Introducción a Numpy

El módulo Numpy (de "Numerical Python") ofrece la posibilidad de trabajar con bloques de datos de forma eficiente tanto en espacio como en tiempo. El objeto principal es el "ndarray" que representa un array de N dimensiones de valores homogéneos (también se da soporte a arrays de valores heterogéneos, pero esto está fuera del ámbito de aplicación para este curso).

La aplicación principal de Numpy es el cómputo numérico y dada su eficiencia es un módulo usado como base para muchos otros módulos con orientación a las ciencias en general y a la Ciencias de Datos en particular.

En el momento de escribir este tutorial, la versión "estable" de Numpy es la versión 1.22. La documentación completa sobre Numpy puede consultarse en la dirección numpy.org/doc/stable.

Inicialización del entorno.

Dependiendo de la plataforma que utilices tendrás distintas formas de instalar Numpy en tu computadora. Consulta la información disponible en el curso relativa a la instalación de paquetes Python.

Vamos a comenzar cargando el módulo y visualizando su versión ya que este dato es importante para consultar la ayuda en línea.

En el momento de escribir este tutorial la versión de Numpy con la que se trabaja es 1.22.4.

Ejercicio 01-01: Comprueba la versión de Numpy que tienes instalada. Lo recomendable sería que coincidiera con la versión indicada arriba.

In [1]:
```
import numpy as np
#Pon tu código aquí.
#Sugerencia: todos los módulos python tiene un atributo "__version__"
#con la versión del módulo.

#
```

¿Por qué usar Numpy?

Como ejemplo introductorio vamos a calcular el producto escalar de dos vectores, primero usando una versión "pura" python usando tipo 'list' para representar los vectores:

Ejercicio 01-02: Prueba a ejecutar en tu máquina para ver cuánto tiempo tarda en realizarse la operación. Para calcular el tiempo que tarda vamos a utilizar la función "mágica" python `%timeit` (ver más información aquí).

El resultado mostrado deberá ser algo parecido a lo siguiente (el tiempo dependerá del ordenará donde se ejecute este script):

```
Producto escalar 332833500
57.4 µs ± 161 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [2]:
```python
def prod_escalar_python(a, b):
    """Calcula el producto escalar de a y b."""
    pe = 0
    for i in range(len(a)):
        pe = pe + a[i]*b[i]
    return pe

a = list(range(1000))
b = list(range(1000))
print('Producto escalar {}'.format(prod_escalar_python(a, b)))
%timeit prod_escalar_python(a,b)
```

```
Producto escalar 332833500
56.9 µs ± 467 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Ahora vamos a realizar el mismo ejemplo pero usando Numpy. No te preocupes si no entiendes bien cada paso, lo importante aquí es que compares la simplicidad de la codificación y eficiencia en la ejecución.

Ejercicio 01-03: Prueba a ejecutar en tu máquina la siguiente celda para ver cuánto tiempo tarda en realizarse la operación.

El resultado mostrado deberá ser algo parecido a lo siguiente:

```
Producto escalar 332833500
2.23 µs ± 14.2 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [3]:
```python
def prod_escalar_numpy(a, b):
    """Calcula el producto escalar de a y b usando numpy."""
    return np.add.reduce(a*b)

a = np.arange(1000)
b = np.arange(1000)
print('Producto escalar {}'.format(prod_escalar_numpy(a, b)))
%timeit prod_escalar_numpy(a,b)
```

```
Producto escalar 332833500
2.29 µs ± 14.6 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Con la versión de python/numpy en la máquina usada para realizar este notebook el speedup obtenido por Numpy es aproximadamente más de 20 veces más rápido, concretamente en la ejecución ejemplo, sería `57,4/2,23 = 27,74` veces más rápida. ¿Cuál ha sido la tuya?

Definición del tipo ndarray.

Como ya hemos dicho el concepto "estrella" del módulo Numpy es la clase "ndarray" que utiliza un bloque compacto de datos en memoria, homogéneo, para presentar un array de N dimensiones.

Un array con una dimensión es lo que denominamos un "vector", si tiene dos dimensiones lo denominamos "matriz". Si tiene 3 o más dimensiones se utiliza el nombre genérico de "tensor". Por ejemplo un tensor de tres dimensiones puede representar una imagen digital en color con Ancho x Alto píxeles (dos de las dimensiones) cada uno codificando un tripleta RGB (la tercera dimensión).

Numpy define la clase "ndarray" para implementarlo. Aquí puedes consultar su definición.

Como puedes ver en la documentación, un objeto ndarray tiene varios atributos que lo definen. Dos de los principales son:

- Atributo `shape` (forma) especificado con un tupla con los tamaños cada dimensión.
- Atributo `dtype` (el tipo de los elementos), un objeto del tipo `numpy.dtype` que indica el tipo de los elementos del array.

Forma de un ndarray.

Un array con una forma shape por ejemplo `(3,)` representa lo que podríamos considerar un vector de 3 elementos. Por otra parte un array con forma `(3, 2)` representa una matriz de 3 filas y 2 columnas. Un array con forma `(480, 640, 3)` representa un array de 3 dimensiones (también denominado 'tensor' de 3 dimensiones) con 480 filas, 640 columnas y cada elemento es a su vez un vector 3 valores. Por ejemplo este tensor podría codificar una imagen en color RGB de tamaño 640x480 píxeles.

Tipo de un ndarray.

Por otro lado Numpy define distintos tipos de datos para representar el tipo de los elementos almacenados en el array. Algunos de ellos son (ver más dtype):

Algunos tipos de elementos

dtype	Carácter	Tipo codificado
np.bool	'?'	boolean.
np.byte	'b'	Entero con signo de 8 bits.
np.ubyte	'B'	Entero sin signo de 8 bits.
np.int32	'i'	Entero con signo de 32 bits.
np.int64	'l'	Entero con signo de 64 bits.
np.uint32	'I'	Entero sin signo de 32 bits.
np.uint64	'L'	Entero sin signo de 64 bits.
np.float	'd'	Flotante de 64 bits.
np.float32	'f'	Flotante de 32 bits.
np.complex	'D'	Complejo de 128 bits.
np.complex64	'F'	Complejo de 64 bits.

Creando un ndarray desde cero.

Sabiendo cómo especificar la forma y el tipo del ndarray, podemos utilizar distintas alternativas para crearlo. Veamos algunos ejemplos.

Crear un ndarray con los elementos sin inicializar.

La función empty() crea un ndarray de forma muy rápida (sólo hay que reservar la memoria necesaria) pero sin inicializar los elementos. Sus valores serán espúreos según el contenido de la memoria cuando se reserva el espacio.

In [4]:
```
def mostrar_informacion_ndarray(a):
    """Muestra información sobre un ndarray."""
```

```
    if (type(a)==np.ndarray):
        print('Forma : {}'.format(a.shape))
        print('Tipo  : {}'.format(a.dtype))
        print('N dims: {}'.format(a.ndim))
        print('Size  : {}'.format(a.size))
        print('Valores:')
        print(a)
    else:
        print('El agumento no es un ndarray!!.')

a = np.empty((3, 2), 'f') #creamos una matriz de 3 filas y 2 columnas de números flotant
mostrar_informacion_ndarray(a)
```

```
Forma : (3, 2)
Tipo  : float32
N dims: 2
Size  : 6
Valores:
[[0.0e+00 0.0e+00]
 [0.0e+00 0.0e+00]
 [4.5e-44 0.0e+00]]
```

Ejercicio 01-04: Crea un ndarray (sin inicializar sus valores) que represente un array de 10 elementos de tipo entero de 32 bits.

El resultado debería ser algo como (los valores son aleatorios):

```
Forma : (10,)
Tipo  : int32
N dims: 1
Size  : 10
Valores:
[0 0 1 0 2 0 0 0 4 0]
```

In [5]:
```
a = None
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Ejercicio 01-05: Crea un ndarray que represente una matriz con 3 filas y 5 columnas y elementos de tipo flotante de 64 bits.

El resultado debería ser algo como (los valores son aleatorios):

```
Forma : (3, 5)
Tipo  : float64
N dims: 2
Size  : 15
Valores:
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

In [6]:
```
a = None
#pon tu código aquí.
```

```
#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Ejercicio 01-06: Crea un ndarray que represente un tensor de 3 dimensiones, con 4 columnas, 3 filas y 3 elementos de profudidad de tipo byte.

El resultado debería ser algo como (los valores son aleatorios):

```
 Forma : (3, 4, 3)
Tipo  : int8
N dims: 3
Size  : 36
Valores:
[[[0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]
 [[0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]
 [[0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]]
```

In [7]:
```
a = None
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Crear ndarray inicializando todos los elementos a un valor.

Si queremos crear el ndarray pero inicializando a un valor concreto sus elementos, podemos utilizar alguno de los siguientes los métodos:

- zeros() para inicializar a ceros.
- ones() para inicializar a unos.
- eye() para inicializar matrices "identidad".
- full() para inicializar con un valor dado.

Veamos algunos ejemplos:

In [8]:
```
a = np.zeros((3,3), 'f') #crea e inicializa a cero.
print('A:\n', a)

b = np.ones((3,3), 'f') #crea e inicializa a unos.
print('B:\n', b)

c = np.eye(3, dtype='f') #crea e inicializa a matriz I
```

```
print('C:\n', c)

d = np.full((2,3), -1, dtype='i') #crea e inicializa con valor -1.
print('D:\n', d)
```

```
A:
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
B:
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
C:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
D:
 [[-1 -1 -1]
 [-1 -1 -1]]
```

Ejercicio 01-07a: Crea un ndarray que represente un array de 10 elementos de tipo entero de 32 bits inicializados a 0.

El resultado debería ser algo así como:

```
 Forma : (10,)
Tipo  : int32
N dims: 1
Size  : 10
Valores:
[0 0 0 0 0 0 0 0 0 0]
```

In [9]:
```
a = None
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Ejercicio 01-07b: Crea un ndarray que represente una matriz con 5 filas y 5 columnas y elementos de tipo flotante de 32 bits inicializado a la matriz Identidad.

El resultado debería ser como:

```
 Forma : (5, 5)
Tipo  : float32
N dims: 2
Size  : 25
Valores:
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

In [10]:
```
a = None
```

```
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Ejercicio 01-07c: Crea un ndarray que representa un tensor de 3 dimensiones, con 5 columnas, 4 filas y 3 elementos de profundidad de tipo byte inicializados a 1.

El resultado debería ser como:

```
 Forma : (4, 5, 3)
Tipo  : int8
N dims: 3
Size  : 60
Valores:
[[[1 1 1]
  [1 1 1]
  [1 1 1]
  [1 1 1]
  [1 1 1]]
 [[1 1 1]
  [1 1 1]
  [1 1 1]
  [1 1 1]
  [1 1 1]]
 [[1 1 1]
  [1 1 1]
  [1 1 1]
  [1 1 1]
  [1 1 1]]
 [[1 1 1]
  [1 1 1]
  [1 1 1]
  [1 1 1]
  [1 1 1]]]
```

In [11]:
```
a = None
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Ejercicio 01-08: Crea un ndarray que represente una matriz con 3 filas y 2 columnas inicializada al valor 2.0 con tipo float 32bits.

El resultado debería ser como:

```
 Forma : (3, 2)
Tipo  : float32
N dims: 2
```

```
Size  : 6
Valores:
[[2. 2.]
 [2. 2.]
 [2. 2.]]
```

In [12]:
```
a = None
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Inicializar un ndarray con valores siguiendo una secuencia.

Podemos crear un array, es decir, un ndarray con 1 dimensión, con valores de una secuencia de valores.

- una secuencia de enteros equiespaciada con arange().
- un intervalo lineal dividido en N partes con linspace().
- un intervalo logarítmico dividido en N partes logspace().

In [13]:
```
a = np.arange(10)
print('A:\n', a)
b = np.linspace(0, 1, 10)
print('B:\n', b)
c = np.logspace(0, 1, 5)
print('C:\n', c)
```

```
A:
 [0 1 2 3 4 5 6 7 8 9]
B:
 [0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
C:
 [ 1.          1.77827941  3.16227766  5.62341325 10.        ]
```

Ejercicio 01-09: Crea un array que representa un rango de números enteros impares en el intervalo $[1, 50)$.

El resultado debería ser algo como:

```
 Forma : (25,)
Tipo  : int64
N dims: 1
Size  : 25
Valores:
[ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49]
```

In [14]:
```
a = None
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Ejercicio 01-10: Crea un array que represente el intervalo lineal $[-1.0, 1.0]$ con 11 valores.

El resultado debería ser algo como:

```
 Forma : (11,)
Tipo  : float64
N dims: 1
Size  : 11
Valores:
[-1.  -0.8 -0.6 -0.4 -0.2  0.   0.2  0.4  0.6  0.8  1. ]
```

In [15]:
```
a = None
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Ejercicio 01-11: Crea un array que representa el intervalo logarítmico $[1.0, 10000.0]$ con un 10 valores.

El resultado sería algo como:

```
 Forma : (10,)
Tipo  : float64
N dims: 1
Size  : 10
Valores:
[1.00000000e+00 2.78255940e+00 7.74263683e+00 2.15443469e+01
 5.99484250e+01 1.66810054e+02 4.64158883e+02 1.29154967e+03
 3.59381366e+03 1.00000000e+04]
```

In [16]:
```
a = None
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Inicializar un ndarray con una rejilla de valores.

También podemos crear rejillas (*grids*) n-dimensionales de puntos. Para ello se utiliza la función meshgrid(). Esta función combina secuencias de valores creadas con alguno de los métodos anteriores para generar los valores de la rejilla.

Por ejemplo vamos a crear un grid de dos dimensiones con 5 particiones para cada eje.

In [17]:
```
X, Y = np.meshgrid(np.linspace(0.0, 1.0, 5), np.linspace(0.0, 1.0, 5))
print ('X:\n', X)
print ('Y:\n', Y)
```

```
X:
 [[0.   0.25 0.5  0.75 1.  ]
 [0.   0.25 0.5  0.75 1.  ]
 [0.   0.25 0.5  0.75 1.  ]
 [0.   0.25 0.5  0.75 1.  ]
 [0.   0.25 0.5  0.75 1.  ]]
Y:
 [[0.   0.   0.   0.   0.  ]
```

```
 [0.25 0.25 0.25 0.25 0.25]
 [0.5  0.5  0.5  0.5  0.5 ]
 [0.75 0.75 0.75 0.75 0.75]
 [1.   1.   1.   1.   1.  ]]
```

Ejercicio 01-12 Crea un rejilla 2D con el espacio lineal $[-1, 1]$ con cuatro muestras 5 para cada eje.

El resultado sería algo como:

```
 X
Forma : (5, 5)
Tipo  : float64
N dims: 2
Size  : 25
Valores:

 [[-1.  -0.5  0.   0.5  1. ]
  [-1.  -0.5  0.   0.5  1. ]
  [-1.  -0.5  0.   0.5  1. ]
  [-1.  -0.5  0.   0.5  1. ]
  [-1.  -0.5  0.   0.5  1. ]]

 Y
Forma : (5, 5)
Tipo  : float64
N dims: 2
Size  : 25
Valores:

 [[-1.  -1.  -1.  -1.  -1. ]
  [-0.5 -0.5 -0.5 -0.5 -0.5]
  [ 0.   0.   0.   0.   0. ]
  [ 0.5  0.5  0.5  0.5  0.5]
  [ 1.   1.   1.   1.   1. ]]
```

In [18]:
```
X = None
Y = None
#pon tu código aquí.

#
print('X')
mostrar_informacion_ndarray(X)
print('')
print('Y')
mostrar_informacion_ndarray(Y)
```

```
X
El agumento no es un ndarray!!.

Y
El agumento no es un ndarray!!.
```

Crear un ndarray a partir de otros objetos.

Podemos crear un ndarray a partir de un objeto python que ofrezca la interfaz de tipo array, por un ejemplo una lista, usando la función array():

In [19]:
```
a = np.array([1, 3, -1, 5, 2, 5]) #crear un array desde una lista.
print('A:\n', a)
```

```
A:
 [ 1  3 -1  5  2  5]
```

Ejercicio 01-13: usando la lista de listas `[[1, 2, 3], [4, 5, 6], [7,8,9]]` crea un ndarray de dos dimensiones con tipo flotante de 32 bits.

El resultado debería ser algo como:

```
Forma : (3, 3)
Tipo  : float32
N dims: 2
Size  : 9
Valores:
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

In [20]:
```
a = None
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Crear un ndarray cargado desde un "string".

Podemos crear un array convirtiendo el texto almacenado en un string usando la función fromstring().

Por ejemplo si tenemos el string `"1 3 -1 5 2 5"` podemos convertir el string en un array de la forma (se usa el caracter "espacio en blanco" como separdor entre valores):

In [21]:
```
a = np.fromstring("1 3 -1 5 2 5", dtype=int, sep=' ')
print('A: ',a)
```

```
A:  [ 1  3 -1  5  2  5]
```

Ejercicio 01-14: dado el string `"1,2,3,4,5,6,7,8,9"` se desea convertirlo en un array de tipo float de 32 bits.

El resultado debería ser algo como:

```
Forma : (9,)
Tipo  : float32
N dims: 1
Size  : 9
Valores:
[1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

In [22]:
```
a = None
#pon tu código aquí.
```

```
#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Crear un array como copia de otro.

Cuando asignamos una ndarray a un variable `a`, lo que se asigna es una "referencia al objeto" y no el objeto en sí.

Por este motivo, si asignamos `b = a`, ahora tanto `a` como `b` referencian al mismo objeto y diremos que "b es un alias de a". Veamos un ejemplo:

In [23]:
```
a = np.arange(5)
print ('A (original)  : ', a)
b = a
a[2] = -1
b[3] = 0
print ('A (modificado): ', a)
print ('B (alias de A): ', b)
```

```
A (original)  :  [0 1 2 3 4]
A (modificado):  [ 0  1 -1  0  4]
B (alias de A):  [ 0  1 -1  0  4]
```

Si queremos crear un ndarray como una copia de otro ndarray, usaremos la función copy().

Veamos cómo cambia el resultado del ejemplo anterior:

In [24]:
```
a = np.arange(5)
print ('A (original)          : ', a)
b = np.copy(a)
a[2] = -1
b[3] = 0
print ('A (modificado)        : ', a)
print ('B (copia de A original): ', b)
```

```
A (original)          :  [0 1 2 3 4]
A (modificado)        :  [ 0  1 -1  3  4]
B (copia de A original):  [0 1 2 0 4]
```

Funciones para manipular un ndarray.

Crear un ndarray como la concatenación de varios.

Podemos crear un ndarray concatenando varios ndarrays a lo largo de una de sus dimensiones (o eje). Para ello usaremos la función concatenate().

Por ejemplo, tenemos un array `a` y queremos obtener un array nuevo como concatenación de `a` con `a`:

In [25]:
```
a = np.arange(10)
print('A:\n', a)
b = np.concatenate((a, a))
print('A concatenado:\n', b)
```

```
A:
 [0 1 2 3 4 5 6 7 8 9]
A concatenado:
 [0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9]
```

En este caso como sólo hay una dimensión no es necesario indicar el eje para concatenar. En el caso de

que tengamos dos o más dimensiones, el eje para concatenar será por defecto el eje 0. Si queremos utilizar otro eje, lo indicaremos con el argumento `axis`.

Ejercicio 01-15: Dadas la matriz `A` como `[[1,2,3],[4,5,6]]` queremos obtener su concatenación en sentido horizontal.

El resultado debería ser algo como:

```
 Forma : (2, 6)
Tipo  : int64
N dims: 2
Size  : 12
Valores:
[[1 2 3 1 2 3]
 [4 5 6 4 5 6]]
```

In [26]:
```
a = np.array([[1,2,3],[4,5,6]])
#pon tu código aquí.
#sugerencia: el sentido horizontal es el eje 1.

#
mostrar_informacion_ndarray(a)
```

```
Forma : (2, 3)
Tipo  : int64
N dims: 2
Size  : 6
Valores:
[[1 2 3]
 [4 5 6]]
```

Observa que con `concatenate()` se utiliza una de las dimensiones (ejes) que tengan los ndarrays. A veces nos interesará crear un ndarray apilando otros en una nueva dimensión. Para ello usaremos la función stack().

Por ejemplo, vamos a apilar un array en vertical.

In [27]:
```
a = np.arange(10)
print('A:\n', a)
a = np.stack ( (a, a), axis=0)
print ('A:\n', a)
```

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

Ejercicio 01-16: Dada la matriz `A` com `[[1,2,3],[4,5,6]]`, queremos apilarlo en profundidad (una nueva tercera dimensión).

El resultado debería ser algo como:

```
 Forma : (2, 3, 2)
Tipo  : int64
N dims: 3
Size  : 12
Valores:
```

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

 [[4 4]
  [5 5]
  [6 6]]]
```

In [28]:
```
a = np.array([[1,2,3],[4,5,6]])
#pon tu código aquí.
#sugerencia: el sentido profundidad es el eje 2.

#
mostrar_informacion_ndarray(a)
```

```
Forma : (2, 3)
Tipo  : int64
N dims: 2
Size  : 6
Valores:
[[1 2 3]
 [4 5 6]]
```

Modificando la forma de un ndarray.

Dado un ndarray podemos cambiarle su forma ('shape') siempre que el número de elementos resultantes sea el mismo.

Para ello utilizamos la función reshape() para obtener una copia con las dimensiones cambiadas. Veamos un ejemplo:

In [29]:
```
a = np.arange(10)
print ('A:\n', a)
b = a.reshape((2,5))
print ('B:\n', b)
```

```
A:
 [0 1 2 3 4 5 6 7 8 9]
B:
 [[0 1 2 3 4]
 [5 6 7 8 9]]
```

Ejercicio 01-17: A partir de un array creado en el rango lineal $[-10, 10]$ con espacio 21 muestras obtener la matriz de 3x7 correspondiente.

El resultado debería ser algo así:

```
 Forma : (3, 7)
Tipo  : float64
N dims: 2
Size  : 21
Valores:
[[-10.  -9.  -8.  -7.  -6.  -5.  -4.]
 [ -3.  -2.  -1.   0.   1.   2.   3.]
 [  4.   5.   6.   7.   8.   9.  10.]]
```

In [30]:
```
a = None
```

```
#pon tu código aquí.

#
mostrar_informacion_ndarray(a)
```

```
El agumento no es un ndarray!!.
```

Aplanar un ndarray.

Alternativamente, a veces será interesante obtener una versión "aplanada" de un ndarray. Para ello podremos utilizar la función ravel(). Veamos un ejemplo:

In [31]:
```
a = np.arange(12).reshape((3,4))
print ('A:\n', a)
print ('A "aplanada": ', np.ravel(a))
```

```
A:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
A "aplanada":  [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Modificando el tipo de los elementos.

Otras veces será interesante obtener una copia de un ndarray con la misma forma pero cambiando el tipo de los elementos. Para esta operación usaremos la función asarray().

Ejercicio 01-18: Dada la matriz de datos enteros `A` definida como `[[1,2,3],[4,5,6]]` queremos obtener una versión de la misma con tipo float de 32 bits.

El resultado debería ser algo como

```
A:
Forma : (2, 3)
Tipo  : int64
N dims: 2
Size  : 6
Valores:

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

B:
Forma : (2, 3)
Tipo  : float32
N dims: 2
Size  : 6
Valores:

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

In [32]:
```
A = np.array([[1,2,3],[4,5,6]])
B = None
#pon tu código aquí.
```

```
#
print('A:')
mostrar_informacion_ndarray(A)
print ('')
print ('B:')
mostrar_informacion_ndarray(B)
```

```
A:
Forma : (2, 3)
Tipo  : int64
N dims: 2
Size  : 6
Valores:
[[1 2 3]
 [4 5 6]]

B:
El agumento no es un ndarray!!.
```

In [ ]: