In [99]:

```
# COMPUTACIÓN BLANDA - Sistemas y Computación
# ------------------------------------------
# Introducción a numpy
# Luis Anibal Loaiza C.
# ------------------------------------------
# Lección 02
#
# ** Técnicas de apilamiento
# ** División de arrays
# ** Propiedades de arrays
#
# ------------------------------------------
```

In [100]:

```
# Se importa la librería numpy

import numpy as np
import random
# 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(4,3)
print('a =\n', a, '\n')
# Matriz b, creada a partir de la matriz a
b = a*24
print('b =\n', b)
# Utilizaremos estas dos matrices para mostrar los mecanismos
# de apilamiento disponibles
c= b//2

print('c =\n', c)
```

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

b =
 [[  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]
c =
 [[  0  12  24]
 [ 36  48  60]
 [ 72  84  96]
 [108 120 132]]
```

In [102]:

```
# APILAMIENTO HORIZONTAL
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
print('c =\n', c, '\n')
# Apilamiento horizontal
print('Apilamiento horizontal =\n', np.hstack((a,b,c)) )
e=np.hstack((a,b,c))
print( 'Recorrer nueva matriz \n')
con=0
for i in e:
    print( 'Posicion {} en matriz \n'.format(con))
    con=con+1
    print(i)
```

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

b =
 [[  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]

c =
 [[  0  12  24]
 [ 36  48  60]
 [ 72  84  96]
 [108 120 132]]

Apilamiento horizontal =
 [[  0   1   2   0  24  48   0  12  24]
 [  3   4   5  72  96 120  36  48  60]
 [  6   7   8 144 168 192  72  84  96]
 [  9  10  11 216 240 264 108 120 132]]
Recorrer nueva matriz

Posicion 0 en matriz

[ 0  1  2  0 24 48  0 12 24]
Posicion 1 en matriz

[  3   4   5  72  96 120  36  48  60]
Posicion 2 en matriz

[  6   7   8 144 168 192  72  84  96]
Posicion 3 en matriz

[  9  10  11 216 240 264 108 120 132]
```

In [103]:

```
# 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,c), axis=1) )

d=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]]

b =
 [[  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]

Apilamiento horizontal con concatenate =
 [[  0   1   2   0  24  48   0  12  24]
 [  3   4   5  72  96 120  36  48  60]
 [  6   7   8 144 168 192  72  84  96]
 [  9  10  11 216 240 264 108 120 132]]
```

In [123]:

```
# APILAMIENTO VERTICAL
# Matrices origen
print('a =\n', a, '\n')

# Apilamiento vertical
print( 'Apilamiento vertical =\n', np.vstack((a,a)) )
```

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

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

In [105]:

```
# 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]]

b =
 [[  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]

Apilamiento vertical con concatenate =
 [[  0   1   2]
 [  3   4   5]
 [  6   7   8]
 [  9  10  11]
 [  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]
```

In [106]:

```
# 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')
print('b =\n', c, '\n')
# Apilamiento en profundidad
print( 'Apilamiento en profundidad =\n', np.dstack((a,b,c)) )
e = np.dstack((a,b,c))
print( 'Recorrer nueva matriz \n')
con=0
for i in e:
    print( 'Posicion {} en matriz \n'.format(con))
    con=con+1
    print(i)
```

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

b =
 [[  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]

b =
 [[  0  12  24]
 [ 36  48  60]
 [ 72  84  96]
 [108 120 132]]

Apilamiento en profundidad =
 [[[  0   0   0]
  [  1  24  12]
  [  2  48  24]]

 [[  3  72  36]
  [  4  96  48]
  [  5 120  60]]

 [[  6 144  72]
  [  7 168  84]
  [  8 192  96]]

 [[  9 216 108]
  [ 10 240 120]
  [ 11 264 132]]]
Recorrer nueva matriz

Posicion 0 en matriz

[[ 0  0  0]
 [ 1 24 12]
 [ 2 48 24]]
Posicion 1 en matriz

[[  3  72  36]
 [  4  96  48]
 [  5 120  60]]
Posicion 2 en matriz

[[  6 144  72]
 [  7 168  84]
 [  8 192  96]]
Posicion 3 en matriz

[[  9 216 108]
 [ 10 240 120]
 [ 11 264 132]]
```

In [107]:

```
# 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')
print('c =\n', c, '\n')
# Apilamiento vertical
e=np.column_stack((a,b,c))
print( 'Apilamiento por columnas =\n',e )
h=[]
for i in e:
    print( 'Posicion {} en matriz \n'.format(con))
    con=con+1
    print(i)
    h.append(i+i)

print('\n',h)
```

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

b =
 [[  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]

c =
 [[  0  12  24]
 [ 36  48  60]
 [ 72  84  96]
 [108 120 132]]

Apilamiento por columnas =
 [[  0   1   2   0  24  48   0  12  24]
 [  3   4   5  72  96 120  36  48  60]
 [  6   7   8 144 168 192  72  84  96]
 [  9  10  11 216 240 264 108 120 132]]
Posicion 4 en matriz

[ 0  1  2  0 24 48  0 12 24]
Posicion 5 en matriz

[  3   4   5  72  96 120  36  48  60]
Posicion 6 en matriz

[  6   7   8 144 168 192  72  84  96]
Posicion 7 en matriz

[  9  10  11 216 240 264 108 120 132]

 [array([ 0,  2,  4,  0, 48, 96,  0, 24, 48]), array([  6,   8,  10, 144,
192, 240,  72,  96, 120]), array([ 12,  14,  16, 288, 336, 384, 144, 168,
192]), array([ 18,  20,  22, 432, 480, 528, 216, 240, 264])]
```

In [108]:

```
# 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
u=np.row_stack((a,b))
print( 'Apilamiento por filas =\n',u )

#Vista de la matriz por elementos
for i in range(len(u)):
      for j in range(len(u[i])):
        print('b[{},{}] ='.format(i,j) , u[i][j],'\n')
```

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

b =
 [[  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]

Apilamiento por filas =
 [[  0   1   2]
 [  3   4   5]
 [  6   7   8]
 [  9  10  11]
 [  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]
b[0,0] = 0

b[0,1] = 1

b[0,2] = 2

b[1,0] = 3

b[1,1] = 4

b[1,2] = 5

b[2,0] = 6

b[2,1] = 7

b[2,2] = 8

b[3,0] = 9

b[3,1] = 10

b[3,2] = 11

b[4,0] = 0

b[4,1] = 24

b[4,2] = 48

b[5,0] = 72

b[5,1] = 96

b[5,2] = 120

b[6,0] = 144

b[6,1] = 168
```

```
b[6,2] = 192

b[7,0] = 216

b[7,1] = 240

b[7,2] = 264
```

In [109]:

```
# 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 [125]:

```
# 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))


print( 'Recorrer nueva matriz \n')
for i in range(len(a)):
      for j in range(len(a[i])):
        print('a[{},{}] ='.format(i,j) , a[i][j],'\n')
```

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

Array con división horizontal =
 [array([[0],
       [3],
       [6],
       [9]]), array([[ 1],
       [ 4],
       [ 7],
       [10]]), array([[ 2],
       [ 5],
       [ 8],
       [11]])]

Array con división horizontal, uso de split() =
 [array([[0],
       [3],
       [6],
       [9]]), array([[ 1],
       [ 4],
       [ 7],
       [10]]), array([[ 2],
       [ 5],
       [ 8],
       [11]])]
Recorrer nueva matriz

a[0,0] = 0

a[0,1] = 1

a[0,2] = 2

a[1,0] = 3

a[1,1] = 4

a[1,2] = 5

a[2,0] = 6

a[2,1] = 7

a[2,2] = 8

a[3,0] = 9

a[3,1] = 10

a[3,2] = 11
```

In [111]:

```
# 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]]

División Vertical =
 [array([[0, 1, 2]]), array([[3, 4, 5]]), array([[6, 7, 8]]), array([[ 9,
10, 11]])]

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

In [112]:

```
# DIVISIÓN EN PROFUNDIDAD
import random
# 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

d = []
for s in range(0,48):
    s=random.randrange(12,68)
    d.append(s)
print(d, '\n')
d=np.reshape(d,(4,4,3))
print(d)
# Se realiza la división
print('División en profundidad =\n', np.dsplit(d,3), '\n')
```

```
[25, 28, 41, 24, 53, 43, 42, 42, 67, 50, 33, 16, 19, 54, 32, 28, 56, 18, 3
9, 43, 38, 23, 12, 50, 59, 23, 30, 52, 42, 63, 42, 39, 56, 32, 22, 44, 16,
44, 14, 66, 27, 40, 20, 59, 14, 18, 12, 28]

[[[25 28 41]
  [24 53 43]
  [42 42 67]
  [50 33 16]]

 [[19 54 32]
  [28 56 18]
  [39 43 38]
  [23 12 50]]

 [[59 23 30]
  [52 42 63]
  [42 39 56]
  [32 22 44]]

 [[16 44 14]
  [66 27 40]
  [20 59 14]
  [18 12 28]]]
División en profundidad =
 [array([[[25],
         [24],
         [42],
         [50]],

        [[19],
         [28],
         [39],
         [23]],

        [[59],
         [52],
         [42],
         [32]],

        [[16],
         [66],
         [20],
         [18]]]), array([[[28],
         [53],
         [42],
         [33]],

        [[54],
         [56],
         [43],
         [12]],

        [[23],
         [42],
         [39],
         [22]],

        [[44],
         [27],
         [59],
         [12]]]), array([[[41],
```

```
         [43],
         [67],
         [16]],

        [[32],
         [18],
         [38],
         [50]],

        [[30],
         [63],
         [56],
         [44]],

        [[14],
         [40],
         [14],
         [28]]])]
```

In [113]:

```
# PROPIEDADES DE LOS ARRAYS
# El atributo ndim calcula el número de dimensiones

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

```
[[  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]

ndim:  2
```

In [114]:

```
# El atributo size calcula el número de elementos

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

```
[[  0  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]

size:  12
```

In [115]:

```
# El atributo itemsize obtiene el número de bytes por cada

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

```
itemsize:  4
```

In [116]:

```
# 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  24  48]
 [ 72  96 120]
 [144 168 192]
 [216 240 264]]

nbytes:  48

nbytes equivalente:  48
```

In [126]:

```
#Multiplicacion de escalar
a=3*a
print(a)
```

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

In [136]:

```
#Determinante de una matriz
A = ([
    [3, 2, 5],
    [1, 6, 7],
    [1, 6, 4]
])
z=np.linalg.det(A)
print(z)
```

```
-48.00000000000001
```

In [137]:

```
#Matriz inversa
k=np.linalg.inv(A)
print(k)
```

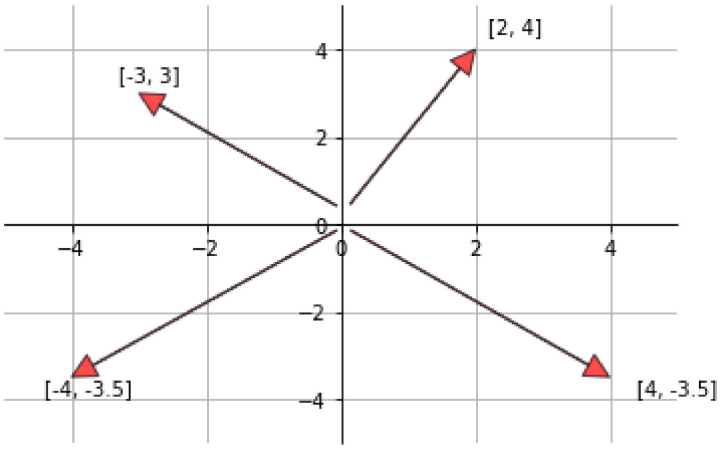
```
[[ 0.375      -0.45833333  0.33333333]
 [-0.0625     -0.14583333  0.33333333]
 [-0.          0.33333333 -0.33333333]]
```

In [142]:

```
import matplotlib.pyplot as plt
from warnings import filterwarnings

%matplotlib inline
filterwarnings('ignore') # Ignorar warnings

def move_spines():
    """Crea la figura de pyplot y los ejes. Mueve las lineas de la izquierda y de abajo
    para que se intersecten con el origen. Elimina las lineas de la derecha y la de arr
iba.
    Devuelve los ejes."""
    fix, ax = plt.subplots()
    for spine in ["left", "bottom"]:
        ax.spines[spine].set_position("zero")

    for spine in ["right", "top"]:
        ax.spines[spine].set_color("none")

    return ax

def vect_fig():
    """Genera el grafico de los vectores en el plano"""
    ax = move_spines()

    ax.set_xlim(-5, 5)
    ax.set_ylim(-5, 5)
    ax.grid()
    vecs = [[2, 4], [-3, 3], [-4, -3.5],[4, -3.5]] # lista de vectores
    for v in vecs:
        ax.annotate(" ", xy=v, xytext=[0, 0],
                    arrowprops=dict(facecolor="red",
                                    shrink=0,
                                    alpha=0.7,
                                    width=0.5))
        ax.text(1.1 * v[0], 1.1 * v[1], v)
```

In [144]:

```
vect_fig() # crea el gráfico
#Recopilado de https://relopezbriega.github.io/blog/2015/06/14/algebra-lineal-con-pytho
n/
```



In [117]:

```
# El atributo T tiene el mismo efecto que la transpuesta de la matriz

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

```
[[  0  24]
 [ 48  72]
 [ 96 120]
 [144 168]
 [192 216]
 [240 264]]

Transpuesta:  [[  0  48  96 144 192 240]
 [ 24  72 120 168 216 264]]
```

In [118]:

```
# Los números complejos en numpy se representan con j

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

```
Complejo:
 [1.+1.j 3.+2.j]
Tipo de dato:
 complex128
```

In [119]:

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

```
real:  [1. 3.]

imaginario:  [1. 2.]
```

In [120]:

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

```
real:  float64

imaginario:  float64
```

In [121]:

```
# 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 0x000002CE9703D490>

0
1
2
3


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

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

In [ ]:

```

```

In [ ]:

```

```