```
1
```

Introducción al Módulo de Numpy

Introducción a Python para Visualización de Datos

Dr. Gerardo Ramón Fox

Julio 2022

Numpy es un módulo de Python que proporciona estructuras de datos para manipulación eficiente de vectores, matrices y (técnicamente) tensores.

Para usar el módulo de Numpy basta ejecutar la instrucción:

```
import numpy as np
```

al inicio de nuestro Notebook o script. El objeto np lo utilizamos para invocar todas las funciones que proporciona el módulo.

```
1 import numpy as np
```

Para más información sobre toda la documentación de Numpy, consultar: https://numpy.org/

Objetos: ndarray

La estructura más básica es un vector, que podemos crear a partir de pasar una lista a la función array:

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

```
a =  [1 2 3 4 5 6]
type(a) =  <class 'numpy.ndarray'>
```

Vemos que el vector a es un objeto de la clase "ndarray". Esta clase de Numpy maneja los arreglos en general. Un objeto de ndarray tiene los siguientes atributos:

```
1 print("a          = ", a)
2 print("a.ndim     = ", a.ndim)  # número de dimensiones: vector = 1, matriz = 2, cubo = 3,
```

```
print("a.shape    = ", a.shape) # tupla con el número de elementos en cada dimensión, por
print("a.size     = ", a.size)  # número de elementos en la estructura, por ejemplo, para
print("a.itemsize = ", a.itemsize) # tamaño en bytes de un elemento del arreglo
print("a.dtype    = ", a.dtype)  # tipo de dato almacenado
```

```
a          =  [1 2 3 4 5 6]
a.ndim     =  1
a.shape    =  (6,)
a.size     =  6
a.itemsize =  8
a.dtype    =  int64
```

Para crear una matriz

```
b = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print("b = ")
print(b)
print(" ")
print("b.ndim  = ", b.ndim)
print("b.shape = ", b.shape)
print("b.size  = ", b.size)
```

```
b =
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

b.ndim  =  2
b.shape =  (3, 4)
b.size  =  12
```

Herramientas para Crear Vectores

Existen varias instrucciones para crear vectores en Numpy. Revisamos algunos ejemplos:

Función np.empty()

```
import numpy as np
# Funcion empty:  x = np.empty(N, dtype=tipo)
# Genera un vector de N elementos del tipo dado sin inicializar valores.

v = np.empty(10, dtype="float")
print("v = ", v)
print("v.shape = ", v.shape, " | ", "v.dtype = ", v.dtype)
```

```
v =  [1.71746230e-316 1.77863633e-322 0.00000000e+000 0.00000000e+000
 0.00000000e+000 1.42526634e+160 3.21442615e-057 4.51425734e-090
 2.62100259e+180 4.50821126e-315]
v.shape =  (10,)  |  v.dtype =  float64
```

Función np.zeros()

```
# Funcion zeros: x = np.zeros(N, dtype=tipo)
# Genera un vector de N elementos del tipo dado inicializando todos a cero.
iv = np.zeros(10, dtype="int")
print("iv = ", iv, " | ", "iv.shape = ", iv.shape, " | ", "iv.dtype = ", iv.dtype)
```

```
iv =  [0 0 0 0 0 0 0 0 0 0]  |  iv.shape =  (10,)  |  iv.dtype =  int64
```

Función np.ones()

```
# Funcion ones: x = np.ones(N, dtype=tipo)
# Genera un vector de N elementos del tipo dado inicializando todos a uno.

z = np.ones(10, dtype="float32")   # float32 es equivalente a precision sencilla.
print("z = ", z, " | ", "z.shape = ", z.shape, " | ", "z.dtype = ", z.dtype)
```

```
z =  [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]  |  z.shape =  (10,)  |  z.dtype =  float32
```

Función np.arange()

```
 # Funcion arange: x = np.arange(ini, fin, paso, dtype=tipo)
 # Genera un vector que inicia en ini y llega hasta fin-paso. El paso es opcional
 # si no se incluye, es 1 por default.

print("np.arange(10)     = ", np.arange(10))         # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print("np.arange(5, 10)  = ", np.arange(5, 10))   # [5, 6, 7, 8, 9]
print("np.arange(5, 11)  = ", np.arange(5, 11))   # [5, 6, 7, 8, 9, 10]
```

```
np.arange(10)     =  [0 1 2 3 4 5 6 7 8 9]
np.arange(5, 10)  =  [5 6 7 8 9]
np.arange(5, 11)  =  [ 5  6  7  8  9 10]
```

```
# Funcion arange con diferentes pasos
print("np.arange(10, step=2) = ", np.arange(10, step=2))

# Es equivalente a
print("np.arange(0, 10, 2)   = ", np.arange(0, 10 ,2))

# Generar un vector de 0 a 10 con paso de 2
print("np.arange(0, 12, 2)   = ", np.arange(0,12,2))

# Con decimales
print("np.arange(-5, 5, 0.5) = ", np.arange(-5, 5, 0.5))

# np.arange(-5, 5, 0.5) es equivalente a np.arange(-5, 5, step=5).
```

```
np.arange(10, step=2) =  [0 2 4 6 8]
np.arange(0, 10, 2)   =  [0 2 4 6 8]
np.arange(0, 12, 2)   =  [ 0  2  4  6  8 10]
np.arange(-5, 5, 0.5) =  [-5.  -4.5 -4.  -3.5 -3.  -2.5 -2.  -1.5 -1.  -0.5  0.   0.5  1
  2.   2.5  3.   3.5  4.   4.5]
```

```
# Si queremos una secuencia -5 a 5 con paso de 0.5, hay que sumar 0.5 al ultimo valor:
print("np.arange(-5, 5.5, 0.5) = ", np.arange(-5, 5.5, 0.5))
```

```
np.arange(-5, 5.5, 0.5) =  [-5.  -4.5 -4.  -3.5 -3.  -2.5 -2.  -1.5 -1.  -0.5  0.   0.5
  2.   2.5  3.   3.5  4.   4.5  5. ]
```

Función np.linspace()

```
# Funcion linspace: x = np.linspace(ini, fin, num=50, dtype=tipo)
# Retorna un arreglo de elementos igualmente espaciados en el intervalo especificado.
x = np.linspace(0,10)
print("x = ")
print(x)
print("x.size = ", x.size)
print(" ")

y  = np.linspace(-5, 5, num=11)
print("y = ")
print(y)
print("y,size = ", y.size, " | ", "y.dtype = ", y.dtype)
```

```
x =
[ 0.          0.20408163  0.40816327  0.6122449   0.81632653  1.02040816
  1.2244898   1.42857143  1.63265306  1.83673469  2.04081633  2.24489796
  2.44897959  2.65306122  2.85714286  3.06122449  3.26530612  3.46938776
  3.67346939  3.87755102  4.08163265  4.28571429  4.48979592  4.69387755
  4.89795918  5.10204082  5.30612245  5.51020408  5.71428571  5.91836735
  6.12244898  6.32653061  6.53061224  6.73469388  6.93877551  7.14285714
  7.34693878  7.55102041  7.75510204  7.95918367  8.16326531  8.36734694
  8.57142857  8.7755102   8.97959184  9.18367347  9.3877551   9.59183673
  9.79591837 10.        ]
x.size =  50

y =
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
y,size =  11  |  y.dtype =  float64
```

Herramientas para Crear Arreglos Multidimensionales

Ahora revisaremos 3 herramientas básicas para generar arreglos multidimensionales:

```
m = np.empty(shape, type=tipo)
m = np.zeros(shape, type=tipo)
m = np.ones(shape, type=tipo)
```

donde shape es una tupla que contiene el numero de elementos en cada dimension: shape = (n1, n2, n3, n4)

Veamos algunos ejemplos:

```
1 # Generar una matriz de 3 filas por 4 columnas de ceros.
2 m = np.zeros((3, 4), dtype="float64")
3
4 print("m = ")
5 print(m)
6 print("m.shape = ", m.shape)
```

```
m =
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
m.shape =  (3, 4)
```

```
1 # Generar un cubo de 3 x 3 x 4 de unos.
2 c = np.ones((3, 3, 4), dtype="float64")
3
4 print("c = ")
5 print(c)
6 print("c.shape = ", c.shape)
```

```
c =
[[[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.]]]
c.shape =  (3, 3, 4)
```

Tipos de Datos de Arreglos de Numpy

Tipo	Descripcion
bool	Booleano (True, False)

Tipo	Descripcion
int	Entero estándar (32 ó 64 segun la implementación)
int8, int16, int32, int64	Entero de 8, 16, 32 ó 64 bits
uint8, uint16, uint32, uint64	Entero unsigned de 8, 16, 32 ó 64 bits
float	Float estándar (doble precisión)
float16	Float: 1 bit de signo, 5 bits exp, 10 bits mantissa
float32	Float de precisión sencilla
float64	Float de precisión doble
complex (complex128)	Números complejos
object	Objetos

Manipulación de Arreglos y Matrices

Podemos acceder a elementos individuales por medio de índices así como por *slices*.

Vectores

```
# Generamos un vector de ejemplo.
v = np.arange(0, 10)
print("v  = ", v)

# Acceder a elementos individuales de un vector.
print("v[0] = ", v[0])
print("v[4] = ", v[4])
print("v[-1] = ", v[-1])

```

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

```
# Extraer un segmento o slice de un vector.
print("v[2:4] = ", v[2:4])
print("v[2:5] = ", v[2:5])
print("v[5:]  = ", v[5:])  # Del indice 5 a el ultimo.
print("v[:5]  = ", v[:5])  # Del inicio al indice 4.
```

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

```
# Extraer un slice con un paso dado de un vector.
print("v = ", v)
```

```
print("v[0:7:3] = ", v[0:7:3])
print("v[::3]   = ", v[::3])
```

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

Matrices

```
# Generamos una matriz de ejemplo de 3 filas y 4 columnas.
m = np.zeros((3, 4), dtype='float64')

# La inicializamos a la suma de los indices por ejemplo.
for row in range(m.shape[0]):
  for col in range(m.shape[1]):
    m[row, col] = row + col        # Accedemos a una matriz de la forma m[renglon, columna

print("m = ")
print(m)
print("m.shape = ", m.shape)
print("----\n")

# Accedemos a elementos individuales.
print("m[0,3] = ", m[0, 3])         # fila 0, columna 3
print("m[1,2] = ", m[1, 2])         # fila 1, columna 2
print("----\n")

# Extraemos una fila entera.
print("m[1,:]   = ", m[1, :])         # Extrae la fila 1.
print("m[1,::2] = ", m[1,::2])        # Extrae cada 2 elementos de la fila 1.
print("----\n")

# Extraemos una columna entera.
print("m[:,2]   = ", m[:, 2])           # Extrae la columna 2, la trata como un vector (ar
print("m[::2,2] = ", m[::2, 2])         # Extrae cada 2 elementos de la columna 2.
print("----\n")

# Extraemos cada dos elementos de la matriz.
print("m[::2, ::2]")
print(m[::2, ::2])
```

```
m =
[[0. 1. 2. 3.]
 [1. 2. 3. 4.]
 [2. 3. 4. 5.]]
m.shape =  (3, 4)
----

m[0,3] =  3.0
m[1,2] =  3.0
----
```

```
m[1,:]   =  [1. 2. 3. 4.]
m[1,::2] =  [1. 3.]
----

m[:,2]   =  [2. 3. 4.]
m[::2,2] =  [2. 4.]
----

m[::2, ::2]
[[0. 2.]
 [2. 4.]]
```

Operaciones Aritméticas con Arreglos de Numpy

Revisamos las operaciones aritméticas de Numpy

```
x = np.array([[1,2],[3,4]], dtype="float64")
y = np.array([[5,6],[7,8]], dtype="int64")

print("x = ")
print(x)
print(" ")
print("y = ")
print(y)
```

```
x =
[[1. 2.]
 [3. 4.]]

y =
[[5 6]
 [7 8]]
```

Suma: x + y

```
print("x + 5 =")
print(x + 5)        # Suma 5 a todos los elementos de x.
print("----")

z = x + y           # Suma x e y, elemento por elemento
print("z = x + y")
print(z)
print(z.dtype)      # Sumamos int y float, y el resultado es de tipo float.
```

```
x + 5 =
[[6. 7.]
 [8. 9.]]
----
```

```
z = x + y
[[ 6.  8.]
 [10. 12.]]
float64
```

Resta: x - y

```
1 z = x - y
2 print("z = x - y")
3 print(z)
```

```
z = x - y
[[-4. -4.]
 [-4. -4.]]
```

Multiplicación: elemento por elemento: x * y

```
1 print("x * 5 =")
2 print(x * 5)        # Multiplica 5 a todos los elementos de x.
3 print("----")
4
5 z = x * y           # Multiplica x e y, elemento por elemento
6 print("z = x * y")
7 print(z)
8 print(z.dtype)      # Multiplicamos int y float, y el resultado es de tipo float.
```

```
x * 5 =
[[ 5. 10.]
 [15. 20.]]
----
z = x * y
[[ 5. 12.]
 [21. 32.]]
float64
```

Multiplicación Matricial (Álgebra Lineal): x @ y

```
1 z = np.matmul(x, y)
2 print("z = x y")
3 print(z)
4
5 z = np.matmul(y, x)
6 print("z = y x")
7 print(z)
8
9 # Notacion corta
```

```
10 z = y @ x
11 print("y @ x")
12 print(z)
```

```
z = x y
[[19. 22.]
 [43. 50.]]
z = y x
[[23. 34.]
 [31. 46.]]
y @ x
[[23. 34.]
 [31. 46.]]
```

División: elemento por elemento: x / y

```
1 print("x / 5 =")
2 print(x / 5)        # Divide todos los elementos de x entre 5.
3 print("----")
4
5 z = x / y           # Divide x e y, elemento por elemento
6 print("z = x / y")
7 print(z)
8 print(z.dtype)      # Dividimos float e int, y el resultado es de tipo float.
```

```
x / 5 =
[[0.2 0.4]
 [0.6 0.8]]
----
z = x / y
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
float64
```

Recíproco: 1/x

```
1 print("1/x")
2 print(1/x)
```

```
1/x
[[1.         0.5       ]
 [0.33333333 0.25      ]]
```

Potencias: x**y

```
1 print("x**2")
```

```
print(x**2)      # Eleva cada elemento de x al cuadrado.

z = x ** y       # Eleva x[i,j] a la potencia guardada en y[i,j]
print("z = x**y")
print(z)
```

```
x**2
[[ 1.  4.]
 [ 9. 16.]]
z = x**y
[[1.0000e+00 6.4000e+01]
 [2.1870e+03 6.5536e+04]]
```

Resumen de Operaciones

Operación	Notación Compacta	Notación Funcional
suma	x + y	np.sum(x, y)
resta	x - y	np.subtract(x, y)
multiplicación	x * y	np.multiply(x, y)
mult. matricial	x @ y	np.matmul(x, y)
división	x / y	np.divide(x, y)
recíproco	1 / x	np.reciprocal(x, y)
potencias	x ** y	np.power(x, y)

Todas se realizan elemento por elemento excepto la multiplicación matricial.

Hay una gran cantidad de funciones matemáticas que podemos aplicar a arreglos de Numpy para evaluarlas. Las iremos revisando en la parte de visualización.

Agregar Elementos (np.append) y Concatenar Arreglos (np.concatenate)

Para agregar elementos a un arreglo o matriz, podemos usar la función np.append():

```
np.append(arreglo, vals, axis=None)
arreglo -> el arreglo base
vals    -> el arreglo de valores a juntar
axis    -> direcci'on por la que se junta (si ndim > 1)
```

```
# Append con vectores.
a = np.array([1., 2., 3., 4.])
b = np.array([5., 6., 7., 8.])

```

```
# Agregar b al final de a. No es necesario especificar axis porque ambos son vectores.
a = np.append(a, b)
print(a)
```

```
[1. 2. 3. 4. 5. 6. 7. 8.]
```

```
# Append con matrices.
a = np.ones((3,3), dtype="float")
b = 2. * np.ones((3,3), dtype="float")

c_axis0 = np.append(a, b, axis=0)   # Agrega los elementos como filas nuevas
c_axis1 = np.append(a, b, axis=1)   # Agrega los elementos como columnas nuevas.

print(c_axis0)
print(" ")
print(c_axis1)
```

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

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

Para concatenar arreglos o matrices podemos usar la función np.concatenate():

```
np.concatenate(iter, axis=None)
iter: tupla o lista de arreglos a concatenar
axis: dirección de concatenación
```

Veamos un ejemplo.

```
a = np.ones((3, 3), dtype="float32")            # Matriz de 3x3 con unos.
b = 2. * np.ones((3, 3), dtype="float32")       # Matriz de 3x3 con todos los elementos i
c = 3. * np.ones((3, 3), dtype="float32")       # Matriz de 3x3 con todos los elementos i

res_axis0 = np.concatenate((a, b, c), axis=0)    # Concatenar a lo largo de las filas.
res_axis1 = np.concatenate([a, b, c], axis=1)    # Concatenar a lo largo de las columnas.

print(res_axis0)
print("\n\n")

print(res_axis1)
```

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



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

Operaciones: Reshape, Transpose y Copy

La operación np.reshape reestructura la forma de un arreglo de una manera compatible.

La operación np.transpose calcula la transpuesta del arreglo.

Veamos algunos ejemmplos:

```
# Generemos una matriz de 2x4.
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print("a = ")
print(a)
print(" ")

# Podemos reacomodar los elementos con la función reshape
print("a.reshape(4,2) = ")
print(a.reshape(4,2))

# Lo anterior es equivalente a np.reshape((4,2), a)
print(" ")

# Ahora calculamos la transpuesta
print("a.transpose() = ")
print(a.transpose())

# Lo anterior es equivalente a np.transpose(a) o a.T
```

Reshape y transpose, en cualquier manera que se invoquen, retornan "por lo general" una vista (view) o referencia al arreglo original y no una copia. Esto puede causar bugs si se quiere modificar el resultado, pero no la estructura original. Veamos el siguiente ejemplo:

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

```

```
b = np.transpose(a)
b[1, :] = np.array([-2, -6])

print(b)
print(" ")
print(a)
```

Evitamos esto con el la función copy del arreglo o la que proporciona del módulo deepcopy

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

# El arreglo a tiene su funcion copy() y la usamos para crear una copia (shallow).
b = np.transpose(a.copy())
b[1, :] = np.array([-2, -6])

print(b)
print(" ")
print(a)
```

```
# Usar la funcion deepcopy si el arreglo almacena objetos.
import copy

a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

b = np.transpose(copy.deepcopy(a))
b[1, :] = np.array([-2, -6])

print(b)
print(" ")
print(a)
```

```
1
```