Entrada y salida con Numpy

Ya sabemos que 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).

En el cuaderno anterior, hemos aprendido a crear los ndarrays desde cero o desde otros objetos Python como las listas. En este cuaderno vamos a estudiar cómo cargar y almacenar ndarrays en disco.

Recuerda que la documentación completa sobre Numpy puede consultarse en la dirección numpy.org/doc/.

Inicializar el entorno.

Vamos a comenzar cargando el módulo y visualizando su versión ya que este dato es importante para consultar la ayuda. En el momento de escribir este tutorial la versión de Numpy con la que se trabaja es 1.22.4:

In [1]:
```
import numpy as np
print('Numpy version: {}'.format(np.__version__))
```

```
Numpy version: 1.22.4
```

Para poder simular ficheros en memoria y usarlos en los ejemplos vamos a cargar los tipos `StringIO` para simular un fichero texto y `BytesIO` para simular un fichero binario. Estos tipos están definidos en el módulo estándar de Python `io`.

In [2]:
```
from io import StringIO, BytesIO
```

Entrada y salida en modo texto.

El formato texto es una de las principales maneras de intercambiar datos entre máquinas ya que ofrece algunas ventajas como:

- Su simplicidad.
- Permite independizarse de la arquitectura hardware.
- Los archivos pueden ser leídos/interpretados por los humanos directamente.

Sin embargo también tiene algunas desventajas como son:

- Uso poco eficiente del espacio.
- Sobrecarga de tiempo debido a la conversión entre representación textual y binaria de los datos.
- Depender del idioma (localización) como por ejemplo el uso de ',' para separar la parte entera de la parte decimal.

Una de los principales formatos de intercambio en modo texto es el conocido formato CSV (Valores Separados por Comas). Se recomienda leer la referencia Comma-separated values.

Entrada en modo texto.

Tenemos varias alternativas para cargar un ndarray desde datos en formato texto en un fichero. La función más rápida (y simple) de usar es loadtxt(). Revisa su documentación.

Ejercicio 01: Cargar un array desde un fichero.

La salida debería ser algo parecido a:

```
Forma : (5,)
Tipo  : float64
N dims: 1
Size  : 5
Valores:
[ 3.  1.  0. -1.  5.]
```

In [3]:
```
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!!.')
```

In [4]:
```
file = StringIO(u'3 1 0 -1 5') #Esto simula un archivo texto.
a = None
#Pon tu código aquí.
#Sugerencia: usa loadtxt con los valores por defecto.

#
mostrar_informacion_ndarray(a)
```

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

Como ves estamos usando el caracter espacio `' '` para separar los valores pero podemos usar otros caracteres para separar valores por ejemplo ',', ':', ...

Ejercicio 02: Cargar un array desde un fichero usando `','` como separador.

Debería salir algo como:

```
Forma : (5,)
Tipo  : float64
N dims: 1
Size  : 5
Valores:
[ 3.  1.  0. -1.  5.]
```

In [5]:
```
file = StringIO(u'3,1,0,-1,5')
a = None
#Pon tu código aquí.
```

```
#Sugerencia: pon el valor apropiado al parámetro 'delimiter'.

#
mostrar_informacion_ndarray(a)
```

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

Ejercicio 03: Cargar un array desde un fichero usando ':' como separador.

Debería salir algo como:

```
Forma : (5,)
Tipo  : float64
N dims: 1
Size  : 5
Valores:
[ 3.  1.  0. -1.  5.]
```

In [6]:
```
file = StringIO(u'3:1:0:-1:5')
a = None
#Pon tu código aquí.
#Sugerencia: pon el valor apropiado al parámetro 'delimiter'.

#
mostrar_informacion_ndarray(a)
```

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

En muchas ocasiones nos encontramos que en el fichero a cargar hay líneas con comentarios sobre los datos normalmente estas líneas comienzan con el carácter `'#'` para indicarlo.

Por defecto `loadtxt()` detecta estos comentarios pero si se utiliza algún otro carácter habría que indicarlo con el parámetro `comments`.

Ejercicio 04: Cargar un array desde un fichero. Hay líneas con comentarios que comienzan con '!'.

Debería salir algo como:

```
Forma : (5,)
Tipo  : float64
N dims: 1
Size  : 5
Valores:
[ 3.  1.  0. -1.  5.]
```

In [7]:
```
file = StringIO(
u'''
!Esto es un ejemplo de comentario
3 1 0 -1 5
''')
a = None
#Pon tu código aquí.
#Sugerencia: pon el valor apropiado al parámetro 'comments'.

#
mostrar_informacion_ndarray(a)
```

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

Como se ve en los ejercicios anteriores, el tipo por defecto de los valores cargados es flotante de 64 bits. Podemos usar el parámetro `dtype` para indicar otro.

Ejercicio 05: Cargar un array desde un fichero. Queremos que el tipo de los valores sea entero con signo de 32 bits.

Debería salir algo como:

```
Forma : (5,)
Tipo  : int32
N dims: 1
Size  : 5
Valores:
[ 3  1  0 -1  5]
```

In [8]:
```
file = StringIO(u'3 1 0 -1 5')
a = None
#Pon tu código aquí.
#Sugerencia: pon el valor apropiado al parámetro 'dtype'.

#
mostrar_informacion_ndarray(a)
```

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

Especificando las dimensiones mínimas.

Por defecto se asume que si el fichero tiene una sola línea con datos, se devuelve un ndarray de una dimensión, en caso contrario se devuelve una matriz. Se asume entonces que todas las líneas tienen el mismo número de columnas.

Ejercicio 06: Cargar un array desde un fichero. El fichero tiene varias líneas por lo que se devuelve una matriz.

El resultado esperado sería:

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

In [10]:
```
file = StringIO(
u'''
#Esto es un ejemplo de matriz
3 1 0 -1 5
4 -1 0 0 2
''')
a=None
#Pon tu código aquí.
#Sugerencia: usa los parámetros por defecto..

#
mostrar_informacion_ndarray(a)
```

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

Sin embargo, podemos forzar el número mínimo de dimensiones con el parámetro 'ndmim'. Este parámetro tiene el valor 0 por defecto, que significa calcular de forma automática.

Ejercicio 07: Cargar un ndarray desde un fichero. Se quiere obtener una matriz.

Debería salir algo como:

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

In [11]:
```
file = StringIO(
u'''
3 1 0 -1 5 4 -1 0 0 2
''')
a=None
#Pon tu código aquí.
#Sugerencia: pon el valor apropiado al parámetro 'ndmin'.

#
mostrar_informacion_ndarray(a)
```

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

Indicando las columnas a cargar.

Cuando los datos de entrada son una tabla o matriz, podremos indicar qué columnas cargar indicando con el parámetro `usecols` una tupla con los índices de las columnas deseadas.

Ejercicio 08: Cargar los datos desde un fichero. Se quiere obtener una matriz matriz con las columnas de 2 y 5.

Debería salir algo como:

```
   Forma : (2, 2)
   Tipo  : float64
   N dims: 2
   Size  : 4
   Valores:
   [[0. 4.]
    [1. 2.]]
```

In [12]:
```
file = StringIO(
u'''
3 1 0 -1 5 4 -1 0 0 2
3 3 1 6 -1 2 -0 2 5 1
''')
a=None
#Pon tu código aquí.
#Sugerencia: pon el valor apropiado al parámetro 'usecols'.

#
mostrar_informacion_ndarray(a)
```

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

Manejando valores perdidos.

La existencia de valores perdidos es común a la mayoría de los conjuntos de datos y por lo tanto

necesitamos buscar alguna estrategia para tratarlos. Se recomienda leer esta referencia para tener una idea completa sobre este concepto.

La función `loadtxt()` no permite manejar valores perdidos fácilmente. Si nuestros datos de entrada sufren de este problema, podremos utilizar la función genfromtxt() que permite indicar cómo gestionar los valores perdidos.

Afortunadamente la función `loadtxt()` es una versión simplificada de `gemfromtxt()` así que todo lo aprendido es de aplicación directa a esta nueva función.

La gestión de los valores perdidos se indica con los parámetros `missing_values` y `filling_values`.

Ejercicio 09: Cargar los datos desde un fichero. Como pueden haber valores "perdidos" utilizaremos 'genfromtxt()'.

Debería salir algo como:

```
Usando loadtxt()...
Detectados valores perdidos.
Usando genfromtxt()...
Forma : (3, 4)
Tipo  : float64
N dims: 2
Size  : 12
Valores:
[[ 1.  2.  3.  4.]
 [ 5. nan  7.  8.]
 [ 9. 10. nan 12.]]
```

In [14]:
```
file = StringIO(
u'''
1,2,3,4
5,,7,8
9,10,,12
''') #Observar que hay valores "perdidos".

a = None
try:
    print("Usando loadtxt()... ")
    #Pon tu código aquí.

    #
except ValueError:
    print("Detectados valores perdidos.")
    print("Usando genfromtxt()... ")
    file.seek(0) #simular cerrar/abrir el fichero.
    #Pon tu código aquí.

    #
mostrar_informacion_ndarray(a)
```

```
Usando loadtxt()...
El agumento no es un ndarray!!.
```

Vemos como el valor por defecto para los valores perdidos será `'np.nan'` (Not An Number). Además se detecta un valor perdido cuando su campo está vacío.

Podemos indicar el valor usado para un "valor perdido" con el parámetro `filling_values`. También es común utilizar algún texto como por ejemplo `"na"` (de Not Applicable) o `'-'` para indicar que un valor no ha podido ser dado. Estas marcas pueden indicarse con el parámetro `missing_values`.

Ejercicio 10: Cargar los datos desde un fichero. Pueden haber valores "perdidos" que son marcados con el texto `'na'`. Se desea utilizar el valor 0 para los valores perdidos.

Debería salir algo como:

```
Forma : (3, 4)
Tipo  : float64
N dims: 2
Size  : 12
Valores:
[[ 1.  2.  3.  4.]
 [ 5.  0.  7.  8.]
 [ 9. 10.  0. 12.]]
```

In [15]:
```
file = StringIO(
u'''
1,2,3,4
5,na,7,8
9,10,na,12
''') #observa que hay valores perdidos marcados con 'na'

a = None
#Pon tu código aquí.
#Sugerencia: revisa los parámetros missing_values y filling_values.

#
mostrar_informacion_ndarray(a)
```

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

Guardando datos en formato texto.

Si queremos guardar un ndarray en un fichero en formato texto (`.txt` , `.csv`) utilizaremos la función savetxt(). Se recomienda revisar su documentación.

Vamos a definir una función auxiliar que utilizaremos para realizar los ejercicios de esta sección:

In [16]:
```
def print_file(file):
    '''
    Imprime un fichero de texto línea a línea, desde la primera línea.

    Params:
        file indica el fichero a imprimir.
    '''
    file.seek(0) # me posiciono al principio del fichero.
    for line in file:
        print(line, end='')

#Ejemplo de uso:
file = StringIO(
u'''
1,2,3,4
5,,7,8
9,10,,12
```

```
''')
print_file(file)
```

```
1,2,3,4
5,,7,8
9,10,,12
```

Ejercicio 11: Guardar un ndarray en un fichero conf formato texto.

El fichero debería contener el texto:

```
0.00000000000000000e+00
2.50000000000000000e+00
5.00000000000000000e+00
7.50000000000000000e+00
1.00000000000000000e+01
```

In [17]:
```
file = StringIO()
a = np.linspace(0, 10, 5)

#Pon tu código aquí.
#Sugerencia: utiliza los valores por defecto de los parámetros.

#

print_file(file)
```

Especificando un formato para almacenar los valores.

Como ves en el ejercicio anterior se está usando toda la precisión a la hora de guardar los datos. Podemos indicar con el parámetro `fmt` el formato deseado. La forma de especificar está explicada en esta referencia.

Ejercicio 12: Guardar el ndarray referenciado por la variable `a` en un fichero con formato texto. Queremos almacenar los datos como flotantes usando una precisión de 3 decimales.

El fichero debería contener el texto:

```
0.000 0.909 1.818 2.727
3.636 4.545 5.455 6.364
7.273 8.182 9.091 10.000
```

In [19]:
```
file = StringIO()
a = np.linspace(0, 10, 12).reshape(3,4)

#Pon tu código aquí.
#Sugerencia: usa el parámetro "fmt"

#

print_file(file)
```

Ejercicio 13: Guardar el ndarray referenciado por la variable `a` en un fichero con formato texto. Queremos almacenar los datos como flotantes usando 7 espacios de los cuales 3 se reservan para los decimales.

El fichero debería contener el texto:

```
0.000   0.909   1.818   2.727
3.636   4.545   5.455   6.364
```

```
   7.273   8.182   9.091  10.000
```

In [20]:
```
file = StringIO()
a = np.linspace(0, 10, 12).reshape(3,4)

#Pon tu código aquí.
#Sugerencia: usa el parámetro "fmt"

#

print_file(file)
```

Ejercicio 14: Guardar el ndarray referenciado por la variable `a` en un fichero con formato texto. Queremos almacenar los datos como enteros usando 3 espacios por valor.

El fichero debería contener el texto:

```
  0
 25
 50
 75
100
```

In [21]:
```
file = StringIO()
a = np.linspace(0, 100, 5)

#Pon tu código aquí.
#Sugerencia: usa el parámetro "fmt"

#

print_file(file)
```

Ejercicio 15: Guardar el ndarray referenciado por la variable `a` en un fichero con formato texto. Queremos almacenar los datos como enteros usando 3 espacios por valor, pero justificados a la izquierda.

El fichero debería contener el texto:

```
0
25
50
75
100
```

In [22]:
```
file = StringIO()
a = np.linspace(0, 100, 5)

#Pon tu código aquí.
#Sugerencia: usa el parámetro "fmt"

#

print_file(file)
```

Como vemos hasta ahora cada valor se guarda en una línea separada ya que los arrays que tienen forma (x,) se consideran como si fueran matrices de la forma (x,1), es decir, tantas filas como elementos.

Si queremos almacenar los valores del array en una fila, debemos cambiar su forma para que sea una matriz con forma (1, x).

Ejercicio 15b: Guardar el array referenciado por la variable `a` en un fichero con formato texto. Queremos almacenar los valores como flotantes con tres decimales y en la misma línea del fichero.

El fichero debería contener el texto:

```
0.000 25.000 50.000 75.000 100.000
```

In [23]:
```
file = StringIO()
a = np.linspace(0, 100, 5)

#Pon tu código aquí.
#Sugerencia: cambia la forma de a de (x,) a (1,x).

#

print_file(file)
```

Almacenar en formato CSV.

Ya sabemos que el formato texto CSV usa un carácter para delimitar los valores. Para indicar qué valor queremos usar para delimitar usaremos el parámetro `delimiter` cuyo valor por defecto es el espacio en blanco.

Ejercicio 16: Guardar el ndarray referenciado por la variable `a` en un fichero con formato CSV usando el carácter `,` para delimitar los valores. Queremos además almacenar los datos como flotantes con 3 decimales.

El fichero debería contener el texto:

```
0.000,9.091,18.182,27.273
36.364,45.455,54.545,63.636
72.727,81.818,90.909,100.000
```

In [25]:
```
file = StringIO()
a = np.linspace(0, 100, 12).reshape(3,4)

#Pon tu código aquí.
#Sugerencia: usa el parámetro "fmt"

#

print_file(file)
```

Entrada y salida en modo binario.

En el anterior apartado hemos visto varias formas de realizar entrada/salida del tipo ndarray en formato texto.

Como vimos, este formato tiene varias desventajas, como hacer un uso ineficiente del espacio y su lentitud ya que requiere convertir texto a valor binarios o viceversa.

Para superar estos inconvenientes podemos cargar/almacenar los datos en formato binario.

Manejando el formato binario básico.

Para cargar/almacenar datos en formato binario lo más fácil es usar las funciones load() y save().

Ejercicio 17: Dado un ndarray en un variable `a` almacenarlo usando formato binario en un fichero `file` y recuperarlo posteriormente en una variable `b`.

Debería aparecer en la salida algo como:

```
Guardando el array A:
Forma : (3, 4)
Tipo  : float64
N dims: 2
Size  : 12
Valores:
[[  0.           9.09090909  18.18181818  27.27272727]
 [ 36.36363636  45.45454545  54.54545455  63.63636364]
 [ 72.72727273  81.81818182  90.90909091 100.        ]]

Cargado el array B:
Forma : (3, 4)
Tipo  : float64
N dims: 2
Size  : 12
Valores:
[[  0.           9.09090909  18.18181818  27.27272727]
 [ 36.36363636  45.45454545  54.54545455  63.63636364]
 [ 72.72727273  81.81818182  90.90909091 100.        ]]
```

In [28]:
```
file = BytesIO()
a = np.linspace(0, 100, 12).reshape(3,4)
print('Guardando el array A:')
mostrar_informacion_ndarray(a)

#Pon tu código aquí.
#Sugerencia: usa la función save().

#

file.seek(0) #simular cerrar/abrir el fichero.
b = None
#Pon tu código aquí.
#Sugerencia: usa la función load()

#
print('')
print('Cargado el array B:')
mostrar_informacion_ndarray(b)
```

```
Guardando el array A:
Forma : (3, 4)
Tipo  : float64
N dims: 2
Size  : 12
Valores:
[[  0.           9.09090909  18.18181818  27.27272727]
 [ 36.36363636  45.45454545  54.54545455  63.63636364]
 [ 72.72727273  81.81818182  90.90909091 100.        ]]
```

```
Cargado el array B:
El agumento no es un ndarray!!.
```

Almacenando más de un ndarray en un mismo fichero.

Se puede almacenar más de un ndarray en un mismo fichero usando la función savez().

En principio a cada ndarray se le asignará una etiqueta `arr_0`, `arr_1`, ..., pero podremos darles las etiquetas que queramos usando argumentos `keywords`, es decir, si queremos asignar la etiqueta `'X'` a un ndarray referenciado por la variable `a` lo indicamos de la forma `X=a` como parámetro.

Ejercicio 18: Generar un grid `X`, `Y` con el intervalo lineal [-1, 1] con 3 muestras para ambos ejes. Almacenar en formato binario los dos ndarrays usando las etiquetas `'X'` e `'Y'` respectivamente. Para comprobar el resultado volver a cargar los intervalos y visualizarlos.

Debería aparecer en la salida algo como:

```
X:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1.  0.  1.]
 [-1.  0.  1.]
 [-1.  0.  1.]]

Y:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1. -1. -1.]
 [ 0.  0.  0.]
 [ 1.  1.  1.]]

NDArrays almacenados:  ['arr_0', 'arr_1']

X cargado:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1.  0.  1.]
 [-1.  0.  1.]
 [-1.  0.  1.]]

Y cargado:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1. -1. -1.]
```

```
 [ 0.  0.  0.]
 [ 1.  1.  1.]]
```

In [30]:

```
file = BytesIO()
X, Y = np.meshgrid(np.linspace(-1, 1, 3), np.linspace(-1, 1, 3))
print('X:')
mostrar_informacion_ndarray(X)
print('')
print('Y:')
mostrar_informacion_ndarray(Y)
print('')

#Pon tu código aquí.
#Sugerencia: usave savez para almacenar mas de un ndarray.

#

file.seek(0) #simular cerrar/abrir el fichero.
ds = None
Y= None
X= None

#Pon tu código aquí.
#Sugerencia: carga el dataset en la variable 'ds' con np.load

#
if ds is not None:
    print('NDArrays almacenados: ', ds.files)
    print('')

#Obtener los ndarrays con etiquetas 'X' e 'Y'
#Pon tu código aquí.
#Sugerencia: utiliza ds['<etiqueta>'] para acceder a los ndarrays
# almacenados.


#

print('X cargado:')
mostrar_informacion_ndarray(X)
print('')
print('Y cargado:')
mostrar_informacion_ndarray(Y)
print('')
```

```
X:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1.  0.  1.]
 [-1.  0.  1.]
 [-1.  0.  1.]]

Y:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1. -1. -1.]
 [ 0.  0.  0.]
 [ 1.  1.  1.]]
```

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

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

Ejercicio 19: Generar un grid `X`, `Y` con el intervalo lineal [-1, 1] con 3 muestras para ambos ejes. Almacenar en formato binario los dos ndarrays asignando la etiqueta "X" para el ndarray `X` y la etiqueta "Y" para el ndarray `Y`. Para comprobar el resultado volver a cargar los intervalos en dos variables `x`, `y`.

Debería aparecer en la salida algo como:

```
X:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1.  0.  1.]
 [-1.  0.  1.]
 [-1.  0.  1.]]

Y:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1. -1. -1.]
 [ 0.  0.  0.]
 [ 1.  1.  1.]]

NDArrays almacenados:  ['X', 'Y']

X cargado:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1.  0.  1.]
 [-1.  0.  1.]
 [-1.  0.  1.]]

Y cargado:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1. -1. -1.]
 [ 0.  0.  0.]
 [ 1.  1.  1.]]
```

In [31]:
```
file = BytesIO()
X, Y = np.meshgrid(np.linspace(-1, 1, 3), np.linspace(-1, 1, 3))
print('X:')
mostrar_informacion_ndarray(X)
```

```
print('')
print('Y:')
mostrar_informacion_ndarray(Y)
print('')

#Pon tu código aquí.
#Sugerencia: usave savez para almacenar mas de un ndarray usando
# usando keywords para asignar las etiquetas indicadas.

#

file.seek(0) #simular cerrar/abrir el fichero.
ds = None
Y= None
X= None

#Pon tu código aquí.
#Sugerencia: carga el dataset en la variable 'ds' con np.load

#
if ds is not None:
    print('NDArrays almacenados: ', ds.files)
    print('')

#Obtener los ndarrays con etiquetas 'X' e 'Y'
#Pon tu código aquí.
#Sugerencia: utiliza ds['<etiqueta>'] para acceder a los ndarrays
# almacenados.

#

print('X cargado:')
mostrar_informacion_ndarray(X)
print('')
print('Y cargado:')
mostrar_informacion_ndarray(Y)
print('')
```

```
X:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1.  0.  1.]
 [-1.  0.  1.]
 [-1.  0.  1.]]

Y:
Forma : (3, 3)
Tipo  : float64
N dims: 2
Size  : 9
Valores:
[[-1. -1. -1.]
 [ 0.  0.  0.]
 [ 1.  1.  1.]]

X cargado:
El agumento no es un ndarray!!.

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

Almacenando de forma comprimida.

Para ahorrar espacio podemos optar por almacenar los datos de forma comprimida, a costa de aumentar el tiempo de almacenamiento y carga. Para ello usaremos la función savez_compressed().

Ejercicio 20: Generar una matriz de 10x10 valores flotantes rellenados a partir del intervalo lineal $[0, 100)$ con 100 muestras. Almacenar el resultado en tres ficheros, 'file1' en modo CSV con ',' para delimitar, 'file2' en modo binario simple y 'file3' en modo binario comprimido. Comparar el tamaño de los ficheros generados.

El resultado que se muestra debería ser algo parecido a:

```
Tamaño CSV           2500 bytes
Tamaño Binario       928 bytes
Tamaño Binario comp. 722 bytes
```

In [32]:
```
a = np.linspace(0, 100, 100).reshape(10,10)
file1 = StringIO()
file2 = BytesIO()
file3 = BytesIO()

#Pon tu código aquí.
#Usa las funciones savetxt, save and savez_compressed según
#  corresponda.

#

print(f"Tamaño CSV           {len(file1.getvalue())} bytes")
print(f"Tamaño Binario       {len(file2.getvalue())} bytes")
print(f"Tamaño Binario comp. {len(file3.getvalue())} bytes")
```

```
Tamaño CSV           0 bytes
Tamaño Binario       0 bytes
Tamaño Binario comp. 0 bytes
```

In [ ]:
```
```