

```
In [72]: #Julian Giraldo Cardona
        #Computación Blanda
```

```
In [73]: # 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(12).reshape(3,4)
print('a =\n', a, '\n')
# Matriz b, creada a partir de la matriz a
b = a*3
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]]
```

```
b =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

```
In [74]: # APILAMIENTO HORIZONTAL
        # Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento horizontal
print('Apilamiento horizontal =\n', np.hstack((b,a)) )
```

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

```
b =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

```
Apilamiento horizontal =
[[ 0  3  6  9  0  1  2  3]
 [12 15 18 21  4  5  6  7]
 [24 27 30 33  8  9 10 11]]
```

```
In [75]: # 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((b,a), axis=1) )
# Si axis=1, el apilamiento es horizontal
```

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

```
b =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

```
Apilamiento horizontal con concatenate =
[[ 0  3  6  9  0  1  2  3]
 [12 15 18 21  4  5  6  7]
 [24 27 30 33  8  9 10 11]]
```

```
In [76]: # APILAMIENTO VERTICAL
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento vertical
print( 'Apilamiento vertical =\n', np.vstack((b,a)) )
```

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

```
b =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

```
Apilamiento vertical =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]
 [ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
In [77]: # 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((b,a), axis=0) )
# Si axis=0, el apilamiento es vertical
```

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

```
b =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

```
Apilamiento vertical con concatenate =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]
 [ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
In [78]: # 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((b,a)) )
```

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

```
b =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

```
Apilamiento en profundidad =
[[[ 0  0]
 [ 3  1]
 [ 6  2]
 [ 9  3]]

 [[12  4]
 [15  5]
 [18  6]
 [21  7]]

 [[24  8]
 [27  9]
 [30 10]
 [33 11]]]
```

```
In [79]: # 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((b,a)) )
```

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

b =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]

Apilamiento por columnas =
[[ 0  3  6  9  0  1  2  3]
 [12 15 18 21  4  5  6  7]
 [24 27 30 33  8  9 10 11]]
```

```
In [80]: # 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((b,a)) )
```

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

b =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]

Apilamiento por filas =
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]
 [ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
In [81]: # 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 [82]: a = np.arange(9).reshape(3,3)
```

```
In [83]: # 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, 3), '\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, 3, axis=1))
```

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

```
Array con división horizontal =
[array([[0],
       [3],
       [6]]), array([[1],
       [4],
       [7]]), array([[2],
       [5],
       [8]])]
```

```
Array con división horizontal, uso de split() =
[array([[0],
       [3],
       [6]]), array([[1],
       [4],
       [7]]), array([[2],
       [5],
       [8]])]
```

```
In [84]: # DIVISIÓN VERTICAL
print(a, '\n')
# La función vsplit divide el array a lo largo del eje vertical:
print('División Vertical = \n', np.vsplit(a, 3), '\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, 3, axis=0))
```

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

```
División Vertical =
[array([[0, 1, 2]]), array([[3, 4, 5]]), array([[6, 7, 8]])]
```

```
Array con división vertical, uso de split() =
[array([[0, 1, 2]]), array([[3, 4, 5]]), array([[6, 7, 8]])]
```

```
In [85]: # 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(27).reshape(3, 3, 3)
print(c, '\n')
# Se realiza la división
print('División en profundidad =\n', np.dsplit(c,3), '\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]]]
```

```
División en profundidad =
[array([[[ 0],
          [ 3],
          [ 6]],

        [[ 9],
          [12],
          [15]],

        [[18],
          [21],
          [24]]]), array([[[ 1],
          [ 4],
          [ 7]],

        [[10],
          [13],
          [16]],

        [[19],
          [22],
          [25]]]), array([[[ 2],
          [ 5],
          [ 8]],

        [[11],
          [14],
          [17]],

        [[20],
          [23],
          [26]]]])]
```



```
In [86]: # PROPIEDADES DE LOS ARRAYS
```

```
In [87]: # El atributo ndim calcula el número de dimensiones
```

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

```
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

```
ndim:  2
```

```
In [88]: # El atributo size calcula el número de elementos
```

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

```
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

```
size:  12
```

```
In [89]: # El atributo itemsize obtiene el número de bytes por cada
```

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

```
itemsize:  4
```

```
In [90]: # 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  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

```
nbytes:  48
```

```
nbytes equivalente:  48
```

In [92]: *# El atributo T tiene el mismo efecto que la transpuesta de la matriz*

```
b.resize(2,6)
print(b, '\n')
print('Transpuesta: ', b.T)
```

```
[[ 0  3  6  9 12 15]
 [18 21 24 27 30 33]]
```

```
Transpuesta: [[ 0 18]
 [ 3 21]
 [ 6 24]
 [ 9 27]
 [12 30]
 [15 33]]
```

In [94]: *# Los números complejos en numpy se representan con j*

```
b = np.array([4.j + 2, 3.j + 6])
print('Complejo: \n', b)
```

```
Complejo:
[2.+4.j 6.+3.j]
```

In [95]: *# 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: [2. 6.]
```

```
imaginario: [4. 3.]
```

In [96]: *# Si el array contiene números complejos, entonces el tipo de datos*

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

```
complex128
```

```
In [97]: # 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(4).reshape(2,2)  
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 = 7  
print(b, '\n')  
b.flat[[1,3]] = 1  
print(b, '\n')
```

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

```
<numpy.flatiter object at 0x08A43FE8>
```

```
0  
1  
2  
3
```

```
Elemento 2:  2  
[[7 7]  
 [7 7]]
```

```
[[7 1]  
 [7 1]]
```