😎 Arrays con Numpy

Numpy Significa 'Python numérico'. Es una biblioteca que proporciona matrices de distintas dimensiones y una variedad de rutinas de operaciones.
Los conjuntos de datos pueden provenir de una amplia gama de fuentes y formatos, incluidas colecciones de documentos, de imágenes, de clips de sonido, de medidas numéricas, etc.

In [1]:
```
import numpy as np
```

Creación de arrays

array: forma simple de crear un array a partir de una una lista o tupla:

In [2]:
```
lista1 = [1, 2, 3, 4, 5]
lista2 = np.array([1, 2, 3, 4, 5])
tupla = (1, 2, 3, 4, 5)
print(f"Array desde una lista: {np.array(lista1)}\nArray desde otro array: {np.array(lista2)}\n\
Array desde una tupla: {np.array(tupla)}")
```

```
Array desde una lista: [1 2 3 4 5]
Array desde otro array: [1 2 3 4 5]
Array desde una tupla: [1 2 3 4 5]
```

In [3]:
```
x = np.array([1,2,3,4,5], dtype = np.int8)
y = np.array([6,7,8,9,0], dtype = np.float32)
x, y
```

Out[3]:
```
(array([1, 2, 3, 4, 5], dtype=int8),
 array([6., 7., 8., 9., 0.], dtype=float32))
```

*Un array de numpy se puede convertir a lista de Python utilizando **tolist()***

In [4]:
```
print(f"Array a lista:\n{x.tolist()}")
```

```
Array a lista:
[1, 2, 3, 4, 5]
```

linspace genera un array formado por n números equiespaciados entre 2 números dados:

In [5]:
```
m = np.linspace(10, 40, 4)
m
```

Out[5]:
```
array([10., 20., 30., 40.])
```

logspace genera un array formado también por n números entre 2 dados, pero en una escala logarítmica. La base a aplicar (por defecto 10) puede especificarse en el argumento base:

In [6]:
```
m = np.logspace(2, 3, 10)
n = np.logspace(2.0, 3.0, num=5, base = 11)
m,n
```

Out[6]:
```
(array([ 100.        ,  129.1549665 ,  166.81005372,  215.443469  ,
         278.25594022,  359.38136638,  464.15888336,  599.48425032,
         774.26368268, 1000.        ]),
 array([ 121.        ,  220.36039471,  401.31159963,  730.8527479 ,
        1331.        ]))
```

arange genera un conjunto de números entre un valor de inicio y uno final, pudiendo especificar un incremento entre los valores:

In [7]:
```
np.arange(24), np.arange(-3, 3, 2, dtype=int)
```

Out[7]:
```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23]),
 array([-3, -1,  1]))
```

randint y **rand** permiten crear elementos aleatorios enteros o flotantes respectivamente, **randn** crea elementos con una disribución normal.

In [8]:
```
v_int = np.random.randint(low = 0, high = 10, size = 5)
v_float = np.random.rand(5)
v_norm = np.random.randn(5)
v_int, v_float, v_norm
```

Out[8]:
```
(array([3, 0, 4, 5, 2]),
 array([0.6577234 , 0.72475865, 0.23252524, 0.67371142, 0.43273407]),
 array([-1.94135785, -0.55530948, -0.36965427, -0.51914419, -0.23018045]))
```

ndim: devuelve el número de dimensiones de la matriz o los ejes. **shape**: devuelve una tupla que consta de dimensiones de matriz.

In [9]:
```
v_float, v_float.ndim, v_float.shape
```

Out[9]:
```
(array([0.6577234 , 0.72475865, 0.23252524, 0.67371142, 0.43273407]), 1, (5,))
```

shape también se puede utilizar para cambiar la dimensión de la matriz:

In [10]:
```
v_m =  np.array([1,2,3,4,5,6])
v_m.shape = (3,2)
v_m
```

Out[10]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

reshape: función para cambiar el tamaño de una matriz:

In [11]:
```
v = np.arange(24)
w = v.reshape(2,4,3)
w.shape
```

Out[11]:
```
(2, 4, 3)
```

Podemos iterar sobre el array de numpy:

In [12]:
```
v=np.random.randint(low = 10, high = 100, size = 5)
for elemento in v:
    print(elemento, end=' ')
```

```
85 95 41 55 66
```

dtype devuelve el tipo de datos. **itemsize** devuelve la longitud en bytes de cada elemento de la matriz. **size**: obtenemos la cantidad de elementos del array.

In [13]:
```
v.dtype, v.itemsize, v.size
```

Out[13]:
```
(dtype('int32'), 4, 5)
```

copy crea una copia del array:

In [14]:
```
copia=np.copy(v)
print("Copia del array 'v':\n", copia)
```

```
Copia del array 'v':
 [85 95 41 55 66]
```

full(): nos permite crear un array de cierto tamaño y tipo completarlo con un valor concreto. **zeros** y **ones** generan arrays de ceros o unos respectivamente

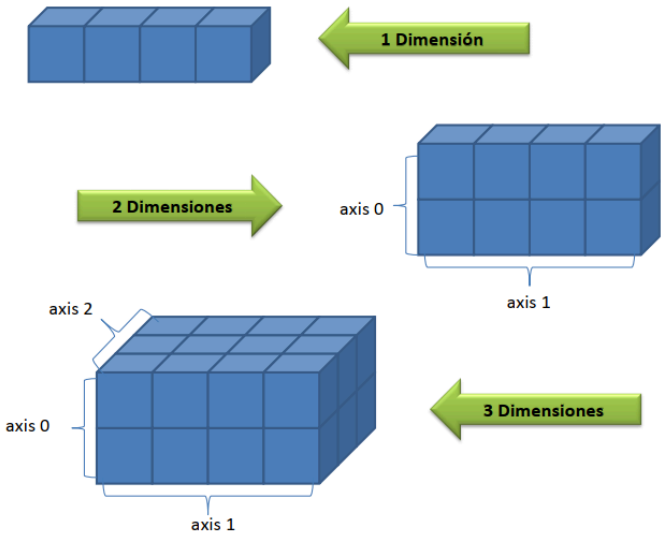
In [15]:
```
x = np.full((10,), fill_value = '-2')
y = np.zeros(5)
z = np.ones(7)
x,y,z
```

Out[15]:
```
(array(['-2', '-2', '-2', '-2', '-2', '-2', '-2', '-2', '-2', '-2'],
       dtype='<U2'),
 array([0., 0., 0., 0., 0.]),
 array([1., 1., 1., 1., 1., 1., 1.]))
```

Ejercicios:

1. Crear un arreglo de 10 cincos.
2. Generar un número aleatorio entre 0 y 1
3. Crear un arreglo con todos los números pares del 10 al 50
4. Generar un arreglo de 25 números aleatorioas con una distribución normal.
5. Convertir el array anterior en una matriz de 5 x 5.

Operaciones básicas



Arrays unidimensionales

Creamos una lista la convertimos en **array** de numpy y con **insert** insertamos un elemento en una posición en particular":

In [16]:
```
v=np.array([25,12,15,66,12.5])
v=np.insert(v, 1, 90)
v
```

Out[16]:
```
array([25. , 90. , 12. , 15. , 66. , 12.5])
```

con **append** agregamos algunos elementos más:

In [17]:
```
v = np.append (v, [10, 11, 12])
v
```

Out[17]:
```
array([25. , 90. , 12. , 15. , 66. , 12.5, 10. , 11. , 12. ])
```

In [18]:
```
v1=np.random.randint(low = 0, high = 10, size = 5)
v2=np.random.randint(low = 11, high = 20, size = 5)
new_v = np.append(v1, v2)
v1, v2, new_v
```

Out[18]:
```
(array([1, 9, 4, 5, 3]),
 array([12, 18, 13, 16, 17]),
 array([ 1,  9,  4,  5,  3, 12, 18, 13, 16, 17]))
```

concatenate: unión de dos arrays en NumPy:

In [19]:
```
x = np.array([1, 2, 3])
y = np.array((4, 5, 6))
z = np.concatenate([x, y])
z
```

Out[19]:
```
array([1, 2, 3, 4, 5, 6])
```

np.concatenate toma una tupla o lista de matrices como primer argumento. También puede concatenar más de dos arrays a la vez:

In [20]:
```
w = [97, 98, 99]
print(np.concatenate([x, y, w]))
```

```
[ 1  2  3  4  5  6 97 98 99]
```

stack: une una secuencia de matrices a lo largo de un nuevo eje:

In [21]:
```
v1=np.random.randint(low = 0, high = 10, size = 5)
v2=np.random.randint(low = 11, high = 20, size = 5)
v1, v2
```

Out[21]:
```
(array([0, 3, 4, 1, 4]), array([13, 16, 18, 17, 18]))
```

In [22]:
```
juntos1 = np.stack((v1, v2), axis=0)
juntos1
```

Out[22]:
```
array([[ 0,  3,  4,  1,  4],
       [13, 16, 18, 17, 18]])
```

In [23]:
```
juntos2 = np.stack((v1, v2), axis=1)
juntos2
```

Out[23]:
```
array([[ 0, 13],
       [ 3, 16],
       [ 4, 18],
       [ 1, 17],
       [ 4, 18]])
```

split: divide el array:

In [24]:
```
separados = np.split(juntos1, 2)
separados
```

Out[24]:
```
[array([[0, 3, 4, 1, 4]]), array([[13, 16, 18, 17, 18]])]
```

con **delete** eliminamos un elemento:

In [25]:
```
v
```

Out[25]:
```
array([25. , 90. , 12. , 15. , 66. , 12.5, 10. , 11. , 12. ])
```

In [26]:
```
v = np.delete(v, 2, axis = 0)
v
```

Out[26]:
```
array([25. , 90. , 15. , 66. , 12.5, 10. , 11. , 12. ])
```

Operaciones de asignación y aritméticas

In [27]:
```
v1 = np.random.randint(low = 0, high = 10, size = 5)
v2 = np.random.randint(low = 0, high = 10, size = 5)
v1, v2
```

Out[27]:
```
(array([9, 4, 0, 1, 6]), array([5, 6, 6, 5, 4]))
```

In [28]:
```
v1[3] = 11
v1
```

Out[28]:
```
array([ 9,  4,  0, 11,  6])
```

In [29]:
```
print("Sumamos 1 a cada elemento del array:\n", v1 + 1)
print("Multiplicamos por 5 cada elemento del array:\n", v1 * 5)
print("Elevamos cada elemento al cuadrado:\n", v1 ** 2)
print("El vector sumado a sí mismo:\n", v1 + v1)
print("Suma de arrays:\n", v1 + v2)
print("Resta de arrays:\n", v1 - v2)
print("Array de ceros con todos los elementos con valor 2:\n", np.zeros(5)+2)
print("Array de unos con todos los elementos con valor 2 (otra forma):\n", np.ones((5))*2)
```

```
Sumamos 1 a cada elemento del array:
 [10  5  1 12  7]
Multiplicamos por 5 cada elemento del array:
 [45 20  0 55 30]
Elevamos cada elemento al cuadrado:
 [ 81  16   0 121  36]
El vector sumado a sí mismo:
 [18  8  0 22 12]
Suma de arrays:
 [14 10  6 16 10]
Resta de arrays:
 [ 4 -2 -6  6  2]
Array de ceros con todos los elementos con valor 2:
 [2. 2. 2. 2. 2.]
Array de unos con todos los elementos con valor 2 (otra forma):
 [2. 2. 2. 2. 2.]
```

power eleva cada base a la potencia correspondiente a la posición:

In [30]:
```
x = np.arange(6)
x, np.power(x, 3)
```

Out[30]:
```
(array([0, 1, 2, 3, 4, 5]), array([  0,   1,   8,  27,  64, 125], dtype=int32))
```

Ejercicios:

1. Generar un array aleatorio de 15 números enteros.
2. Agregar el valor 0 en la posición 5
3. Generar un array de 4 números enteros y agregalo al array del punto 1
4. Generar un array de 20 elementos enteros, une éste con el anterior en un nuevo eje.
5. Suma los elementos de ambos arrays.

Aplicando operadores relaciones

In [31]:
```
v=np.array([25,12,15,66,12.5,7])
print(f"Original: {v}\nNos quedamos con los >= 15: {v >= 15}")
print(f"Verificamos los elementos pares: {v % 2 == 0}")
```

```
Original: [25.  12.  15.  66.  12.5  7. ]
Nos quedamos con los >= 15: [ True False  True  True False False]
Verificamos los elementos pares: [False  True False  True False False]
```

Algunas funciones

where devuelve la posición de un elemento.

In [32]:
```
print(f"Dónde está el número 15?: {np.where(v == 15)}")
```

```
Dónde está el número 15?: (array([2], dtype=int64),)
```

sort ordena el array. **argsort** devuelve los índices de los elementos ordenados:

In [33]:
```
print(f"Array original: {v}\nArray ordenado: {np.sort(v)}\nÍndices de elementos ordenados: {np.argsort(v)}")
```

```
Array original: [25.  12.  15.  66.  12.5  7. ]
Array ordenado: [ 7.  12.  12.5 15.  25.  66. ]
Índices de elementos ordenados: [5 1 4 2 0 3]
```

El primer elemento de este resultado da el índice del elemento más pequeño, el segundo valor da el índice del segundo más pequeño y así sucesivamente.

sum, **mean**, **max**, **min**

In [34]:
```
print(f"Array: {v}, Suma: {np.sum(v)}, Promedio: {np.mean(v)}, Máximo: {np.max(v)}, Mínimo: {np.min(v)}")
```

```
Array: [25.  12.  15.  66.  12.5  7. ], Suma: 137.5, Promedio: 22.916666666666668, Máximo: 66.0, Mínimo: 7.0
```

Como parámetro de una función lambda

In [35]:
```
v=np.array([25,12,15,66,12.5,7])
sumo_dos = lambda x: x + 2
print(f"Array:{v}\nArray después de la función sumo_dos:\n {sumo_dos(v)}")
```

```
Array:[25.  12.  15.  66.  12.5  7. ]
Array después de la función sumo_dos:
 [27.  14.  17.  68.  14.5  9. ]
```

Ejercicios:

1. Generar un array de 10 elementos de tipo float.
2. Determinar los elementos menores a 5 del array generado en el punto 1
3. Emitir el máximo y el mínimo del array.
4. Crear una función lambda que devuelva los elementos del array elevados al cubo.
5. Ordenar el array.

Arrays bidimensionales

Creación. **randint** y **rand**

In [36]:
```
m1 = np.random.randint(100, size=(2, 4))
m2 = np.random.rand(2,3)
m3 = np.random.randn(3,4)
m1, m2, m3
```

Out[36]:
```
(array([[82, 16, 11, 91],
        [90, 93, 59, 98]]),
 array([[0.8203538 , 0.82679124, 0.50927197],
        [0.0195013 , 0.61226397, 0.69211116]]),
 array([[ 1.38015252, -0.97193288,  0.12229416,  0.38297534],
        [-0.81301758,  0.34644905, -0.52958326,  0.19982498],
        [ 0.31274673, -0.57511304, -1.49421832,  1.04379165]]))
```

iteramos de distintas formas:

In [37]:
```
for elemento in m1:
    print(elemento)
```

```
[82 16 11 91]
[90 93 59 98]
```

In [38]:
```
for i in m1:
    for j in i:
        print(j, end=' ')
    print()
```

```
82 16 11 91
90 93 59 98
```

In [39]:
```
for i in range(len(m1)):
    for j in range(len(m1[i])):
        print(m1[i][j], end=' ')
    print()
```

```
82 16 11 91
90 93 59 98
```

zeros y **ones**

In [40]:
```
print(f"Matriz de ceros:\n {np.zeros((2,3))}\n Matriz de unos:\n {np.ones((4,3))}")
```

```
Matriz de ceros:
 [[0. 0. 0.]
 [0. 0. 0.]]
 Matriz de unos:
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

append

In [41]:
```
m1 = np.array([[1 ,-4], [12 , 3], [7.2, 5]])
m1 = np.append(m1, [[400], [800], [260]], axis = 1)
m1 = np.append(m1, [[50, 60, 70]], axis = 0)
m1
```

Out[41]:
```
array([[  1. ,  -4. , 400. ],
       [ 12. ,   3. , 800. ],
       [  7.2,   5. , 260. ],
       [ 50. ,  60. ,  70. ]])
```

concatenate: también se puede usar para matrices bidimensionales:

In [42]:
```
m1 = np.array([[1, 2, 3], [4, 5, 6]])
m2 = np.array([[10, 20, 30], [40, 50, 60]])
print(f"Concatenar a lo largo del primer eje:\n{np.concatenate([m1, m2])},\n \
Concatenar a lo largo del segundo eje:\n{np.concatenate([m1, m2], axis=1)}")
```

```
Concatenar a lo largo del primer eje:
[[ 1  2  3]
 [ 4  5  6]
 [10 20 30]
 [40 50 60]],
 Concatenar a lo largo del segundo eje:
[[ 1  2  3 10 20 30]
 [ 4  5  6 40 50 60]]
```

vstack: apila matrices verticalmente en cuanto a filas. **hstack**: apila matrices horizontalmente, en cuanto a columnas.

In [43]:
```
m1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
m2 = np.array([[9, 10, 11, 12], [13, 14, 15, 16]])
print(f"Mezcla Vertical:\n {np.vstack((m1, m2))}\nMezcla Horizontal:\n {np.hstack((m1, m2))}")
```

```
Mezcla Vertical:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
Mezcla Horizontal:
 [[ 1  2  3  4  9 10 11 12]
 [ 5  6  7  8 13 14 15 16]]
```

vsplit y **hsplit**: dividen la matriz, en cuanto a filas y columnas, respectivamente:

In [44]:
```
print(f"División vertical de m1 en 2:\n {np.vsplit(m1, 2)}\nDivisión horizontal de m1 en 2:\n {np.hsplit(m1, 2)}")
```

```
División vertical de m1 en 2:
 [array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]])]
División horizontal de m1 en 2:
 [array([[1, 2],
       [5, 6]]), array([[3, 4],
       [7, 8]])]
```

delete

In [45]:
```
m1, m2
```

Out[45]:
```
(array([[1, 2, 3, 4],
        [5, 6, 7, 8]]),
 array([[ 9, 10, 11, 12],
        [13, 14, 15, 16]]))
```

In [46]:
```
m1 = np.delete(m1, 0, axis = 0)
m2 = np.delete(m2, 1, axis = 1)
m1, m2
```

Out[46]:
```
(array([[5, 6, 7, 8]]),
 array([[ 9, 11, 12],
        [13, 15, 16]]))
```

Ejercicios:

1. Generar una matriz con una distribución normal de 3 por 4.
2. Agregar una nueva columna y una fila a la matriz anterior.
3. Generar otra matriz con la mismas dimensiones de la anterior y concatena ambas.
4. Apilar las matrices horizontalmente.

Operaciones de asignación y aritméticas

In [47]:
```
m1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
m1, m1 + 5
```

Out[47]:
```
(array([[1, 2, 3, 4],
        [5, 6, 7, 8]]),
 array([[ 6,  7,  8,  9],
        [10, 11, 12, 13]]))
```

In [48]:
```
print("Asignamos el valor 40 a los elementos de la columna 0:\n")
m1[:,0] = 40
m1
```

```
Asignamos el valor 40 a los elementos de la columna 0:
```

Out[48]:
```
array([[40,  2,  3,  4],
       [40,  6,  7,  8]])
```

In [49]:
```
print("- Dividimos por 3 la columna 1:\n")
m1[:,1] = m1[:,1]/3.0
m1
```

```
- Dividimos por 3 la columna 1:
```

Out[49]:
```
array([[40,  0,  3,  4],
       [40,  2,  7,  8]])
```

In [50]:
```
print("Multiplicamos por 5 la fila 1:\n")
m1[1,:] = m1[1,:]*5
m1
```

```
Multiplicamos por 5 la fila 1:
```

Out[50]:
```
array([[ 40,   0,   3,   4],
       [200,  10,  35,  40]])
```

Algunas funciones

sort

In [51]:
```
m1 = np.array([[1 ,-4], [12 , 3], [7.2, 5]])
print(f"Matriz ordenada:\n{np.sort(m1)}")
```

```
Matriz ordenada:
[[-4.   1. ]
 [ 3.  12. ]
 [ 5.   7.2]]
```

transpose: invierte o permuta los ejes de una matriz; devuelve la matriz modificada. **ravel**: devuelve la matriz aplanada.

In [52]:
```
m1.transpose(), m1.ravel()
```

Out[52]:
```
(array([[ 1. , 12. ,  7.2],
        [-4. ,  3. ,  5. ]]),
 array([ 1. , -4. , 12. ,  3. ,  7.2,  5. ]))
```

Ejercicios:

1. Generar una matriz de números enteros aleatorios de 4 por 5.
2. Dividir por 3 la columna 2
3. Tranponer la matriz.
4. Ordenar la matriz.

Más información: Álgebra Lineal con Numpy (https://numpy.org/doc/stable/reference/routines.linalg.html#)

Más información: Álgebra Lineal con Python (https://deepnote.com/@anthonymanotoa/Apuntes-de-Algebra-Lineal-con-Python-bf71a544-4b58-430c-9e81-79153c6ef73d)

Arrays booleanos

In [53]:
```
rng = np.random.RandomState(0)
m1 = rng.randint(10, size=(3, 4))
m1
```

Out[53]:
```
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

In [54]:
```
m1 < 6
```

Out[54]:
```
array([[ True,  True,  True,  True],
       [False, False,  True,  True],
       [ True,  True, False, False]])
```

Para contar el número de entradas verdaderas (True) en una matriz booleana, **np.count_nonzero** es útil:

In [55]:
```
np.count_nonzero(m1 < 6)
```

Out[55]: 8

Vemos que hay 8 entradas de matriz que son menores que 6. Otra forma de obtener esta información es usar np.sum; en este caso, False se interpreta como 0 y True se interpreta como 1:

In [56]:
```
np.sum(m1 < 6)
```

Out[56]: 8

El beneficio de sum() es que, al igual que con otras funciones de agregación de NumPy, esta suma también se puede realizar a lo largo de filas o columnas:

In [57]:
```
m1, np.sum(m1, axis=0), np.sum(m1, axis=1), np.sum(m1 < 6, axis=1)
```

Out[57]:
```
(array([[5, 0, 3, 3],
        [7, 9, 3, 5],
        [2, 4, 7, 6]]),
 array([14, 13, 13, 14]),
 array([11, 24, 19]),
 array([4, 2, 2]))
```

Si estamos interesados en verificar rápidamente si alguno o todos los valores son verdaderos, podemos usar **np.any()** o **np.all()**

¿hay valores mayores que 8?, ¿hay valores menores que 0?, ¿todos los valores son menores que 10?, ¿Son todos los valores iguales a 6?

In [58]:
```
np.any(m1 > 8), np.any(m1 < 0), np.all(m1 < 10), np.all(m1 == 6)
```

Out[58]: (True, False, True, False)

any() y all() están destinados a matrices booleanas. any() devuelve True si hay valores que son iguales a True en el array. all() devuelve True si todos los valores de la matriz son iguales a True. Para enteros o flotantes, la funcionalidad es similar, excepto que regresan True si el valor 0 no se encuentra en la matriz

In [59]:
```
a = np.array([1,2,3])
b = np.array([-1,0,1])
c = np.array([True, False])

print(a.any(), a.all())
print(b.any(), b.all())
print(c.any(), c.all())
```

```
True True
True False
True False
```

np.all() y np.any() también se pueden usar a lo largo de ejes particulares

¿todos los valores de cada fila son inferiores a 8?

In [60]:
```
m1, np.all(m1 < 8, axis=1)
```

Out[60]:
```
(array([[5, 0, 3, 3],
        [7, 9, 3, 5],
        [2, 4, 7, 6]]),
 array([ True, False,  True]))
```

Python tiene funciones integradas sum(), any() y all(). Estos tienen una sintaxis diferente a las versiones de NumPy y, en particular, fallarán o producirán resultados no deseados cuando se usen en arreglos multidimensionales. Asegúrese de usar np.sum(), np.any() y np.all() para estos ejemplos.

Arrays booleanos como máscaras

In [61]:
```
x, x < 3
```

Out[61]:
```
(array([0, 1, 2, 3, 4, 5]), array([ True,  True,  True, False, False, False]))
```

Para seleccionar estos valores de la matriz, simplemente podemos indexar en esta matriz booleana; esto se conoce como una **operación de enmascaramiento**

In [62]:
```
x, x[x < 3]
```

Out[62]:
```
(array([0, 1, 2, 3, 4, 5]), array([0, 1, 2]))
```

Se devuelve es una matriz unidimensional llena de todos los valores que cumplen esta condición; en otras palabras, todos los valores en las posiciones en las que la matriz de máscaras es verdadera.

Uso de las palabras clave and / or frente a los operadores & / |, cuál es la diferencia?

La diferencia es esta: **"and"** y **"or** miden la verdad o falsedad de todo el objeto, mientras que **"&"** y **"|"** se refieren a bits dentro de cada objeto. Cuando se usa and u or, es equivalente a pedirle a Python que trate el objeto como una sola entidad booleana. En Python, todos los enteros distintos de cero se evaluarán como verdaderos.

Cuando se usa **"&"** y **"|"** en números enteros, la expresión opera sobre los bits del elemento, aplicando el **"and"** o el **"or** a los bits individuales que componen el número:

In [63]:
```
A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
A | B
```

Out[63]:
```
array([ True,  True,  True, False,  True,  True])
```

Usar **"or"** en estas matrices intentará evaluar la verdad o la falsedad de todo el objeto de la matriz, que no es un valor bien definido:

In [64]:
```
A or B
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [64], in <cell line: 1>()
----> 1 A or B

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

De manera similar, al hacer una expresión booleana en una matriz dada, debe usar **"&"** o **"|"** en lugar de **"or"** o **"and"**:

In [65]:
```
x = np.arange(10)
(x > 4) & (x < 8)
```

Out[65]:
```
array([False, False, False, False, False,  True,  True,  True, False,
       False])
```

Tratar de evaluar la verdad o la falsedad de toda la matriz dará el mismo error que vimos antes:

In [66]:
```
(x > 4) and (x < 8)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [66], in <cell line: 1>()
----> 1 (x > 4) and (x < 8)

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

Recordar: and y or realizan una única evaluación booleana en un objeto completo, mientras que & y | realizar múltiples evaluaciones booleanas

Función universal (abreviado ufunc)

Operan en arrays elemento por elemento, soportando broadcasting, conversión de tipos, y varias otras características estándar. Es decir, un ufunc es un wrapper "vectorizado" de una función que toma un número fijo de entradas escalares y produce un número fijo de salidas escalares.

Las funciones universales son instancias de la clase numpy.ufunc. Muchas de las funciones disponibles son implementadas en código C compilado. Los ufuncs existen en dos formas: ufuncs **unarios**, que operan en una sola entrada, y **binarios**, que operan en dos entradas.

Operadores aritméticos implementados en NumPy

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

Fuente: Source-Python for data science handbook

Operadores de comparación como ufuncs

Operator	Equivalent ufunc
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

Fuente: Source-Python for data science handbook

operadores booleanos bit a bit y sus ufuncs equivalentes

Operator	Equivalent ufunc
&	np.bitwise_and
|	np.bitwise_or
^	np.bitwise_xor
~	np.bitwise_not

Fuente: Source-Python for data science handbook

Especificación de salida

En lugar de crear una matriz temporal, **out** puede usarse para escribir los resultados de los cálculos.

In [67]:
```
v1 = np.arange(5)
v2 = np.zeros(5)
v1, np.multiply(v1,10,out=v2)
```

Out[67]:
```
(array([0, 1, 2, 3, 4]), array([ 0., 10., 20., 30., 40.]))
```

Si quisiéramos reducir una matriz con una operación en particular, podemos usar el método **reduce**. Una reducción aplica repetidamente una operación determinada a los elementos de una matriz hasta que solo queda un único resultado.

In [68]:
```
x = np.arange(1, 6)
x, np.add.reduce(x), np.multiply.reduce(x), np.add.accumulate(x), np.multiply.accumulate(x)
```

Out[68]:
```
(array([1, 2, 3, 4, 5]),
 15,
 120,
 array([ 1,  3,  6, 10, 15], dtype=int32),
 array([  1,   2,   6,  24, 120], dtype=int32))
```

Cualquier ufunc puede calcular la salida de todos los pares de dos entradas diferentes usando el método externo **outer**. Esto le permite, en una línea, hacer cosas como crear una tabla de multiplicar:

In [69]:
```
1 x, np.multiply.outer(x, x)
```

Out[69]:
```
(array([1, 2, 3, 4, 5]),
 array([[ 1,  2,  3,  4,  5],
        [ 2,  4,  6,  8, 10],
        [ 3,  6,  9, 12, 15],
        [ 4,  8, 12, 16, 20],
        [ 5, 10, 15, 20, 25]]))
```

Agregaciones

Cuando existe una gran cantidad de datos a procesar, un primer paso es calcular estadísticas de resumen para los datos en cuestión. Las estadísticas de resumen más comunes son la media y la desviación estándar, que le permiten resumir los valores “típicos” en un conjunto de datos, pero también son útiles otros agregados como la suma, el producto, la mediana, el mínimo y el máximo, los cuantiles, etc.

Como ejemplo, considere calcular la suma de todos los valores en una matriz. Python mismo puede hacer esto usando la función de suma incorporada

In [70]:
```
1 array = np.random.random(100)
2 sum(array)
```

Out[70]:
```
50.84406067150769
```

In [71]:
```
1 np.sum(array)
```

Out[71]:
```
50.844060671507684
```

Una sintaxis más corta es usar métodos del propio objeto:

In [72]:
```
1 big_array = np.random.rand(100000)
2 big_array.min(), big_array.max(), big_array.sum()
```

Out[72]:
```
(2.037581791292098e-06, 0.9999992595236278, 49903.69790802508)
```

Podemos encontrar el valor mínimo dentro de cada columna especificando el axis:

In [73]:
```
1 m1 = np.random.random((3, 4))
2 m1, m1.min(axis=0), m1.max(axis=1)
```

Out[73]:
```
(array([[0.34845145, 0.44589961, 0.16307191, 0.71161085],
        [0.57033305, 0.49592335, 0.01193752, 0.08616538],
        [0.32011183, 0.691695  , 0.0246659 , 0.4665668 ]]),
 array([0.32011183, 0.44589961, 0.01193752, 0.08616538]),
 array([0.71161085, 0.57033305, 0.691695  ]))
```

Funciones de agregación disponibles en NumPy

Function Name	NaN-safe Version	Description
np.sum	np.nansum	Compute sum of elements
np.prod	np.nanprod	Compute product of elements
np.mean	np.nanmean	Compute median of elements
np.std	np.nanstd	Compute standard deviation
np.var	np.nanvar	Compute variance
np.min	np.nanmin	Find minimum value
np.max	np.nanmax	Find maximum value
np.argmin	np.nanargmin	Find index of minimum value
np.argmax	np.nanargmax	Find index of maximum value
np.median	np.nanmedian	Compute median of elements
np.percentile	np.nanpercentile	Compute rank-based statistics of elements
np.any	N/A	Evaluate whether any elements are true
np.all	N/A	Evaluate whether all elements are true

Fuente: Source-Python for data science handbook

```
Ejercicios:

1. Genera una matriz con una distribución normal de 4 por 3
2. Calcular la suma por fila y por columna.
3. Utilizando funciones universales calcular el máximo, el mínimo la media, la desviación estándar,
la varianza y los percentiles.
```

Broadcasting

Las funciones universales de NumPy se pueden usar para **vectorizar** operaciones y, por lo tanto, eliminar los bucles de Python. Otra forma de vectorizar operaciones es usar la funcionalidad de **transmisión** de NumPy. La transmisión es simplemente un conjunto de reglas para aplicar ufuncs binarios como suma, resta, multiplicación, etc. y en matrices de diferentes tamaños.

In [74]:
```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

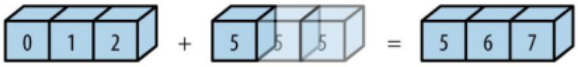
Out[74]:
```
array([5, 6, 7])
```

La transmisión permite que estos tipos de operaciones binarias se realicen en matrices de diferentes tamaños; por ejemplo, podemos agregar fácilmente un escalar a una matriz:

In [75]:
```
a + 5
```

Out[75]:
```
array([5, 6, 7])
```

Podemos pensar en esto como una operación que estira o duplica el valor 5 en la matriz [5, 5, 5] y suma los resultados:



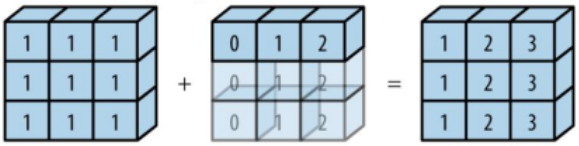
Fuente: Source-Python for data science handbook

De manera similar, podemos extender esto a arreglos de mayor dimensión. Observe el resultado cuando agregamos una matriz unidimensional a una matriz bidimensional:

In [76]:
```
m = np.ones((3, 3))
m, a, m + a
```

Out[76]:
```
(array([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]),
 array([0, 1, 2]),
 array([[1., 2., 3.],
        [1., 2., 3.],
        [1., 2., 3.]]))
```

Aquí, la matriz unidimensional 'a' se estira, o transmite, a través de la segunda dimensión para que coincida con la forma de 'matriz':

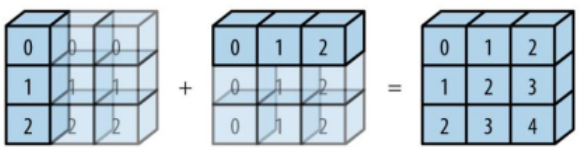


Fuente: Source-Python for data science handbook

In [77]:
```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]
a, b, a + b
```

Out[77]:
```
(array([0, 1, 2]),
 array([[0],
        [1],
        [2]]),
 array([[0, 1, 2],
        [1, 2, 3],
        [2, 3, 4]]))
```

Aquí hemos estirado tanto a como b para que coincidan con una forma común, y el resultado es una matriz bidimensional:



Fuente: Source-Python for data science handbook

La transmisión en NumPy sigue un estricto conjunto de reglas para determinar la interacción entre las dos matrices:

• Regla 1: si las dos matrices difieren en el número de dimensiones, la forma de la que tiene menos dimensiones se rellena con unos en su lado inicial (izquierdo).

• Regla 2: si la forma de las dos matrices no coincide en ninguna dimensión, la matriz con forma igual a 1 en esa dimensión se estira para que coincida con la otra forma.

• Regla 3: Si en alguna dimensión los tamaños no están de acuerdo y ninguno es igual a 1, se comete un error.

Ejemplo de transmisión 1

In [78]:
```
m = np.ones((2, 3))
v = np.arange(3)
m, v
```

Out[78]:
```
(array([[1., 1., 1.],
        [1., 1., 1.]]),
 array([0, 1, 2]))
```

Las formas de las matrices son:

In [79]:
```
m.shape, v.shape
```

Out[79]:
```
((2, 3), (3,))
```

Vemos por la regla 1 que el arreglo vector tiene menos dimensiones, entonces lo rellenamos a la izquierda con unos:

```
m.shape -> (2, 3)
v.shape -> (1, 3)
```

Por la regla 2, ahora vemos que la primera dimensión no está de acuerdo, por lo que estiramos esta dimensión para que coincida:

```
m.shape -> (2, 3)
v.shape -> (2, 3)
```

Las formas coinciden, y vemos que la forma final será (2, 3)

In [80]:
```
m + v
```

Out[80]:
```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

Ejemplo de transmisión 2

In [81]:
```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
a, b
```

Out[81]:
```
(array([[0],
        [1],
        [2]]),
 array([0, 1, 2]))
```

In [82]:
```
a.shape, b.shape
```

Out[82]:
```
((3, 1), (3,))
```

La regla 1 dice que debemos rellenar la forma de b con unos:

```
a.shape -> (3, 1)
b.shape -> (1, 3)
```

Y la regla 2 nos dice que actualizamos cada uno de estos para que coincida con el tamaño correspondiente de la otra matriz:

```
a.shape -> (3, 3)
b.shape -> (3, 3)
```

Debido a que el resultado coincide, estas formas son compatibles.

In [83]:
```
a + b
```

Out[83]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Ejemplo de transmisión 3

Un ejemplo en el que las dos matrices no son compatibles:

In [84]:
```
m = np.ones((3, 2))
v = np.arange(3)
m, v
```

Out[84]:
```
(array([[1., 1.],
        [1., 1.],
        [1., 1.]]),
 array([0, 1, 2]))
```

Esta es una situación ligeramente diferente a la del primer ejemplo: la matriz se transpone. ¿Cómo afecta esto al cálculo? Las formas de las matrices son:

In [85]:
```
m.shape, v.shape
```

Out[85]:
```
((3, 2), (3,))
```

De nuevo, la regla 1 nos dice que debemos rellenar la forma de vector con unos:

```
m.shape -> (3, 2)
v.shape -> (1, 3)
```

Por la regla 2, la primera dimensión de a se estira para que coincida con la de matriz:

```
m.shape -> (3, 2)
v.shape -> (3, 3)
```

Llegamos a la regla 3, las formas finales no coinciden, por lo que estas dos matrices son incompatibles, como podemos observar al intentar esta operación:

In [86]:
```
1 m + v
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [86], in <cell line: 1>()
----> 1 m + v

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Podría imaginarse hacer que vector y matriz sean compatibles, por ejemplo, rellenando la forma de vector con unos a la derecha en lugar de a la izquierda, pero así no funcionan las reglas de transmisión. Si lo que desea es el relleno del lado derecho, puede hacerlo explícitamente remodelando la matriz con **np.newaxis**

In [87]:
```
1 v[:, np.newaxis].shape
```

Out[87]: (3, 1)

In [88]:
```
1 m + v[:, np.newaxis]
```

Out[88]:
```
array([[1., 1.],
       [2., 2.],
       [3., 3.]])
```

Estas reglas de transmisión se aplican a cualquier ufunc binario

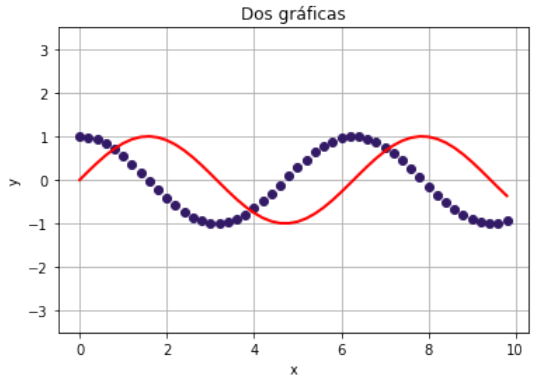
Gráficos: Numpy + Matplotlib

In [91]:
```
1 import matplotlib.pyplot as plt
2 import numpy as np
```

Ejemplo

In [92]:
```
1  x = np.arange(0,10,0.2)
2  y1 = np.cos(x)
3  y2 = np.sin(x)
4
5  plt.rcParams["figure.figsize"] = (6,4)
6  plt.plot(x,y1,'o',linewidth=3,color=(0.2,0.1,0.4))
7  plt.plot(x,y2,'-',linewidth=2,color='r')
8  plt.grid()
9  plt.axis('equal')
10 plt.xlabel('x')
11 plt.ylabel('y')
12 plt.title('Dos gráficas')
```

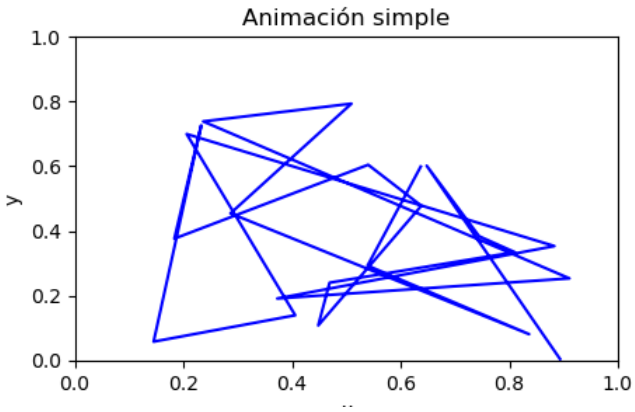
Out[92]: Text(0.5, 1.0, 'Dos gráficas')



Ejemplo

In [93]:

```
%matplotlib notebook
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np

def update_line(num, data, line):
    line.set_data(data[...,:num])
    return line,


fig1 = plt.figure()
fig1.set_size_inches(5,3)

data = np.random.rand(2, 25)
l, = plt.plot([], [], 'b-')
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Animación simple')
line_ani = animation.FuncAnimation(fig1, update_line, 25, fargs=(data, l), interval=50, blit=True)
```



Ejemplo Creamos un archivo .csv con los siguientes datos:

```
 2,0.75,2,1,0.5
1,0.125,1,1,0.125
2.75,1.5,1,0,1
4,0.5,2,2,0.5
```

In [94]:

```
csv_array = np.genfromtxt('archs/archivo.csv', delimiter = ',')
csv_array
```

Out[94]:

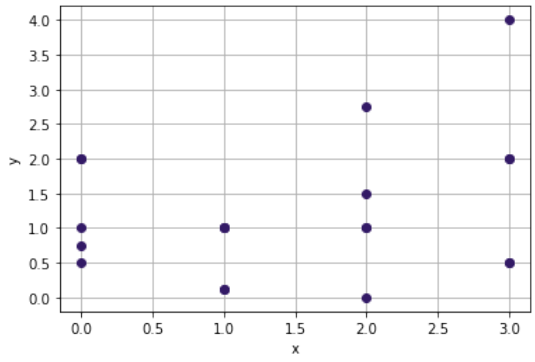
```
array([[2.   , 0.75 , 2.   , 1.   , 0.5  ],
       [1.   , 0.125, 1.   , 1.   , 0.125],
       [2.75 , 1.5  , 1.   , 0.   , 1.   ],
       [4.   , 0.5  , 2.   , 2.   , 0.5  ]])
```

In [95]:

```
%matplotlib inline
plt.rcParams["figure.figsize"] = (6,4)

plt.plot(csv_array,'o',linewidth=3,color=(0.2,0.1,0.4))
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
```

Out[95]:

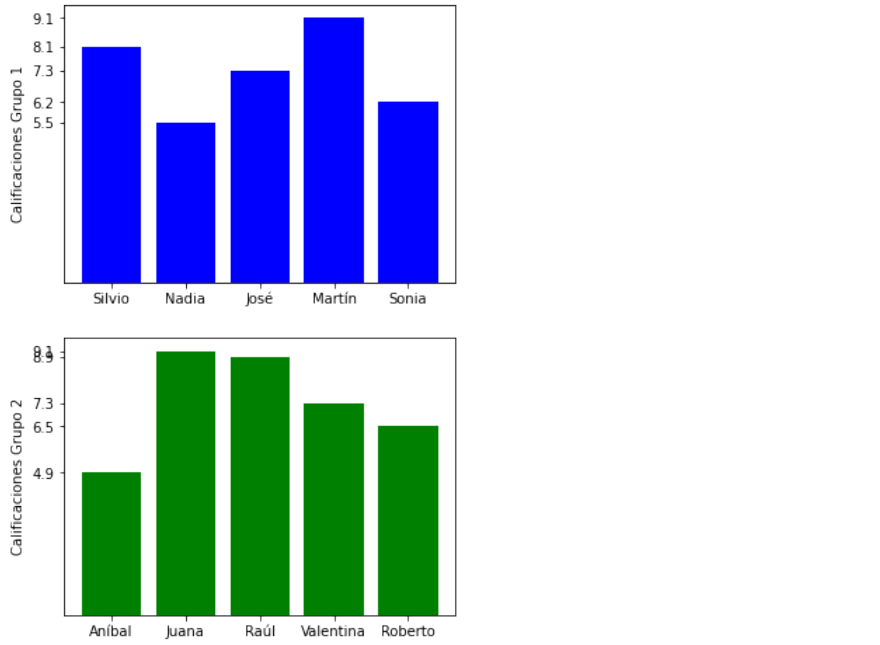
```
Text(0, 0.5, 'y')
```



In [96]:

```
#creamos la ventana
fig=plt.figure('Calificaciones')
fig.set_size_inches(5,8)
#agregamos dos gráficas
grupo1=fig.add_subplot(211)
grupo2=fig.add_subplot(212)

#obtenemos datos para el primer conjunto
alu_1 = ['Silvio', 'Nadia','José', 'Martín','Sonia']
calif_1 = np.array([8.1, 5.5, 7.3, 9.1, 6.2])

#obtenemos datos para el segundo conjunto
alu_2 = ['Aníbal', 'Juana','Raúl', 'Valentina','Roberto']
calif_2 = np.array([4.9, 9.1, 8.9, 7.3, 6.5])

#barras para el primer conjunto
grupo1.bar(alu_1, calif_1, color='b', align='center')
grupo1.set_xticks(alu_1)
grupo1.set_xticklabels(alu_1)
grupo1.set_yticks(calif_1)
grupo1.set_ylabel('Calificaciones Grupo 1')

# si queremos las etiquetas debajo debemos usar el método xlabel
#barras para el segundo conjunto
grupo2.bar(alu_2, calif_2, color='g',align='center')
grupo2.set_xticks(alu_2)
grupo2.set_xticklabels(alu_2)
grupo2.set_yticks(calif_2)
grupo2.set_ylabel('Calificaciones Grupo 2')
```

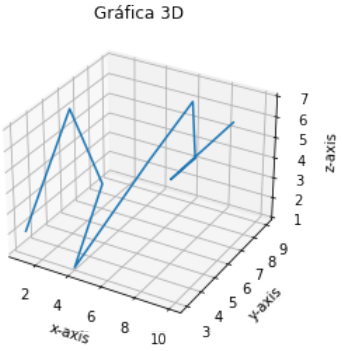
Out[96]: Text(0, 0.5, 'Calificaciones Grupo 2')



In [97]:

```
from mpl_toolkits.mplot3d import Axes3D

#creamos la ventana
fig=plt.figure()

#agregamos plano 3D
plano=fig.add_subplot(111, projection='3d')

#obtenemos datos para x, y, z
x=np.array([[1,2,3,4,5,6,7,8,9,10]])
y=np.array([[3,5,6,3,7,9,8,5,5,7]])
z=np.array([[2,7,3,1,4,6,4,5,6,7]])

#agregamos los arrays al plano
plano.plot_wireframe(x,y,z)
plano.set_xlabel('x-axis')
plano.set_ylabel('y-axis')
plano.set_zlabel('z-axis')
plt.title('Gráfica 3D')
```

Out[97]: Text(0.5, 0.92, 'Gráfica 3D')



Más información: Procesamiento de imágenes con Python y Numpy (https://facundoq.github.io/courses/aa2018/res/04_imagenes_numpy.html)