bi)):
...         b[i,j] += 100
...
>>> b
array([[100, 101, 102],
       [110, 111, 112]])
```

"Vistas" (*views*) e cópias

- "Fatias" retornam "vistas":

```
>>> a = np.arange(12)
>>> a.shape = 3,4
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> s = a[ : , 1:3]
>>> s[:] = 10              # s[:] is a view of s. Note the difference between s=1
>>> a
array([[ 0, 10, 10,  3],
       [ 4, 10, 10,  7],
       [ 8, 10, 10, 11]])
```

- Método copy → cópia completa dos dados

```
>>> d = a.copy()          # a new array object with new data is created
>>> d[0,0] = 9999
>>> a
array([[ 0, 10, 10,  3],
       [ 4, 10, 10,  7],
       [ 8, 10, 10, 11]])
```

Álgebra linear

Produto matricial

```
>>> A = np.array([[1,0],
...               [0,1]])
>>> B = np.array([[2,7],
...               [3,4]])
>>> A * B  # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B  # matrix product
array([[2, 7],
       [3, 4]])
>>> A.dot(B)  # matrix product (another way)
array([[2, 7],
       [3, 4]])
```

Alguns métodos para operações com matrizes

- Matriz transposta, matriz inversa, traço

```
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> a
[[ 1.  2.]
 [ 3.  4.]]

>>> a.transpose()
array([[ 1.,  3.],
       [ 2.,  4.]])

>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

>>> u = np.eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[ 1.,  0.],
       [ 0.,  1.]])

>>> np.trace(u)  # trace
2.0
```

Resolução de sistemas de equações

- Solução de $Ax = b$

```
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> a
array([[1., 2.],
       [3., 4.]])

>>> b = np.array([[5.], [7.]])

>>> np.linalg.solve(a, b)
array([[-3.],
       [ 4.]])
```

Funções agregadoras

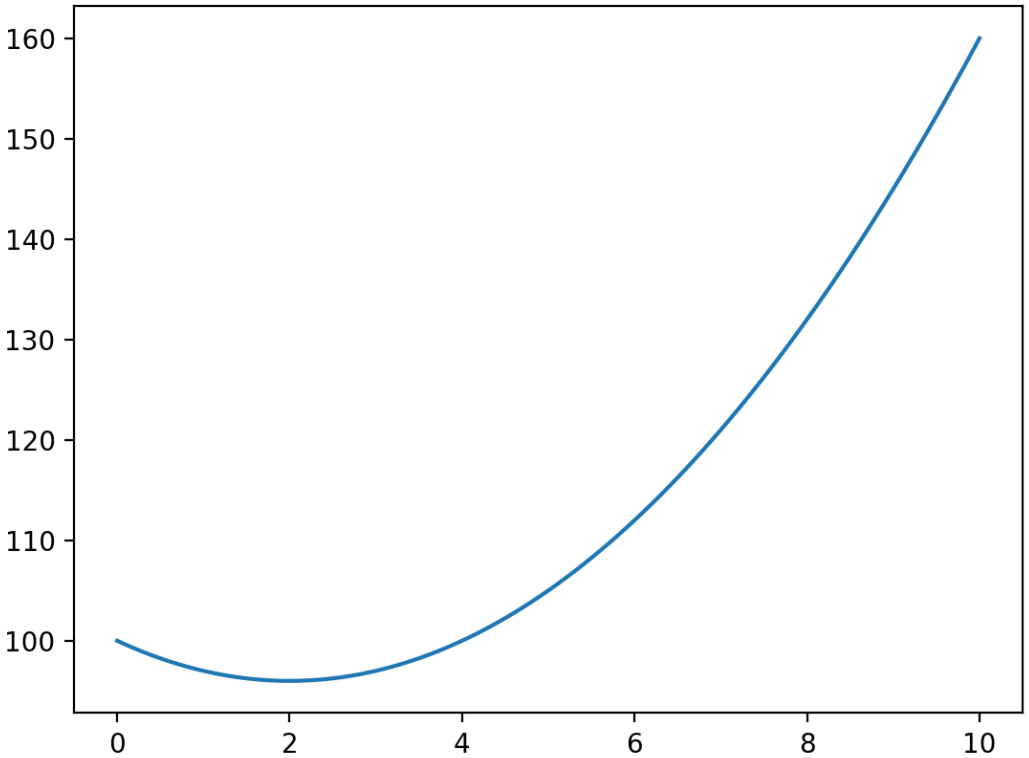
- Algumas funções do NumPy agregam e calculam valores em matrizes multidimensionais

```
>>> x =np.random.randint(low=1, high=100, size=10)
>>> x
array([92, 73, 76, 27, 35, 61, 23, 46, 73, 23])
>>> np.min(x)
23
>>> np.argmin(x)
6
>>> np.sum(x)
529
>>> np.std(x)
24.006040906405207
>>> A =np.random.randint(low=1,high=100, size=(3,5))
>>> A
array([[14, 96, 96,  3, 11],
       [ 4, 62, 21, 89, 11],
       [40, 99, 79, 53, 51]])
>>> np.sum(A, axis=0)
array([ 58, 257, 196, 145,  73])
>>> np.sum(A, axis=1)
array([220, 187, 322])
```

Gráficos: matplotlib

Gráficos: matplotlib

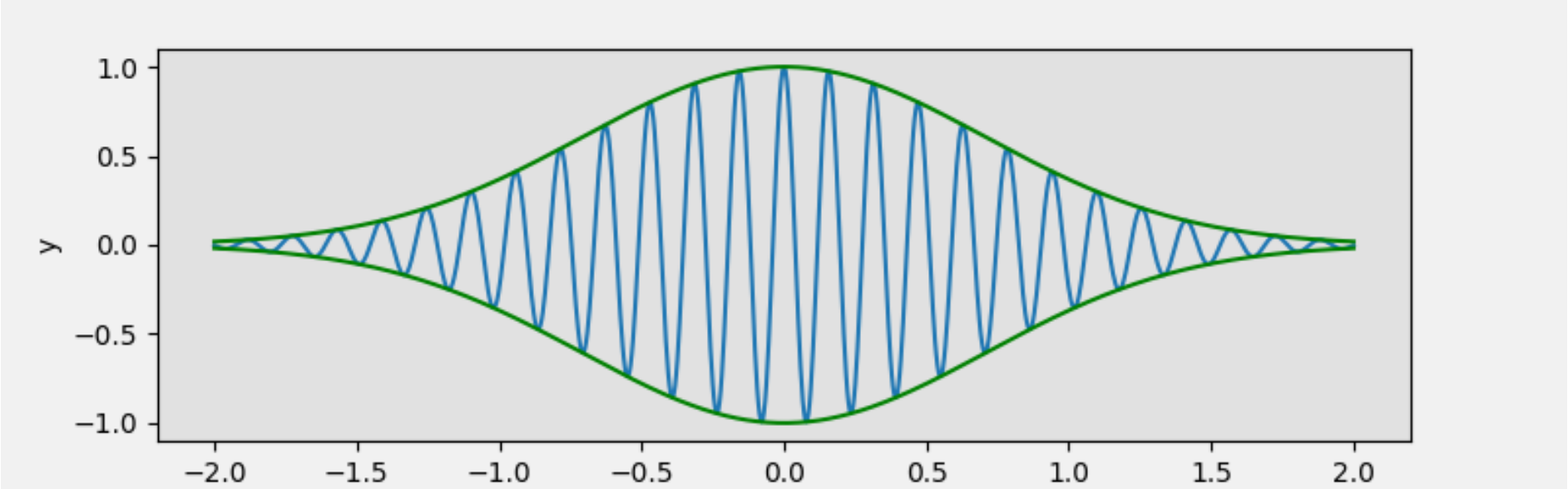
```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0,10,101)  # 101 values, equally spaced in [0,10]
>>> y = 100 - 4*x + x**2  # evaluate y on the function to plot (based on ufunc's)
>>> plt.plot(x,y)
>>> plt.show()
```



Figuras

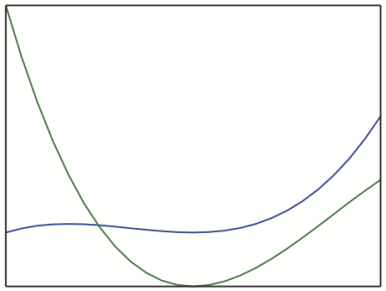
- Usadas para representar gráficos
- Criadas com `plt.figure`
  - add_axes → método para criar novo sistema de eixos

```
fig = plt.figure(figsize=(8, 2.5), facecolor="#f1f1f1")
left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
ax = fig.add_axes((left, bottom, width, height), facecolor="#e1e1e1")
x = np.linspace(-2, 2, 1000)
y1 = np.cos(40 * x)
y2 = np.exp(-x**2)
ax.plot(x, y1 * y2)
ax.plot(x, y2, 'green')
ax.plot(x, -y2, 'green')
ax.set_xlabel("x")
ax.set_ylabel("y")
fig.savefig("graph.png", dpi=100, facecolor="#f1f1f1")
```

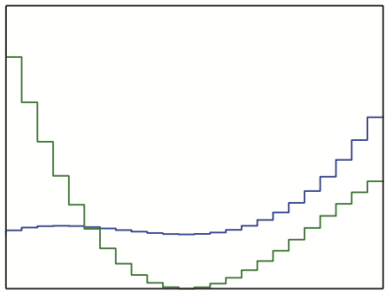

1.0
0.5
0.0
−0.5
−1.0
y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

Tipos de gráficos

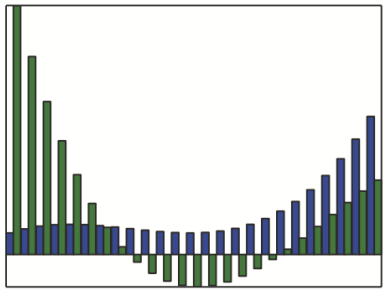
`Axes.plot`



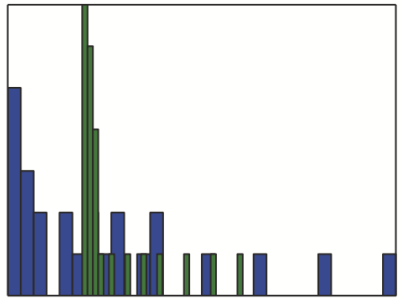
`Axes.step`



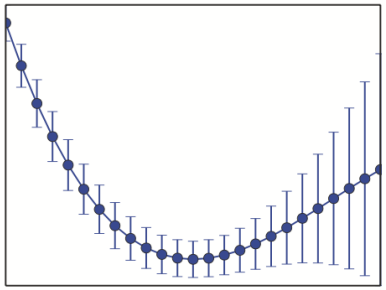
`Axes.bar`



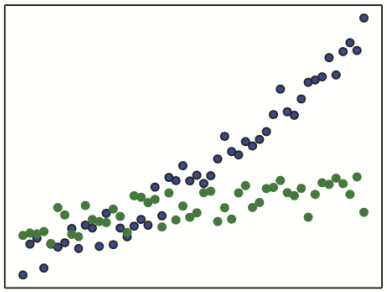
`Axes.hist`



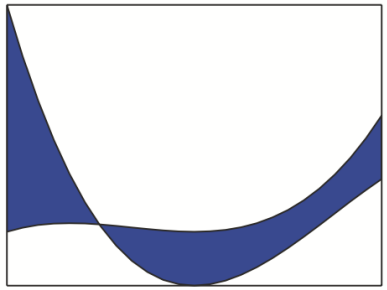
`Axes.errorbar`



`Axes.scatter`



`Axes.fill_between`



`Axes.quiver`

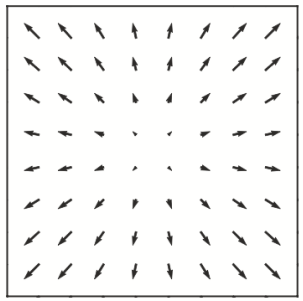


Ilustração: *data set* `iris`

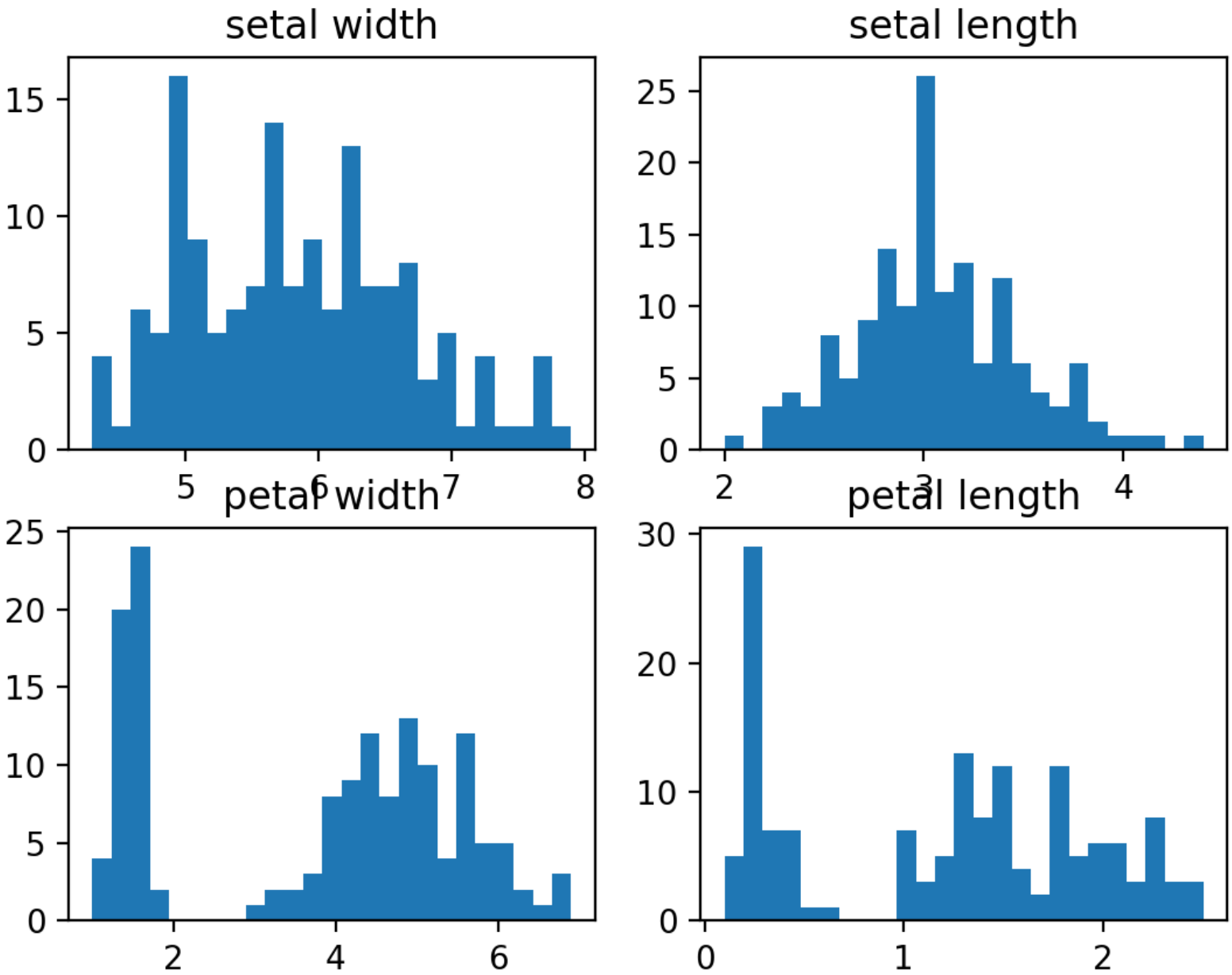
- Iris flower data set:

  `https://en.wikipedia.org/wiki/Iris_flower_data_set`
- Muito usado em *machine learning*
- Classificação de flores com base em:
  - comprimneto e largura da sépala
  - comprimento e largura da pétala
- Data set: por esta ordem:
  - *sepal length, sepal width, petal length, petal width*

```
>>> from sklearn.datasets import load_iris
>>> iris=load_iris()
>>> data = iris['data']
>>> data
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [...]
>>> setal_w = data[:,0]
>>> setal_l = data[:,1]
>>> petal_w = data[:,2]
>>> petal_l = data[:,3]
```

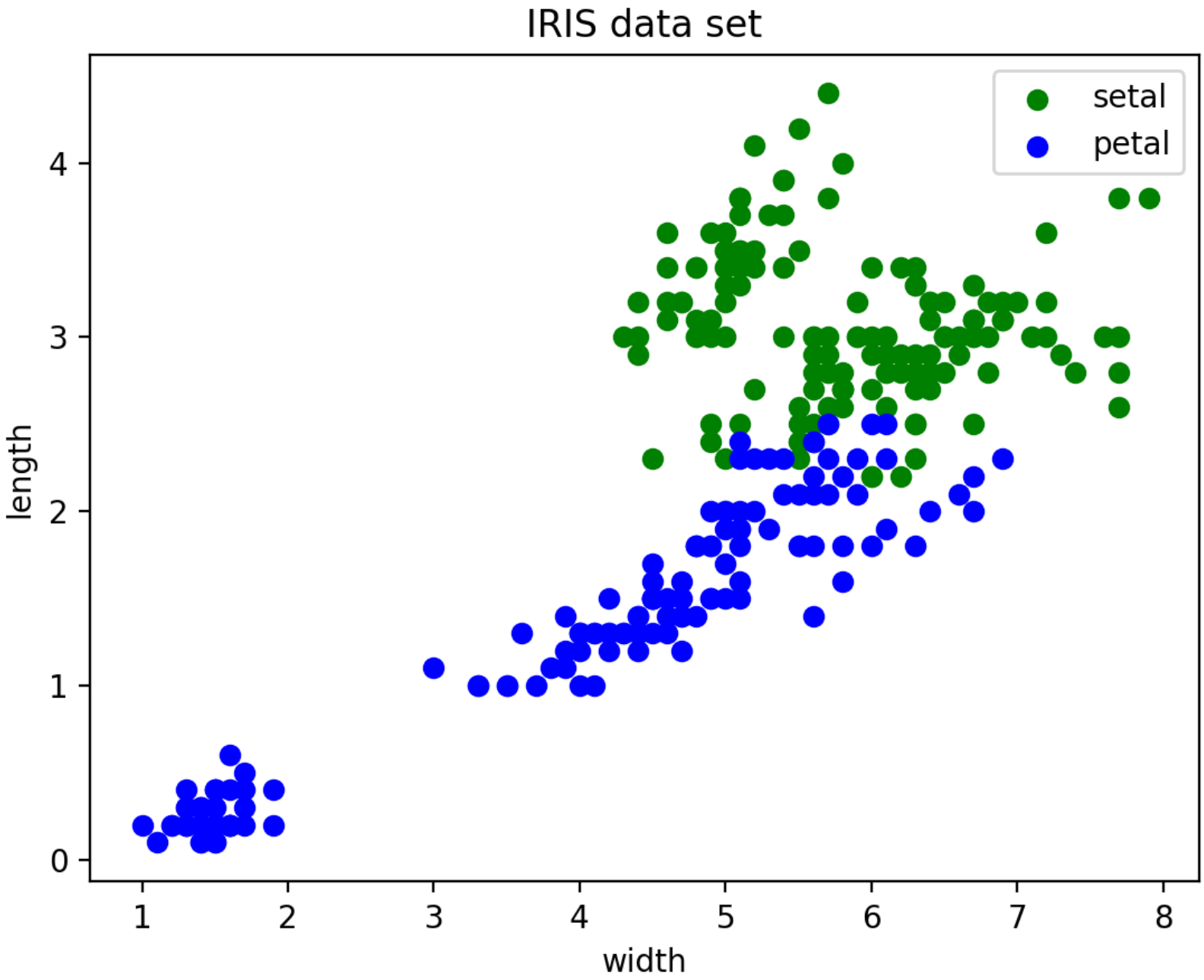
Histogramas

```
fig, axes = plt.subplots(nrows=2, ncols=2)
axes[0,0].hist(setal_w, bins=25)
axes[0,1].hist(setal_l, bins=25)
axes[1,0].hist(petal_w, bins=25)
axes[1,1].hist(petal_l, bins=25)
axes[0,0].set_title("setal width")
axes[0,1].set_title("setal length")
axes[1,0].set_title("petal width")
axes[1,1].set_title("petal length")
```


setal width
setal length
petal width
petal length

Scatter plots

```
fig, axes = plt.subplots(nrows=1, ncols=1)
axes.set_xlabel("width")  # title for x axis
axes.set_ylabel("length")
axes.scatter(setal_w, setal_l, color='green', label='setal')
axes.scatter(petal_w, petal_l, color='blue', label='petal')
axes.set_title("IRIS data set")  # title for the graphic
axes.legend()  # show what is being plotted
fig.show()
```


IRIS data set
setal
petal
length
width
0
1
2
3
4
1
2
3
4
5
6
7
8

Mais informação

- Documentação: `https://matplotlib.org`
- Exemplos: `https://matplotlib.org/gallery.html`

Aula de hoje:

- Computação científica: NumPy
  - mais informação: `https:/numpy.org`

Próximas aulas

- Biblioteca standard do Python, módulo `sympy`