1 Numerical Python o NumPy

NumPy es una librería para realizar cálculo en el campo del algebra lineal. Originalmente fue derivado de un paquete de rutinas hecho para Fortran, conocido como BLAS.

NumPy es un paquete fundamental para utilizar python en computación de cálculos científicos. Esta es una librería que provee de objetos equivalentes a arreglos multidimensionales y todo un conjunto de rutinas para una operación rápida en los campos de la matemáticas, lógica, manipulación de formas, ordenamientos, entre/salida de datos, Trasformada discreta de Fourier, álgebra lineal básica, operaciones estadísticas básicas, simulaciones con números al azar y otros muchos temas.

Es una librería más importante en el uso científico de python.

Se puede conseguir muchas mas información en www.numpy.org

2 Numpy en un notebook

2.1 Importando NumPy

Como toda librería externa se debe cargar para que pueda ser usada por el notebook. El Python de por si no tiene esta librería en memoria, por lo cual, debe ser el usuario el que decida utilizarla y para ello debe "importarla". Al hacerlo cargará no sólo los algoritmos que lleva programados, sino además las Clases con las cuales se definene las variables propias de la librería. Los objetos que se usan en NumPy son básicametne arreglos que mantienen cierta similitud con los vectores y matrices del álgebra.

Primero tenemos que cargar la librería, es bastante popular llamar "np" a la libreria numPy. Aunque no es obligatorio llamarla como "np" ya es un estandard reconocido.

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

Este comando carga la librería NumPy, pero con un cambio de nombre, la traducción literal del comando sería "Carga NumPy como np". desde esta manera Numpy de ahora en adelante se llamará np en mi programa. Es normal renombrarla como "np", aunque nada evita que se pudiera haber elegido cualquier otro nombre.

2.2 La clase de los arreglos (array)

NumPy usa sus propias variables, es decir como Python es un lenguaje orientado a objetos se definió una clase para generar objetos con las propiedades necesarios para hacer álgebra vectorial. Es decir uso de vectores, matrices, cubos, etc, con mucha eficiencia y respetando propiedades del álgebra. Básicamente NumPy utiliza arreglos que son propios de su librería

2.2.1 Veamos el uso de la función array( ) que crea objetos NumPy a partir de variables de Python

Puedo crear un arreglo numpy tomándolo de una lista o tupla, o creándolos nuevos.

Veamos el ejemplo de usar una lista y con ella crear un arreglo NUmpy.

Para ello uso la función array de NumPy, y como a NumPy lo llamé "np", a la función la tengo que llamar np.array( )

El arreglo NumPy creado ya no es una lista (aunque se le parece mucho), tiene otras propiedades.

[2]:
```
a_lista = [1,2,3,4,5,6]

a = np.array(a_lista)

print(a)
```

```
[1 2 3 4 5 6]
```

Notar que ahora no tenemos las "," típicas de una lista.

O podría hacer directamente

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

Si pregunto el tipo de variable que tengo ahora

[4]:
```
print("Es de la clase:", type(a))
```

```
Es de la clase: <class 'numpy.ndarray'>
```

[5]:
```
# funciona también con tuplas:

b = np.array((1,2,3))

print(b)
print("Es de la clase:",type(b)) # Ya NO es una Tupla.... y además
                                 # desaparecen las comas de la lista o tupla
```

```
[1 2 3]
Es de la clase: <class 'numpy.ndarray'>
```

[6]:
```
# Creo una lista
L = [1, 2, 3, 4]
# Convierto mi lista a un arreglo NumPy
a = np.array(L)

print("La lista L es del tipo: ",type(L))
print("a es de tipo:",type(a))
print("El arreglo a es del tipo:",a.dtype)
print("")
print(L)
print(a)
```

```
La lista L es del tipo:  <class 'list'>
a es de tipo: <class 'numpy.ndarray'>
El arreglo a es del tipo: int64

[1, 2, 3, 4]
[1 2 3 4]
```

El dtype es un comando de Numpy. Me indica que el arreglo “a” no es una lista de Python. dtype me indica el tipo de arreglo dentro de las posibilidades de arreglos Numpy. En este caso es tipo Integer*8 (64 bits).

A diferencia de las listas los arreglos Numpy engloban a variables que son del mismo tipo. Es decir que todo el arreglo Numpy es del mismo tipo de variable (Real*8, Int*8, complejo, etc)

Una vez definido, todos los agregados se modifican para que sean válidos en el tipo de arreglo final.

[7]:
```
Lista2=[1,2,22.] # <-- 1 y 2 enteros, 22. es real
Lista_numpy=np.array(Lista2)

print(Lista_numpy)
```

```
[ 1.  2. 22.]
```

Todo el arreglo terminó siendo real.

Los arreglos NumPy son estructuras donde los cálculos son rápidos y eficientes.

Para ver la ventaja real de esta librería usaremos el comando del notebook %timeit (que es una función del notebook, no es una orden de Python) que me permite tomar tiempos.

range es una función de python.

arange es una función que se agrega porque viene con numPy. Esta última crea los mismos valores de la lista, pero ahora en array NumPy

[8]:
```
L = range(1000)  # Crea una lista

%timeit for L in range(1000):    A=L**2

print('Pero usando NumPy')

A = np.arange(1000)  # Crea un arreglo Numpy

%timeit A2 = A**2
```

```
191 µs ± 4.53 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
Pero usando NumPy
1.12 µs ± 25.2 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

2.2.2 Moraleja 1:

La idea que hay que tener presente cuando se usa NumPy, es que se va a trabajar con arreglos como variables. No hay que pensar en escribir código que apunte a algoritmos donde se trabaja con los elementos de a uno en un arreglo. El código se escribe pensando las operaciones matemáticas que se realizan sobre vectores y matrices. Veremos esto con detalle durante el curso.

Los arreglos NumPy llevan consigo al ser objetos, atributos y funciones. Estas funciones en particular son mucho más eficientes que las originales de Python. Probemos creando un arreglo usar primero la función que busca el mínimo valor de Python (min(a)) y luego lo que hace el mismo trabajo pero con la función

que reside en el objeto NumPy (a.min()). Noten que tiene el mismo nombre pero que se las llama para su uso de forma diferente.

[9]:
```
# construyo un vector de 10.000.000 números con primer valor 1000 y último 0,␣
 ↪con paso -0.00001

a = np.arange(1000,0,-0.0001)

# veo que tengo en a
print("Largo de a: ",len(a))

%timeit min(a)
%timeit a.min()
```

```
Largo de a:  10000000
868 ms ± 28.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
4.51 ms ± 46.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

2.2.3 Moraleja 2:

Hay que usar las funciones que vienen en las librerías que son especifícas para un tema particular, en el caso de álgebra vectorial (como en este último ejemplo) es prácticamente obligatorio usar las funciones de los objetos NumPy.

2.2.4 Pueden ser de varias dimensiones 1D, 2D, 3D, . . .

[10]:
```
a = np.array([1,2,3,4,5,6])
b = np.array([[1,2],[1,4]])
c=np.array([[[1,2], [2,3]], [[3,4], [4,5]]])

print("Imprimo el atributo shape")
print( a.shape, b.shape, c.shape)
print("")
print("Pero si me fijo el len()")
print(len(a),len(b), len(c))
```

```
Imprimo el atributo shape
(6,) (2, 2) (2, 2, 2)

Pero si me fijo el len()
6 2 2
```

El atributo shape me cuenta de las dimensiones del arreglo, en cambio la función len( ) me cuenta de su largo.

El comando shape me devuelve una **tupla** con la información, en cambio len( ) me devuelve un número.

Si tengo varias dimensiones y quiero saber la cantidad total de elementos es conveniente imprimir el atributo size.

Veamos como imprimimos el array c y sus componentes ya que este arreglo es trimensional.

[11]:
```
print(c[0])
```

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

[12]:
```
print(c[0][0])
```

```
[1 2]
```

[13]:
```
print(c[0][0][0])  #<-- recién aquí me devuelve un número
```

```
1
```

[14]:
```
# size me da la cantidad de valores en el array
b.size
```

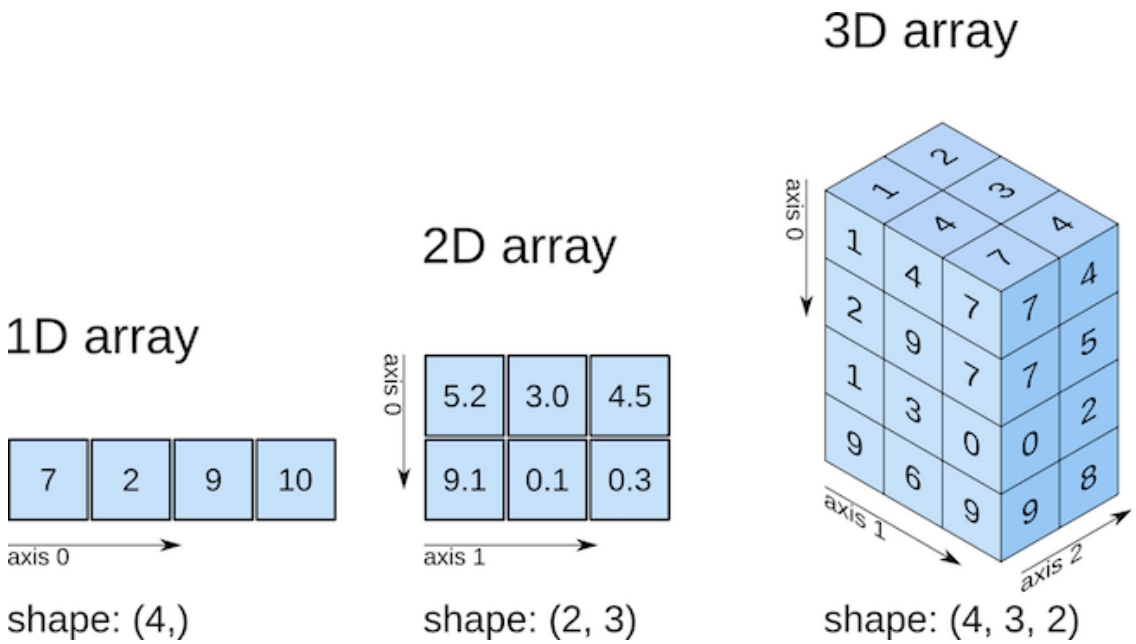
[14]: 4

Pero estrictamente la dimensión me la da el atributo ndim

[15]:
```
print (a.ndim, b.ndim, c.ndim )
```

```
1 2 3
```

La forma como NumPy numera los ejes, se conoce como el estilo “C” (por el lenguaje C que lo hace de esa manera) y es diferente al estilo Fortran. Aunque siempre es más cómodo visualizar las distintas dimensiones como *arreglos de arreglos de otros arreglos*. . .



Crédito del dibujo:https://www.tomasbeuzen.com/python-programming-for-data-science/chapters/chapter5-numpy.html

---

La librería numPy trae funciones asociadas.

Las puedo buscar en el manual (www.numpy.org), veamos algunos ejemeplos:

[16]:
```
a = np.array([1,2,3,4,5,6,7])
```

Por ejemplo, probemos con dos, la que calcula promedios de un array a.mean( ) y la que me encuentra el valor máximo a.max( )

[17]:
```
print(a.mean(), a.max())
```

```
4.0 7
```

mean( ) y max( ) son métodos (funciones) de la clase del arreglo, necesitan los paréntesis ( ).

[18]:
```
print(a.mean) # Esto así imprime de que trata la funcion no su resultado
```

```
<built-in method mean of numpy.ndarray object at 0x14d72c976c10>
```

[19]:
```
print ("Imprimo b: ",b)
print ("Imprimo el promedio de b: ",b.mean()) # Promedio sobre todo el arreglo
print ("Imprimo el promedio de las columnas de b: ",b.mean(axis=0)) # Promedio␣
 ↪sobre el primer eje (columnas)
print ("Imprimo el promedio de las filas de b: ", b.mean(1)) # Promedio sobre␣
 ↪las filas, se sobreentiende que es "axis=1"
```

```
Imprimo b:  [[1 2]
 [1 4]]
Imprimo el promedio de b:  2.0
Imprimo el promedio de las columnas de b:  [1. 3.]
Imprimo el promedio de las filas de b:  [1.5 2.5]
```

3 Creando arreglos usando sentencias específicas

La idea es que el arreglo reciba una cantidad de elementos creados en una forma automática y que cumplen reglas que se especifican como bucles. Es decir con un inicio, un final y un paso. Donde este último no tiene que ser obligtoriamente lineal. Por ejemplo, podríamos tener un paso geométrico, logarítmico, exponencial, etc.

También veremos que existen órdenes que copian la forma de otro arreglo, pero no sus elementos, sólo su forma, es decir su tamaño y dimensión.

[20]:
```
print(np.arange(10))
```

```
[0 1 2 3 4 5 6 7 8 9]
```

[21]:
```
print(np.linspace(0, 1, 10)) # Comienzo, final, y número de puntos
```

```
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
```

[22]:
```
print(np.linspace(0, 1, 10, endpoint=False)) # No incluimos el punto final, o lo␣
↪hacemos hasta 11.
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

[23]:
```
print(np.zeros(2)) # Lleno con ceros
```

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

[24]:
```
print(np.zeros((2,2))) # es un arreglo de dos dimensiones lleno con ceros
```

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

[25]:
```
print(np.ones_like(a)) # Esto es muy bueno, creo un arreglo (o tupla)
                       # usando las propiedades de otro que ya tengo
```

```
[1 1 1 1 1 1 1]
```

[26]:
```
print(np.zeros_like(a)+3) # Me sirve para llenar con otro valor que no son ceros␣
↪o 1's
```

```
[3 3 3 3 3 3 3]
```

[27]:
```
print(np.ones_like([1,2,3]))
```

```
[1 1 1]
```

3.0.1 Puedo crear arreglos vacíos de números pero con la forma final que deben tomar.

[28]:
```
a22=np.empty([2, 2])
a11=np.empty([10])
```

[29]:
```
print(np.logspace(0, 2, 10)) # va de 10**comienzo hasta 10**final y calculo 10␣
↪valores
```

```
[  1.           1.66810054   2.7825594    4.64158883   7.74263683
  12.91549665  21.5443469   35.93813664  59.94842503 100.        ]
```

Una manera muy típica de trabajar en python es primero crea el array como unidimensional y luego darle forma con el comando **reshape( )**, que lo vimos en Fortran 90.

[30]:
```
a = np.array([1,2,3,4,5,6])
b = a.reshape((3,2)) # Este comando no cambia la "forma" de  a
```

```
print(a)
print('')
print(b)
```

```
[1 2 3 4 5 6]

[[1 2]
 [3 4]
 [5 6]]
```

Mientras que la función **ravel( )** lo vuelve unidimensional.

[31]:
```
print(b.ravel())
```

```
[1 2 3 4 5 6]
```

3.0.2 OJO - CUIDADO, los arreglos son objetos y cuando se reasignan no se copian, entonces apuntan a la misma memoria

[32]:
```
b = a.reshape((3,2))
c = a.reshape((3,2))
print(a.shape, b.shape)
```

```
(6,) (3, 2)
```

[33]:
```
print(b)
print("")
b[1,1] = 100 # modifico un valor del arreglo
print(b)
```

```
[[1 2]
 [3 4]
 [5 6]]

[[  1   2]
 [  3 100]
 [  5   6]]
```

[34]:
```
print(a) # !!! a y b son el mismo objeto comparten el mismo espacio de memoria,
         # es decir apuntan a los mismos valores
```

```
[  1   2   3 100   5   6]
```

[35]:
```
print(b[1,1],a[3]) # el mismo valor
```

```
100 100
```

[36]:
```
c = a.reshape((2,3)).copy() # Esta es la solución
```

[37]:
```
print(a)
print('')
print(c)
```

```
[  1   2   3 100   5   6]

[[  1   2   3]
 [100   5   6]]
```

[38]:
```
c[0,0] = 8888
print(a)
print(c)
```

```
[  1   2   3 100   5   6]
[[8888    2    3]
 [ 100    5    6]]
```

3.0.3 Números al azar (random) en Numpy

Tengo funciones específicas ya programadas en NumPy para realizar la tarea.

[39]:
```
ran_uniform = np.random.rand(5) # Entre 0 y 1
ran_normal = np.random.randn(5) # Gausiana promedio 0 varianza 1
print("Números con distribución Uniforme: ",ran_uniform)
print ('')
print ("Números con distribución Gaussiana: ",ran_normal)
print ('')
ran_normal_2D = np.random.randn(5,5) # Gausiana promedio 0 varianza 1
print("Números con distribución Gaussiana, pero ahora en una matriz:␣
 ↪\n\n",ran_normal_2D)
```

```
Números con distribución Uniforme:  [0.0811311  0.75597703 0.27802981 0.2280084
0.23190263]

Números con distribución Gaussiana:  [-1.50575338 -0.90617856 -0.17009338
-0.22264365 -0.37075671]

Números con distribución Gaussiana, pero ahora en una matriz:

 [[-2.16126674 -0.06864267 -1.1587345   0.33457461  0.16279986]
 [ 0.51610614  0.55802463 -0.30501822  0.38553794 -0.68100637]
 [ 0.34532073 -1.18763026  1.22090744 -1.49297653 -1.16314819]
 [-1.47282281 -0.24667535  1.25375259 -0.13985277 -1.25335474]
 [ 0.08598072 -1.66795764 -0.43849998 -0.4110718   0.23842044]]
```

Si en la celda siguiente elimino el # en la línea y corro la celda me da como resultado un “help” del comando. Estas ayudas que se agregan en las funciones son los comentarios que se agregaron en el

código de la función como docstrings.

[40]:
```
#np.random.randn?
```

3.0.4 Slicing (cortando fetas)

Es tal como hemos visto con los comandos originales de Python. El rabanado de un objetos de Numpy se hace con los mismos comandos.

[41]:
```
a = np.arange(10)
print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

[42]:
```
print(a[1:8:3])
```

```
[1 4 7]
```

3.0.5 Usando arreglos y máscaras

Puedo en NumPy cambiar varios números en una sólo línea con un lista de estos elementos. Veamos un ejemplo:

[43]:
```
print(a)
a[[2,4,6]] = -999   # Puse como índices en el arreglo a, una lista de elementos␣
 ↪2,4,6
print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
[   0    1 -999    3 -999    5 -999    7    8    9]
```

Veamos como construir un arreglo NumPy donde sus elementos son variables lógicas

[44]:
```
a = np.random.randint(0, 100, 20) # min, max, N
print(a)
```

```
[40 80 70 44 79 65 12 97 72 63 50 80 42 95  3 64 69 15 56 16]
```

[45]:
```
a < 50
```

[45]:
```
array([ True, False, False,  True, False, False,  True, False, False,
       False, False, False,  True, False,  True, False, False,  True,
       False,  True])
```

La respuesta al operador de relación fue un arreglo con valores booleanos, es decir es la respueta si es verdadera o falsa la pregunta que se realizó en cada elemento.

Veamos esto con más detalle y como usarlo en un caso real:

4 Operaciones con arreglos - Filtrando con máscaras

Utilizo el arreglo a del ejemplo anterior:

[46]:
```
a
```

[46]:
```
array([40, 80, 70, 44, 79, 65, 12, 97, 72, 63, 50, 80, 42, 95,  3, 64, 69,
       15, 56, 16])
```

Y lo uso para realizar operacion matemáticas simples:

[47]:
```
a + 1
```

[47]:
```
array([41, 81, 71, 45, 80, 66, 13, 98, 73, 64, 51, 81, 43, 96,  4, 65, 70,
       16, 57, 17])
```

Hay que recordar que ahora la variable para los cálculos es todo el arreglo. Y podemos hacer operaciones bastante más complejas, como la que sigue:

[48]:
```
a**2 + 3*a**3
```

[48]:
```
array([ 193600, 1542400, 1033900,  257488, 1485358,  828100,    5328,
       2747428, 1124928,  754110,  377500, 1542400,  224028, 2581150,
            90,  790528,  990288,   10350,  529984,   12544])
```

Creo un arreglo más pequeño:

[49]:
```
a = np.arange(10)
print("a es:",a)
```

```
a es: [0 1 2 3 4 5 6 7 8 9]
```

Y realizo una operación más compleja y asigno el resultado a otro arreglo que llamo “b”

[50]:
```
b = a**2 + (a+1)**2
print("b es:",b)
```

```
b es: [  1   5  13  25  41  61  85 113 145 181]
```

Hago otra operación a la que asigno el resultado al arreglo “c”

[51]:
```
c = (a+2)**2
```

[52]:
```
print("b es:",b)
print("c es:",c)
```

```
b es: [  1   5  13  25  41  61  85 113 145 181]
c es: [  4   9  16  25  36  49  64  81 100 121]
```

Podría entonces preguntar que valores están repetidos en el mismo lugar en los arreglos b y c

[53]:
```
print(b == c)
```

```
[False False False  True False False False False False False]
```

Guardo el resultado en el arreglo “máscara”

[54]:
```
mascara = b==c
print("mascara es:", mascara)
```

```
mascara es: [False False False  True False False False False False False]
```

¿Y si la máscara la uso para filtrar el vector? ¿Cómo funciona eso?

Es decir pido que imprima b[mascara], pero mascara ahora funciona como un arreglo de los índices de b al poner de esa manera el comando. Sólo pasarán el filtro los elementos verdaderos.

[55]:
```
print("b es:",b)
print("b filtrado por la mascara es: ",b[mascara])
```

```
b es: [  1   5  13  25  41  61  85 113 145 181]
b filtrado por la mascara es:  [25]
```

También podría haber usado un expresión más compleja

[56]:
```
mascara= b < 15
print(mascara)
```

```
[ True  True  True False False False False False False False]
```

Podría entonces usar el arreglo máscara para filtrar los valores que cumplen con la condición solicitada.

[57]:
```
d=b[mascara]
print(d)
```

```
[ 1  5 13]
```

Incluso podría haber hecho todo esta operación en un sólo comando:

[58]:
```
print(b>c)
print(a[b>c])
```

```
[False False False False  True  True  True  True  True  True]
[4 5 6 7 8 9]
```

Cuidado que esto último lo puedo hacer porque a,b y c tienen el mismo tamaño y los elementos se corresponden en la numeración

NumPy maneja casi todas las expresiones matemáticas, log, trigonométricas, etc

[59]:
```
a = np.arange(18)
print("a es:",a)
print("")
print("El log10 de a es:",np.log10(a))
```

```
a es: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17]

El log10 de a es: [      -inf 0.         0.30103    0.47712125 0.60205999
0.69897
 0.77815125 0.84509804 0.90308999 0.95424251 1.         1.04139269
 1.07918125 1.11394335 1.14612804 1.17609126 1.20411998 1.23044892]

/home/carlos/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:4:
RuntimeWarning: divide by zero encountered in log10
  after removing the cwd from sys.path.
```

Notar que hubo un error al sacar el log10 de 0. Pero el programa siguió corriendo.

Ese error fue notificado en el cartel que sale con el “RuntimeWarning”.

Podemos encontrarnos elementos que dieron errores y estos pueden ser: -inf, +inf, y NaN.

5 Operaciones con vectores y matrices

[60]:
```
a = [[1, 0], [0, 1]]
b = [[4, 1], [2, 2]]

c = np.matmul(a, b)

print("c es:",c)
```

```
c es: [[4 1]
 [2 2]]
```

[61]:
```
a = [[1, 0], [0, 1]]
b = [1, 2]
c= np.matmul(a, b)

print(c)

c= np.matmul(b, a)
print(c)
```

```
[1 2]
[1 2]
```

[62]:
```
a = np.arange(1000000).reshape(1000,1000)
b = np.ones_like(a)
print(a)
print("")
print(b)
```

```
[[     0      1      2 ...    997    998    999]
 [  1000   1001   1002 ...   1997   1998   1999]
```

```
 [  2000   2001   2002 ...   2997   2998   2999]
 ...
 [997000 997001 997002 ... 997997 997998 997999]
 [998000 998001 998002 ... 998997 998998 998999]
 [999000 999001 999002 ... 999997 999998 999999]]

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

[63]:
```
c=np.matmul(a,b)
print(c)
```

```
[[   499500    499500    499500 ...    499500    499500    499500]
 [  1499500   1499500   1499500 ...   1499500   1499500   1499500]
 [  2499500   2499500   2499500 ...   2499500   2499500   2499500]
 ...
 [997499500 997499500 997499500 ... 997499500 997499500 997499500]
 [998499500 998499500 998499500 ... 998499500 998499500 998499500]
 [999499500 999499500 999499500 ... 999499500 999499500 999499500]]
```

6 Solución de un sistema lineal de ecuaciones

A = floor(random.rand(4000,4000)*20-10) <– genera una matriz de 4000 x 4000 valores llena con números al azar

[64]:
```
A = np.random.rand(4000,4000)
print("Imprimo A")
print (A)

b = np.floor(np.random.rand(4000,1)*20-10)
print("")
print("Imprimo b")
print(b)

# Resolvamos Ax = b usando el comando NumPy para la tarea:
x = np.linalg.solve(A,b)

print("")
print("Imprimo el resultado")
print(x)
```

```
Imprimo A
[[0.49581908 0.20872043 0.78153798 ... 0.14717675 0.09790914 0.78191698]
```

```
 [0.34291123 0.88346862 0.54834274 ... 0.6450303  0.83337163 0.09230528]
 [0.13789077 0.5060304  0.97280969 ... 0.41948824 0.5654873  0.63943   ]
 ...
 [0.76218377 0.58580148 0.2721973  ... 0.1214925  0.63820361 0.35963161]
 [0.98309218 0.37306244 0.57645748 ... 0.938292   0.02599148 0.19659851]
 [0.83350541 0.48478324 0.90017178 ... 0.08599545 0.15524257 0.29067635]]

Imprimo b
[[ 7.]
 [ 4.]
 [-8.]
 ...
 [ 0.]
 [-8.]
 [ 1.]]

Imprimo el resultado
[[ 3.37526956]
 [-0.98691674]
 [-1.31103021]
 ...
 [ 5.18041178]
 [10.07516089]
 [11.16027651]]
```

7 Broadcasting

El broadcasting es la manera que tiene Python de compensar operaciones elemento a elemento entre arreglos de diferentes tamaños.
La situación más simple es cuando se realiza una operación entre un arreglo por un escalar. El escalar en virtualmente repetido todas las veces necesarias para repetir el tamaño del escalar.

Se suele considerar que el broadcasting es un fuerte consumidor de memoria de la computadora.

[65]:
```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
```

[65]:
```
array([2., 4., 6.])
```

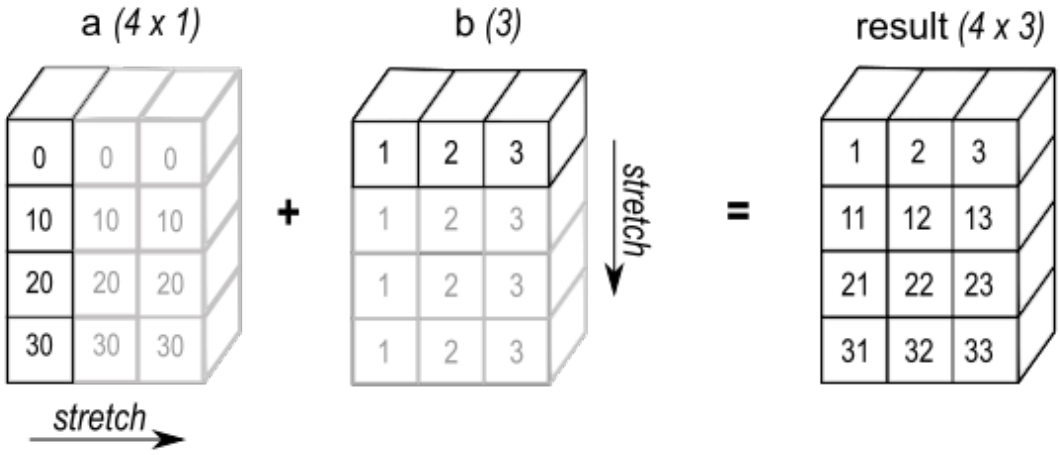
En este caso hubiese sido similar a que conviertiera b en b=np.array([2.0, 2.0, 2.0])

[66]:
```
a = np.array([1.0, 2.0, 3.0])
b = b=np.array([2.0, 2.0, 2.0])
a * b
```

[66]:
```
array([2., 4., 6.])
```

[67]:
```
a = np.array([[0.0], [10.0], [20.0], [30.0]])
b = np.array([1.0, 2.0, 3.0])
a+b
```

[67]:
```
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])
```



Moraleja: Los arreglos sirven para hacer álgebra vectorial, pero NO son vectores, ni matrices...

No tienen las mismas propiedades...

El broadcast sólo se puede realizar si:

- Los arreglos son iguales o
- uno de ellos tiene dimensión 1 arreglos son iguales

7.0.1 Forma de hacer las cosas con NumPy

Veamos como hay que razonar para usar NumPy y de paso calculemos PI (una vez más...)

Esta vez usando series de Taylor. En este caso arctan(1) como hizo en su momento Leibniz

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

Veamos un la forma de plantearlo en Python, y usando NumPy:

Por lo que calculo la serie de dos maneras, primero estilo Fortran: término por término de la serie mientras los sumo.

La segunda manera en modo NumPy, todos los términos al mismo tiempo, creando un vector de términos y que luego sumo.

La función pow(x, y) is igual a hacer el cálculo x^y, y funciona mejor que x ∗ ∗y, porque permite valores fracionarios y negativos de y.

[68]:
```
# Estilo Fortran
n = 10000000
total = 0
for k in range(n):
    total = total + pow(-1,k)/(2*k+1.0)
print("Cálculo un elemento y lo voy sumando              :",total*4)
```

```
Cálculo un elemento y lo voy sumando              : 3.1415925535897915
```

[69]:
```
#Estilo NumPy
k=np.arange(n)
term=pow(-1,k)/(2*k+1.0)
total=term.sum()
print("Todos los términos al mismo tiempo y sumo al final:", total*4)
```

```
Todos los términos al mismo tiempo y sumo al final: 3.1415925535897977
```

Noten que la forma NumPy es mucho más veloz.

8 Resumen de lo importante para usar NumPy

- Python con NumPy es muy bueno, pero hay que pensar en términos de matrices (o arreglos). Nadie usa Python sin usar NumPy en el área científica.
- Para usar NumPy en forma eficiente y que valga la pena, es necesario que al algoritmo a programar sea posible describirlo en términos de álgebra vectorial.
- Puede paralelizarse para soportar un hardware específico (GPUs o TPUs) en varias encarnaciones, aunque aún no hay una versión oficial.
- Queda claro que como existen una cantidad importante de órdenes, uno sólo necesita aprender los comandos básicos y para alguna tarea muy específica hay leer la documentación primero para verificar si el comando ya existe.
- En soporte a la idea anterior, existen demasiados comandos, puede ser una mala idea pensar que se pueden aprender todos.
- Se siguen hoy en día agregando y modificando comandos. A NumPy aún se lo continúa construyéndo.