# Numpy

January 17, 2026

INSTITUTO SUPERIOR TECNOLÓGICO QUITO

Formamos tu PROPÓSITO DE VIDA

Numpy


## 0.1 Nombre: *Adriel Bedoya*

## 0.2 NumPy ndarray

Significa ***Num***erical ***Py***thon, es la libería estandar para el análisis numérico de Python. Su estructura de datos es multidimensional y es bastante eficiente porque esta escrita en **C**: ndarray.

Además, es una colección de funciones para álgebra lineal, estadística descriptiva y ayuda al procesamiento de datos con **np.where**

[2]:
```
#Primero abrimos la consola de jupiter y ponemos "pip install numpy"
#Segundo, importamos dentro del code
import numpy as np
```

### 0.2.1 Performance Sample, Numpy vs Plain Python

***En cristiano.*** **Muestra de Rendimiento: Numpy vs Python Puro**

[6]:
```
my_list = list(range(10000000)) #Utilizando listas en python
"""
Se duplican todos los valores de una lista de 10 millones de elementos.
Se mide el tiempo de ejecución de esa operación usando Python Puro.
"""
%time my_list = [x * 2 for x in my_list]
```

```
CPU times: total: 328 ms
Wall time: 338 ms
```

[5]:
```
my_arr = np.arange(10000000) # Utilizando NumPy Arrays
"""
Se duplican 10 millones de valores usando NumPy para medir el tiempo de␣
 ↪ejecución
Normalmente es mucho más rápido que hacerlo con listas y list comprehension en␣
 ↪Python puro.
Usa menos sobrecarga de interpretación, por eso NumPy destaca en operaciones␣
 ↪numéricas masivas.
"""
%time my_arr2 = my_arr * 2
```

```
CPU times: total: 15.6 ms
Wall time: 15.8 ms
```

### 0.2.2 Formas de Crear ndarrays

[7]:
```
array = np.array([ [2,71,0,34], [2,171,-35,34] ]) # 2D
print(array)
```

```
[[  2  71   0  34]
 [  2 171 -35  34]]
```

[9]:
```
lista_compras = [["carne", "huevos","pollo"],
                 ["manzana","pera","mandarina"]] # Estará en 2D
array = np.array(lista_compras)
print(array)
print(type(array))
```

```
[['carne' 'huevos' 'pollo']
 ['manzana' 'pera' 'mandarina']]
<class 'numpy.ndarray'>
```

[11]:
```
print(np.array([100,10,1])) # array a partir de una lista
#para el arange es (inicio, fin, salto) , como un rango normal
print(np.arange(2,10,2.1)) # array a partir de una secuencia(range) 1D
```

```
[100  10   1]
[2.  4.1 6.2 8.3]
```

[16]:
```
#GENERAR ARRAYS CON VALORES FIJOS

# Generar arrays con valores fijos
print("Crea un array 1D con 2 elementos")
print(np.zeros(2)) #Crea un array 1D con 2 elementos y todos los valores son 0
print()
#Matriz 2D de tamaño 4x3
print("Crea una matriz 2D de tamaño 4x3")
print(np.ones((4,3))) # Todos los valores son 1
print()

#Crea un array 3D con forma de 2x2x2
print("Crea un array 3D con forma de 2x2x2")
print(np.empty((2,2,2))) # no inicializa valores, contiene datos trash de la␣
 ↪memoria
print()

#Crea una matriz 2D de tamaño 5x2
print("Crea una matriz 2D de tamaño 5x2") #5 abajo 2 al frente
print(np.full((5,2),2)) #Todos los valores serán 2
print()

#Crea una matriz identidad 3x3
print("Crea una matriz identidad 3x3")
print(np.eye(3)) #Matriz identidad es que la diagonal principal tenga 1 y el␣
 ↪resto 0.
print()
```

```
Crea un array 1D con 2 elementos
[0. 0.]

Crea una matriz 2D de tamaño 4x3
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]

Crea un array 3D con forma de 2x2x2
[[[9.88e-324 3.51e-322]
  [0.00e+000 1.68e-322]]
```

```
 [[9.88e-324 8.45e-322]
  [      nan 1.68e-322]]]

Crea una matriz 2D de tamaño 5x2
[[2 2]
 [2 2]
 [2 2]
 [2 2]
 [2 2]]

Crea una matriz identidad 3x3
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

### 0.2.3 Obtener Información sobre ndarray

[21]:
```
#Convierte la lista en un array de numpy
numpy_array = np.array([2,71,0,34])
#Array 1D
print(numpy_array)
```

```
[ 2 71  0 34]
```

[22]:
```
#argmax() devuelve la posición del valor más grande del array.
#No devuelve el valor, sino dónde está.
print(numpy_array.argmax())
#np.amax() devuelve el valor máximo del array.
#Es equivalente a max() del python puro.
print(np.amax(numpy_array))
```

```
1
71
```

[24]:
```
#argmin() devuelve la posición del valor más bajo
print(numpy_array.argmin())
#np.amin() devuelve el valor más bajo del array
#equivalente al min()
print(np.amin(numpy_array))
```

```
2
0
```

[28]:
```
#Devuelve las posiciones de elementos que no sean 0
print(numpy_array.nonzero()) #ojo, solo las posiciones, no los valores
```

```
(array([0, 1, 3]),)
```

[29]:
```
#np.full crea un array numpy con valores fijos
#es 3D, 5(bloques), 4(filas), 2(columnas)
#En cristiano, 5 matrices; cada una tiene un tamaño de 4x2 y todas se
#llenan con el valor 2
my_ndarray = np.full((5, 4,2), 2) # matrix 5x2x2
print(my_ndarray)
```

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

 [[2 2]
  [2 2]
  [2 2]
  [2 2]]

 [[2 2]
  [2 2]
  [2 2]
  [2 2]]

 [[2 2]
  [2 2]
  [2 2]
  [2 2]]

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

[30]:
```
"""
recordemos:
my_ndarray = np.full((5, 4,2), 2)
"""

#.size devuelve el total de elementos del array.
#Se calcula multiplicando sus dimensiones = 5x4x2
print(my_ndarray.size)

#devuelve una tupla con el tamaño de cada dimensión.
print(my_ndarray.shape) #5 repeticiones, 4 filas, 2 columnas

#.ndim indica el número de dimensiones.
print(my_ndarray.ndim) #como es 3D, entonces dará 3
```

```
40
```

```
(5, 4, 2)
3
```

[32]:
```
#Es un array numpy de 1D
my_ndarray = np.array([2,71,0,34])
print(my_ndarray)

print(my_ndarray.size) #devuelve 4 elementos
print(my_ndarray.shape) #Como solo es una dimensión, solo devuelve 4 columnas
print(my_ndarray.ndim) # 1D
```

```
[ 2 71  0 34]
4
(4,)
1
```

### Redimensionar los Datos

[34]:
```
#Es una matriz 2D que tiene 8 columnas
my_ndarray = np.array([[0, 71, 21, 19, 213, 412, 111, 98]]) # matrix 1x8

print(my_ndarray)
print(my_ndarray.shape) #1 fila porque tiene doble corchete
print(my_ndarray.ndim) # 2D
```

```
[[  0  71  21  19 213 412 111  98]]
(1, 8)
2
```

[37]:
```
#reshape cambia la organización  2 filas x 4 columnas
new_dims = my_ndarray.reshape(2, 4) #en vez de 1x8, ahora es 2x4
print(new_dims) #antes era 2D 1x8, ahora es 2D pero 2x4
print(new_dims.ndim) #Es 2D

"""
IMPORTANTE:

El número total de elementos no puede cambiar
Solo cambia la forma, no los valores
Si no coincide, NumPy lanza error
"""
print()
```

```
[[  0  71  21  19]
 [213 412 111  98]]
2
```

[39]:
```
new_dims = my_ndarray.reshape(4, 2) #ahora es al revés, 4 filas x 2 col
print(new_dims)
```

```
print(new_dims.ndim)#Sigue siendo 2D
```

```
[[  0  71]
 [ 21  19]
 [213 412]
 [111  98]]
2
```

[41]:
```
#Generamos un array numpy 1D y luego lo reorganizamos 3 filas 1 col
x = np.array([100, 10, 1]).reshape(3,1)
print(x.shape) #tamaño de dimensión = 3x1 = (3,1) recuerda que devuelve una␣
 ↪tupla
print(x) #print
```

```
(3, 1)
[[100]
 [ 10]
 [  1]]
```

[44]:
```
#REDUCIR 1D

#creamos un array numpy con números del 0 al 8, siendo una matriz 3x3 2D
new_dims = np.arange(9).reshape(3,3) #si hay 2 valores en el reshape se hace 2␣
 ↪dimensiones, 3 valores = 3 dimensiones y así
print(new_dims)
print()
#Flatten transforma el array en una dimensión independientemente a la dimensión␣
 ↪que tenga
print("Con flatten")
print(new_dims.flatten()) #Siempre convertirá en 1D
```

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

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

[47]:
```
#Creamos un array con valores de 0 a 15 y le hacemos una matriz 4x4 de 2D
new_dims = np.arange(16).reshape(4,4)
print(new_dims)
print("--------------------")
## el 2 fija el número de filas
#El -1 le dice a NumPy : Calcula automáticamente cuántas columnas necesito
#16 elementos / 2 filas = 8 columnas
print(new_dims.reshape(2,-1))
#En cristiano, Cambia la forma de 4x4 a 2x8
#el -1 calcula automaticamente cuantas columnas caben
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
--------------------
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]]
```

### 0.2.4 Manipulación de ndarrays

[51]:
```
# Cambiar ejes
#Creamos un array numpy de 0 a 15 donde será en una dimensión 2D con 8 filas x 2 columnas
new_dims = np.arange(16).reshape(8,2)
print(new_dims)
#Cambia los ejes
#actualmente axe 0 son filas y axe 1 son col
#Con esto, axe 0 ahora serán col y axe 1 ahora serán filas
#Ahora es una matriz de 2 filas x 8 columnas
print("--------------------------")
print(new_dims.swapaxes(0,1))
```

```
[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]
 [12 13]
 [14 15]]
--------------------------
[[ 0  2  4  6  8 10 12 14]
 [ 1  3  5  7  9 11 13 15]]
```

[53]:
```
# Flip

#Creamos un nuevo array numpy de 0 a 16 con dimensión 2D de 4 filas x 4 col
new_dims = np.arange(16).reshape(4,4)
print(new_dims)
print("----------------------------------------")
#Axis = None invierte el array completo como si fuera una sola secuencia
#No distingue filas ni columnas
#Primero "aplana" conceptualmente el array y luego lo invierte.
print(np.flip(new_dims, axis = None))
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
```

```
 [12 13 14 15]]
----------------------------------------
[[15 14 13 12]
 [11 10  9  8]
 [ 7  6  5  4]
 [ 3  2  1  0]]
```

[55]:
```
# ordenar
#usamos el sort como antes para ordenarlos
array1 = np.array([10,2,9,17])
array1.sort()
print(array1)
```

```
[ 2  9 10 17]
```

[58]:
```
# Juntar dos arrays

#Creamos 2 listas
a1 = [0,0,0,0,0,0]
a2 = [1,1,1,1,1,1]

#Horizontal Stack
#une arrays uno al lado del otro
print(np.hstack((a1,a2)))
print(np.hstack((a1,a2)).shape) #Es un array 1D, solo que más largo
print("----------------------------------------")

#Vertical Stack
#Apila uno encima del otro
print(np.vstack((a1,a2)))  #Convierte los arrays 1D en filas de una matriz 2D
print(np.vstack((a1,a2)).shape)
```

```
[0 0 0 0 0 0 1 1 1 1 1 1]
(12,)
----------------------------------------
[[0 0 0 0 0 0]
 [1 1 1 1 1 1]]
(2, 6)
```

[62]:
```
#Ambos son arrays 2D
array1 = np.array([[2, 4], [6, 8]])
array2 = np.array([[3, 5], [7, 9]])
#concatenate une arrays a lo largo de un eje específico
#Se concatenan uno debajo del otro
#Se suman las filas
#IMPORTANTE: Las columnas deben coincidir

"""
```

```
concatenate une arrays por el eje indicado:
axis=0 agrega filas, axis=1 agrega columnas.
"""

print(np.concatenate((array1, array2), axis = 0))#Es decir, unimos lso dos␣
 ↪arrays agregando filas
print("------------------------")
print(np.concatenate((array1, array2), axis = 1))#Unimos arrays mediante␣
 ↪columnas
```

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

[67]:
```
# dividir un array

#Creamos un array de 0 a 15, matriz 2D 4 filas x 4 col
array = np.arange(16).reshape(4,4)
print("ORIGINAL")
print(array)
print("--------------")
#Divide el array en 2 partes
#Como es eje 0, quiere decir que será mediante filas
#4 filas / 2 = 2 filas por parte
print(np.array_split(array, 2, axis = 0))
print("--------------------")
#no se especifíca el eje, pero por defecto es axis = 0
"""
NumPy no puede dividir exactamente, entonces
Reparte las filas lo más equitativamente posible
Las primeras partes reciben una fila extra
"""
print(np.array_split(array, 3)) # divide array en 3 partes "iguales"
```

```
ORIGINAL
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
--------------
[array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])]
--------------------
```

```
[array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11]]), array([[12, 13, 14, 15]])]
```

[71]:
```
#Crea un array de 0 al 15, 2D siendo matriz 4x4
array = np.arange(16).reshape(4,4)
print(array)
print("------------------------------")
#Divide en el índice 3 del array mediante columnas (por el axis = 1)
#Las col tienen 0,1,2,3,4
#Antes de llegar a la tercera columna, lo divide
print(np.array_split(array, [3], axis = 1)) # divide array, por la fila 3
print("----------------------")
#probemos con fila
#hay 4 filas: indice 0,1,2,3
print(np.array_split(array, [2], axis = 0)) #En el 1 ya lo corta
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
------------------------------
[array([[ 0,  1,  2],
       [ 4,  5,  6],
       [ 8,  9, 10],
       [12, 13, 14]]), array([[ 3],
       [ 7],
       [11],
       [15]])]
----------------------
[array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])]
```

[73]:
```
#Creamos un array del 0 al 63, 2D siendo matriz de 8x8
array = np.arange(64).reshape(8,8)
print(array)
print("-----------------------------")
#Corta antes de la fila 1 y antes de la fila 3
print(np.array_split(array, [1, 3], axis = 0))
print("-----------------------------")
#Corta antes de la col 1 y antes de la col 3
print(np.array_split(array, [1, 3], axis = 1)) # divide array por la columna
```

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]
 [32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47]
```

```
 [48 49 50 51 52 53 54 55]
 [56 57 58 59 60 61 62 63]]
----------------------------
[array([[0, 1, 2, 3, 4, 5, 6, 7]]), array([[ 8,  9, 10, 11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20, 21, 22, 23]]), array([[24, 25, 26, 27, 28, 29, 30,
31],
       [32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47],
       [48, 49, 50, 51, 52, 53, 54, 55],
       [56, 57, 58, 59, 60, 61, 62, 63]])]
----------------------------
[array([[ 0],
       [ 8],
       [16],
       [24],
       [32],
       [40],
       [48],
       [56]]), array([[ 1,  2],
       [ 9, 10],
       [17, 18],
       [25, 26],
       [33, 34],
       [41, 42],
       [49, 50],
       [57, 58]]), array([[ 3,  4,  5,  6,  7],
       [11, 12, 13, 14, 15],
       [19, 20, 21, 22, 23],
       [27, 28, 29, 30, 31],
       [35, 36, 37, 38, 39],
       [43, 44, 45, 46, 47],
       [51, 52, 53, 54, 55],
       [59, 60, 61, 62, 63]])]
```

### 0.2.5 Índices y slicing

**- De manera análoga a índices y slicing para listas**

[75]:
```
#Creamos un array empezamos en 10 y terminando en 15
#Es 1D
array = np.arange(10, 16)
print(array)
print(array[-1]) # Se va al índice final
print(array[:-1]) # slice
```

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

[78]:
```
# bidimensional
#Es 2D
#Creamos un array del 0 al 15, 2D en matriz de 4 filas x 4 col
array = np.arange(16).reshape(4,4)
print(array)

print("------------------")
print(array[2, 2]) # índice 2 fila e índice 2 columna
print(array[3,2]) # fila 3 2 columna
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
------------------
10
14
```

[81]:
```
# multidimensional

#Creamos un array de 0 a 15, 3D 2 matrices, 3 filas 6 columnas
array = np.arange(36).reshape(2,3,6)
print(array)
print("---------------")
print(array[0,2,4]) # primera matriz, segundo índice fila, cuarto indice col
print(array[1][2][4]) # segunda matriz, segundo índice fila, cuarto indice col
```

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

 [[18 19 20 21 22 23]
  [24 25 26 27 28 29]
  [30 31 32 33 34 35]]]
---------------
16
34
```

### 0.2.6 NumPy es row major

[82]:
```
#Creamos un array de 1D
row_major = np.array([1, 2, 3, 4, 5, 6])
#row major llena la matriz, fila por fila
row_major = row_major.reshape(2,3)
print(row_major)
```

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

[86]:
```
#Es lo mismo que el otro pero lo hace con C-row
row_major = np.array([1, 2, 3, 4, 5, 6])
row_major = row_major.reshape(2,3,order='C') # by default, row major
print(row_major)

print("--------------------")

#Para ver la diferencia, lo hacemos con F
col_major = np.array([1, 2, 3, 4, 5, 6])
col_major = col_major.reshape(2,3,order='F') # con el F, se ordena por columnas
print(col_major)
```

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

**Slice**

[88]:
```
# se puede hacer slice por fila o columna
#Creamos un array numpy de 0 al 80, 2D 9 filas x 9 col
array = np.arange(81).reshape(9,9)
print(array)
print("-----------------------------------")
#Empiza por la fila uno y termina en la fila 2 (excluye la fila 3),
#empieza en la columna 2 hasta el final y va saltando cada 2
print(array[1:3, 2::2])
```

```
[[ 0  1  2  3  4  5  6  7  8]
 [ 9 10 11 12 13 14 15 16 17]
 [18 19 20 21 22 23 24 25 26]
 [27 28 29 30 31 32 33 34 35]
 [36 37 38 39 40 41 42 43 44]
 [45 46 47 48 49 50 51 52 53]
 [54 55 56 57 58 59 60 61 62]
 [63 64 65 66 67 68 69 70 71]
 [72 73 74 75 76 77 78 79 80]]
-----------------------------------
[[11 13 15 17]
 [20 22 24 26]]
```

[90]:
```
# se puede hacer slice por fila o columna
array = np.arange(9).reshape(3,3)
print(array)
print("----------")
print(array[0,:])
```

```
[[0 1 2]
 [3 4 5]
```

```
 [6 7 8]]
----------
[0 1 2]
```

### 0.2.7 Diferencias entre índices y slicing en listas y ndarrays:

***Retornan una diferencia, no una copia***

[93]:
```
# Slice en Listas, retorna una copia
#Creamos una lista
lista = [0,1,2,3,4,5,6,7,8,9]
#Almacenamos una copia donde veremos desde el indice 0 al 2
my_copy = lista[0:2]
print(my_copy)
#Cambiamos que el indice 0 será de 222
my_copy[0] = 222
print(lista)
print(my_copy) #Devuelve una nueva lista en forma de copia
```

```
[0, 1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[222, 1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[98]:
```
# slicing retorna una referencia en ndarrays

#Creamos un array numpy del 0 al 9, 1D
array = np.arange(10)
my_copy = array[2:5] # devuelve una referencia
print(array)
print("--------------------------")
print(my_copy)
print("-----------------")
my_copy[0] = 222
print(array)
print("-*-------------------------")
print(my_copy)
```

```
[0 1 2 3 4 5 6 7 8 9]
--------------------------
[2 3 4]
-----------------
[  0   1 222   3   4   5   6   7   8   9]
-*-------------------------
[222   3   4]
```

[100]:
```
# Para copiar ND ARRAYS
array = np.arange(10)
my_copy = array[2:5].copy() # hay que hacer la copia explicitamente
```

```
my_copy[0] = 222
print(array)
print("----------------------")
print(my_copy)
```

```
[0 1 2 3 4 5 6 7 8 9]
----------------------
[222   3   4]
```

[103]:
```
#Agregar valores en un corte
#1D
array = np.arange(10)
print(array)
print("----------")
print(array[0:3])
print("---------------")
#Cambiamos el valor de -1 desde el índice 0 hasta el 2 (recuerda que el 3 se␣
 ↪excluye)
array[0:3] = -1
print(array)
```

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

**Slicing condicionales**

[109]:
```
#Creamos un array numpy desde el -3 hasta el 3, 1D
array = np.arange(-3,4)
print(array)
print("---------------")
#Creamos una variable donde decimos si array%2 == 0, imprima true, sino false
mask = array %2 == 0
print(mask)
print("---------------")
#Imprimimos los valores por índice
print(array[mask])
print("---------------")
#imprimimos la condicional dentro
print(array[array %2 == 0])
print("---------------")
#Podemos asignar valores a la condicional, ahora valdrá 0 los pares
array[mask] = 0
print(array)
```

```
[-3 -2 -1  0  1  2  3]
---------------
```

```
[False  True False  True False  True False]
---------------
[-2  0  2]
---------------
[-2  0  2]
---------------
[-3  0 -1  0  1  0  3]
```

[110]:
```
array = np.arange(-3,4)
print(array)
print("---------------")
array[array < 0] = 0
print(array)
print("---------------")
```

```
[-3 -2 -1  0  1  2  3]
---------------
[0 0 0 0 1 2 3]
---------------
```

[113]:
```
# slicing condicional
#Creamos un array numpy desde el -3 hasta el 3, 1D
array = np.arange(-3,4)
print(array)
print("---------------")

#De la condicional si es par
mask = array % 2 == 0
print(mask)
print("---------------")

#Hacemos que devuelva una copia
array_par = array[mask] #Devolverá los pares
print(array_par)
print("---------------")

array_par[0] = -10 #decimos que de los pares, el primer índice sea -10
print(array_par)
print("---------------")
```

```
[-3 -2 -1  0  1  2  3]
---------------
[False  True False  True False  True False]
---------------
[-2  0  2]
---------------
[-10   0   2]
---------------
```

- Acceso a múltiples valores con listados de índices

[120]:
```
#Creamos un array del 0 al 15, 2D matriz de 4 filas x 4 col
array = np.arange(0,16).reshape(4,4)
print(array)
print("------------------")

print("Seleccionado filas")
#Seleccionamos mediante índice la fila 1 y 2
print(array[[1,2]])
print("------------------")

print("Seleccionando Columnas")
#Seleccionamos mediante índice todas las filas de la columna 1 y 2
print(array[:,[1,2]])
print("------------------")
print("Juntamos filas y columnas")
#Primero, seleccionamos filas mediante el índice 1 y 2 a todas las col
#Segundo, seleccionamos col mediante indice 1 y 2 a todas las filas
print(array[[1,2],:][:,[1,2]])
print("------------------")
print("Lo mismo de arriba pero método de numpy")
print(array[np.ix_([1,2],[1,2])])
print("------------------")
print("Copia slice")
# copia del array de la fila índice 2, fila índice 3 y índice fila 1
ar = array[[2,3,1]]
print(ar)
print("------------------")
#En la fila índice 0, todos los valores serán 999
ar[0] = 999
print(ar)
print(array)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
------------------
Seleccionado filas
[[ 4  5  6  7]
 [ 8  9 10 11]]
------------------
Seleccionando Columnas
[[ 1  2]
 [ 5  6]
 [ 9 10]
 [13 14]]
```

```
------------------
Juntamos filas y columnas
[[ 5  6]
 [ 9 10]]
------------------
Lo mismo de arriba pero método de numpy
[[ 5  6]
 [ 9 10]]
------------------
Copia slice
[[ 8  9 10 11]
 [12 13 14 15]
 [ 4  5  6  7]]
------------------
[[999 999 999 999]
 [ 12  13  14  15]
 [  4   5   6   7]]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

### 0.3 Funciones Matemáticas

Aplicar operaciones entre escalares y arrays o entre arrays

[126]:
```
# listas y escalares
lista = [0,1,2,3]
lista *= 3 #Se repetirá 3 veces
print(lista)
print("------------------------------------------------")
lista += [1] #Le agregamos uno al final
print(lista)
```

```
[0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
------------------------------------------------
[0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 1]
```

[125]:
```
# ndarray hace las operaciones sobre los elementos

#Crea 4 elementos con valor de 1
#cada elemento es un entero de 8 bits
#int8 puede guardar valores entre -128 y 127.
array = np.ones(4, dtype = np.int8)
print(array)
print("*------------------------------------------------")
#NumPy NO multiplica el array como un todo, sino cada elemento
#[1 1 1 1]  luego [2 2 2 2]
```

```
array *= 2
print(array)
print("-------------------------------------------------------")
array += 10
print(array)
print(array.dtype) # check data type of array
```

```
[1 1 1 1]
*-----------------------------------------------
[2 2 2 2]
-------------------------------------------------------
[12 12 12 12]
int8
```

[137]:
```
# operaciones entre arrays es como operaciones entre matrices

#Creamos un array de 4 valores 1, luego se hace 2 y son 2,2,2,2
array_1 = np.ones(4) * 2
#Creamos un array del 0 al 4, porque el +1 lo incluye
array_2 = np.arange(4) + 1
print(array_1)
print("-------------------------------------------------------")
print(array_2)
print("-------------------------------------------------------")
print("Suma de Array") #mediante índices
print(array_1 + array_2)
print("-------------------------------------------------------")
print("Resta de Array")
print(array_1 - array_2)
print("-------------------------------------------------------")
print("Multiplicación de Array")
print(array_1 * array_2)
print("-------------------------------------------------------")
print("División de Array")
print(array_2 / array_1)
print("-------------------------------------------------------")
```

```
[2. 2. 2. 2.]
-------------------------------------------------------
[1 2 3 4]
-------------------------------------------------------
Suma de Array
[3. 4. 5. 6.]
-------------------------------------------------------
Resta de Array
[ 1.  0. -1. -2.]
-------------------------------------------------------
Multiplicación de Array
```

```
[2. 4. 6. 8.]
------------------------------------------------------------
División de Array
[0.5 1.  1.5 2. ]
------------------------------------------------------------
```

### 0.4 Broadcasting (transmisión de operaciones)

Si las arrays no tienen las mismas dimensiones, se hace broadcast del pequeño al grande ———- broadcasting

| Tipo | Operación | Descripción |
|---|---|---|
| Unario | abs | Valor absoluto de cada elemento |
| Unario | sqrt | Raíz cuadrada de cada elemento |
| Unario | exp | e^x, siendo x cada elemento |
| Unario | log, log10, log2 | Logaritmos en distintas bases de cada elemento |
| Unario | sign | Retorna el signo de cada elemento: -1 si es negativo, 0 si es cero, 1 si es positivo |
| Unario | ceil | Redondea cada elemento hacia arriba |
| Unario | floor | Redondea cada elemento hacia abajo |
| Unario | isnan | Retorna True si el elemento es NaN |
| Unario | cos, sin, tan | Operaciones trigonométricas básicas |
| Unario | arccos, arcsin, arctan | Operaciones trigonométricas inversas |
| Binario | add, subtract, multiply, divide | Suma, resta, multiplicación y división de dos arrays |
| Binario | maximum, minimum | Retorna el valor máximo o mínimo de cada pareja de elementos |
| Binario | equal, not_equal | Retorna la comparación de cada pareja de elementos (igual o no igual) |
| Binario | greater, greater_equal, less, less_equal | Comparaciones de cada pareja de elementos: > , >= , < , <= |

[133]:
```
#EJEMPLOS DE OPERADORES
array = np.arange(5)
print(array)
#Calcula la raiz cuadrada del elemento del array
#Devuelve un nuevo array con resultados
"""
| Elemento | sqrt       |
| -------- | ---------- |
| 0        | 0.0        |
| 1        | 1.0        |
| 2        | 1.41421356 |
| 3        | 1.73205081 |
| 4        | 2.0        |


"""
```

```
np.sqrt(array)
```

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

[133]:
```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ])
```

[135]:
```
# ejemplos de operadores
#Creamos un array del 0 al 3
array1 = np.arange(4)
#Creamos un array con los valores ya dados
array2 = np.array([0,-1,2,-3])

print(array1)
print(array2)
#greater compara los valores de los índices entre ellos
#Devuelve true si es mayor el primero que el segundo
#array1[i] > array2[i]
print(np.greater(array1,array2))
#array1[i] < array2[i]
print(np.less(array2, array1))
```

```
[0 1 2 3]
[ 0 -1  2 -3]
[False  True False  True]
[False  True False  True]
```

### 0.5 Estadística Descriptiva

- Importante saber la naturaleza de los datos
- Valores máximos, mínimos, distribución, etc

| Función | Descripción |
|---|---|
| sum(arr) | Suma de todos los elementos de `arr` |
| mean(arr) | Media aritmética de los elementos de `arr` |
| std(arr) | Desviación estándar de los elementos de `arr` |
| cumsum(arr) | Devuelve un array con la suma acumulada de cada elemento con todos los anteriores |
| cumprod(arr) | Devuelve un array con el producto acumulado de cada elemento con todos los anteriores |
| min(arr), max(arr) | Mínimo y máximo de `arr` |
| any(arr) | En un array de tipo booleano, retorna True si **algún elemento es True** |
| all(arr) | En un array de tipo booleano, retorna True si **todos los elementos son True** (o <>0 en valores numéricos) |
| unique(arr) | Devuelve un array con los valores únicos de `arr` |
| in1d(arr1, arr2) | Devuelve un array booleano indicando si cada elemento de `arr1` está en `arr2` |
| union1d(arr1, arr2) | Devuelve la unión de ambos arrays |

| Función | Descripción |
|---|---|
| intersect1d(arr1, arr2) | Devuelve la intersección de ambos arrays |

[138]:
```
# sum != cumsum
#Creamos un array del 0 al 9
array1 = np.arange(9)
print(array1)
print("------------------------------------------------------------")
print(array1.sum()) #Suma todos los elementos del array
print("------------------------------------------------------------")
print(array1.cumsum()) #Devuelve de uno en uno la suma de elementos
```

```
[0 1 2 3 4 5 6 7 8]
------------------------------------------------------------
36
------------------------------------------------------------
[ 0  1  3  6 10 15 21 28 36]
```

[139]:
```
# any vs all

#Creamos un array desde el -8 hasta el 1, 2D matriz de 2 filas x 5 col
array1 = np.arange(-8,2).reshape(2,5)
print(array1)
print("------------------------------------------------------------")
#cualquier número distinto de 0 se considera True, y 0 se considera False.
#verifica si al menos un elemento es "True".
print(array1.any())
print("------------------------------------------------------------")
#verifica que todos los elementos sean true, hay un 0 así que ya no lo es
print(array1.all())
```

```
[[-8 -7 -6 -5 -4]
 [-3 -2 -1  0  1]]
------------------------------------------------------------
True
------------------------------------------------------------
False
```

[140]:
```
# todas las funciones aceptan parametro 'axis'
# 0 para columnas, 1 para filas, 2 para profundidad...

#Creamos un array del 0 al 9, 2D matriz de 2 filas x 5 col
array1 = np.arange(10).reshape(2,5)
print(array1)
print("------------------------------------------------------------")
#Verifica mediante columnas si todos no tienen 0
#la primera tiene 0, asi que es False
```

```
#las demás no tienen 0, así que son True
print(array1.all(axis=0))
print("-----------------------------------------------------------")
#Ahora hacemos lo mismo pero mediante filas
#En la primera fila "0 1 2 3 4" hay un 0, así que es False
#En la otra fila, no hay 0, así que es True
print(array1.all(axis=1))
print("-----------------------------------------------------------")
print(array1.all())
```

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

## 0.6 Álgebra Lineal

- numpy.linalg contiene funciones para álgebra lineal.
- dot product, multiplicación de matrices, cálculo del determinante, factorizaciones, etc

| Función | Descripción |
|---|---|
| dot(mat1, mat2) | Devuelve el producto escalar entre dos arrays. Si son matrices 2D, es equivalente a la multiplicación de ambas |
| matmul(mat1, mat2) | Devuelve el producto entre dos matrices |
| trace(mat) | Suma de los elementos de la diagonal de la matriz |
| det(mat) | Devuelve el determinante de la matriz |
| eig(mat) | Computa los autovalores y autovectores de la matriz cuadrada |
| inv(mat) | Devuelve la inversa de la matriz |
| qr(mat) | Computa la factorización QR de la matriz |
| solve(A, b) | Resuelve el sistema lineal de ecuaciones Ax = b, cuando A es cuadrada |
| transpose(mat) | Devuelve la transpuesta de la matriz |

[143]:
```
#Creamos un array del 0 al 11, 2D matriz de 4 filas x 3 col
A = np.arange(12).reshape(4,3)
#Creamos un array del 0 al 5, 2D matriz de 3 filas x 2 col
B = np.arange(6).reshape(3,2)
print(A)
print(B)
print("-----------------------------------")
#Devuelve el producto entre dos matices (multiplica)
print(np.matmul(A, B))
#Se multiplica filas x columnas
#0*0 + 1*2 + 2*4 = 0 + 2 + 8 = 10
```

```
#0*1 + 1*3 + 2*5 = 0 + 3 + 10 = 13 (como no alcanzo, completo con la de abajo)
print("-----------------------------------")
#Devuelve el producto escalar entre dos matices, si es 2D es quivalente a la␣
 ↪multiplicación entre las 2
print(np.dot(A, B))
print("-----------------------------------")
print(A @ B) #es el matmul en símbolo
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
[[0 1]
 [2 3]
 [4 5]]
-----------------------------------
[[10 13]
 [28 40]
 [46 67]
 [64 94]]
-----------------------------------
[[10 13]
 [28 40]
 [46 67]
 [64 94]]
-----------------------------------
[[10 13]
 [28 40]
 [46 67]
 [64 94]]
```

[146]:
```
import numpy.linalg as lg
import math

# resolver sistema de ecuaciones
# 3x + 2y + 5z = 12
# x - 3y -z = 2
# 2x -y +12z = 32
# Ax = b --> resolver para x

a = np.array([ [3,2,5], [1,-3,-1], [2,-1,12] ])
"""
a = np.array([
    [3, 2, 5],   # coeficientes de la 1ra ecuación
    [1, -3, -1], # coeficientes de la 2da ecuación
    [2, -1, 12]  # coeficientes de la 3ra ecuación
])
```

```
"""
b = np.array([12,2,32]) # resultados de cada ecuación
print(a)
print(b)

"""
a= matriz de coeficientes (3x3)
b= vector de resultados (3 elementos)
"""
print("-------------------------------------")

#Devuelve los valores
#1) lg.solve resuelve Ax = b
#2) Devuelve el vector x , y, z con la solución del sistema
x,y,z = lg.solve(a,b)
print(f'x={x} y={y} z={z}')
print("-------------------------------------")

#Verificamos la solución
#math.isclose(a, b) devuelve True si a y b son iguales o muy cercanos
print(math.isclose(3*x + 2*y +5*z, 12))
print(math.isclose(x - 3*y - z, 2))
print(math.isclose(2*x -y + 12*z, 32))
```

```
[[ 3  2  5]
 [ 1 -3 -1]
 [ 2 -1 12]]
[12  2 32]
-------------------------------------
x=0.7543859649122807 y=-1.2280701754385965 z=2.43859649122807
-------------------------------------
True
True
True
```

[147]:
```
#matriz identidad
#Propiedad clave: multiplicar cualquier matriz A por I no cambia A:

#Crea una matriz identidad 4x4
I = np.identity(4)
print(I)
print("-------------------------------------")
#Crea una matriz de 0 a 15, 2D 4 filas x 4 columnas
A = np.arange(16).reshape(4,4)
print(A)
print("-------------------------------------")
```

```
#producto matricial A × I
#La propiedad de la identidad asegura que el resultado es exactamente A
#Cada fila de A se mantiene igual, porque la identidad no altera los valores.
print(np.matmul(A,I))
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
-------------------------------------
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
-------------------------------------
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]]
```

## 0.7 Filtrado de Datos

- Filtrar y modificar datos numéricos con np.where
- Retorna A o B en función de una condición en un array

[153]:
```
#Creamos un array 1D
prices = np.array([0.99, 14.49, 19.99, 20.99, 0.49])
#Creamos una condicional que si los precios son menores a 1
mask = prices < 1
#slicing condicional
print(prices[prices < 1]) #Se muestran los valores que son menores a 1
print("-------------------------------------")
# mask con los elementos menores de 1
mask = np.where(prices < 1, True, False) #Según los resultados, pondrá True o␣
 ↪False
print("Mask")
print(mask) #print True o False
print(prices[mask]) #Se muestran los valores de la condicional
print("-------------------------------------")
prices[mask] = 1.0 #Cambia todos los elementos menores de 1 a 1.0
print(prices)#print
print("-------------------------------------")
#Usar np.where para reemplazar valores sin modificar el array original
#Verifica la condición prices < 1
#Si es True entonces reemplaza por 1.0
#Si no lo es, deja el precio
print(np.where(prices < 1, 1.0, prices))
```

```
[0.99 0.49]
------------------------------------
Mask
[ True False False False  True]
[0.99 0.49]
------------------------------------
[ 1.   14.49 19.99 20.99  1.  ]
------------------------------------
[ 1.   14.49 19.99 20.99  1.  ]
```

[157]:
```python
#Creamos en una variable un array con 6 valores, el cual se hará una matriz 2D␣
 ↪2 filas x 3 col
bank_transfers_values = np.array([0.99, -1.49, 19.99, 20.99, -0.49, 12.1]).
 ↪reshape(2, 3)
print(bank_transfers_values)
print("------------------------------------")
#Guarda en una variable la condicional: si la transferencias son mayores a 0 se␣
 ↪quedan igual, caso contrario se hacen 0
clean_data = np.where(bank_transfers_values > 0, bank_transfers_values, 0)
print(clean_data) #todos los negativos se hacen 0
print("------------------------------------")
# limpiar NaN (sustitución)

#nan = not a number
#significa Not A Number, usado para representar valores faltantes o indefinidos.
data = [10, 12, -143, np.nan, 1, -3] # np.nan --> NotANumber, elemento especial
#np.isnan(data) = devuelve True para los elementos que son NaN, False para los␣
 ↪demás
#np.where(np.isnan(data), 0, data) = reemplaza NaN por 0, mantiene los demás:
data_clean = np.where(np.isnan(data), 0, data)
print(data_clean)
print("------------------------------------")
print(id(data))
print("------------------------------------")
print(id(data_clean))
```

```
[[ 0.99 -1.49 19.99]
 [20.99 -0.49 12.1 ]]
------------------------------------
[[ 0.99  0.   19.99]
 [20.99  0.   12.1 ]]
------------------------------------
[  10.   12. -143.    0.    1.   -3.]
------------------------------------
1707251499072
------------------------------------
1707252117744
```

[161]:
```
# np.where puede seleccionar elementos

#Creamos array 2x3
bank_transfers_values = np.array([0.99, -1.49, 19.99, 20.99, -0.49, 12.1]).
 ↪reshape(2, 3)
print(bank_transfers_values)
print("----------------------------------")
# array de índices para los que la condición es True
credits = np.where(bank_transfers_values > 0) #Todos los numeros menores a 0
print(credits)
print("----------------------------------")
#bank_transfers_values[credits] devuelve los valores que cumplen la condición
print(bank_transfers_values[credits])
#Esto es equivalente a usar el slicing condicional directamente:
print(bank_transfers_values[bank_transfers_values > 0])
```

```
[[ 0.99 -1.49 19.99]
 [20.99 -0.49 12.1 ]]
----------------------------------
(array([0, 0, 1, 1]), array([0, 2, 0, 2]))
----------------------------------
[ 0.99 19.99 20.99 12.1 ]
[ 0.99 19.99 20.99 12.1 ]
```

[165]:
```
# valor de array dependiendo del valor en array referencia

#Crear un array de 0 a 5
rewards_default = np.arange(6)
print(rewards_default)
print("----------------------------------")
#Creamos un array del 0 al 5 y luego se multiplica cada uno de ellos * 10
rewards_upgrade = np.arange(6) * 10
print(rewards_upgrade)
print("----------------------------------")
#Creamos un array con datos pre establecidos
daily_points = np.array([0, 0, 1, 6, 0, 2])
print(daily_points)
print("----------------------------------")
#Creamos una variable con una condicional de que si los
#daily_points son mayores a 1, entonces da rewards_upgrade, caso contrario
#da rewards_default
final_rewards = np.where(daily_points > 1, rewards_upgrade, rewards_default)
print(final_rewards)
```

```
[0 1 2 3 4 5]
----------------------------------
[ 0 10 20 30 40 50]
----------------------------------
```

```
[0 0 1 6 0 2]
----------------------------------
[ 0  1  2 30  4 50]
```

## 0.8 Números aleatorios

- Python módulo random

[166]:
```
import random
```

[169]:
```
#Da un número random entre el 0 y el 1
print(random.random())
#Da un número random entero entre el 0 y el 5
print(random.randint(0,5))
#Da un número real entre 0 a 5 (float)
print(random.uniform(0,5))
```

```
0.412566289226884
0
2.8522817582556415
```

[172]:
```
# ELEGIR UN POKEMON WOAAAAAAAAAAAAAAAAAAAAAAA
names = ['Pikachu','Eevee','Charmander']
#Elección al azar dentro de una colección
random.choice(names)
```

[172]:
```
'Pikachu'
```

[ ]:
```
#lo estuve probando hasta que saliera el pikachu jeje
```

### 0.8.1 NumPy.random

- Generar fácilmente listas con valores aleatorios

[179]:
```
#Genera un número random entre 0 y 1
print(np.random.rand())
## Genera un número entero entre dos valores
print(np.random.randint(0,5))
#numero random máximo (excluyendo el número), el size indica las dimensiones
print(np.random.randint(3, size=(2, 4)))
```

```
0.0026124702875048866
1
[[0 0 2 1]
 [2 0 0 0]]
```

[185]:
```
# array de elementos aleatorios
#(distribucion normal gaussiana, media 0, desviación 1)
print(np.random.randn(4,2))
```

```
[[-0.35882895  0.6034716 ]
 [-1.66478853 -0.70017904]
 [ 1.15139101  1.85733101]
 [-1.51117956  0.64484751]]
```

[184]:
```
# otras distribuciones

#'n' intentos, 'p' probabilidad
#n es cuando intentos lo hace y p es la probabilidad que tiene de hacerlo
print(np.random.binomial(n=5, p=0.3))
# distribución uniforme, probabilidad del 0% al 10%
print(np.random.uniform(low=0, high=10))
# 'lam' número de ocurrencias esperadas, retornar 'size' valores
#lam=2 = promedio esperado de ocurrencias por intervalo
#size=(2,2) = forma del array de salida, en este caso es un 2 filas x 2 col
print(np.random.poisson(lam=2, size=(2,2)))
```

```
1
9.186109079379216
[[3 1]
 [0 2]]
```

### 0.8.2 Random seed

- Números pseudo-aleatorios
- Ordenadores generan números a partir de ecuaciones
- Basadas en un número inicial (seed)
- Bueno para reproducibilidad de experimentos

[187]:
```
i = 5
#Buscamos
#Hacemos un for para recorrer 5 números
for _ in range(5):
    #fija la semilla de la generación de números aleatorios
    #Aquí siempre usamos i=5
    #por eso cada iteración genera exactamente los mismos números.
    np.random.seed(i) # misma semilla
    #np.random.rand() genera un número aleatorio uniforme entre 0 y 1
    print(np.random.rand())
    print(np.random.rand())
    #Con la misma semilla, los números aleatorios son reproducibles.
    print("--------------------------------------")
print()
#range(6) genera los números 0, 1, 2, 3, 4, 5
for i in range(6):
    #Aquí cada iteración tiene una semilla diferente: 0, 1, 2, 3, 4, 5
    #Por eso cada par de números aleatorios será diferente en cada iteración.
    np.random.seed(i) # diferentes semillas
```

```
    #genera un número aleatorio uniforme entre 0 y 1.
    #Se hace dos veces por iteración,
    #por eso verás dos números distintos por semilla.
    print(np.random.rand())
    print(np.random.rand())
```

```
0.22199317108973948
0.8707323061773764
--------------------------------------
0.22199317108973948
0.8707323061773764
--------------------------------------
0.22199317108973948
0.8707323061773764
--------------------------------------
0.22199317108973948
0.8707323061773764
--------------------------------------
0.22199317108973948
0.8707323061773764
--------------------------------------

0.5488135039273248
0.7151893663724195
0.417022004702574
0.7203244934421581
0.43599490214200376
0.025926231827891333
0.5507979025745755
0.7081478226181048
0.9670298390136767
0.5472322491757223
0.22199317108973948
0.8707323061773764
```

## 0.9 Ejercicios Github

[189]:
```
"""
2. Print the numpy version and the configuration ( )
4. How to find the memory size of any array ( )
6. Create a null vector of size 10 but the fifth value which is 1 ( )
8. Reverse a vector (first element becomes last) ( )
10. Find indices of non-zero elements from [1,2,0,0,4,0] ( )
12. Create a 3x3x3 array with random values ( )
14. Create a random vector of size 30 and find the mean value ( )
16. How to add a border (filled with 0's) around an existing array? ( )
18. Create a 5x5 matrix with values 1,2,3,4 just below the diagonal ( )
```

```
20. Consider a (6,7,8) shape array, what is the index (x,y,z) of the 100th␣
 ↪element? ( )
"""
#En caso que no entiendan inglés, dejo la versión en español xd

"""
2. Imprimir la versión de numpy y la configuración ( )
4. Cómo encontrar el tamaño en memoria de cualquier array ( )
6. Crear un vector nulo de tamaño 10 pero con el quinto valor igual a 1 ( )
8. Invertir un vector (el primer elemento se convierte en el último) ( )
10. Encontrar los índices de los elementos distintos de cero de [1,2,0,0,4,0]␣
 ↪( )
12. Crear un array 3x3x3 con valores aleatorios ( )
14. Crear un vector aleatorio de tamaño 30 y encontrar su valor medio ( )
16. ¿Cómo agregar un borde (relleno con 0) alrededor de un array existente? ( )
18. Crear una matriz 5x5 con los valores 1,2,3,4 justo debajo de la diagonal␣
 ↪( )
20. Considerar un array con forma (6,7,8), ¿cuál es el índice (x,y,z) del␣
 ↪elemento número 100? ( )
"""
print()
```

[190]:
```
#Print the numpy version and the configuration
import numpy as np
print(np.__version__)
```

```
2.4.0
```

[194]:
```
#How to find the memory size of any array
import numpy as np

soy_ejemplo = np.arange(2,6)
print("El array es: ", soy_ejemplo)
#¿Por qué? recordemos que son bits que se van sumando
print("Tamaño en la memoria", soy_ejemplo.nbytes)
```

```
El array es:  [2 3 4 5]
Tamaño en la memoria 32
```

[198]:
```
#Create a null vector of size 10 but the fifth value which is 1 ( )

import numpy as np

# I create vector size 10
vector = np.zeros(10)
```

```
# Change the fifth value to 1
vector[4] = 1

# Print
print(vector)
```

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

[205]:
```
#Reverse a vector (first element becomes last) ( )
import numpy as np

# First Vector
vector = np.array([1, 2, 3, 4, 5])

# Reversed Vector
vector_invertido = vector[::-1]

# Print sssssssssss
print(vector)
print(vector_invertido)
```

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

[208]:
```
#Find indices of non-zero elements from [1,2,0,0,4,0]
import numpy as np

#Create an array
arraysito = np.array([1, 2, 0, 0, 4, 0])
#print
print("Array:", arraysito)

# I use np.nonzero
index = np.nonzero(arraysito)
print("Índices de elementos distintos de cero:", index[0])

#Another technique to do
index2 = np.where(arraysito != 0)
print("Index : ", index2[0])
```

```
Array: [1 2 0 0 4 0]
Índices de elementos distintos de cero: [0 1 4]
Index :  [0 1 4]
```

[210]:
```
#Create a 3x3x3 array with random values ( )
import numpy as np

# Create a random array
arrayote = np.random.rand(3, 3, 3)
```

```
print(arrayote)
```

```
[[[0.18841466 0.02430656 0.20455555]
  [0.69984361 0.77951459 0.02293309]
  [0.57766286 0.00164217 0.51547261]]

 [[0.63979518 0.9856244  0.2590976 ]
  [0.80249689 0.87048309 0.92274961]
  [0.00221421 0.46948837 0.98146874]]

 [[0.3989448  0.81373248 0.5464565 ]
  [0.77085409 0.48493107 0.02911156]
  [0.08652569 0.11145381 0.25124511]]]
```

[211]:
```
#Create a random vector of size 30 and find the mean value ( )
vector = np.random.rand(30)  # vector with 30 random values between 0 and 1
mean = vector.mean()# calculate the mean
print("Vector:", vector)
print("Mean:", mean)
```

```
Vector: [0.96491529 0.63176605 0.8166602  0.566082   0.63535621 0.81190239
 0.92668262 0.91262676 0.82481072 0.09420273 0.36104842 0.03550903
 0.54635835 0.79614272 0.0511428  0.18866774 0.36547777 0.24429087
 0.79508747 0.35209494 0.63887768 0.49341505 0.58349974 0.93929935
 0.94354008 0.11169243 0.84355497 0.34602815 0.10082727 0.38340907]
Mean: 0.5434989623336113
```

[212]:
```
#How to add a border (filled with 0's) around an existing array
arrayloco = np.array([[1, 2], [3, 4]])
# Add a border of 0's around the array
arr_border = np.pad(arrayloco, pad_width=1, mode='constant', constant_values=0)
print(arr_border)
```

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

[217]:
```
#Create a 5x5 matrix with values 1,2,3,4 just below the diagonal
# Lower diagonal

#In NumPy, the main diagonal have offset k=0.
#If i want the diagonal one row down, i use k=-1
matrixPOWERRR = np.diag([1,2,3,4], k=-1)
print(matrixPOWERRR)
```

```
[[0 0 0 0 0]
 [1 0 0 0 0]
 [0 2 0 0 0]
```

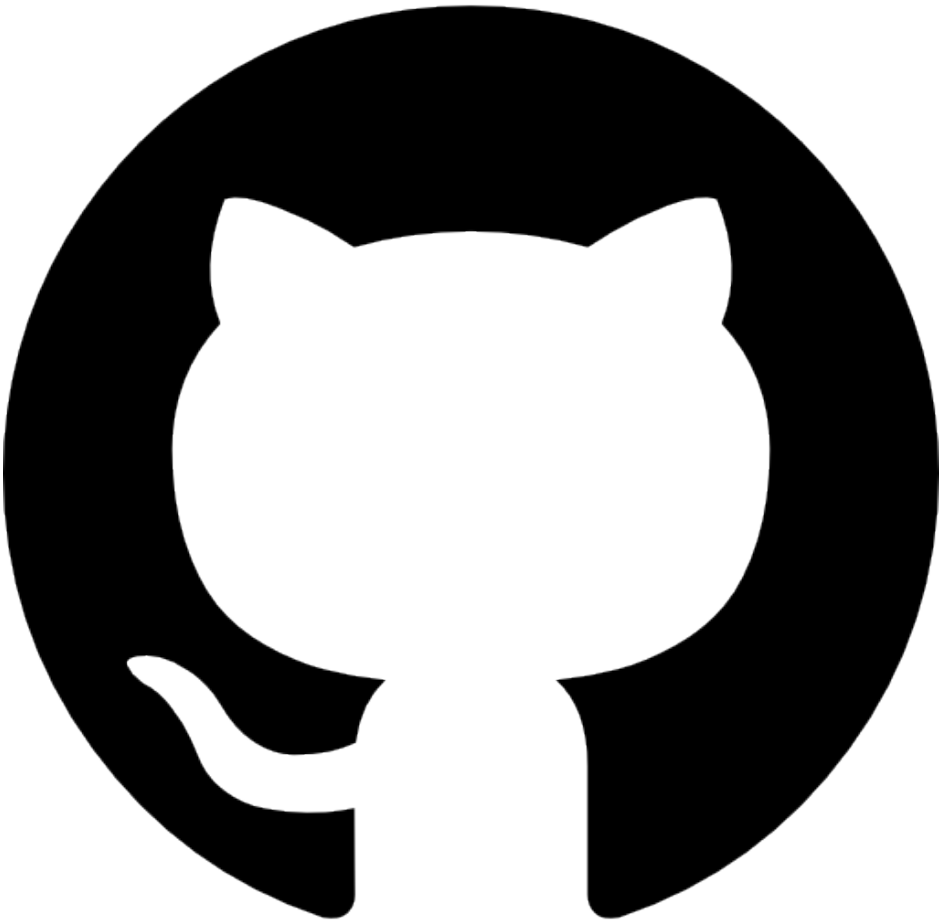
```
[0 0 3 0 0]
[0 0 0 4 0]]
```

[218]:
```
#Consider an array with shape (6,7,8), what is the index (x,y,z) of the 100th␣
↪element?

# Consider an array with shape (6,7,8)
shape = (6,7,8)

# We want to find the 3D index (x,y,z) of the 100th element
flat_index = 100

# Use np.unravel_index to convert a flat index into coordinates in a␣
↪multi-dimensional array
# The function returns a tuple with one element per dimension
index_3d = np.unravel_index(flat_index, shape)

print(index_3d)
```

```
(np.int64(1), np.int64(5), np.int64(4))
```

[220]:
```
#El resultaod que sale arriba devuelve en forma de tupla, en formato
#normal sería así (1,5,4)
```

# 1 Github



### 1.0.1 Click aquí para ver el repositorio