Numpy

Camacho Vázquez Vanessa Alejandra.

Su documentación menciona:

Provides

1. *An array object of arbitrary homogeneous items*
2. *Fast mathematical operations over arrays*
3. *Linear Algebra, Fourier Transforms, Random Number Generation*

¿Qué es un **arreglo** (***array***)?

Un **arreglo** (***array***) es una **colección finita de datos del mismo tipo**, que se almacenan en posiciones consecutivas de memoria y reciben un nombre común.

En el caso de un **arreglo unidimensional** estamos hablando de una estructura de datos que tendría cierta equivalencia o similaridad con el objeto matemático: **vector** $\left(\vec{v} = (v_1, v_2, \ldots, v_n) = (v_i)_{1 \leq i \leq n}\right)$, ya que ambos corresponden a una secuencia de valores ordenados, que son indexados por un único índice asociado a una única dimensión.

Por ejemplo, el arreglo unidimensional representado aquí:

índice:	**0**	**1**	**2**	**3**	**4**
valor:	1.2	3.4	5.6	7.8	9

contiene la misma información o los mismos datos que el vector:

$$\vec{v} = (v_1, v_2, \ldots, v_n) = (1.2, 3.4, 5.6, 7.8, 9)$$

Los **arreglos multidimensionales** son aquellos en donde cada elemento del arreglo está indexado por más de un índice (por más de un subíndice cuando los elementos del arreglo se denotan de la siguiente manera: $a_{i,j,k,\ldots}$

Por ejemplo, el arreglo multidimensional (bidimensional) representado aquí:

	2do índice:	**0**	**1**	**2**
1er índice:	**0**	1.2	3.4	5.6
	1	7.8	9	0

contiene la misma información o los mismos datos que la matriz:

$$A = (a_{ij})_{1 \leq i \leq n, 1 \leq j \leq m} = \begin{bmatrix} 1.2 & 3.4 & 5.6 \\ 7.8 & 9 & 0 \end{bmatrix}$$

¿Por qué usar arrays de numpy y no listas o tuplas de python?

Las listas y tuplas nos podrían servir para el manejo de *arrays* pero las operaciones con listas o tuplas son demasiado lentas (lo que se gana en versatilidad se pierde en velocidad). Se ha dicho que las operaciones con arreglos de `numpy` es 50 veces más veloz que con listas.

La mayoría, sino todas, las librerías principales de IA y AP (dícese Tensorflow y Pytorch) suelen hacer sus operaciones con arrays de numpy.

Creación básica de arrays

Primero importamos (cargamos) el módulo `numpy` y le asociamos el alias `np`:

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

Vamos a crear arrays de distintas dimensiones. Para esto podemos usar números, listas o tuplas.

In [2]:
```
a = np.array(13) #array de dimensión 0
print("type(a):", type(a), "\n")
print("a:", a, sep = "\n")
a
```

```
type(a): <class 'numpy.ndarray'>

a:
13
```

Out[2]:
```
array(13)
```

In [3]:
```
b = np.array([1, 2, 3, 4]) #array de dimensión 1
print("b:", b, sep = "\n")
b
```

```
b:
[1 2 3 4]
```

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

In [4]:
```
c = np.array([[1, 2, 3, 4], [5, 6, 7, 8]]) #array de dimensión 2
print("c:", c, sep = "\n")
c
```

```
c:
[[1 2 3 4]
 [5 6 7 8]]
```

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

In [5]:
```
d = np.array([[[1, 2, 3, 4], [5, 6, 7, 8]], [[9, 10, 11, 12], [13, 14, 15, 16]]]) #array de dimensión 3
print("d:", d, sep = "\n")
d
```

```
d:
[[[ 1  2  3  4]
  [ 5  6  7  8]]

 [[ 9 10 11 12]
  [13 14 15 16]]]
```

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

       [[ 9, 10, 11, 12],
        [13, 14, 15, 16]]])
```

Los escalares son arreglos/tensores de dimensión 0, los vectores son arreglos/tensores de dimensión 1, las matrices son arreglos/tensores de dimensión 2, etc.

El atributo `ndim` almacena la dimension del arreglo/tensor (no confundir con el tamaño de un vector o una matriz).

In [6]:
```
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
0
1
2
3
```

Indexación básica

El proceso de indexación en numpy es similar al de las listas en python. Cada número implica buscar la posición en cada dimensión del array

In [7]:
```
print("a =", a)
print("b[2] = ", b[2])
print("c[1] =", c[1])
print("d[1] =", d[1])
```

```
a = 13
b[2] =  3
c[1] = [5 6 7 8]
d[1] = [[ 9 10 11 12]
 [13 14 15 16]]
```

In [8]:
```
print("a =", a)
print("b[2] =", b[2])
print("c[1,2] =", c[1,2])
print("d[1,0,2] =", d[1,0,2])
```

```
a = 13
b[2] = 3
c[1,2] = 7
d[1,0,2] = 11
```

Similarmente podemos segmentar **arrays** como en las listas de python,

In [9]:
```
print(b[1:4]) #b = [1 2 3 4]
print(c[1,1:3]) #c = [[1 2 3 4][5 6 7 8]]
print(d[1:3,1:3,0:2])
```

```
[2 3 4]
[6 7]
[[[13 14]]]
```

Diferentes tipos de datos en los arrays

Los arrays no están limitados a números enteros, pueden también tener cadenas, booleanos, o números de punto flotante.

In [10]:
```
frutas = np.array(['Manzana', 'Naranja', 'Uva']) #cadenas
print(frutas.dtype)
```

```
<U7
```

Podemos manipular el tipo de los datos del array dentro de la función de creación del array `np.array`, siempre y cuando el cambio sea posible. Por ejemplo podemos pasar enteros a cadenas (integers a strings) pero no viceversa.

In [11]:
```
number_to_string = np.array([6, 1, 2, 4, 623, 8], dtype=str) #b denota un string
print(number_to_string.dtype)
print(number_to_string)
```

```
<U3
['6' '1' '2' '4' '623' '8']
```

In [12]:
```
try:
    error_array = np.array(['a', '2', '3'], dtype=int)
except:
    print("hay un error en su lógica")
```

```
hay un error en su lógica
```

Podemos también cambiar el tipo en arrays ya existentes

In [13]:
```
floating_array = np.array([1.1, 2.1, 3.1]) #punto flotante

int_array = floating_array.astype(int)
print(int_array)
```

```
[1 2 3]
```

In [14]:
```
vector = np.array([127, -127, 32767, -32767], dtype=np.int8) # entero de 1 byte = 8 bits (1 bit se necesita para almacenar el signo)
print(``vector`:\t', vector)
print(``vector.dtype`: ', vector.dtype)
vector = np.array([127, -127, 32767, -32767], dtype=np.int16) # entero de 2 bytes = 16 bits (1 bit se necesita para almacenar el signo)
print(``vector`:\t', vector)
print(``vector.dtype`: ', vector.dtype)
```

```
`vector`:       [ 127 -127   -1    1]
`vector.dtype`:  int8
`vector`:       [   127   -127  32767 -32767]
`vector.dtype`:  int16
```

Tamaño de los arrays

El tamaño de un arreglos es diferente a su dimensión. El tamaño del arreglo es el número de elementos posibles por cada dimensión.

In [15]:
```
print(a.shape) # a = 13
print(b.shape) # b = [1 2 3 4]
print(c.shape) # c = [[1 2 3 4][5 6 7 8]]
print(d.shape) # d = [[[ 1  2  3  4][ 5  6  7  8]] [[ 9 10 11 12][13 14 15 16]]]
```

```
()
(4,)
(2, 4)
(2, 2, 4)
```

Cambio de forma de los arreglos

Podemos cambiar la forma de los arreglos. Esto significa aumentar el número de dimensiones o el número de elementos por dimensión.

In [16]:
```
print("c =", c, "\n") # c = [[1 2 3 4][5 6 7 8]]
c_1D = c.reshape(1, 8)
print("c_1D =", c_1D, "\n")
```

```
c = [[1 2 3 4]
 [5 6 7 8]]
```

In [17]:
```
c_4D = c.reshape(4, 2)
print("c_4D =", c_4D, "\n")
```

```
c_4D = [[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

¿Podemos hacer cualquier cambio en la forma? Si, mientras la cantidad de elementos sea coincidente.

In [18]:
```
print(d)
```

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

 [[ 9 10 11 12]
  [13 14 15 16]]]
```

`d` es un array con 16 elementos, así que podemos hacer reshape con tamaños por ejemplo 1x16, 4x4, 8x2, 2x4x2.

In [19]:
```
try:
    d.reshape(3, 5)
except:
    print('Hay un error al intentar cambiar la forma del arreglo')
```

```
Hay un error al intentar cambiar la forma del arreglo
```

In [20]:
```
d.reshape(2, 2, 2, 2)
```

Out[20]:
```
array([[[[ 1,  2],
         [ 3,  4]],

        [[ 5,  6],
         [ 7,  8]]],


       [[[ 9, 10],
         [11, 12]],

        [[13, 14],
         [15, 16]]]])
```

Para no especificar explícitamente el número de elementos de una (solamente una) de las dimensiones se le puede dar el valor `-1` (solamente una vez) al método `reshape()`.

In [21]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
arr.reshape(3, -1)
```

Out[21]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Podemos usar dicho valor `-1` para "aplanar" el array a una dimensión.

In [22]:
```
d.reshape(-1)
```

Out[22]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16])
```

Transpuesta

Para transponer el array utilizamos `.T`

In [23]:
```
print("c =", c, "\n")
print("c.T =", c.T)
```

```
c = [[1 2 3 4]
 [5 6 7 8]]

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

Recorrer los valores de un arreglo

Podemos usar los métodos tradicionales para recorrer los valores de un arreglo.

In [24]:
```
for x in b:
    print(x, "\n")
```

```
1

2

3

4
```

In [25]:
```
for x in c:
    print(x, "\n")
```

```
[1 2 3 4]

[5 6 7 8]
```

In [26]:
```
for x in d:
    print(x, "\n")
```

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

[[ 9 10 11 12]
 [13 14 15 16]]
```

In [27]:
```
for x in d:
    for y in x:
        for z in y:
            print(z)
```

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

Esto por supuesto no es óptimo. Así que numpy tiene la función `nditer()` para estos casos.

In [28]:
```
for x in np.nditer(d):
    print(x)
```

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

Se puede usar la función `ndenumerate`, en caso de que además de los valores dentro de un arreglo, también se requiera la indexación respectiva.

In [29]:
```
for i, x in np.ndenumerate(d):
    print('i:', i, '\t\tx:', x)
```

```
i: (0, 0, 0)            x: 1
i: (0, 0, 1)            x: 2
i: (0, 0, 2)            x: 3
i: (0, 0, 3)            x: 4
i: (0, 1, 0)            x: 5
i: (0, 1, 1)            x: 6
i: (0, 1, 2)            x: 7
i: (0, 1, 3)            x: 8
i: (1, 0, 0)            x: 9
i: (1, 0, 1)            x: 10
i: (1, 0, 2)            x: 11
i: (1, 0, 3)            x: 12
i: (1, 1, 0)            x: 13
i: (1, 1, 1)            x: 14
i: (1, 1, 2)            x: 15
i: (1, 1, 3)            x: 16
```

Generación de arreglos

`numpy` permite crear arreglos específicos rápidamente:

In [30]:
```
print(f'`np.arange(0, 1, 0.1)`:\n{np.arange(0, 1, 0.1)}\n')
```

```
`np.arange(0, 1, 0.1)`:
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

In [31]:
```
print(f'`np.zeros((2,3,5))`:\n{np.zeros((2,3,5))}\n')
```

```
`np.zeros((2,3,5))`:
[[[0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]]

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

In [32]:
```
print(f'`np.ones((2,3,5))`:\n{np.ones((2,3,5))}\n')
```

```
`np.ones((2,3,5))`:
[[[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.]]]
```

In [33]:
```
print(f'`np.full((3,4), 1.23)`:\n{np.full((2,3,5), 1.23)}\n')
```

```
`np.full((3,4), 1.23)`:
[[[1.23 1.23 1.23 1.23 1.23]
  [1.23 1.23 1.23 1.23 1.23]
  [1.23 1.23 1.23 1.23 1.23]]

 [[1.23 1.23 1.23 1.23 1.23]
  [1.23 1.23 1.23 1.23 1.23]
  [1.23 1.23 1.23 1.23 1.23]]]
```

In [34]:
```
print(f'`np.identity(3)`:\n{np.identity(3)}\n')
```

```
`np.identity(3)`:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

In [35]:
```
print(f'`np.eye(3)`:\n{np.eye(3)}\n')
```

```
`np.eye(3)`:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

In [36]:
```
print(f'`np.empty((2,3,5))`:\n{np.empty((2,3,5))}\n')
```

```
`np.empty((2,3,5))`:
[[[1.23 1.23 1.23 1.23 1.23]
  [1.23 1.23 1.23 1.23 1.23]
  [1.23 1.23 1.23 1.23 1.23]]

 [[1.23 1.23 1.23 1.23 1.23]
  [1.23 1.23 1.23 1.23 1.23]
  [1.23 1.23 1.23 1.23 1.23]]]
```

Operaciones con arrays

La función `where` devuelve los índices de los valores para los cuales se cumple una condición dada. Estos índices podrían ser utilizados para "filtrar". Sin embargo, si el único objetivo es "filtrar", se puede lograr usando una colección de valores booleanos.

In [37]:
```
vector = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
indx = np.where(vector%3 == 0)
print(f'`type(indx)`:\t{type(indx)}')
print(f'`indx`:\t{indx}')
```

```
`type(indx)`:   <class 'tuple'>
`indx`: (array([2, 5, 8]),)
```

In [38]:
```
filtrado = vector[indx]
print(f'`filtrado`:\t{filtrado}')
```

```
`filtrado`:     [3 6 9]
```

In [39]:
```
vector_bool = vector%3 == 0
print(f'`type(vector_bool)`:\t{type(vector_bool)}')
print(f'`vector_bool`:\t\t{vector_bool}')
filtrado = vector[vector_bool]
print(f'`filtrado`:\t\t{filtrado}')
```

```
`type(vector_bool)`:    <class 'numpy.ndarray'>
`vector_bool`:          [False False  True False False  True False False  True]
`filtrado`:             [3 6 9]
```

Además de proveer la clase `ndarray`, `numpy` contiene un conjunto de funciones (que suelen llamar **ufuncs**, *universal functions*) que se ejecutan elemento a elemento sin necesidad de utilizar un `for` o un `while` (cuando una función se ejecuta de esa manera se dice que está **vectorizada** o que es una **función vectorizada**). Es más eficiente usar una función vectorizada (si está muy bien hecha, probada, etc.) que iterar mediante un `while` o un `for`.

- Operaciones aritmeticas: `add()`, `subtract()`, `multiply()`, `divide()`, `power()`, `mod()`, `remainder()`, `divmod()`, `absolute()`.
- "Redondeo": `trunc()`, `fix()`, `around()`, `floor()`, `ceil()`.
- Mínimo común múltiplo y máximo común divisor: `lcm()`, `lcm.reduce()`, `gcd()`, `gcd.reduce()`
- Logaritmos y exponencial: `log2()`, `log10()`, `log()`, `exp()`.
- Trigonométricas y relacionadas: `sin()`, `cos()`, `tan()`, `arcsin()`, `arccos()`, `arctan()`, `sinh()`, `cosh()`, `tanh()`, `arcsinh()`, `arccosh()`, `arctanh()`, `deg2rad()`, `rad2deg()`.
- Conjuntos: `unique()`, `union1d()`, `intersect1d()`, `setdiff1d()`, `setxor1d()`.

Todas las ufuncs tienen como parámetros adicionales:

- `dtype` para dar el tipo de elementos para la salida.
- `out` para dar un arreglo en donde la salida será copiada.
- `where` para dar un arreglo booleano o una condición para seleccionar los elementos sobre los que se ejecutará la función.

In [40]:
```
import math
# en vez de algo como:
opt1 = [math.sin(i) for i in [1, 2, 3]]
print(opt1)
```

```
[0.8414709848078965, 0.9092974268256817, 0.1411200080598672]
```

In [41]:
```
import math
# en vez de algo como:
opt1 = [math.sin(i) for i in [1, 2, 3]]
print(f'`opt1`: \t{opt1}')
print(f'`opt1[0]`: \t{opt1[0]:.30f}')
```

```
`opt1`:         [0.8414709848078965, 0.9092974268256817, 0.1411200080598672]
`opt1[0]`:      0.841470984807896504875657228695
```

In [42]:
```
# es mucho más eficiente hacer algo como:
opt2 = np.sin(np.array([1, 2, 3], dtype=np.uint8), dtype=np.float64)
print(f'`opt2`: \t{opt2}')
print(f'`opt2[0]`:\t{opt2[0]:.30f}')
```

```
`opt2`:         [0.84147098 0.90929743 0.14112001]
`opt2[0]`:      0.841470984807896504875657228695
```

In [43]:
```
x = np.array([1, 2, 3])
y = np.array([0.0, 0.0, 0.0])
np.sin(x, where = x%2==1, out = y)
print(f'`y`: \t{y}')
```

```
`y`:    [0.84147098 0.         0.14112001]
```

In [44]:
```
x = np.array([1, 2, 3, 4])
y = np.array([4, 5, 6, 7])
z = np.add(x, y)
print(f'`z`:\t{z}')
```

```
`z`:    [ 5  7  9 11]
```

In [45]:
```
z = x + y
print(f'`z`:\t{z}')
```

```
`z`:    [ 5  7  9 11]
```

In [46]:
```
z = np.add(x, y, dtype=float)
print(f'`z`:\t{z}', )
```

```
`z`:    [ 5.  7.  9. 11.]
```

In [47]:
```
try:
    print(np.array([1, 2, 3, 4]) + np.array([5, 6]))
except:
    print("No se pudo hacer la operación")
```

```
No se pudo hacer la operación
```

In [48]:
```
try:
    print(np.array([1, 2, 3, 4]) + 0.9)
except:
    print("No se pudo hacer la operación")
```

```
[1.9 2.9 3.9 4.9]
```

Adicionalmente tenemos las funciones: `sum()`, `cumsum()`, `prod()`, `cumprod()`, `diff()`.

In [49]:
```
matriz = np.array([[3, 2, 4], [5, 2, 1], [8, 7, 6]])
print(f'`matriz`:\n{matriz}\n')
print(f'`np.prod(matriz)`:\n{np.prod(matriz)}\n')
print(f'`np.prod(matriz, axis=0)`:\n{np.prod(matriz, axis=0)}\n')
print(f'`np.prod(matriz, axis=1)`:\n{np.prod(matriz, axis=1)}\n')
```

```
`matriz`:
[[3 2 4]
 [5 2 1]
 [8 7 6]]

`np.prod(matriz)`:
80640

`np.prod(matriz, axis=0)`:
[120  28  24]

`np.prod(matriz, axis=1)`:
[ 24  10 336]
```

In [50]:
```
print(f'`matriz`:\n{matriz}\n')
print(f'`np.cumprod(matriz)`:\n{np.cumprod(matriz)}\n')
print(f'`np.cumprod(matriz, axis=0)`:\n{np.cumprod(matriz, axis=0)}\n')
print(f'`np.cumprod(matriz, axis=1)`:\n{np.cumprod(matriz, axis=1)}\n')
```

```
`matriz`:
[[3 2 4]
 [5 2 1]
 [8 7 6]]

`np.cumprod(matriz)`:
[    3     6    24   120   240   240  1920 13440 80640]

`np.cumprod(matriz, axis=0)`:
[[  3   2   4]
 [ 15   4   4]
 [120  28  24]]

`np.cumprod(matriz, axis=1)`:
[[  3   6  24]
 [  5  10  10]
 [  8  56 336]]
```

Además de las funciones para operaciones matemáticas, `numpy` también tiene funciones para realizar algunas operaciones que se suelen requerir desde la estadística (https://numpy.org/doc/stable/reference/routines.statistics.html)

In [51]:
```
print(f'media: {np.mean(matriz)}')
print(f'desviación estándar muestral: {np.std(matriz, ddof=1)}')
```

```
media: 4.222222222222222
desviación estándar muestral: 2.438123139721299
```

Multiplicación Matricial

Para la multiplicación de matrices usamos `@`, `matmul()`, o incluso `dot()`. Naturalmente, es necesario tener una especial atención a los tamaños de los arreglos a operar.

In [52]:
```
arr1 = np.array([[1,2,3],
                 [4,5,6]]) # tamaño 2,3
arr2 = np.array([[6,5,4],
                 [3,2,1]]) # tamaño 2,3
```

In [53]:
```
print(arr1 @ arr2.T) # 2,3 y 3,2. Resultado: 2,2
```

```
[[28 10]
 [73 28]]
```

In [54]:
```
print(np.matmul(arr1, arr2.T))
```

```
[[28 10]
 [73 28]]
```

In [55]:
```
print(np.dot(arr1, arr2.T))
```

```
[[28 10]
 [73 28]]
```

Con `matmul` no se puede hacer la multiplicación cA, en donde c es un escalar y A una matriz.

In [56]:
```
try:
    print(0.9 @ arr1) # float y arreglo de tamaño 2,3
except:
    print("No se pudo hacer la operación")
```

```
No se pudo hacer la operación
```

In [57]:
```
try:
    print(np.array([0.9]) @ arr1) # tamaño 1 y tamaño 2,3
except:
    print("No se pudo hacer la operación")
```

```
No se pudo hacer la operación
```

In [58]:
```
try:
    print(np.array([200, 100]) @ arr1) # tamaño 2 y tamaño 2,3
except:
    print("No se pudo hacer la operación")
```

```
[ 600  900 1200]
```

In [59]:
```
try:
    print(np.matmul(arr1, np.array([10, 20, 30]))) # 2,3 y 3
except:
    print("No se pudo hacer la operación")
```

```
[140 320]
```

Aunque el resultado de `matmul()` y `dot()` coincide para arreglos de dimensión dos (matrices), `dot()` en general no hace lo mismo que `matmul()`. Para más información acerca de la función `dot()`, consultar https://numpy.org/doc/stable/reference/generated/numpy.dot.html#numpy.dot