NumPy

Cursada 2021 - Computación FACG/UNLP - Carlos Feinstein

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

Se caracteriza por proveer objetos que puedn ser arreglos multidimensionales y por tener consigo una colección de rutinas de alta velocidad con el fin de la manipulación y cálculo de álgebra lineal, transformadas de Fourier, números al azar, ordenamiento, estadística básica, y muchas funciones más.

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

Mucha info en www.numpy.org

Importemos NumPy para que pueda ser usado por el notebook Python

In [1]:
```
# Primero tenemos que cargar la librería, es bastante popular llamar "np" a la numpy:

import numpy as np
```

Llamar a librería numpy como "np" es un estandard, todo el mundo le pone "np". Aunque se podría haber elegido cualquier otro nombre.

Para estar seguro que tengo la última versión, o si quiero ver que versión tengo en la computadora, pongo el siguiente comando:

In [2]:
```
print(np.__version__)
```

```
1.19.2
```

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.

Creando arreglos...

In [3]:
```
# Puedo crear un array numpy de una lista ----> Ojo es un ARRAY NUMPY ahora no una lista
a = np.array([1,2,3,4,5,6])

print(a)
print(type(a))

#funciona también con tuplas:
b = np.array((1,2,3))

print(b)
print(type(b)) # Ya NO es una Tupla....
```

```
[1 2 3 4 5 6]
<class 'numpy.ndarray'>
[1 2 3]
<class 'numpy.ndarray'>
```

Los arreglos numpy son estructuras donde los cálculos son rápidos y eficientes

Uso el decorador %timeit que me permite tomar tiempos.

range es una función de python.

arange es una función que se agrega porque viene con numpy.

In [4]:
```
L = range(1000)

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

print('Pero usando NumPy')

A = np.arange(1000)
%timeit A2 = A**2
```

```
286 µs ± 35.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
Pero usando NumPy
1.23 µs ± 80.2 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [9]:
```
L = [1, 2, 3, 4]
a = np.array(L)
print(a.dtype)
print(a)
```

```
# el dtype es de Numpy, y no es el type de Python
```

```
int64
[1 2 3 4]
```

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

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

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

print( a.shape, b.shape, c.shape)
print(a.shape)
```

```
(6,) (2, 2) (2, 2, 1)
(6,)
```

In [11]:
```
print (len(a), len(b), len(c)) # tamaño de la primera dimensión
```

```
6 2 2
```

In [14]:
```
b.size
```

Out[14]:
```
4
```

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

```
1 2 3
```

La librería numpy trae funciones asociadas.

Las puedo buscar en el manual (www.numpy.org)

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

```
4.0 7 (7,)
```

mean y max son métodos (funciones) de la clase del arreglo, necesitan los paréntesis (). Shape en cambio es un atributo.

In [17]:
```
print(a.mean) # Esto así imprime de que trata la funcion no su resultado
```

```
<built-in method mean of numpy.ndarray object at 0x7fed62de0760>
```

In [18]:
```
print (b)
print (b.mean()) # Promedio sobre todo el arreglo
print (b.mean(axis=0)) # Promedio sobre el primer eje (columnas)
print (b.mean(1)) # Promedio sobre las filas, se sobreentiende que es "axis=1"
```

```
[[1 2]
 [1 4]]
2.0
[1. 3.]
[1.5 2.5]
```

Creando arreglos desde el scratch (sería como decir desde el comienzo)

In [19]:
```
print(np.arange(10))
```

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

In [20]:
```
print(np.linspace(0, 1, 10)) # Comienzo, final, y número de puntos
print('---------------------------------')
print(np.linspace(0, 1, 10, endpoint=False)) # No incluimos el punto final,
```

```
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
---------------------------------
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

In [21]:
```
print(np.zeros(2)) # Lleno con zeros
print('---------------------------------')
print(np.zeros((2,2))) # es un arreglo de dos dimensiones lleno con ceros
print('---------------------------------')
print(np.ones_like(a)) # Esto es muy bueno, creo un arreglo (o tupla) usando las propiedades de otro que ya tengo
print('---------------------------------')
print(np.zeros_like(a)+3) # Me sirve para lenar con otro valor que no son ceros o 1s
print('---------------------------------')
print(np.ones_like([1,2,3]))
```

```
[0. 0.]
---------------------------------
[[0. 0.]
 [0. 0.]]
---------------------------------
[1 1 1 1 1 1 1]
---------------------------------
[3 3 3 3 3 3 3]
---------------------------------
[1 1 1]
```

In [22]:
```
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.        ]
```

In [23]:
```
a = np.array([1,2,3,4,5,6])
b = a.reshape((3,2)) # This does NOT change the shape of a
print(a)
print('-------------')
print(b)
```

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

In [24]:
```
print(b.ravel())
```

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

OJO - CUIDADO, los arreglos comparten memoria - WARNING arrays share memory

In [25]:
```
b = a.reshape((3,2))
c = a.reshape((3,2))
print(a.shape, b.shape)
```

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

In [26]:
```
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]]
```

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

In [28]:
```
print(b[1,1],a[3]) # el mismo valor
```

```
100 100
```

In [29]:
```
b is a # a y b son diferentes
```

Out[29]:
```
False
```

In [30]:
```
c = a.reshape((2,3)).copy() # Esta es la solución
```

In [32]:
```
print(a)
print(c)
```

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

In [33]:
```
c[0,0] = 8888
print(a)
print(c)
```

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

Random

In [34]:
```
ran_uniform = np.random.rand(5) # Entre 0 y 1
ran_normal = np.random.randn(5) # Gausiana promedio 0 varianza 1
print(ran_uniform)
print ('-----------------------------')
print (ran_normal)
print ('-----------------------------')
ran_normal_2D = np.random.randn(5,5) # Gausiana promedio 0 varianza 1
print(ran_normal_2D)
```

```
[0.94167538 0.33020713 0.73333124 0.10457935 0.67881422]
-----------------------------
[ 0.21070602  1.44546101 -0.22192138 -0.48030352  0.92547449]
-----------------------------
[[ 0.26267193  0.09359075 -0.02646266 -1.2031766  -0.85154595]
 [ 0.00282569  0.13273235 -0.17034849 -1.21160469 -0.31244954]
 [ 1.11037684 -1.09638183 -0.20347803 -0.07240237  0.3873073 ]
```

```
[-0.77629442 -1.06225285  0.59904299  0.58597344  1.03304104]
[ 0.11103272 -0.64771423  1.05484691  0.54722373 -0.16145073]]
```

In [35]:
```
%np.random.randn?
```

Slicing (cortando fetas)

In [36]:
```
a = np.arange(10)
print(a)
# print(a[1:8:3])
```

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

In [37]:
```
print(a[1:8:3])
```

```
[1 4 7]
```

Usando arreglos y máscaras

In [38]:
```
print(a)
a[[2,4,6]] = -999
print(a)
```

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

In [39]:
```
a = np.random.randint(0, 100, 20) # min, max, N
print(a)
```

```
[ 1 50 83 91 11 93 54 87 80 34 19 16 89 86  1 40 96 74 51 72]
```

In [40]:
```
a < 50
```

Out[40]:
```
array([ True, False, False, False,  True, False, False, False, False,
        True,  True,  True, False, False,  True,  True, False, False,
       False, False])
```

Operaciones con arreglos

In [41]:
```
a
```

Out[41]:
```
array([ 1, 50, 83, 91, 11, 93, 54, 87, 80, 34, 19, 16, 89, 86,  1, 40, 96,
       74, 51, 72])
```

In [42]:
```
a + 1
```

Out[42]:
```
array([ 2, 51, 84, 92, 12, 94, 55, 88, 81, 35, 20, 17, 90, 87,  2, 41, 97,
       75, 52, 73])
```

In [43]:
```
a**2 + 3*a**3
```

Out[43]:
```
array([      4,  377500, 1722250, 2268994,    4114, 2421720,  475308,
       1983078, 1542400,  119068,   20938,   12544, 2122828, 1915564,
             4,  193600, 2663424, 1221148,  400554, 1124928])
```

In [44]:
```
# look for the integers I so that i**2 + (i+1)**2 = (i+2)**2
a = np.arange(10)
print(a)
b = a**2 + (a+1)**2
print(b)
```

```
[0 1 2 3 4 5 6 7 8 9]
[  1   5  13  25  41  61  85 113 145 181]
```

In [45]:
```
c = (a+2)**2
```

In [46]:
```
print(b)
print(c)
```

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

In [47]:
```
b == c
```

Out[47]:
```
array([False, False, False,  True, False, False, False, False, False,
       False])
```

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

In [48]:
```
a = np.arange(18)
print(a)
print(np.log10(a))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17]
[      -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]
/Users/carlos/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by zero encountered
```

```
in log10
  This is separate from the ipykernel package so we can avoid doing imports until
```

Numpy hace operaciones matriciales

In [49]:
```
a = [[1, 0], [0, 1]]
b = [[4, 1], [2, 2]]

c = np.matmul(a, b)

print(c)
```

```
[[4 1]
 [2 2]]
```

In [50]:
```
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]
```

In [51]:
```
a = np.arange(1000000).reshape(1000,1000)
b = np.ones_like(a)
print(a)
```

```
[[     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]]
```

In [52]:
```
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]]
```

In [53]:

```
#Solución de un sistema lineal de ecuaciones

#A = floor(random.rand(4000,4000)*20-10) # random matrix
A = np.random.rand(4000,4000)

print (A)

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

print(b)

# solve Ax = b using the standard numpy function
x = np.linalg.solve(A,b)
print(x)
```

```
[[0.92876846 0.70556694 0.90924381 ... 0.76560061 0.48918951 0.05352418]
 [0.07569249 0.45352815 0.42065    ... 0.33693727 0.97824361 0.67129038]
 [0.50588115 0.13999435 0.63487406 ... 0.1095268  0.73119003 0.77253283]
 ...
 [0.80111455 0.72330952 0.22128345 ... 0.47809887 0.44070475 0.33805835]
 [0.96948356 0.68377217 0.99537495 ... 0.89085754 0.71117478 0.16955667]
 [0.35208074 0.65394129 0.01556624 ... 0.53001193 0.23654106 0.10398452]]
[[ 7.]
 [ 6.]
 [ 2.]
 ...
 [-8.]
 [ 6.]
 [-2.]]
[[  19.02584735]
 [  -8.7690453 ]
 [-109.46918255]
 ...
 [ -42.11001547]
 [ -34.68361764]
 [ -62.07767231]]
```

Resumen

- ### 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.
- ### No pensar en término de variables, sino de arreglos.
- ### Puede vectorizarse a hardware.
- ### Python sin Numpy no nos sería tan útil.
- ### Queda claro, hay muchas órdenes, uno solo necesita aprender los comandos que le sirven. Para una tarea nueva, averiguar si la orden ya está creada. Sacarse la idea de encima que se puede aprender todos los comandos
- ### Se siguen agregando y modificando las órdenes todo el tiempo.