😎 Arrays con Numpy

NumPy es el paquete fundamental para la computación científica en Python. Numpy Significa 'Python numérico'. Es una biblioteca que proporciona matrices de distintas dimensiones y una variedad de rutinas de operaciones.
Los arrays en NumPy brindan operaciones de datos y almacenamiento mucho más eficientes a medida que crecen en tamaño y forman el núcleo de casi todo el ecosistema de herramientas de ciencia de datos en Python.

Los conjuntos de datos pueden provenir de una amplia gama de fuentes y formatos, incluidas colecciones de documentos, colecciones de imágenes, colecciones de clips de sonido, colecciones de medidas numéricas, etc. A pesar de esta aparente heterogeneidad, nos ayudará a pensar en todos los datos fundamentalmente como matrices de números, es decir, no importa cuáles sean los datos, el primer paso para hacerlos analizables será transformarlos en arrays de números.

Por esta razón, el almacenamiento y la manipulación eficientes de matrices numéricas son absolutamente fundamentales para el proceso de hacer ciencia de datos.

Operaciones usando NumPy

• Operaciones matemáticas y lógicas en matrices.

• Transformadas de Fourier y rutinas para la manipulación de formas.

• Operaciones relacionadas con álgebra lineal. NumPy tiene funciones integradas para álgebra lineal y generación de números aleatorios.

La distribución estándar de Python no viene incluida con el módulo NumPy

pip install numpy

Importamos la librería

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

Creación de arrays

array: Es la forma más simple de crear un array a partir de un iterador como una simple lista, el iterador también puede ser otro array NumPy

In [2]:
```
lista = [1, 2, 3, 4, 5]
x = np.array(lista)
print("El array de numpy creado desde una lista es = ", x)
```

```
El array de numpy creado desde una lista es =  [1 2 3 4 5]
```

In [3]:
```
listas = np.array([[1,2,3],[4,5,6]])
x = np.array(listas)
print("El array de numpy creado desde dos listas es = ", x)
```

```
El array de numpy creado desde dos listas es =  [[1 2 3]
 [4 5 6]]
```

In [4]:
```
x = np.array([1,2,3,4,5], dtype = np.int8)
x
```

Out[4]:
```
array([1, 2, 3, 4, 5], dtype=int8)
```

In [5]:
```
x = np.array([1,2,3,4,5], dtype = np.float32)
x
```

Out[5]:
```
array([1., 2., 3., 4., 5.], dtype=float32)
```

Se pueden crear arrays desde una tupla

In [6]:
```
tupla = (1, 2, 3, 4, 5)
x = np.array(tupla)
print("El array de numpy creado desde una tupla es = ", x)
```

```
El array de numpy creado desde una tupla es =  [1 2 3 4 5]
```

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

In [7]:
```
print("Array a lista = ", x.tolist())
```

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

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

In [8]:
```
m = np.linspace(10, 40, 4)
m
```

Out[8]:
```
array([10., 20., 30., 40.])
```

In [9]:
```
m = np.linspace(2, 3, 10)
m
```

Out[9]:
```
array([2.        , 2.11111111, 2.22222222, 2.33333333, 2.44444444,
       2.55555556, 2.66666667, 2.77777778, 2.88888889, 3.        ])
```

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

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

In [11]:
```
m = np.linspace(10, 40, 4, dtype = "float")
m
```

Out[11]:
```
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 [12]:
```
m = np.logspace(2, 3, 10)
m
```

Out[12]:
```
array([ 100.        ,  129.1549665 ,  166.81005372,  215.443469  ,
        278.25594022,  359.38136638,  464.15888336,  599.48425032,
        774.26368268, 1000.        ])
```

In [13]:
```
m = np.logspace(2.0, 3.0, num=5, base = 11)
m
```

Out[13]:
```
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 [14]:
```
un_rango = np.arange(24)
un_rango
```

Out[14]:
```
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])
```

randint y **rand** permiten crear elementos aleatorios enteros o flotantes respectivamente

In [15]:
```
vector_enteros = np.random.randint(low = 0, high = 10, size = 5)
vector_enteros
```

Out[15]:
```
array([2, 0, 8, 0, 3])
```

In [16]:
```
vector_flotantes = np.random.rand(5)
vector_flotantes
```

Out[16]:
```
array([0.48061772, 0.87653503, 0.04116929, 0.47197347, 0.15992837])
```

Podemos iterar sobre el array de numpy

In [17]:
```
vector=np.random.randint(low = 10, high = 100, size = 5)
vector
```

Out[17]:
```
array([50, 34, 73, 12, 82])
```

In [18]:
```
for elemento in vector:
    print(elemento)
```

```
50
34
73
12
82
```

full(): nos permite crear un array de cierto tamaño y tipo completarlo con un valor concreto:

In [19]:
```
a = np.full((2,3), fill_value = '-2')
a
```

Out[19]:
```
array([['-2', '-2', '-2'],
       ['-2', '-2', '-2']], dtype='<U2')
```

zeros y **ones** generan arrays de ceros o unos respectivamente

In [20]:
```
z = np.zeros(5)
z
```

Out[20]:
```
array([0., 0., 0., 0., 0.])
```

In [21]:
```
z = np.ones(5)
z
```

Out[21]:
```
array([1., 1., 1., 1., 1.])
```

con **copy** creamos una copia del array

In [22]:
```
copia=np.copy(vector)
print("Copia del array:", copia)
```

```
Copia del array: [50 34 73 12 82]
```

ndim: devuelve el número de dimensiones de la matriz o los ejes

In [23]:
```
z
```

Out[23]:
```
array([1., 1., 1., 1., 1.])
```

In [24]:
```
z.ndim
```

Out[24]:
```
1
```

In [25]:
```
a
```

Out[25]:
```
array([['-2', '-2', '-2'],
       ['-2', '-2', '-2']], dtype='<U2')
```

In [26]:
```
a.ndim
```

Out[26]:
```
2
```

shape: devuelve una tupla que consta de dimensiones de matriz.

In [27]:
```
z
```

Out[27]:
```
array([1., 1., 1., 1., 1.])
```

In [28]:
```
z.shape
```

Out[28]:
```
(5,)
```

In [29]:
```
a
```

Out[29]:
```
array([['-2', '-2', '-2'],
       ['-2', '-2', '-2']], dtype='<U2')
```

In [30]:
```
a.shape
```

Out[30]:
```
(2, 3)
```

También se puede usar para cambiar la dimensión de la matriz

In [31]:
```
matriz =  np.array([[1,2,3],[4,5,6]])
matriz
```

Out[31]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [32]:
```
matriz.shape = (3,2)
matriz
```

Out[32]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [33]:
```
matriz.shape
```

Out[33]:
```
(3, 2)
```

reshape: función para cambiar el tamaño de una matriz.

In [34]:
```
a = np.arange(24)
a
```

Out[34]:
```
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])
```

In [35]:
```
b = a.reshape(2,4,3)
b
```

Out[35]:
```
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]]])
```

In [36]:
```
b.shape
```

Out[36]:
```
(2, 4, 3)
```

dtype devuelve el tipo de datos

In [37]:
```
b.dtype
```

Out[37]:
```
dtype('int32')
```

itemsize devuelve la longitud en bytes de cada elemento de la matriz.

In [38]:
```
a.itemsize
```

Out[38]:
```
4
```

size: obtenemos la cantidad de elementos del array

In [39]:
```
a.size
```

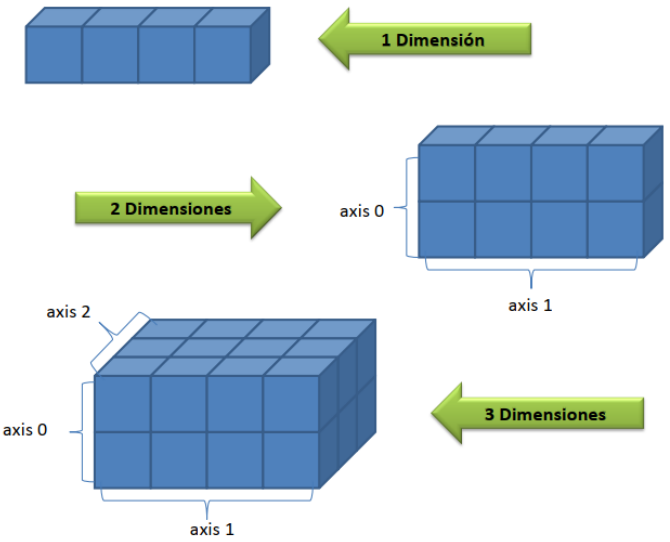
Out[39]:
```
24
```

flags: atributos del objeto ndarray. Sus valores actuales son devueltos por esta función.

In [40]:
```
a.flags
```

Out[40]:
```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False


C_CONTIGUOS (C): Los datos están en un solo segmento contiguo de estilo C
F_CONTIGUOS (F): Los datos están en un solo segmento contiguo de estilo Fortran
OWNDATA : La matriz posee la memoria que usa o la toma prestada de otro objeto
WRITEABLE : Se puede escribir en el área de datos. Establecer esto en False bloquea los datos, haciéndolos de
solo lectura
ALIGNED : Los datos y todos los elementos están alineados adecuadamente para el hardware
WRITEBACKIFCOPY : Esta matriz es una copia de alguna otra matriz
UPDATEIFCOPY : (Obsoleto, use WRITEBACKIFCOPY)
```

🧐 Operaciones básicas



Arrays unidimensionales

Creamos una lista la convertimos en **array** de numpy

In [41]:
```
lista=[25,12,15,66,12.5]
vector=np.array(lista)
vector
```

Out[41]:
```
array([25. , 12. , 15. , 66. , 12.5])
```

con **insert** insertamos un elemento en una posición en particular

In [42]:
```
vector = np.insert(vector, 1, 90)
vector
```

Out[42]:
```
array([25. , 90. , 12. , 15. , 66. , 12.5])
```

con **append** agregamos algunos elementos más

In [43]:
```
vector = np.append (vector, [10, 11, 12])
vector
```

Out[43]:
```
array([25. , 90. , 12. , 15. , 66. , 12.5, 10. , 11. , 12. ])
```

In [44]:
```
vector1=np.random.randint(low = 0, high = 10, size = 5)
vector1
```

Out[44]:
```
array([1, 3, 8, 4, 7])
```

In [45]:
```
vector2=np.random.randint(low = 11, high = 20, size = 5)
vector2
```

Out[45]:
```
array([19, 18, 13, 11, 14])
```

In [46]:
```
nuevo_Array = np.append(vector1, vector2)
nuevo_Array
```

Out[46]:
```
array([ 1,  3,  8,  4,  7, 19, 18, 13, 11, 14])
```

*con **savetxt** podemos grabar los datos en un archivo*

In [47]:
```
np.savetxt("archs/myArray.csv", nuevo_Array)
np.savetxt("archs/myArray2.csv", nuevo_Array, fmt='%.2f')
```

concatenate: unión de dos arrays en NumPy

In [48]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

Out[48]:
```
array([1, 2, 3, 3, 2, 1])
```

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

In [49]:
```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

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

In [50]:
```
vector1=np.random.randint(low = 0, high = 10, size = 5)
vector1
```

Out[50]: array([2, 1, 2, 8, 2])

In [51]:
```
vector2=np.random.randint(low = 11, high = 20, size = 5)
vector2
```

Out[51]: array([13, 16, 16, 14, 16])

In [52]:
```
juntos = np.stack((vector1, vector2), axis=0)
juntos
```

Out[52]:
```
array([[ 2,  1,  2,  8,  2],
       [13, 16, 16, 14, 16]])
```

In [53]:
```
juntos = np.stack((vector1, vector2), axis=0)
juntos
```

Out[53]:
```
array([[ 2,  1,  2,  8,  2],
       [13, 16, 16, 14, 16]])
```

split: divide el array

In [54]:
```
separados = np.split(juntos, 2)
separados
```

Out[54]: [array([[2, 1, 2, 8, 2]]), array([[13, 16, 16, 14, 16]])]

In [55]:
```
separados[0]
```

Out[55]: array([[2, 1, 2, 8, 2]])

In [56]:
```
type(separados[0])
```

Out[56]: numpy.ndarray

con **delete** eliminamos un elemento

In [57]:
```
vector
```

Out[57]: array([25. , 90. , 12. , 15. , 66. , 12.5, 10. , 11. , 12.])

In [58]:
```
vector = np.delete(vector, 2, axis = 0)
vector
```

Out[58]: array([25. , 90. , 15. , 66. , 12.5, 10. , 11. , 12.])

Acceso a los elementos y slicing

In [59]:
```
lista=[25, 12, 15, 66, 12.5, 7]
vector=np.array(lista)
vector
```

Out[59]: array([25. , 12. , 15. , 66. , 12.5, 7.])

In [60]:
```
vector[3]
```

Out[60]: 66.0

In [61]:
```
print("Un subset del vector:", vector[-3:])
```

```
Un subset del vector: [66.  12.5  7. ]
```

In [62]:
```
print("Otro subset del vector:", vector[2:5])
```

```
Otro subset del vector: [15.  66.  12.5]
```

In [63]:
```
vector[1:4]
```

Out[63]: array([12., 15., 66.])

In [64]:
```
vector[1:]
```

Out[64]:
```
array([12. , 15. , 66. , 12.5,  7. ])
```

In [65]:
```
vector[:4]
```

Out[65]:
```
array([25., 12., 15., 66.])
```

In [66]:
```
vector[:]
```

Out[66]:
```
array([25. , 12. , 15. , 66. , 12.5,  7. ])
```

Operaciones de asignación y aritméticas

In [67]:
```
vector[4] = 7.5
```

In [68]:
```
print("Sumamos 1 a cada elemento del vector:", vector + 1)
```

```
Sumamos 1 a cada elemento del vector: [26.  13.  16.  67.   8.5  8. ]
```

In [69]:
```
print("Multiplicamos por 5 cada elemento del vector:", vector * 5)
```

```
Multiplicamos por 5 cada elemento del vector: [125.   60.   75.  330.   37.5  35. ]
```

In [70]:
```
print("Elevamos cada elemento al cuadrado:", vector ** 2)
```

```
Elevamos cada elemento al cuadrado: [ 625.    144.    225.   4356.     56.25   49.  ]
```

In [71]:
```
print("El vector sumado a si mismo:", vector + vector)
```

```
El vector sumado a si mismo: [ 50.  24.  30. 132.  15.  14.]
```

In [72]:
```
vector2=np.array([11, 55, 1.2, 7.4, -8, 32])
```

In [73]:
```
print("Suma de vectores vector1 y vector2:", vector + vector2)
```

```
Suma de vectores vector1 y vector2: [36.  67.  16.2 73.4 -0.5 39. ]
```

In [74]:
```
print("Resta de vectores vector1 y vector2:", vector - vector2)
```

```
Resta de vectores vector1 y vector2: [ 14.  -43.   13.8  58.6  15.5 -25. ]
```

In [75]:
```
print("Vector de ceros con todos los elementos con valor 2:", np.zeros(5)+2)
```

```
Vector de ceros con todos los elementos con valor 2: [2. 2. 2. 2. 2.]
```

In [76]:
```
print("Vector de unos con todos los elementos con valor 2 (otra forma):", np.ones((5))*2)
```

```
Vector de unos con todos los elementos con valor 2 (otra forma): [2. 2. 2. 2. 2.]
```

power eleva cada base en x1 a la potencia correspondiente a la posición en x2

In [77]:
```
x1 = np.arange(6)
x1
```

Out[77]:
```
array([0, 1, 2, 3, 4, 5])
```

In [78]:
```
np.power(x1, 3)
```

Out[78]:
```
array([  0,   1,   8,  27,  64, 125], dtype=int32)
```

Aplicando operadores relaciones

In [79]:
```
lista=[25,12,15,66,12.5,7]
vector=np.array(lista)
vector
```

Out[79]:
```
array([25. , 12. , 15. , 66. , 12.5,  7. ])
```

In [80]:
```
print("Nos quedamos con los >= 15:", vector >= 15)
```

```
Nos quedamos con los >= 15: [ True False  True  True False False]
```

In [81]:
```
print("Verificamos los elementos pares:", vector % 2 == 0)
```

```
Verificamos los elementos pares: [False  True False  True False False]
```

Algunas funciones

data devuelve la dirección dónde comienza el array

In [82]:
```
vector.data
```

Out[82]:
```
<memory at 0x000001AFC18A7AC0>
```

where

In [83]:
```
index = np.where(vector == 15)
index
```

Out[83]:
```
(array([2], dtype=int64),)
```

In [84]:
```
print("Dónde está el número 15?: ", index)
```

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

con **sort** ordenamos el array

In [85]:
```
print("Array original", vector)
```

```
Array original [25.  12.  15.  66.  12.5  7. ]
```

In [86]:
```
print("Array ordenado = ", np.sort(vector))
```

```
Array ordenado =  [ 7.  12.  12.5 15.  25.  66. ]
```

con **argsort** devuelve los índices de los elementos ordenados

In [87]:
```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)
```

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

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

In [88]:
```
print("La suma de los elementos es:", np.sum(vector))
```

```
La suma de los elementos es: 137.5
```

mean

In [89]:
```
print("Promedio (media) de los elementos:", np.mean(vector))
```

```
Promedio (media) de los elementos: 22.916666666666668
```

max

In [90]:
```
print("El máximo de los elementos:", np.max(vector))
```

```
El máximo de los elementos: 66.0
```

min

In [91]:
```
print("El mínimo de los elementos:", np.min(vector))
```

```
El mínimo de los elementos: 7.0
```

Como parámetro de una función lambda

In [92]:
```
lista=[25,12,15,66,12.5,7]
vector=np.array(lista)
vector
```

Out[92]:
```
array([25. , 12. , 15. , 66. , 12.5,  7. ])
```

In [93]:
```
sumo_dos = lambda x: x + 2
```

In [94]:
```
print("Array después de la función sumo_dos: ", sumo_dos(vector))
```

```
Array después de la función sumo_dos:  [27.  14.  17.  68.  14.5  9. ]
```

Arrays bidimensionales

Ejemplos

randint y rand

In [95]:
```
matriz = np.random.randint(100, size=(2, 4))
matriz
```

Out[95]:
```
array([[51, 31, 79,  3],
       [77, 48, 11, 44]])
```

iteramos

In [96]:
```
for elemento in matriz:
    print(elemento)
```

```
[51 31 79  3]
[77 48 11 44]
```

In [97]:
```
matriz = np.random.rand(2,3)
matriz
```

Out[97]:
```
array([[0.54384727, 0.45764203, 0.96145739],
       [0.71798441, 0.57001613, 0.70484603]])
```

iteramos

In [98]:
```
for i in matriz:
    for j in i:
        print(j)
```

```
0.5438472702603325
0.4576420250717257
0.9614573933908576
0.717984405846717
0.570016127708234
0.7048460295237344
```

In [99]:
```
for i in range(len(matriz)):
    for j in range(len(matriz[i])):
        print(matriz[i][j], end=' ')
    print()
```

```
0.5438472702603325 0.4576420250717257 0.9614573933908576
0.717984405846717 0.570016127708234 0.7048460295237344
```

zeros y ones

In [100]:
```
print("Matriz de ceros:",np.zeros((2,3)))
```

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

In [101]:
```
print("Matriz de unos:",np.ones((4,3)))
```

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

append

In [102]:
```
lista_de_listas=[ [1 ,-4], [12 , 3], [7.2, 5]]
matriz = np.array(lista_de_listas)
matriz
```

Out[102]:
```
array([[ 1. , -4. ],
       [12. ,  3. ],
       [ 7.2,  5. ]])
```

In [103]:
```
col = np.array([[400], [800], [260]])
col
```

Out[103]:
```
array([[400],
       [800],
       [260]])
```

In [104]:
```
matriz = np.append(matriz, col, axis = 1)
matriz
```

Out[104]:
```
array([[  1. ,  -4. , 400. ],
       [ 12. ,   3. , 800. ],
       [  7.2,   5. , 260. ]])
```

In [105]:
```
matriz = np.append(matriz, [[50, 60, 70]], axis = 0)
matriz
```

Out[105]:
```
array([[  1. ,  -4. , 400. ],
       [ 12. ,   3. , 800. ],
       [  7.2,   5. , 260. ],
       [ 50. ,  60. ,  70. ]])
```

concatenate: también se puede usar para matrices bidimensionales

In [106]:
```
grid = np.array([[1, 2, 3], [4, 5, 6]])
grid
```

Out[106]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

concatenar a lo largo del primer eje

In [107]:
```
np.concatenate([grid, grid])
```

Out[107]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

concatenar a lo largo del segundo eje

In [108]:
```
np.concatenate([grid, grid], axis=1)
```

Out[108]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

eye(): devuelve un array de dos dimensiones con unos en la diagonal principal y ceros en el resto del array

In [109]:
```
a = np.eye(5, dtype=int)
a
```

Out[109]:
```
array([[1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 1]])
```

identity(): devuelve una matriz cuadrada con unos en la diagonal principal y ceros en el resto del array

In [110]:
```
a = np.identity(5, dtype=int)
a
```

Out[110]:
```
array([[1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 1]])
```

Para ver la diferencia de ambas en un ejemplo: Creamos una matriz de 4 x 4 con la diagonal principal con 1's

In [111]:
```
arr1 = np.eye(4)
arr1
```

Out[111]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

Con eye() se puede cambiar la posición diagonal (prueba con otro número como k= -2)

In [112]:
```
arr2 = np.eye(4, k=1)
arr2
```

Out[112]:
```
array([[0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.]])
```

Con identity() no se puede cambiar la diagonal en la matriz identidad

In [113]:
```
arr3 = np.identity(4)
print(arr3)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

vstack: apila matrices verticalmente en cuanto a filas

In [114]:
```
m1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
m2 = np.array([[9, 10, 11, 12], [13, 14, 15, 16]])
print("Matriz 1:\n", m1)
print("Matriz 2:\n", m2)
```

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

In [115]:
```
print("Mezcla Vertical:")
print(np.vstack((m1, m2)))
```

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

hstack: apila matrices horizontalmente, en cuanto a columnas

In [116]:
```
print("Mezcla Horizontal:")
print(np.hstack((m1, m2)))
```

```
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 [117]:
```
print("División vertical de m1 en 2:")
r = np.vsplit(m1, 2)
r # r[0],r[1]
```

```
División vertical de m1 en 2:
```

Out[117]:
```
[array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]])]
```

In [118]:
```
print("División horizontal de m1 en 2:")
r = np.hsplit(m1, 2)
r # r[0],r[1]
```

```
División horizontal de m1 en 2:
```

Out[118]:
```
[array([[1, 2],
        [5, 6]]),
 array([[3, 4],
        [7, 8]])]
```

In [119]:
```
np.savetxt("archs/myMatrix.csv", m1, fmt='%.2f')
```

delete

In [120]:
```
matriz = np.delete(matriz, 0, axis = 0)
matriz
```

Out[120]:
```
array([[ 12. ,   3. , 800. ],
       [  7.2,   5. , 260. ],
       [ 50. ,  60. ,  70. ]])
```

In [121]:
```
matriz = np.delete(matriz, 1, axis = 1)
matriz
```

Out[121]:
```
array([[ 12. , 800. ],
       [  7.2, 260. ],
       [ 50. ,  70. ]])
```

Acceso a los elementos y slicing

In [122]:
```
lista_de_listas=[ [1 ,-4],[12 , 3], [7.2, 5]]
matriz = np.array(lista_de_listas)
matriz
```

Out[122]:
```
array([[ 1. , -4. ],
       [12. ,  3. ],
       [ 7.2,  5. ]])
```

In [123]:
```
print("Elemento en los índices [1][1]:", matriz[1][1])
```

```
Elemento en los índices [1][1]: 3.0
```

In [124]:
```
print("Elementos individuales:", matriz[0,1])
```

```
Elementos individuales: -4.0
```

In [125]:
```
print("Elementos individuales:", matriz[2,1])
```

```
Elementos individuales: 5.0
```

In [126]:
```
print("Vector de elementos de la fila 1:", matriz[1,:])
```

```
Vector de elementos de la fila 1: [12.  3.]
```

In [127]:
```
print("Vector de elementos de la columna 0:", matriz[:,0])
```

```
Vector de elementos de la columna 0: [ 1.  12.   7.2]
```

In [128]:
```
print("Submatriz de 2x2 con las primeras dos filas:", matriz[0:2,:])
```

```
Submatriz de 2x2 con las primeras dos filas: [[ 1. -4.]
 [12.  3.]]
```

In [129]:
```
print("Submatriz de 2x2 con las ultimas dos filas:", matriz[1:3,:])
```

```
Submatriz de 2x2 con las ultimas dos filas: [[12.   3. ]
 [ 7.2  5. ]]
```

Operaciones de asignación y aritméticas

In [130]:
```
matriz + 5
```

Out[130]:
```
array([[ 6. ,  1. ],
       [17. ,  8. ],
       [12.2, 10. ]])
```

In [131]:
```
print("Asignamos el valor 4 a los elementos de la columna 0:")
matriz[:,0] = 4
matriz
```

```
Asignamos el valor 4 a los elementos de la columna 0:
```

Out[131]:
```
array([[ 4., -4.],
       [ 4.,  3.],
       [ 4.,  5.]])
```

In [132]:
```
print("- Dividimos por 3 la columna 1:")
matriz[:,1]=matriz[:,1]/3.0
matriz
```

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

Out[132]:
```
array([[ 4.        , -1.33333333],
       [ 4.        ,  1.        ],
       [ 4.        ,  1.66666667]])
```

In [133]:
```
print("Multiplicamos por 5 la fila 1:")
matriz[1,:]=matriz[1,:]*5
matriz
```

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

Out[133]:
```
array([[ 4.        , -1.33333333],
       [20.        ,  5.        ],
       [ 4.        ,  1.66666667]])
```

In [134]:
```
print("- Le sumamos 1 a toda la matriz:")
matriz= matriz + 1
matriz
```

```
- Le sumamos 1 a toda la matriz:
```

Out[134]:
```
array([[ 5.        , -0.33333333],
       [21.        ,  6.        ],
       [ 5.        ,  2.66666667]])
```

Algunas funciones

sort

In [135]:
```
lista_de_listas=[ [1 ,-4], [12 , 3], [7.2, 5]]
matriz = np.array(lista_de_listas)
matriz
```

Out[135]:
```
array([[ 1. , -4. ],
       [12. ,  3. ],
       [ 7.2,  5. ]])
```

In [136]:
```
print("Matriz ordenada:", np.sort(matriz))
```

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

sum

In [137]:
```
print("Suma por columna:", matriz.sum(axis=0))
```

```
Suma por columna: [20.2  4. ]
```

In [138]:
```
print("Suma por fila:", matriz.sum(axis=1))
```

```
Suma por fila: [-3.  15.  12.2]
```

max

In [139]:
```
print("Máximo:", matriz.max())
```

```
Máximo: 12.0
```

min

In [140]:
```
print("Mínimo:", matriz.min())
```

```
Mínimo: -4.0
```

transpose: invierte o permuta los ejes de una matriz; devuelve la matriz modificada

In [141]:
```
matriz
```

Out[141]:
```
array([[ 1. , -4. ],
       [12. ,  3. ],
       [ 7.2,  5. ]])
```

In [142]:
```
matriz.transpose()
```

Out[142]:
```
array([[ 1. , 12. ,  7.2],
       [-4. ,  3. ,  5. ]])
```

ravel: devuelve la matriz aplanada

In [143]:
```
matriz.ravel()
```

Out[143]:
```
array([ 1. , -4. , 12. ,  3. ,  7.2,  5. ])
```

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 [144]:
```
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
x
```

Out[144]:
```
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

In [145]:
```
x < 6
```

Out[145]:
```
array([[ True,  True,  True,  True],
       [False, False,  True,  True],
       [ True,  True, False, False]])
```

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

In [146]:
```
np.count_nonzero(x < 6)
```

Out[146]:
```
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 [147]:
```
np.sum(x < 6)
```

Out[147]:
```
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 [148]:
```
x
```

Out[148]:
```
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

In [149]:
```
np.sum(x < 6, axis=1)
```

Out[149]:
```
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?

In [150]:
```
np.any(x > 8)
```

Out[150]:
```
True
```

¿hay valores menores que cero?

In [151]:
```
np.any(x < 0)
```

Out[151]:
```
False
```

¿todos los valores son menores que 10?

In [152]:
```
np.all(x < 10)
```

Out[152]:
```
True
```

¿Son todos los valores iguales a 6?

In [153]:
```
np.all(x == 6)
```

Out[153]:
```
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 [154]:
```
a = np.array([1,2,3])
b = np.array([-1,0,1])
c = np.array([True, False])

print(a.any())
print(a.all())

print(b.any())
print(b.all())

print(c.any())
print(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 [155]:
```
np.all(x < 8, axis=1)
```

Out[155]:
```
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

Rescantando nuestra matriz x

In [156]:
```
x
```

Out[156]:
```
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

In [157]:
```
x < 5
```

Out[157]:
```
array([[False,  True,  True,  True],
       [False, False,  True, False],
       [ True,  True, 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 [158]:
```
x[x < 5]
```

Out[158]:
```
array([0, 3, 3, 3, 2, 4])
```

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 or 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

In [159]:
```
bool(42), bool(0)
```

Out[159]:
```
(True, False)
```

In [160]:
```
bool(42 and 0)
```

Out[160]:
```
False
```

In [161]:
```
bool(42 or 0)
```

Out[161]:
```
True
```

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 [162]: bin(42)
Out[162]: '0b101010'
```

```
In [163]: bin(59)
Out[163]: '0b111011'
```

```
In [164]: bin(42 & 59)
Out[164]: '0b101010'
```

```
In [165]: bin(42 | 59)
Out[165]: '0b111011'
```

Los bits correspondientes de la representación binaria se comparan para obtener el resultado. Cuando tiene una matriz de valores booleanos en NumPy, esto se puede considerar como una cadena de bits donde 1 = Verdadero y 0 = Falso, y el resultado de **"&"** y **"|"** opera de manera similar a la anterior

```
In [166]: 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[166]: 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 [167]: A or B
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-167-ea2c97d9d9ee> in <module>
----> 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 [168]: x = np.arange(10)
          (x > 4) & (x < 8)
Out[168]: 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 [169]: (x > 4) and (x < 8)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-169-eecf1fdd5fb4> in <module>
----> 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 sobre el contenido (los bits o bytes individuales) de un objeto 😉

😲 Función universal (abreviado ufunc)

Es una función que opera en ndarrays 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.

En Numpy, 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

ufuncs especializados

NumPy tiene muchos más ufuncs disponibles, incluidas funciones trigonométricas hiperbólicas, aritmética bit a bit, operadores de comparación, conversiones de radianes a grados, redondeo y residuos, y muchas más. La documentación de NumPy revela muchas funcionalidades interesantes.

Especificación de salida

Para cálculos grandes, a veces es útil poder especificar la matriz donde se almacenará el resultado del cálculo. En lugar de crear una matriz temporal, puede usarla para escribir los resultados de los cálculos utilizando el argumento **out**

In [170]:
```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
```

```
[ 0. 10. 20. 30. 40.]
```

Para **ufuncs binarios**, hay algunos agregados interesantes que se pueden calcular directamente desde el objeto. Por ejemplo, si nos quisiéramos reducir una matriz con una operación en particular, podemos usar el método **reduce** de cualquier ufunc. Una reducción aplica repetidamente una operación determinada a los elementos de una matriz hasta que solo queda un único resultado

In [171]:
```
x = np.arange(1, 6)
x
```

Out[171]:
```
array([1, 2, 3, 4, 5])
```

In [172]:
```
np.add.reduce(x)
```

Out[172]:
```
15
```

In [173]:
```
np.multiply.reduce(x)
```

Out[173]:
```
120
```

In [174]:
```
np.add.accumulate(x)
```

Out[174]:
```
array([ 1,  3,  6, 10, 15], dtype=int32)
```

In [175]:
```
np.multiply.accumulate(x)
```

Out[175]:
```
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. Esto le permite, en una línea, hacer cosas como crear una tabla de multiplicar

In [176]:
```
x = np.arange(1, 6)
x
```

Out[176]:
```
array([1, 2, 3, 4, 5])
```

In [177]:
```
np.multiply.outer(x, x)
```

Out[177]:
```
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: Mín., Máx. y otros

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 [178]:
```
array = np.random.random(100)
array
```

Out[178]:
```
array([0.26036175, 0.36251954, 0.24607597, 0.3405522 , 0.11110959,
       0.60269475, 0.61865009, 0.75538646, 0.21131875, 0.94743664,
       0.58327345, 0.95220988, 0.08276258, 0.36903081, 0.76845001,
       0.77997844, 0.30460492, 0.06544506, 0.44411178, 0.96538938,
       0.65043192, 0.55158547, 0.06255588, 0.22581376, 0.23111567,
       0.36495238, 0.63490974, 0.57517261, 0.52022067, 0.28591858,
       0.85494659, 0.33020637, 0.38766016, 0.69344478, 0.15200542,
       0.81577264, 0.04108286, 0.57172516, 0.1456308 , 0.15779923,
       0.29965957, 0.50675182, 0.01259653, 0.11043666, 0.43856685,
       0.44360408, 0.7872315 , 0.31269627, 0.81003555, 0.63630676,
       0.21293868, 0.68421142, 0.2327092 , 0.537494  , 0.82545579,
       0.29162667, 0.72627629, 0.76149699, 0.38718089, 0.14008971,
       0.33518485, 0.47853527, 0.18708415, 0.42840501, 0.95740581,
       0.52817399, 0.82341091, 0.23442242, 0.07931714, 0.70827178,
       0.40364237, 0.4630678 , 0.30412999, 0.41637062, 0.40510931,
       0.84952853, 0.69710322, 0.24625053, 0.50611069, 0.59432267,
       0.47749644, 0.94526316, 0.34354643, 0.90386498, 0.03547314,
       0.16114879, 0.62648817, 0.03109119, 0.78028723, 0.83441774,
       0.05558148, 0.64266354, 0.80057866, 0.93193375, 0.10365397,
       0.3022184 , 0.00225331, 0.03005975, 0.51508507, 0.99034484])
```

In [179]:
```
sum(array)
```

Out[179]:
```
46.34097503154862
```

La sintaxis es similar a la de la función de suma de NumPy, y el resultado es el mismo

In [180]:
```
np.sum(array)
```

Out[180]:
```
46.34097503154864
```

Sin embargo, debido a que ejecuta la operación en código compilado, la versión de NumPy se calcula mucho más rápido

In [181]:
```
big_array = np.random.rand(1000000)
```

In [182]:
```
%timeit sum(big_array)
```

```
175 ms ± 17.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [183]:
```
%timeit np.sum(big_array)
```

```
1.75 ms ± 318 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [184]:
```
min(big_array), max(big_array)
```

Out[184]:
```
(1.4764038746006847e-06, 0.9999995138219229)
```

In [185]:
```
np.min(big_array), np.max(big_array)
```

Out[185]:
```
(1.4764038746006847e-06, 0.9999995138219229)
```

In [186]:
```
%timeit min(big_array)
```

```
134 ms ± 22.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [187]:
```
%timeit np.min(big_array)
```

```
965 µs ± 41.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Para min, max, sum y varios otros agregados NumPy, una sintaxis más corta es usar métodos del propio objeto

In [188]:
```
print(big_array.min(), big_array.max(), big_array.sum())
```

```
1.4764038746006847e-06 0.9999995138219229 499801.8847596226
```

Las funciones de agregación toman un argumento adicional que especifica el eje a lo largo del cual se calcula el agregado. Por ejemplo, podemos encontrar el valor mínimo dentro de cada columna especificando axis=0:

In [189]:
```
matriz = np.random.random((3, 4))
matriz
```

Out[189]:
```
array([[0.96067887, 0.50094468, 0.22768667, 0.26318141],
       [0.13800703, 0.57297291, 0.29738591, 0.57848494],
       [0.98223531, 0.98506125, 0.95141812, 0.28392943]])
```

In [190]:
```
matriz.min(axis=0)
```

Out[190]:
```
array([0.13800703, 0.50094468, 0.22768667, 0.26318141])
```

In [191]:
```
matriz.max(axis=1)
```

Out[191]:
```
array([0.96067887, 0.57848494, 0.98506125])
```

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

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.

Recordemos que para matrices del mismo tamaño, las operaciones binarias se realizan elemento por elemento

In [192]:
```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

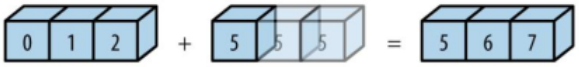
Out[192]:
```
array([5, 6, 7])
```

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 [193]:
```
a + 5
```

Out[193]:
```
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

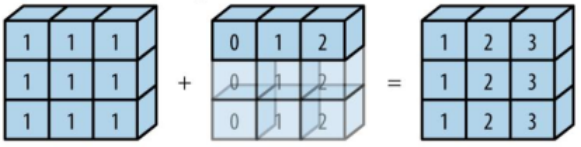
In [194]:
```
matriz = np.ones((3, 3))
matriz
```

Out[194]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

In [195]:
```
matriz + a
```

Out[195]:
```
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

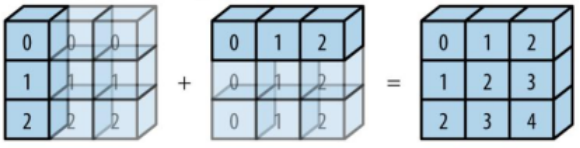
In [196]:
```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]
a, b
```

Out[196]:
```
(array([0, 1, 2]),
 array([[0],
        [1],
        [2]]))
```

In [197]:
```
a + b
```

Out[197]:
```
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 [198]:
```
matriz = np.ones((2, 3))
vector = np.arange(3)
matriz, vector
```

Out[198]:
```
(array([[1., 1., 1.],
        [1., 1., 1.]]),
 array([0, 1, 2]))
```

Las formas de las matrices son

In [199]:
```
matriz.shape, vector.shape
```

Out[199]:
```
((2, 3), (3,))
```

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

```
matriz.shape -> (2, 3)
vector.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

```
matriz.shape -> (2, 3)
vector.shape -> (2, 3)
```

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

In [200]:
```
matriz + vector
```

Out[200]:
```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

Ejemplo de transmisión 2

In [201]:
```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
a, b
```

Out[201]:
```
(array([[0],
        [1],
        [2]]),
 array([0, 1, 2]))
```

In [202]:
```
a.shape, b.shape
```

Out[202]:
```
((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 [203]:
```
a + b
```

Out[203]:
```
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 [204]:
```
matriz = np.ones((3, 2))
vector = np.arange(3)
matriz, vector
```

Out[204]:
```
(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 [205]:
```
matriz.shape, vector.shape
```

Out[205]:
```
((3, 2), (3,))
```

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

```
matriz.shape -> (3, 2)
vector.shape -> (1, 3)
```

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

```
matriz.shape -> (3, 2)
vector.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 [206]:
```
matriz + vector
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-206-2b5bfd13b797> in <module>
----> 1 matriz + vector

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 [207]:
```
vector[:, np.newaxis].shape
```

Out[207]:
```
(3, 1)
```

In [208]:
```
matriz + vector[:, np.newaxis]
```

Out[208]:
```
array([[1., 1.],
       [2., 2.],
       [3., 3.]])
```

Estas reglas de transmisión se aplican a cualquier ufunc binario

Centrar una matriz

Supongamos que se tiene una matriz de 10 observaciones, cada una de las cuales consta de 3 valores

In [209]:
```
X = np.random.random((10, 3))
X
```

Out[209]:
```
array([[0.35697451, 0.98050904, 0.86771305],
       [0.91279271, 0.61654084, 0.43668492],
       [0.70856622, 0.7023956 , 0.58838194],
       [0.58886633, 0.90542789, 0.11778138],
       [0.86081773, 0.6077116 , 0.01915647],
       [0.31441641, 0.72930665, 0.05819754],
       [0.10712474, 0.06261349, 0.38701826],
       [0.74290004, 0.05828458, 0.24740289],
       [0.61854159, 0.31713954, 0.78854518],
       [0.0740446 , 0.27684028, 0.17820678]])
```

Podemos calcular la media de cada característica utilizando la media agregada en la primera dimensión

In [210]:
```
Xmean = X.mean(0)
Xmean
```

Out[210]:
```
array([0.52850449, 0.52567695, 0.36890884])
```

Y ahora podemos centrar la matriz X restando la media (esta es una operación de transmisión)

In [211]:
```
X_centered = X - Xmean
X_centered
```

Out[211]:
```
array([[-0.17152998,  0.45483209,  0.49880421],
       [ 0.38428822,  0.09086389,  0.06777608],
       [ 0.18006173,  0.17671865,  0.2194731 ],
       [ 0.06036184,  0.37975094, -0.25112746],
       [ 0.33231324,  0.08203465, -0.34975237],
       [-0.21408808,  0.2036297 , -0.3107113 ],
       [-0.42137974, -0.46306346,  0.01810942],
       [ 0.21439555, -0.46739237, -0.12150595],
       [ 0.0900371 , -0.20853742,  0.41963634],
       [-0.45445989, -0.24883667, -0.19070206]])
```

Para comprobar que hemos hecho esto correctamente, podemos comprobar que la matriz centrada tiene una media cercana a cero

In [212]:
```
X_centered.mean(0)
```

Out[212]:
```
array([ 8.88178420e-17, -3.33066907e-17,  0.00000000e+00])
```

Con precisión dentro de la máquina, la media ahora es cero

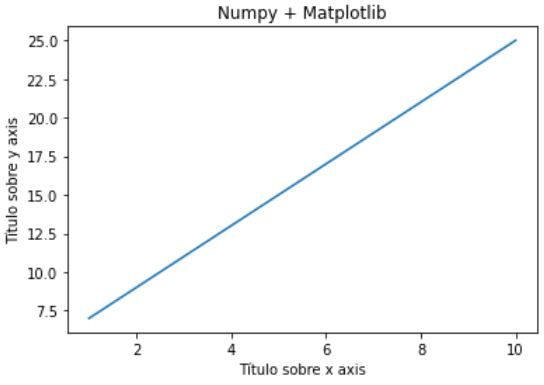
🤨 Gráficos: Numpy + Matplotlib

In [213]:
```
import matplotlib.pyplot as plt
import numpy as np
```

Ejemplo

In [214]:
```
plt.rcParams["figure.figsize"] = (6,4)

x = np.arange(1,11)
y = 2 * x + 5

plt.title("Numpy + Matplotlib")
plt.xlabel("Título sobre x axis")
plt.ylabel("Título sobre y axis")
plt.plot(x,y)
```

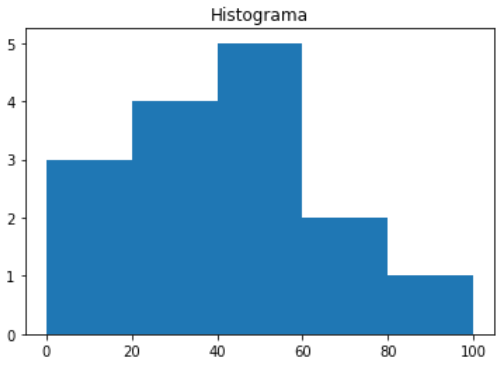
Out[214]:
```
[<matplotlib.lines.Line2D at 0x1afc6035a60>]
```



Ejemplo

In [215]:
```
a = np.array([22,87,5,43,56,73,55,54,11,20,51,5,79,31,27])

plt.rcParams["figure.figsize"] = (6,4)
plt.hist(a, bins = [0,20,40,60,80,100])
plt.title("Histograma")
```

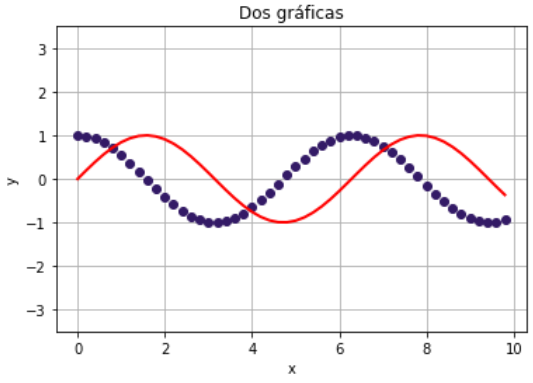
Out[215]: Text(0.5, 1.0, 'Histograma')



Ejemplo

In [216]:
```
x = np.arange(0,10,0.2)
y1 = np.cos(x)
y2 = np.sin(x)

plt.rcParams["figure.figsize"] = (6,4)
plt.plot(x,y1,'o',linewidth=3,color=(0.2,0.1,0.4))
plt.plot(x,y2,'-',linewidth=2,color='r')
plt.grid()
plt.axis('equal')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Dos gráficas')
```

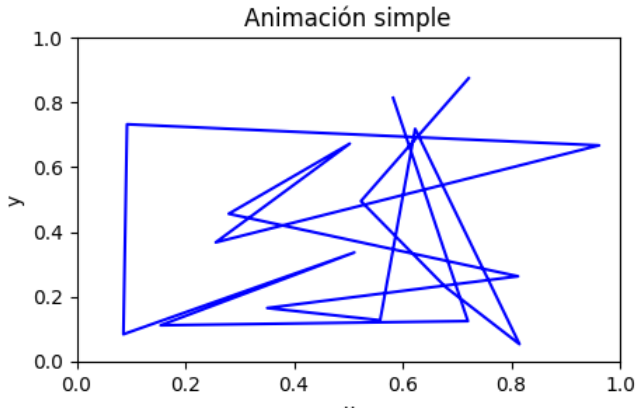
Out[216]: Text(0.5, 1.0, 'Dos gráficas')



Ejemplo

In [217]:
```
%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)
```

```
<IPython.core.display.Javascript object>
```



Ejemplo Creamos un archivo .csv con los siguientes datos:

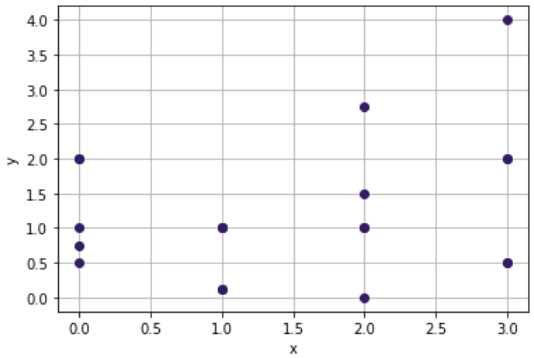
```
 2,0.75,2,1,0.5
1,0.125,1,1,0.125
2.75,1.5,1,0,1
4,0.5,2,2,0.5
```

In [218]:
```
csv_array = np.genfromtxt('archs/archivo.csv', delimiter = ',')
csv_array
```

Out[218]:
```
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 [219]:
```
%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')
```

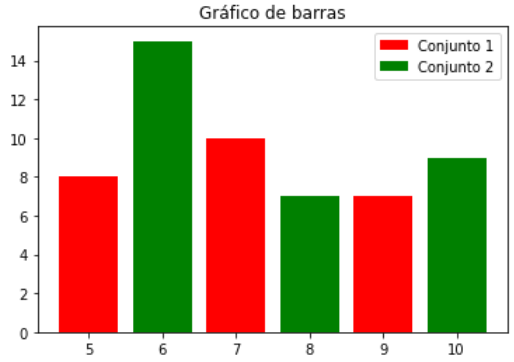
Out[219]:
```
Text(0, 0.5, 'y')
```



In [220]:

```
#primer conjunto de datos
x1 = np.array([5,7,9])
y1 = np.array([8,10,7])

#segundo conjunto de datos
x2 = np.array([6,8,10])
y2 = np.array([15,7,9])

plt.rcParams["figure.figsize"] = (6,4)
#barras para el primer conjunto
plt.bar(x1,y1, color='r',align='center')

#barras para el segundo conjunto
plt.bar(x2,y2, color='g',align='center')

#titulo
plt.title('Gráfico de barras')

#leyenda
plt.legend(['Conjunto 1', 'Conjunto 2'])
```

Out[220]: <matplotlib.legend.Legend at 0x1afc6322d90>

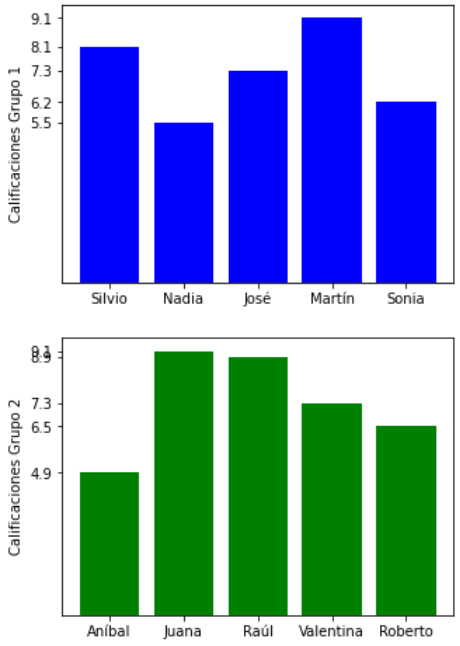


In [221]:

```
#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[221]:

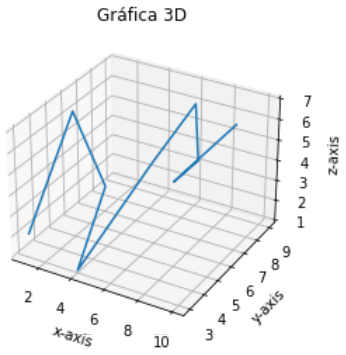
```
Text(0, 0.5, 'Calificaciones Grupo 2')
```



In [222]:

```
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[222]: 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)

In [ ]: