In [11]:
```
import numpy as np
#Apilado
#Las matrices se pueden apilar horizontalmente, en profundidad o
#verticalmente. Podemos utilizar para este proposito, las funciones
#vstack,dstack,hstack

#creamos dos Arrays
a = np.arange(9).reshape(3,3)
print ('a=\n',a,'\n')
#matriz b en base a A
b=a*2
```

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

In [6]:
```
print ('b=\n',b,'\n')
```

```
b=
 [[ 0  2  4]
 [ 6  8 10]
 [12 14 16]]
```

In [7]:
```
print ('a=\n',a,'\n')
print ('b=\n',b,'\n')
print ('Apilamiento horizontal = \n',np.hstack((a,b)))
#Apilamiento horizontal
```

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

b=
 [[ 0  2  4]
 [ 6  8 10]
 [12 14 16]]

Apilamiento horizontal =
 [[ 0  1  2  0  2  4]
 [ 3  4  5  6  8 10]
 [ 6  7  8 12 14 16]]
```

In [4]:
```
print ('a =\n',a,'\n')
print ('b =\n',b,'\n')
print ('Apilamiento vertical =\n',np.vstack((a,b)))
#Apilamiento vertical
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-f8230b020bf7> in <module>
----> 1 print ('a =\n',a,'\n')
      2 print ('b =\n',b,'\n')
      3 print ('Apilamiento vertical =\n',np.vstack((a,b)))
      4 #Apilamiento vertical

NameError: name 'a' is not defined
```

In [9]:
```
#Apilamiento vertical- variante
#función concatenate()
#Matrices origen
print ('a =\n',a,'\n')
print ('b =\n',b,'\n')
print ('Apilamiento vertical con concatenate =\n',
       np.concatenate((a,b),axis=0))
#Si axis=0 apilamiento vertical
```

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

b =
 [[ 0  2  4]
 [ 6  8 10]
 [12 14 16]]

Apilamiento vertical con concatenate =
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 0  2  4]
 [ 6  8 10]
 [12 14 16]]
```

In [12]:
```
#Apilamiento en profundidad
#En el apilamiento en profundidad se crean bloques utilizando
#parejas de datos tomados de las dos matrices
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]]

b =
 [[ 0  2  4]
 [ 6  8 10]
 [12 14 16]]

Apilamiento en profundidad =
 [[[ 0  0]
  [ 1  2]
  [ 2  4]]

 [[ 3  6]
  [ 4  8]
  [ 5 10]]

 [[ 6 12]
  [ 7 14]
  [ 8 16]]]
```

In [13]:
```
#Apilamiento por columnas
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]]

b =
 [[ 0  2  4]
 [ 6  8 10]
 [12 14 16]]

Apilamiento por columnas =
 [[ 0  1  2  0  2  4]
 [ 3  4  5  6  8 10]
 [ 6  7  8 12 14 16]]
```

In [14]:
```
#Apilamiento por filas
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]]

b =
 [[ 0  2  4]
 [ 6  8 10]
 [12 14 16]]

Apilamiento por filas =
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 0  2  4]
 [ 6  8 10]
 [12 14 16]]
```

In [15]:
```
# 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 [16]:
```
#Division Horizontal
print (a,'\n')
#El codigo resultante divide una matriz a lo largo de su eje
#horizontal
#en tres piezas del mismo tamaño y forma
print('Array con division horizontal =\n',np.hsplit(a,3),'\n')
#El mismo efecto se consigue con split y utilizando bandera  a 1
print(' Array con division horizontal con split()=\n', np.split(a,3,axis=1))
```

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

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

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

In [17]:
```
#Division Vertical
print(a,'\n')
#la funcion vsplit divide el array a lo largo del eje vertical:
print('Division vertical =\n',np.vsplit(a,3),'\n')
#mismo efecto split con bandera 0
print('Array con division vertica, uso de split()=\n'
      ,np.split(a,3,axis=0))
```

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

Division vertical =
 [array([[0, 1, 2]]), array([[3, 4, 5]]), array([[6, 7, 8]])]

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

In [18]:
```
#Division en profundidad
#displit realiza division en profundidad dentro del array
#ejemplo
c=np.arange(27).reshape(3,3,3)
print(c,'\n')
#Se realiza division
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 [19]:
```
#El atributo ndim calcula el # de dimensiones
print(b,'\n')
print('ndim: ',b.ndim)
```

```
[[ 0  2  4]
 [ 6  8 10]
 [12 14 16]]

ndim:  2
```

In [20]:
```
#Size # de elementos
print(b,'\n')
print('size: ',b.size)
```

```
[[ 0  2  4]
 [ 6  8 10]
 [12 14 16]]

size:  9
```

In [21]:
```
#itemsize obtiene el numero de bytes
print('itemsize',b.itemsize)
```

```
itemsize 4
```

In [22]:
```
#nbytes calcula el número total de bytes dela rray
print(b,'\n')
print('nbytes: ',b.nbytes,'\n')
#Es ifual a
print('nbytes equivalente: ',b.size*b.itemsize)
```

```
[[ 0  2  4]
 [ 6  8 10]
 [12 14 16]]

nbytes:  36

nbytes equivalente:  36
```

In [23]:
```
#El atributo T tiene el mismo efecto que la transpuesta
b.resize(6,4)
print(b,'\n')
print('Transpuesta: ',b.T)
```

```
[[ 0  2  4  6]
 [ 8 10 12 14]
 [16  0  0  0]
 [ 0  0  0  0]
 [ 0  0  0  0]
 [ 0  0  0  0]]

Transpuesta:  [[ 0  8 16  0  0  0]
 [ 2 10  0  0  0  0]
 [ 4 12  0  0  0  0]
 [ 6 14  0  0  0  0]]
```

In [24]:
```
#Los numeros complejos en numpy se representa con j

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

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

In [25]:
```
#El atributoo real nos da la parte real del array
#o el array en si mismo solo contiene # reales
print('real: ',b.real,'\n')
#atributo imag contiene la parte imaginaria del array
print('imaginario: ',b.imag)
```

```
real:  [1. 3.]

imaginario:  [1. 2.]
```

In [26]:
```
#Si el array contiene numeros complejos, entonces el tipo de datos
#Se convierete  acomplejos
print(b.dtype)
```

```
complex128
```

In [27]:
```
#Flat devuelve un objeto numpy.flattier
#Esta es la unica forma de adquirir un flattier
#no tenemos acceso a un constructor de flattier.
#El apartamento el iterador nos permite recorre una matriz
#como si fuera una matriz plana, como se muestr a acontinuación

b=np.arange(4).reshape(2,2)
print(b,'\n')

f=b.flat
print(f,'\n')

#ciclo que itera a lo largo d e
for item in f:print(item)
#Selección de un elemento
print('\n')
print ('Elemento 2: ',b.flat[2])
#operaciones con flat
b.flat = 7
print(b,'\n')
b.flat[[1,3]]=1
print(b,'\n')
```

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

<numpy.flatiter object at 0x00000226D72D64E0>

0
1
2
3


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

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

In [ ]:
```
```