

```
In [249]: #Presentado por:Angie Paola Villada Ortiz.
#codigo:1089721336

# COMPUTACIÓN BLANDA - Sistemas y Computación
# -----
# Introducción a numpy
# -----
# Lección 02
#
# ** Técnicas de apilamiento
# ** División de arrays
# ** Propiedades de arrays
#
# -----
# Se importa la librería numpy

import numpy as np
# APILAMIENTO
# -----
# Apilado
# Las matrices se pueden apilar horizontalmente, en profundidad o
# verticalmente. Podemos utilizar, para ese propósito,
# las funciones vstack, dstack, hstack, column_stack, row_stack y concatenate.
# Para empezar, vamos a crear dos arrays
# Matriz a
a = np.arange(16).reshape(4,4)
print('a =\n', a, '\n')
# Matriz b, creada a partir de la matriz a
b = a*4
print('b =\n', b)
# Utilizaremos estas dos matrices para mostrar los mecanismos
# de apilamiento disponibles
```

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

```
b =
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

```
In [250]: # APILAMIENTO HORIZONTAL
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento horizontal
print('Apilamiento horizontal =\n', np.hstack((a,b)) )
```

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

```
b =
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

```
Apilamiento horizontal =
[[ 0  1  2  3  0  4  8 12]
 [ 4  5  6  7 16 20 24 28]
 [ 8  9 10 11 32 36 40 44]
 [12 13 14 15 48 52 56 60]]
```

```
In [251]: # APILAMIENTO HORIZONTAL - Variante
# Utilización de la función: concatenate()
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento horizontal
print( 'Apilamiento horizontal con concatenate = \n',
np.concatenate((a,b), axis=1) )
# Si axis=1, el apilamiento es horizontal
```

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

```
b =
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

```
Apilamiento horizontal con concatenate =
[[ 0  1  2  3  0  4  8 12]
 [ 4  5  6  7 16 20 24 28]
 [ 8  9 10 11 32 36 40 44]
 [12 13 14 15 48 52 56 60]]
```

```
In [252]: # APILAMIENTO VERTICAL
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento vertical
print( 'Apilamiento vertical =\n', np.vstack((a,b)) )
```

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

```
b =
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

```
Apilamiento vertical =
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

```
In [253]: # APILAMIENTO VERTICAL - Variante

# Utilización de la función: concatenate()
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento vertical
print( 'Apilamiento vertical con concatenate =\n',
np.concatenate((a,b), axis=0) )
# Si axis=0, el apilamiento es vertical
```

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

```
b =
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

```
Apilamiento vertical con concatenate =
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

```
In [254]: # APILAMIENTO EN PROFUNDIDAD

# En el apilamiento en profundidad, se crean bloques utilizando
# parejas de datos tomados de las dos matrices
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento en profundidad
print( 'Apilamiento en profundidad =\n', np.dstack((a,b)) )
```

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

```
b =
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

Apilamiento en profundidad =

```
[[[ 0  0]
 [ 1  4]
 [ 2  8]
 [ 3 12]]
```

```
[[ 4 16]
 [ 5 20]
 [ 6 24]
 [ 7 28]]
```

```
[[ 8 32]
 [ 9 36]
 [10 40]
 [11 44]]
```

```
[[12 48]
 [13 52]
 [14 56]
 [15 60]]]
```

```
In [255]: # APILAMIENTO POR COLUMNAS

# El apilamiento por columnas es similar a hstack()
# Se apilan las columnas, de izquierda a derecha, y tomándolas
# de los bloques definidos en la matriz
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento vertical
print( 'Apilamiento por columnas =\n',
np.column_stack((a,b)) )
```

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

```
b =
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

```
Apilamiento por columnas =
[[ 0  1  2  3  0  4  8 12]
 [ 4  5  6  7 16 20 24 28]
 [ 8  9 10 11 32 36 40 44]
 [12 13 14 15 48 52 56 60]]
```

```
In [256]: # APILAMIENTO POR FILAS

# El apilamiento por fila es similar a vstack()
# Se apilan las filas, de arriba hacia abajo, y tomándolas
# de los bloques definidos en la matriz
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento vertical
print('Apilamiento por filas =\n',
      np.row_stack((a,b)) )

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

b =
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]

Apilamiento por filas =
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

```
In [257]: # DIVISIÓN DE ARRAYS

# Las matrices se pueden dividir vertical, horizontalmente o en profundidad.
# Las funciones involucradas son hsplit, vsplit, dsplit y split.
# Podemos hacer divisiones de las matrices utilizando su estructura inicial
# o hacerlo indicando la posición después de la cual debe ocurrir la división
```

```
In [258]: # DIVISIÓN HORIZONTAL
print(a, '\n')
# El código resultante divide una matriz a lo largo de su eje horizontal
# en tres piezas del mismo tamaño y forma:}
print('Array con división horizontal =\n', np.hsplit(a, 4), '\n')
# El mismo efecto se consigue con split() y utilizando una bandera a 1
print('Array con división horizontal, uso de split() =\n',
np.split(a, 4, axis=1))
```

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

Array con división horizontal =

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

Array con división horizontal, uso de split() =

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



```
In [259]: # DIVISIÓN VERTICAL
print(a, '\n')
# La función vsplit divide el array a lo largo del eje vertical:
print('División Vertical = \n', np.vsplit(a, 4), '\n')
# El mismo efecto se consigue con split() y utilizando una bandera a 0
print('Array con división vertical, uso de split() =\n',
np.split(a, 4, axis=0))
```

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

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

```
Array con división vertical, uso de split() =
[array([[0, 1, 2, 3]]), array([[4, 5, 6, 7]]), array([[ 8,  9, 10, 11]]), ar
ray([[12, 13, 14, 15]])]
```

```
In [260]: # DIVISIÓN EN PROFUNDIDAD

# La función dsplit, como era de esperarse, realiza división
# en profundidad dentro del array
# Para ilustrar con un ejemplo, utilizaremos una matriz de rango tres
c = np.arange(64).reshape(4, 4, 4)
print(c, '\n')
# Se realiza la división
print('División en profundidad =\n', np.dsplit(c,4), '\n')
```

```
[[[ 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]]]
```

División en profundidad =

```
[array([[ 0],
        [ 4],
        [ 8],
        [12]],

        [[16],
         [20],
         [24],
         [28]],

        [[32],
         [36],
         [40],
         [44]],

        [[48],
         [52],
         [56],
         [60]]]), array([[ 1],
        [ 5],
        [ 9],
        [13]],

        [[17],
         [21],
         [25],
         [29]],

        [[33],
         [37],
         [41],
         [45]],

        [[49],
         [53],
         [57],
```

```

[61]]]), array([[[ 2],
[ 6],
[10],
[14]],

[[18],
[22],
[26],
[30]],

[[34],
[38],
[42],
[46]],

[[50],
[54],
[58],
[62]]]), array([[[ 3],
[ 7],
[11],
[15]],

[[19],
[23],
[27],
[31]],

[[35],
[39],
[43],
[47]],

[[51],
[55],
[59],
[63]]]])]

```

```

In [261]: # PROPIEDADES DE LOS ARRAYS
# El atributo ndim calcula el número de dimensiones

print(b, '\n')
print('ndim: ', b.ndim)

```

```

[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]

```

```
ndim: 2
```

In [262]: *# El atributo size calcula el número de elementos*

```
print(b, '\n')
print('size: ', b.size)
```

```
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

size: 16

In [263]: *# El atributo itemsize obtiene el número de bytes por cada*

```
# elemento en el array
print('itemsize: ', b.itemsize)
```

itemsize: 4

In [264]: *# El atributo nbytes calcula el número total de bytes del array*

```
print(b, '\n')
print('nbytes: ', b.nbytes, '\n')
# Es equivalente a la siguiente operación
print('nbytes equivalente: ', b.size * b.itemsize)
```

```
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

nbytes: 64

nbytes equivalente: 64

In [265]: *# El atributo T tiene el mismo efecto que la transpuesta de la matriz*

```
b.resize(4,4)
print(b, '\n')
print('Transpuesta: ', b.T)
```

```
[[ 0  4  8 12]
 [16 20 24 28]
 [32 36 40 44]
 [48 52 56 60]]
```

```
Transpuesta: [[ 0 16 32 48]
 [ 4 20 36 52]
 [ 8 24 40 56]
 [12 28 44 60]]
```

In [266]: *# Los números complejos en numpy se representan con j*

```
b = np.array([1.j + 1, 2.j + 3])  
print('Complejo: \n', b)
```

Complejo:
[1.+1.j 3.+2.j]

In [267]: *# El atributo real nos da la parte real del array,
o el array en sí mismo si solo contiene números reales*
El atributo imag contiene la parte imaginaria del array
print('real: ', b.real, '\n')
print('imaginario: ', b.imag)

real: [1. 3.]

imaginario: [1. 2.]

In [268]: *# Si el array contiene números complejos, entonces el tipo de datos*

```
# se convierte automáticamente a complejo  
print(b.dtype)
```

complex128

```
In [279]: # El atributo flat devuelve un objeto numpy.flatiter.  
# Esta es la única forma de adquirir un flatiter:  
# no tenemos acceso a un constructor de flatiter.  
# El apartamento El iterador nos permite recorrer una matriz  
# como si fuera una matriz plana, como se muestra a continuación:  
# En el siguiente ejemplo se clarifica este concepto  
b = np.arange(16).reshape(4,4)  
print(b, '\n')  
f = b.flat  
print(f, '\n')  
# Ciclo que itera a lo largo de f  
for item in f: print (item)  
# Selección de un elemento  
print('\n')  
print('Elemento 2: ', b.flat[2])  
# Operaciones directas con flat  
b.flat = 9  
print(b, '\n')  
b.flat[[1,3]] = 1  
print(b, '\n')
```

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

```
<numpy.flatiter object at 0x000001BEAFDF15F0>
```

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

```
Elemento 2:  2
```

```
[[9 9 9 9]
 [9 9 9 9]
 [9 9 9 9]
 [9 9 9 9]]
```

```
[[9 1 9 1]
 [9 9 9 9]
 [9 9 9 9]
 [9 9 9 9]]
```