# La librería *NumPy*

*NumPy* es una librería bastante popular en el ámbito de la computación científica, que nos permite trabajar con datos en forma de vector o tabla n-dimensional (llamada *ndarray*). Dispone de funciones adicionales que nos van a permitir realizar operaciones matemáticas, lógicas, de selección o de ordenación sobre ese conjunto de datos, entre otras cosas. Podemos consultar más información que la que aquí mostraremos en la web oficial de la librería.

De hecho, otras librerías como *Pandas*, y paquetes algo más avanzados como Keras para gestión de redes neuronales, hacen uso de esta librería, con lo que conviene conocerla y saber qué cosas permite hacer, para poderla emplear también en otros ámbitos.

## 1. Instalación e importación

Para empezar, necesitamos instalar el módulo `numpy` en el sistema utilizando la herramienta `pip`:

```
pip install numpy
```

Ahora ya podremos emplear la librería en nuestros programas, con una instrucción como esta (el alias dado con *as* es opcional, pero muy habitual):

```
import numpy as np
```

## 2. Algunas operaciones elementales

Como decíamos antes, *NumPy* dispone de mecanismos para crear arrays o tablas unidimensionales o n-dimensionales, y darles unos valores iniciales fijos o aleatorios, para posteriormente poderlos modificar y realizar operaciones con ellos. También podemos consultar algunas propiedades básicas de los arrays, como su forma (tamaño de cada dimensión), tamaño (número total de elementos), etc. Veremos aquí algunas de las opciones básicas más habituales en la gestión de arrays con *NumPy*.

### 2.1. Creación de arrays

El método `array` permite crear un array (unidimensional o n-dimensional) con los datos que indiquemos entre paréntesis. Estos datos pueden venir en forma de lista o de tupla. Podemos indicar un segundo parámetro `dtype` que indica el tipo de datos. Algunos de los más habituales son `np.float32` o `np.int32`.

```
vector = np.array([1, 2, 3, 4, 5])
vector2 = np.array((2, 4, 6, 8))
vector3 = np.array([1, 2, 3], dtype = np.float32)
tabla = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Los métodos `zeros` y `ones` permiten inicializar un vector o tabla con ceros o unos, respectivamente. Hay que tener en cuenta que estos métodos devuelven un array de números *reales*. Si queremos que sean enteros, podemos emplear adicionalmente el método `astype` para indicar el tipo final:

```
ceros = np.zeros(2)                       # [0. 0.]
unos = np.ones(3)                         # [1. 1. 1.]
unosEnteros = np.ones(3).astype(np.int32) # [1 1 1]
unos2 = np.ones((2, 3))                     # [[1. 1. 1.][1. 1. 1.]]
```

El método `full` se emplea para rellenar un array con un valor predeterminado. Puede ser un array unidimensional con un valor, o un array n-dimensional con valores diferentes para cada columna.

```
cuatros = np.full(6, 4)          # [4 4 4 4 4 4]
tabla = np.full((3, 2), 4)       # [[4 4][4 4][4 4]]
tabla2 = np.full((3,2), [3,4])   # [[3 4][3 4][3 4]]
```

El método `arange` inicializa el vector o tabla con datos consecutivos entre dos límites dados. Opcionalmente se le puede dar un tercer parámetro que sería el incremento entre cada par de datos adyacentes.

```
secuencia1 = np.arange(4)          # [0 1 2 3]
secuencia2 = np.arange(1, 5)       # [1 2 3 4]
secuencia3 = np.arange(0, 10, 2)   # [0 2 4 6 8]
```

El método `linspace` construye un array donde sus elementos están uniformemente distribuidos entre un valor mínimo y un valor máximo, con una cantidad de datos determinada

```
# 5 datos distribuidos del 0 al 20
secuencia = np.linspace(0, 20, 5)   # [0. 5. 10. 15. 20.]
```

## 2.2. Tipos de datos en *NumPy*

*NumPy* dispone de numerosos tipos de datos, con distintos tamaños internamente y, además, se pueden definir en el código de múltiples formas. Aquí podéis consultar esos tipos de datos desde la documentación oficial, aunque aquí no seremos tan exhaustivos. Simplemente debéis tener en cuenta que:

- Para cada tipo de dato existen diferentes tamaños de memoria en bytes. Así, por ejemplo, para el tipo entero existe un entero de 32 bits (4 bytes, `np.int32`) y otro de 64 (8 bytes, `np.int64`). Lo mismo ocurre para el *float*.
- Los tipos de datos se pueden definir con literales que los referencian, como `np.int32`, `np.float64` ... o también con una cadena de texto que indica, en su primer carácter, el tipo de dato, y en el resto de caracteres, el número de bytes que se necesitan. Así, por ejemplo, `i4` representaría a un entero de 4 bytes, y `f8` a un real de 8 bytes.

Esto permite definir este mismo array de formas distintas:

```
v = np.array([1, 2, 3], dtype = np.float64)
v = np.array([1, 2, 3], dtype = 'f8')
v = np.array([1, 2, 3]).astype(np.float64)
v = np.array([1, 2, 3]).astype('f8')
```

## 2.3. Acceso y recorrido de arrays

Veremos a continuación que *NumPy* permite tanto acceder a casillas concretas de un array o tabla, como también seleccionar un rango de celdas con una sintaxis bastante cómoda.

### 2.3.1. Acceder a casillas concretas

Podemos acceder a cada casilla del array con su posición entre corchetes, como ocurre con las listas en Python. En el caso de más de una dimensión, ponemos un índice por dimensión, separados por comas.

```
vector = np.array([1, 2, 3, 4, 5])
tabla = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

print(vector[0])    # 1
print(tabla[1, 2])  # 7
```

También podemos acceder a un conjunto de casillas usando una característica llamada *fancy indexing*, en la que le pasamos una lista de índices:

```
vector = np.array([1, 2, 3, 4, 5])
seleccionados = vector[[1, 3]]
# seleccionados = [2 4]
```

### 2.3.2. Selección de rangos de celdas

Podemos seleccionar toda una fila o toda una columna, poniendo dos puntos `:` en la dimensión que no queremos seleccionar

```
tabla = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(tabla[1, :])  # [5 6 7 8]
print(tabla[:, 2])  # [3 7 11]
```

Adicionalmente, también podemos seleccionar un rango de casillas de un vector o tabla. Para ello usamos el símbolo `:` para especificar el rango. Finalmente, podemos emplear índices negativos para referenciar posiciones desde el final.

```
vector = np.array([1, 2, 3, 4, 5])
tabla = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

subvector1 = vector[1:]   # [2 3 4 5]
subvector2 = vector[1:3]  # [2 3]
subvector3 = vector[:4]   # [1 2 3 4]
subvector4 = vector[:-1]  # [1 2 3 4]

subtabla = tabla[1:3, 2:] # [[7 8] [11 12]]
```

### 2.3.3. Recorrido de arrays

A la hora de recorrer los datos de un array podemos emplear bucles `for`: un bucle para arrays unidimensionales, y dos bucles anidados para arrays bidimensionales.

```
# Recorrido unidimensional
for numero in vector:
    print(numero)

# Recorrido bidimensional
for fila in tabla:
    for dato in fila:
        print(dato)
```

## 2.4. Atributos de los arrays

Una vez hemos creado un array, podemos acceder a una serie de atributos del mismo, como son:

- `ndim`: indica el número de dimensiones del array

- `shape` : indica la longitud de cada una de las dimensiones. Devuelve una tupla con la longitud de cada dimensión.
- `size` : indica el número total de elementos del array
- `dtype` : tipo de dato de los elementos del array
- `itemsize` : tamaño en bytes de cada elemento del array
- `T` : obtiene la transposición del array actual (intercambiar filas por columnas)

Veamos un ejemplo de uso de cada atributo:

```
import numpy as np
vector = np.array([1, 2, 3, 4])
datos = np.array([[1, 2, 3],[4, 5, 6]])
print(vector.ndim)     # 1
print(datos.ndim)      # 2
print(vector.shape)    # (4,)
print(datos.shape)     # (2, 3) (2 filas, 3 columnas)
print(datos.size)      # 6 (elementos)
print(datos.dtype)     # int32
print(datos.itemsize)  # 4 (bytes)
print(datos.T)         # [[1 4][2 5][3 6]]
```

> **Ejercicio 1:**
>
> Crea un programa en Python llamado **ArrayCeros.py** que defina un array unidimensional de 10 ceros (enteros) en *NumPy*. Después, cambia el tercer cero por un 1. Imprime el resultado final.

> **Ejercicio 2:**
>
> Crea un programa en Python llamado **ArraySecuencia.py** que defina un array unidimensional con los números del 10 al 50 (inclusive). Imprime el resultado final.

# 3. Algunas operaciones complejas

Veamos ahora algunas operaciones que podemos hacer para transformar arrays previamente creados.

## 3.1. Concatenación de arrays

La operación `concatenate` permite unir dos o más arrays en uno solo. Puede tener un parámetro `axis` que indica el sentido de la concatenación: 0 para concatenación vertical (añadir más filas), 1 para concatenación horizontal (añadir más columnas) o *None* para linealizar la concatenación (ponerlo todo secuencial en un vector).

```
array1 = np.array([[1, 2],[3, 4]])
array2 = np.array([[5, 6]])
resultado = np.concatenate((array1, array2), axis = 0)
# resultado = [[1 2][3 4][5 6]]
resultado2 = np.concatenate((array1, array2), axis = 1)
# resultado2 = [[1 2 5][3 4 6]]
resultado3 = np.concatenate((array1, array2), axis = None)
# resultado3 = [1 2 3 4 5 6]
```

También podemos hacer uso de las operaciones `hstack` y `vstack` para apilar horizontalmente o verticalmente dos arrays (siempre que sean compatibles en tamaño). Debemos pasar los arrays a apilar en forma de tupla.

```
array1 = np.array([[1, 2],[3, 4]])
array2 = np.array([[5, 6],[7, 8]])

array_vert = np.vstack((array1, array2))
# [[1, 2],
#  [3, 4],
#  [5, 6],
#  [7, 8]]

array_horiz = np.hstack((array1, array2))
# [[1, 2, 5, 6],
#  [3, 4, 7, 8]]
```

## 3.2. Redimensionado

La operación `reshape` redimensiona el array. Puede ser útil para convertir un array unidimensional en bidimensional con filas y columnas, o al revés.

```
array1 = np.array([1, 2, 3, 4, 5, 6, 7, 8])
array2 = np.reshape(array1, (2, 4))
# También array2 = array1.reshape((2, 4))
# array2 = [[1 2 3 4][5 6 7 8]] (2 filas, 4 columnas)
```

Esta operación es bastante utilizada en redes neuronales, para adaptar las dimensiones de los datos de entrada a lo que espera la red en cuestión. Así, por ejemplo, una imagen en escala de grises de tamaño M x N (es decir, M filas y N columnas con valores de gris de 0 a 255) podemos redimensionarla en un vector unidimensional de tamaño M x N casillas, con esos valores. También en algunos casos nos puede venir bien añadir una dimensión adicional, para especificar, por ejemplo, el número de canales de la imagen (3 para imágenes a color), o simplemente para añadir una dimensión más a un array ya existente.

A la inversa, podemos emplear la operación `flatten` para convertir cualquier array N-dimensional en un vector unidimensional donde los elementos se disponen consecutivamente:

```
array = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
vector = array.flatten()
# vector = [1 2 3 4 5 6 7 8]
```

> **Ejercicio 3**:
>
> Crea un programa llamado **MatrizSecuencia.py** que defina una matriz de 3 filas y 3 columnas con los números del 0 al 8. Imprime el resultado por pantalla.

## 3.3. Operaciones estadísticas

Disponemos también de funciones estadísticas como `sum`, `max`, `min`, `mean`, `std`, `median`... que calculan en este caso la suma, el máximo, mínimo, media, desviación típica y mediana del array correspondiente, respectivamente. De forma alternativa, también podemos usar las funciones `np.sum`, `np.min`, `np.max`, etc.

```
array1 = np.array([1, 2, 3, 4, 5, 6, 7, 8])
print(array1.sum())   # 36
print(array1.min())   # 1
print(np.min(array1)) # 1
```

> **Ejercicio 4:**
>
> Crea un programa en Python llamado **NotasAlumnos.py** que le pida al usuario las 4 notas de 3 alumnos diferentes y las almacene en un array *NumPy* (3 filas, una para cada alumno). Después, deberá calcular la media de cada alumno y añadir esas medias como una columna al final del array, mostrando el array resultado.
>
> **AYUDA:** vídeo con la solución del ejercicio

## 3.4. Operaciones aritméticas escalares

Podemos hacer operaciones aritméticas sobre todos los elementos de un array, sumando/restando/multiplicando/dividiendo/elevando a una potencia todos los datos por un número, y hacer operaciones aritméticas entre arrays, llamadas *wise operations*, donde se opera cada casilla de un array con su correspondiente del otro array. Notar que estas operaciones NO se permiten entre listas convencionales en Python, o se obtienen resultados diferentes. Por ejemplo, en el caso de la multiplicación por un escalar, repetiría la lista entera, y en el caso de la suma de dos listas, las concatenaría. Es una de las ventajas que ofrece trabajar con *NumPy* en estos casos. Notar también que la multiplicación de dos arrays

*NumPy* con el símbolo `*` NO es una multiplicación algebraica de arrays, sino que multiplica cada elemento de un array por su correspondiente del otro.

```
datos = np.array([1, 2, 3, 4])
datos2 = datos + 1
# datos2 = [2 3 4 5]
datos3 = datos ** 2
# datos3 = [1 4 9 16]
resta = datos2 - datos
# resta = [1 1 1 1]
```

## 3.5. Operaciones aritméticas matriciales

Si queremos multiplicar algebraicamente dos arrays, debemos usar la instrucción `np.matmul` o bien la instrucción `np.dot`, indicando los dos arrays a multiplicar, que deben ser compatibles entre sí. Es decir, el número de columnas del primero debe coincidir con el número de filas del segundo.

```
mat1 = np.array([[1, 2, 3],[4, 5, 6]])  # Matriz 2 x 3
mat2 = np.array([[1, 2],[3, 4],[5, 6]]) # Matriz 3 x 2
resultado = np.dot(mat1, mat2)          # Obtenemos una matriz 2 x 2
```

> **NOTA**: en principio, ambas operaciones son equivalentes (*matmul* y *dot*) aunque presentan alguna diferencia en la forma de presentar los datos para matrices de más de 2 dimensiones. Pero no entraremos a detallar estas diferencias en estos apuntes.

## 3.6. Ordenaciones

Las ordenaciones de arrays se hacen con la instruccion `sort`. Esta instrucción ordena de menor a mayor. Para ordenar en orden inverso podemos invertir el signo del array, y también del resultado final. Aquí vemos un ejemplo:

```
datos = np.array([3, 5, 2, 1, 4])
datosAsc = np.sort(datos)           # [1 2 3 4 5]
datosNeg = -datos                   # [-1 -2 -3 -4 -5]
datosNegSort = np.sort(-datos)      # [-5 -4 -3 -2 -1]
datosDesc = -np.sort(-datos)        # [5 4 3 2 1]
```

## 3.7. Filtrado

Podemos también filtrar los datos que nos interesen de un array, poniendo la condición entre los corchetes

```
datos = np.array([3, 5, 2, 1, 4])
datosPares = datos[datos % 2 == 0]  # [2, 4]
```

El filtrado también puede resultar útil, por ejemplo, para asignar un valor a todos los elementos de un array que cumplan una condición. Por ejemplo, así dejamos a 0 los números de un array que sean negativos:

```
datos = np.array([1, 3, -3, 4, -7, 9])
negativos = datos < 0
# negativos = [False, False, True, False, True, False]
datos[negativos] = 0
# datos = [1, 3, 0, 4, 0, 9]
```

> **Ejercicio 5**:
>
> Crea un programa llamado **NumerosNoNulos.py** que le pida al usuario una secuencia de números separada por espacios, y cree con ella un array unidimensional de *NumPy*. Después, deberá quedarse con los números que no sean ceros, y mostrar el array resultado por pantalla. Deberá también indicar el valor mayor y menor introducido por el usuario (sin contar los ceros previamente filtrados).

> **Ejercicio 6**:
>
> Crea un programa llamado **NotasNumPy.py** que le pida al usuario una secuencia de notas numéricas, separadas por espacios. Con ellas deberá:
>
> - Crear un array unidimensional con NumPy
> - Ordenar las notas de mayor a menor
> - Crear un segundo array con las notas aprobadas
> - Calcular la media de las notas aprobadas
>
> **AYUDA:** vídeo con la solución del ejercicio

# 4. Números aleatorios con *NumPy*

Podemos generar fácilmente números aleatorios a través de *NumPy* mediante su objeto `random`. Esto permitirá generar tanto datos aleatorios simples (un número real/entero al azar en un rango dado) como arrays aleatorios.

Si queremos generar un solo dato aleatorio entero/real podemos emplear el método `randint` o `rand`. En el primer caso le indicaremos como parámetros el tope máximo para el número a generar (exclusive) o bien el rango mínimo (inclusive) a máximo (exclusive). En el segundo caso, genera un número real aleatorio entre 0 y 1. Adicionalmente, el método `uniform` permite generar una secuencia aleatoria entre unos límites dados.

```
import numpy as np

# Entero de 0 a 100 (sin contar el 100)
entero = np.random.randint(100)
# Entero de 1 a 10 (sin contar el 10)
entero2 = np.random.randint(1, 10)
# Real de 0 a 1
real = np.random.rand()
# Real de 0.2 a 0.7
real2 = np.random.uniform(low = 0.2, high = 0.7, size = 1)
```

También podemos aprovechar estas mismas instrucciones para crear vectores o arrays aleatorios:

```
# Vector unidimensional de 5 enteros de 0 a 100
vector = np.random.randint(100, size=5)
# Vector de 10 enteros del 1 al 10 (sin contar el 10)
vector2 = np.random.randint(1, 10, size=10)
# Matriz de 3 x 5 de enteros de 0 a 100
matriz = np.random.randint(100, size=(3, 5))
# Vector unidimensional de 5 reales de 0 a 1
vector3 = np.random.rand(5)
# Matriz de 3 x 5 reales de 0 a 1
matriz2 = np.random.rand(3, 5)
# Vector de 5 reales entre 0.2 y 0.7
vector4 = np.random.uniform(low = 0.2, high = 0.7, size = 5)
```

En el caso de que queramos asegurarnos de que siempre se genera la misma secuencia de números aleatorios (algo útil cuando estamos probando el funcionamiento de un determinado programa, por ejemplo), podemos emplear la instrucción `seed` indicando el número "semilla" para la generación de la secuencia:

```
np.random.seed(123)
# ... Generar aquí los números aleatorios
```

# 5. Arrays de *NumPy* o listas de Python

Podríamos pensar que gran parte de las cosas que podemos hacer con *NumPy* las podemos hacer también con las listas de Python, pero existen ciertas diferencias importantes por las que es más conveniente usar *NumPy* en algunos casos:

- Las listas de Python son heterogéneas (podemos almacenar en cada casilla datos de distintos tipos), mientras que los arrays de *NumPy* son homogéneos, lo que permite gestionar la información de forma más eficiente

- Las listas tienen un tamaño variable. Podemos añadir o quitar elementos en cualquier momento. El tamaño de los arrays en *NumPy* es fijo: una vez creamos el array, el tamaño no varía (sólo podemos alterar el contenido). Esto hace que el intérprete de Python no tenga que preocuparse de controlar tamaños y elementos, y que la gestión de la información ocupe menos memoria y sea más eficiente.
- Además, en matrices bidimensionales de *NumPy*, por ejemplo, cada fila debe tener el mismo número de columnas que el resto. No ocurre lo mismo con listas bidimensionales de Python que, en realidad, es una lista que contiene otras (y cada una puede tener un tamaño propio y distinto al resto).