Indexación avanzada

Ya hemos visto que podemos acceder a los elementos de un ndarray de forma directa indicando su posición para acceder a un elemento concreto, por ejemplo `x[1]` o `x[3,2]`, o bien utilizando rangos, por ejemplo `x[3:5]`. Este tipo de acceso es denominado *indexación básica* y lo que devuelve es una "vista" del ndarray.

En este cuaderno vamos a estudiar otras formas de acceder a los elementos de ndarray denominada *indexación avanzada* y que es utilizada cuando se utiliza para indexar un objeto que no es una tupla secuencial, por ejemplo `x[[3, 5, 1]]`, o se utiliza para indexar un objeto ndarray `obj` cuyos elementos sean enteros o valores lógicos, por ejemplo `x[obj]`. Cuando se utiliza este tipo de indexación lo que se obtiene son "copias" del ndarray accedido.

Este cuaderno se basa en la documentación oficial que puedes consultar aquí.

Configuración del entorno.

Como siempre, vamos a comenzar cargando el módulo y visualizando su versión ya que este dato es importante para consultar la ayuda. Aprovechamos para indicar que queremos mostrarar los números flotantes con una precisión de 3 decimales.

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
from numpy import random
print('Numpy version: {}'.format(np.__version__))
np.set_printoptions(precision=3)
```

```
Numpy version: 1.22.4
```

Consideraciones generales.

La indexación avanza `ndarray[` *index* `]` se activa siempre que el objeto selección *index* no sea una tupla secuencial. Por ejemplo `x[(1,2,3)]`, que es equivalente a `x[1,2,3]` utiliza indexación "básica", mientras que por ejemplo `x[ [1,2,3] ]` implica usar indexación "avanzada".

Esta distinción es muy importante ya la indexación "básica" nos devuelve una *vista* del ndarray que está siendo indexado y eso significa que, si modificamos la *vista*, también modificaremos el ndarray original.

En contraposición, la indexación "avanzada" devuelve una copia de los datos por lo que si modificamos la copia, el ndarray original no será afectado.

Usando ndarrays lógicos como máscaras.

La indexación avanza usando un array de valores lógicos nos permite seleccionar elementos de un ndarray basados en que se cumpla o no un predicado lógico.

La notación usada para activar la indexación avanzada sería: `ndarray[ obj ]` donde `obj` es un array de valores lógicos indicando qué índices del ndarray queremos seleccionar.

El array `obj` de valores lógicos se suele denominar con el térmico *máscara de selección* y para calcularlo podemos usar funciones universales para comparar valores y combinar las comparaciones con funciones universales lógicas para crear predicados lógicos complejos. Se recomienda repasar esta referencia antes de realizar los siguientes ejercicios.

Ejercicio 01: Dado un ndarray con valores flotantes, se desea detectar cuáles valores pertenecen al intervalo $[-0.5, 0.5]$.

La salida debería ser algo parecido a:

```
Datos:
 [[ 0.126 -0.132  0.64   0.105]
 [-0.536  0.362  1.304  0.947]
 [-0.704 -1.265 -0.623  0.041]]
Valores en el intervalo [-0.5, 0.5]
 [[ True  True False  True]
 [False  True False False]
 [False False False  True]]
```

In [2]:
```
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
data = gen.normal(size=(3,4))
mascara = np.full(data.shape, False)
#Pon tu código aquí.

#
print("Datos: \n", data)
print("Valores en el intervalo [-0.5, 0.5]\n", mascara)
```

```
Datos:
 [[ 0.126 -0.132  0.64   0.105]
 [-0.536  0.362  1.304  0.947]
 [-0.704 -1.265 -0.623  0.041]]
Valores en el intervalo [-0.5, 0.5]
 [[False False False False]
 [False False False False]
 [False False False False]]
```

Ejercicio 02: Dado un ndarray con valores flotantes, se desea crear una máscara (array de valores lógicos) para visualizar los valores del ndarray que están fuera del intervalo $[-0.5, 0.5]$.

La salida debería ser algo parecido a:

```
Datos:
 [[ 0.126 -0.132  0.64   0.105]
 [-0.536  0.362  1.304  0.947]
 [-0.704 -1.265 -0.623  0.041]]
Valores fuera del intervalo [-0.5, 0.5]
 [ 0.64  -0.536  1.304  0.947 -0.704 -1.265 -0.623]
```

In [3]:
```
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
data = gen.normal(size=(3,4))
mascara = np.full(data.shape, False)
```

```
#Pon tu código aquí.

#
print("Datos: \n", data)
print("Valores fuera del intervalo [-0.5, 0.5]:\n", data[mascara])
```

```
Datos:
 [[ 0.126 -0.132  0.64   0.105]
 [-0.536  0.362  1.304  0.947]
 [-0.704 -1.265 -0.623  0.041]]
Valores fuera del intervalo [-0.5, 0.5]:
 []
```

Ejercicio 03: Tras cargar un dataset desde un fichero CSV, sabemos que los valores "perdidos" se les ha asignado el valor `np.nan`. Queremos corregir los valores perdidos rellenandolos con un valor que siga una distribución gaussiana con parámetros μ_j, σ_j obtenidos a partir de la columna j dónde correspondiente al valor perdido, es decir,

$$\text{Si } \text{data}[i,j] \equiv \text{np.nan}, \text{data}[i,j] = \alpha, \alpha \sim N(\mu_j, \sigma_j)$$

La salida debería ser algo parecido a:

```
Datos:
 [[ 0.126 -0.132  0.64 ]
 [ 0.105 -0.536    nan]
 [ 1.304  0.947 -0.704]
 [-1.265 -0.623  0.041]
 [-2.325 -0.219 -1.246]
 [   nan -0.544 -0.316]
 [ 0.412  1.043 -0.129]
 [ 1.366    nan  0.352]
 [ 0.903  0.094 -0.743]
 [-0.922 -0.458  0.22 ]]

Medias      :  [-0.033 -0.048 -0.209]
Desviaciones:  [1.175 0.598 0.568]

Máscara:
 [[False False False]
 [False False  True]
 [False False False]
 [False False False]
 [False False False]
 [ True False False]
 [False False False]
 [False  True False]
 [False False False]
 [False False False]]

Relleno:
 [[-1.219 -0.173 -0.3  ]
 [ 0.603  0.081 -0.007]
 [-0.801 -0.125  0.236]
 [ 1.722 -0.801  0.651]
 [ 1.549  0.42  -0.059]
 [-0.402  0.824  0.905]
 [ 2.084  0.739 -0.006]
 [-1.453 -0.05   0.164]
 [-1.547  0.189  0.035]
```

```
 [ 0.785 -0.756 -0.585]]

Datos corregidos:
 [[ 0.126 -0.132  0.64 ]
 [ 0.105 -0.536 -0.007]
 [ 1.304  0.947 -0.704]
 [-1.265 -0.623  0.041]
 [-2.325 -0.219 -1.246]
 [-0.402 -0.544 -0.316]
 [ 0.412  1.043 -0.129]
 [ 1.366 -0.05   0.352]
 [ 0.903  0.094 -0.743]
 [-0.922 -0.458  0.22 ]]
```

In [4]:

```python
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
data = gen.normal(size=(10,3))
#Simulamos varios valores np.nan
data[1, 2]=np.nan
data[5, 0]=np.nan
data[7, 1]=np.nan

print("Datos: \n", data)

medias = np.zeros(data.shape[1])
sigmas = np.zeros(data.shape[1])
#1. calcular los parámetros media,desviación por columna.
#Pon tu código aquí.
#Sugerencia: usar funciones de agregación,
#recuerda que hay valores np.nan.

#
print('')
print("Medias      : ", medias)
print("Desviaciones: ", sigmas)

#2. calcular la máscara que identfica los valores np.nan.
#Pon tu código aquí.
#Sugerencia: contempla dos alternativas, comparar con el valor nan
#o utilizar la ufunc isnan()

#
print('')
print("Máscara: \n", mascara)

relleno = np.zeros_like(data)
#3. Generar una matriz con la misma forma que la original
#con valores aleatorios usando la distribución normal
#con las medias/desviaciones calculadas por columna.
#Pon tu código aquí.

#
print('')
print("Relleno:\n", relleno)

#4. Asigna, usando la máscara como objeto de indexación, los valores
# de ruido calculados.
#Pon tu código aquí.

#
print('')
print("Datos corregidos:\n", data)
```

```
Datos:
 [[ 0.126 -0.132  0.64 ]
```

```
 [ 0.105 -0.536    nan]
 [ 1.304  0.947 -0.704]
 [-1.265 -0.623  0.041]
 [-2.325 -0.219 -1.246]
 [   nan -0.544 -0.316]
 [ 0.412  1.043 -0.129]
 [ 1.366    nan  0.352]
 [ 0.903  0.094 -0.743]
 [-0.922 -0.458  0.22 ]]

Medias      :  [0. 0. 0.]
Desviaciones:  [0. 0. 0.]

Máscara:
 [[False False False False]
 [False False False False]
 [False False False False]]

Relleno:
 [[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.]]

Datos corregidos:
 [[ 0.126 -0.132  0.64 ]
 [ 0.105 -0.536    nan]
 [ 1.304  0.947 -0.704]
 [-1.265 -0.623  0.041]
 [-2.325 -0.219 -1.246]
 [   nan -0.544 -0.316]
 [ 0.412  1.043 -0.129]
 [ 1.366    nan  0.352]
 [ 0.903  0.094 -0.743]
 [-0.922 -0.458  0.22 ]]
```

Ejercicio 04: Tenemos un dataset donde la primera columna representa el género 0 para hombre, 1 para mujer. Se desea obtener promedios del resto de columnas según el género.

La salida debería ser algo parecido a:

```
   Datos:
    [[ 1.    -0.132  0.64 ]
    [ 1.    -0.536  0.362]
    [ 1.     0.947 -0.704]
    [ 0.    -0.623  0.041]
    [ 1.    -0.219 -1.246]
    [ 0.    -0.544 -0.316]
    [ 1.     1.043 -0.129]
    [ 1.    -0.665  0.352]
    [ 1.     0.094 -0.743]
    [ 1.    -0.458  0.22 ]]
   Promedio para "hombre":  [ 0.    -0.584 -0.137]
   Promedio para "mujer":  [ 1.     0.009 -0.156]
```

In [5]:
```
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
data = gen.normal(size=(10,3))
data[:,0] = gen.integers(2, size=10)
print("Datos: \n", data)
avg_hombre = np.zeros(2)
avg_mujer = np.zeros(2)
#Pon tu código aquí.
#Sugerencia: usar la ufunc equal() con la primera columna para
# crear la máscara de acceso.

#
print('Promedio para "hombre": ', avg_hombre)
print('Promedio para "mujer": ', avg_mujer)
```

```
Datos:
 [[ 1.    -0.132  0.64 ]
 [ 1.    -0.536  0.362]
 [ 1.     0.947 -0.704]
 [ 0.    -0.623  0.041]
 [ 1.    -0.219 -1.246]
 [ 0.    -0.544 -0.316]
 [ 1.     1.043 -0.129]
 [ 1.    -0.665  0.352]
 [ 1.     0.094 -0.743]
 [ 1.    -0.458  0.22 ]]
Promedio para "hombre":  [0. 0.]
Promedio para "mujer":  [0. 0.]
```

Ejercicio 05: Tenemos un dataset donde la primera columna representa el género 0 para hombre, 1 para mujer y la segunda columna representa la edad. Se desea obtener las medias de las columnas separando por "género" y "edad" asumiendo que "niños" serían edades en el rango $[0, 16)$, jóvenes el rango $[16, 30)$, adultos el rango $[30, 65)$ y ancianos el rango $[65, -)$

La salida debería ser algo parecido a:

```
Promedio para "hombres":
    Niño    :  [0.    7.987 4.763 5.378]
    jóven   :  [ 0.    21.277  5.064  5.126]
    adulto  :  [ 0.    48.286  5.044  4.84 ]
    anciano :  [ 0.    83.027  4.652  4.811]
Promedio para "mujer":
    Niño    :  [1.    7.194 4.611 5.332]
    jóven   :  [ 1.    21.7    4.684  5.089]
    adulto  :  [ 1.    47.027  4.888  4.955]
    anciano :  [ 1.    82.151  5.081  4.726]
```

(**Nota**: razonar por qué aparece np.nan en algunos resultados)

In [6]:
```
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
#Simulamos el dataset.
data = gen.random(size=(1000,4))*10.0
data[:, 0] = gen.integers(2, size=data.shape[0]) #género.
data[:, 1] = gen.integers(100, size=data.shape[0]) #edad.

#Para almacenar los resultados.
avg_h_nin = 0
avg_m_nin = 0
avg_h_jov = 0
avg_m_jov = 0
avg_h_adult = 0
avg_m_adult = 0
```

```
avg_h_anc = 0
avg_m_anc = 0

#Pon tu código aquí.
#Sugerencia: crea máscaras con ufuncs de comparación para las
#categorías hombre, mujer para el género y para niño, joven, adulto y
# anciano para la edad y después combinarlas con ufunc lógicas
# para calcular los promedios por separado.

#
print('Promedio para "hombres":')
print('\tNiño    : ', avg_h_nin)
print('\tjóven   : ', avg_h_jov)
print('\tadulto  : ', avg_h_adult)
print('\tanciano : ', avg_h_anc)
print('Promedio para "mujer": ')
print('\tNiño    : ', avg_m_nin)
print('\tjóven   : ', avg_m_jov)
print('\tadulto  : ', avg_m_adult)
print('\tanciano : ', avg_m_anc)
```

```
Promedio para "hombres":
        Niño    :  0
        jóven   :  0
        adulto  :  0
        anciano :  0
Promedio para "mujer":
        Niño    :  0
        jóven   :  0
        adulto  :  0
        anciano :  0
```

Ejercicio 06: Tenemos un dataset y queremos dividirlo en dos partes en función de un valor umbral t sobre la tercera columna. Así la parte 1 del dataset estará formada por todas las muestras i tales que $\text{data}[i,j] \leq t$ y la otra parte con las restantes.

La salida debería ser algo parecido a:

```
Datos:
 [[0.637 0.27  0.041 0.017]
 [0.813 0.913 0.607 0.729]
 [0.544 0.935 0.816 0.003]
 [0.857 0.034 0.73  0.176]
 [0.863 0.541 0.3   0.423]
 [0.028 0.124 0.671 0.647]
 [0.615 0.384 0.997 0.981]
 [0.686 0.65  0.688 0.389]
 [0.135 0.721 0.525 0.31 ]
 [0.486 0.889 0.934 0.358]]

Partición 1:
 [[0.637 0.27  0.041 0.017]
 [0.863 0.541 0.3   0.423]]

Partición 2:
 [[0.813 0.913 0.607 0.729]
 [0.544 0.935 0.816 0.003]
 [0.857 0.034 0.73  0.176]
 [0.028 0.124 0.671 0.647]
 [0.615 0.384 0.997 0.981]
 [0.686 0.65  0.688 0.389]
```

```
 [0.135 0.721 0.525 0.31 ]
 [0.486 0.889 0.934 0.358]]
```

In [7]:

```
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
#Simulamos el dataset.
data = gen.random(size=(10,4))
print("Datos: \n", data)

t = 0.5

data_1 = 0 #primera parte
data_2 = 0 #segunda parte
#Pon tu código aquí.
#Sugerencia: usa ufunc de comparación con el valor t para crear
#la máscara que se aplica para la columna 3.

#
print('')
print('Partición 1: \n', data_1)
print('')
print('Partición 2: \n', data_2)
```

```
Datos:
 [[0.637 0.27  0.041 0.017]
 [0.813 0.913 0.607 0.729]
 [0.544 0.935 0.816 0.003]
 [0.857 0.034 0.73  0.176]
 [0.863 0.541 0.3   0.423]
 [0.028 0.124 0.671 0.647]
 [0.615 0.384 0.997 0.981]
 [0.686 0.65  0.688 0.389]
 [0.135 0.721 0.525 0.31 ]
 [0.486 0.889 0.934 0.358]]

Partición 1:
 0

Partición 2:
 0
```

Ejercicio 07: Tenemos un dataset y queremos dividirlo en dos partes separando las columnas. La primera parte estará formada por aquellas columnas cuya media sea menor o igual que un umbral t y la otra por el resto de columnas. Así la parte 1 del dataset estará formada por todas las columnas j tales que $\mu_j \leq t$ y la parte 2 las restantes columnas.

La salida debería ser algo parecido a:

```
Datos:
 [[0.637 0.27  0.041 0.017]
 [0.813 0.913 0.607 0.729]
 [0.544 0.935 0.816 0.003]
 [0.857 0.034 0.73  0.176]
 [0.863 0.541 0.3   0.423]
 [0.028 0.124 0.671 0.647]
 [0.615 0.384 0.997 0.981]
 [0.686 0.65  0.688 0.389]
 [0.135 0.721 0.525 0.31 ]
 [0.486 0.889 0.934 0.358]]
Promedios:  [0.566 0.546 0.631 0.403]
Parte 1:
 [[0.017]
```

```
 [0.729]
 [0.003]
 [0.176]
 [0.423]
 [0.647]
 [0.981]
 [0.389]
 [0.31 ]
 [0.358]]
Parte 2:
 [[0.637 0.27  0.041]
 [0.813 0.913 0.607]
 [0.544 0.935 0.816]
 [0.857 0.034 0.73 ]
 [0.863 0.541 0.3  ]
 [0.028 0.124 0.671]
 [0.615 0.384 0.997]
 [0.686 0.65  0.688]
 [0.135 0.721 0.525]
 [0.486 0.889 0.934]]
```

In [8]:

```python
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
#Simulamos el dataset.
data = gen.random(size=(10,4))
print("Datos: \n", data)
umbral = 0.55
promedios = 0
data_1 = 0
data_2 = 0
#Pon tu código aquí.
#Sugerencia: usa ufunc de agregación para calcular la media y
#compara el resultado con el valor umbral para crear
#la máscara. Esta máscara se aplica a la segunda dimensión
#(columnas) para obtener una parte y su versión negada para obtener
#la otra parte.

#
print('Promedios: ', promedios)
print('Parte 1: \n', data_1)
print('Parte 2: \n', data_2)
```

```
Datos:
 [[0.637 0.27  0.041 0.017]
 [0.813 0.913 0.607 0.729]
 [0.544 0.935 0.816 0.003]
 [0.857 0.034 0.73  0.176]
 [0.863 0.541 0.3   0.423]
 [0.028 0.124 0.671 0.647]
 [0.615 0.384 0.997 0.981]
 [0.686 0.65  0.688 0.389]
 [0.135 0.721 0.525 0.31 ]
 [0.486 0.889 0.934 0.358]]
Promedios:  0
Parte 1:
 0
Parte 2:
 0
```

Indexación avanzada con enteros.

La indexación avanzada usando un array de enteros nos permite seleccionar elementos de un ndarray

basados en su índice n-dimensional.

La forma de indicar esto sería `ndarray[ obj ]` donde `obj` es un ndarray (o lista) de enteros indicando qué valores queremos seleccionar.

Ejercicio 08: Dado un dataset `data`, queremos obtener un subconjunto de datos con las filas 1, 3, 4 y 7.

La salida debería ser algo parecido a:

```
Datos:
 [[0.637 0.27  0.041 0.017]
 [0.813 0.913 0.607 0.729]
 [0.544 0.935 0.816 0.003]
 [0.857 0.034 0.73  0.176]
 [0.863 0.541 0.3   0.423]
 [0.028 0.124 0.671 0.647]
 [0.615 0.384 0.997 0.981]
 [0.686 0.65  0.688 0.389]
 [0.135 0.721 0.525 0.31 ]
 [0.486 0.889 0.934 0.358]]

Sub data:
 [[0.813 0.913 0.607 0.729]
 [0.857 0.034 0.73  0.176]
 [0.863 0.541 0.3   0.423]
 [0.686 0.65  0.688 0.389]]
```

In [9]:
```
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
#Simulamos el dataset.
data = gen.random(size=(10,4))
print("Datos: \n", data)
sub_data = 0
#Pon tu código aquí.
#Sugerencia crea un array 'idxs' con los índices de fila
#deseados: 1, 3, 4 y 7.

#
print('')
print('Sub data:\n', sub_data)
```

```
Datos:
 [[0.637 0.27  0.041 0.017]
 [0.813 0.913 0.607 0.729]
 [0.544 0.935 0.816 0.003]
 [0.857 0.034 0.73  0.176]
 [0.863 0.541 0.3   0.423]
 [0.028 0.124 0.671 0.647]
 [0.615 0.384 0.997 0.981]
 [0.686 0.65  0.688 0.389]
 [0.135 0.721 0.525 0.31 ]
 [0.486 0.889 0.934 0.358]]

Sub data:
 0
```

Ejercicio 09: Dado un dataset `data`, queremos obtener un subconjunto de datos con las columnas 1, 3, 4 y 7.

Observa que en este ejemplo se combina la indexación básica con la avanzada.

La salida debería ser algo parecido a:

```
Datos:
 [[0.637 0.27  0.041 0.017 0.813 0.913 0.607 0.729 0.544 0.935]
 [0.816 0.003 0.857 0.034 0.73  0.176 0.863 0.541 0.3   0.423]
 [0.028 0.124 0.671 0.647 0.615 0.384 0.997 0.981 0.686 0.65 ]
 [0.688 0.389 0.135 0.721 0.525 0.31  0.486 0.889 0.934 0.358]]

Sub data:
 [[0.27  0.017 0.813 0.729]
 [0.003 0.034 0.73  0.541]
 [0.124 0.647 0.615 0.981]
 [0.389 0.721 0.525 0.889]]
```

In [10]:
```
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
#Simulamos el dataset.
data = gen.random(size=(4,10))
print("Datos: \n", data)
sub_data = 0
#Pon tu código aquí.
#Sugerencia crea un array 'idxs' con los índices de fila
#deseados: 1, 3, 4 y 7.

#
print('')
print('Sub data:\n', sub_data)
```

```
Datos:
 [[0.637 0.27  0.041 0.017 0.813 0.913 0.607 0.729 0.544 0.935]
 [0.816 0.003 0.857 0.034 0.73  0.176 0.863 0.541 0.3   0.423]
 [0.028 0.124 0.671 0.647 0.615 0.384 0.997 0.981 0.686 0.65 ]
 [0.688 0.389 0.135 0.721 0.525 0.31  0.486 0.889 0.934 0.358]]

Sub data:
 0
```

Ejercicio 10: Dado un ndarray 'data' queremos obtener la tercera columna de las filas 1, 3, 4 y 7.

Observa que en este ejemplo se combina la indexación básica con la avanzada.

La salida debería ser algo parecido a:

```
Data
 [[ 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 27 28 29]
 [30 31 32 33 34]
 [35 36 37 38 39]
 [40 41 42 43 44]
 [45 46 47 48 49]]
sub data:  [ 7 17 22 37]
```

In [11]:
```
data = np.arange(50).reshape(10,5)
print('Data\n', data)
sub_data = 0
#Pon tu código aquí.
#Sugerencia: crea un array idx con los índices de filas
#requeridos.
```

```
#
print('sub data: ', sub_data)
```

```
Data
 [[ 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 27 28 29]
 [30 31 32 33 34]
 [35 36 37 38 39]
 [40 41 42 43 44]
 [45 46 47 48 49]]
sub data:  0
```

Ejercicio 11: Dada un matriz 3x3 queremos obtener las cuatro esquinas usando indexación avanzada con enteros.

Para resolver este ejercicio deberemos usar dos objetos para indexar, uno indicando las filas y el otro las columnas con la misma forma que el objeto que se está accediendo, es este caso es (3,3).

La salida debería ser algo parecido a:

```
La matriz original es:
 [[0 1 2]
 [3 4 5]
 [6 7 8]]

Índices de fila:
 [[0 0]
 [2 2]]

Índices de columna:
 [[0 2]
 [0 2]]

La matriz con las esquinas es:
 [[0 2]
 [6 8]]
```

In [12]:
```
m = np.arange(9).reshape(3,3)
print('La matriz original es:\n', m)
esquinas = np.zeros((2,2))
fil = np.zeros((2,2)) #índices de fila.
col = np.zeros((2,2)) #índices de columna.
#Pon tu código aquí.
#Sugerencia: queremos los valores con índice de
#fila/columna {0,2}.
#Crea dos matrices 2x2 fil y col una para indicar las filas
#y otra para las columnas.

#
print('')
print('Índices de fila:\n', fil)
print('')
print('Índices de columna:\n', col)
print('')
print('La matriz con las esquinas es:\n', esquinas)
```

```
La matriz original es:
```

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

Índices de fila:
 [[0. 0.]
 [0. 0.]]

Índices de columna:
 [[0. 0.]
 [0. 0.]]

La matriz con las esquinas es:
 [[0. 0.]
 [0. 0.]]
```

Ejercicio 12: Tenemos un dataset representado por la matriz X con las características medidas para cada muestra y la matriz Y con una sóla columna con la etiqueta asociada a cada muestra.

Se desea seleccionar de manera aleatoria tres de muestras, generando las matrices muestra_X, muestra_Y. El muestro aleatorio debe ser sin reemplazo.

La salida debería ser algo parecido a:

```
Datos X|Y:
 [[6.370e-01 2.698e-01 4.097e-02 8.000e+00]
 [1.653e-02 8.133e-01 9.128e-01 6.000e+00]
 [6.066e-01 7.295e-01 5.436e-01 7.000e+00]
 [9.351e-01 8.159e-01 2.739e-03 3.000e+00]
 [8.574e-01 3.359e-02 7.297e-01 8.000e+00]
 [1.757e-01 8.632e-01 5.415e-01 1.000e+00]
 [2.997e-01 4.227e-01 2.832e-02 5.000e+00]
 [1.243e-01 6.706e-01 6.472e-01 7.000e+00]
 [6.154e-01 3.837e-01 9.972e-01 8.000e+00]
 [9.808e-01 6.855e-01 6.505e-01 5.000e+00]]

Índices seleccionados:  [3 4 2]

Muestra X|Y:
 [[9.351e-01 8.159e-01 2.739e-03 3.000e+00]
 [8.574e-01 3.359e-02 7.297e-01 8.000e+00]
 [6.066e-01 7.295e-01 5.436e-01 7.000e+00]]
```

In [13]:

```python
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
#Simulamos el dataset.
X = gen.random(size=(10,3))
Y = gen.integers(10, size=(10,1))
print("Datos X|Y:\n", np.concatenate((X, Y), axis=1))

idx = 0
muestra_X = np.zeros_like(X)
muestra_Y = np.zeros_like(Y)

#Pon tu código aquí.
#Sugerencia: usando random.choice genera el vector idx con
#tres valores aleatorios en el rango entero
#[0, número de filas del dataset).
#Recuerda que no queremos repeticiones.
#Utiliza el vector idx para cómo índice para las matrices
#X e Y.
```

```
#
print('')
print('Índices seleccionados: ', idx)
print('')
print('Muestra X|Y:\n', np.concatenate((muestra_X, muestra_Y),axis=1))
```

```
Datos X|Y:
 [[6.370e-01 2.698e-01 4.097e-02 8.000e+00]
 [1.653e-02 8.133e-01 9.128e-01 6.000e+00]
 [6.066e-01 7.295e-01 5.436e-01 7.000e+00]
 [9.351e-01 8.159e-01 2.739e-03 3.000e+00]
 [8.574e-01 3.359e-02 7.297e-01 8.000e+00]
 [1.757e-01 8.632e-01 5.415e-01 1.000e+00]
 [2.997e-01 4.227e-01 2.832e-02 5.000e+00]
 [1.243e-01 6.706e-01 6.472e-01 7.000e+00]
 [6.154e-01 3.837e-01 9.972e-01 8.000e+00]
 [9.808e-01 6.855e-01 6.505e-01 5.000e+00]]

Índices seleccionados:  0

Muestra X|Y:
 [[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.]
 [0. 0. 0. 0.]]
```

Ejercicio 13: Tenemos un dataset con dos matrices X|Y que representan las características y la etiqueta asociadas (las columnas) a cada muestra (las filas). Queremos dividir de forma aleatoria el dataset en dos partes una para entrenar (*train*) y la otra para comprobar (*test*). Se utilizará un factor f para indicar la fracción del tamaño total utilizado para *train* y el resto será para *test*.

La salida debería ser algo parecido a:

```
Datos X|Y:
 [[6.370e-01 2.698e-01 4.097e-02 8.000e+00]
 [1.653e-02 8.133e-01 9.128e-01 6.000e+00]
 [6.066e-01 7.295e-01 5.436e-01 7.000e+00]
 [9.351e-01 8.159e-01 2.739e-03 3.000e+00]
 [8.574e-01 3.359e-02 7.297e-01 8.000e+00]
 [1.757e-01 8.632e-01 5.415e-01 1.000e+00]
 [2.997e-01 4.227e-01 2.832e-02 5.000e+00]
 [1.243e-01 6.706e-01 6.472e-01 7.000e+00]
 [6.154e-01 3.837e-01 9.972e-01 8.000e+00]
 [9.808e-01 6.855e-01 6.505e-01 5.000e+00]]

Train X|Y:
 [[2.997e-01 4.227e-01 2.832e-02 5.000e+00]
 [9.808e-01 6.855e-01 6.505e-01 5.000e+00]
 [1.243e-01 6.706e-01 6.472e-01 7.000e+00]
 [8.574e-01 3.359e-02 7.297e-01 8.000e+00]
 [6.066e-01 7.295e-01 5.436e-01 7.000e+00]
 [9.351e-01 8.159e-01 2.739e-03 3.000e+00]]

Test  X|Y:
 [[0.017 0.813 0.913 6.   ]
```

```
 [0.637 0.27  0.041 8.   ]
 [0.176 0.863 0.541 1.   ]
 [0.615 0.384 0.997 8.   ]]
```

In [14]:

```
gen = random.default_rng(0) #Semilla 0 para poder reproducir.
#Simulamos el dataset.
X = gen.random(size=(10,3))
Y = gen.integers(10, size=(10,1))
print("Datos X|Y:\n", np.concatenate((X, Y), axis=1))

f = 0.6 #Queremos el 60% para Train y 40% para Test.
X_train = np.zeros_like(X)
Y_train = np.zeros_like(Y)
X_test = np.zeros_like(X)
Y_test = np.zeros_like(Y)
#Pon tu código aquí
#Sugerencia: genera un array de índices con el rango del total
#de muestras (filas) del dataset. Permuta el array de índices
#La primera fracción de índices serán los índices para train
#y el resto para Test.

#
print('')
print("Train X|Y:\n", np.concatenate((X_train, Y_train), axis=1))
print('')
print("Test  X|Y:\n", np.concatenate((X_test, Y_test), axis=1))
```

```
Datos X|Y:
 [[6.370e-01 2.698e-01 4.097e-02 8.000e+00]
 [1.653e-02 8.133e-01 9.128e-01 6.000e+00]
 [6.066e-01 7.295e-01 5.436e-01 7.000e+00]
 [9.351e-01 8.159e-01 2.739e-03 3.000e+00]
 [8.574e-01 3.359e-02 7.297e-01 8.000e+00]
 [1.757e-01 8.632e-01 5.415e-01 1.000e+00]
 [2.997e-01 4.227e-01 2.832e-02 5.000e+00]
 [1.243e-01 6.706e-01 6.472e-01 7.000e+00]
 [6.154e-01 3.837e-01 9.972e-01 8.000e+00]
 [9.808e-01 6.855e-01 6.505e-01 5.000e+00]]

Train X|Y:
 [[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.]
 [0. 0. 0. 0.]]

Test  X|Y:
 [[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.]
 [0. 0. 0. 0.]]
```

In [ ]: