El paquete Numpy

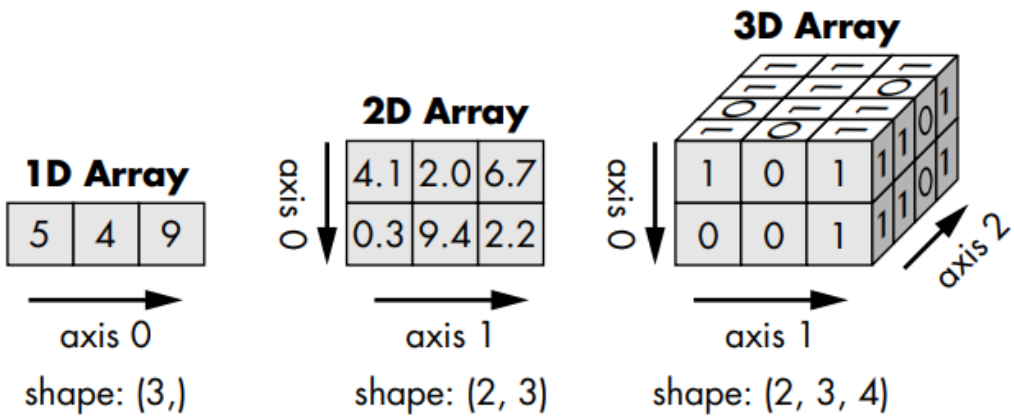
Características:

- **Arrays**
- **Estadística**
- **Eficiente (x50 vs listas)**

NumPy es una librería de Python especializada en el cálculo numérico y el análisis de datos, especialmente para un gran volumen de datos.

Incorpora una nueva clase de objetos llamados arrays que permite representar colecciones de datos de un mismo tipo en varias dimensiones, y funciones muy eficientes para su manipulación.

La ventaja de Numpy frente a las listas predefinidas en Python es que el procesamiento de los arrays se realiza mucho más rápido (hasta 50 veces más) que las listas, lo cual la hace ideal para el procesamiento de vectores y matrices de grandes dimensiones.



Instalar paquetes

Numpy viene instalado por defecto, pero si queremos instalar un nuevo paquete, debemos hacerlo desde el terminal (powershell en windows) con la siguiente instrucción:

pip install numpy

Alternativamente, los "comandos mágicos" de Jupyter Notebook lo permiten:

In [ ]:

```
! pip install numpy
```

Vamos a importar el paquete y crear un array:

In [ ]:

```
import numpy

miarray = numpy.array([2,4,5,6,1])

miarray
```

Y podemos experimentar usando lo que ya sabíamos con listas

In [ ]:

```
miarray[0]
```

In [ ]:

```
miarray[0:4:2]
```

Aunque lo más estándar es importar numpy de la siguiente forma:

In [ ]:

```
import numpy as np


miarray = np.array([2,4,5,6,1])

miarray
```

Dimensiones de un Array

Los arrays pueden tener varias dimensiones (o ninguna), veamos algunos ejemplos

In [ ]:

```
np.array(3)
```

In [ ]:

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

In [ ]:

```
np.array([[3,2,1],[6,7,9]])
```

In [ ]:

```
np.array([[3,2,1],[6,7,9],[1,0,1]])
```

Pero no nos dejemos engañar... el ejemplo anterior tiene 2D!!!

Para obtener 3 dimensiones, debemos hacer lo siguiente:

In [ ]:

```
np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
```

Para comprobar las dimensiones...

In [ ]:

```
miarray = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

miarray.ndim
```

O para definirlas, aunque puede ser un poco confuso al principio

In [ ]:

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

In [ ]:

```
miarray.ndim
```

Coordenadas internas de los arrays

In [14]:

```
arr = np.array([[3,2,1,2],[6,7,9,5]])
```

In [ ]:

```
arr[1,3] # Filas y columnas
```

In [ ]:

```
arr[:,3]
```

In [ ]:

```
arr[1,:]
```

Y con 3 o más dimensiones...

In [ ]:

```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]], [[12, 14, 15],[16, 17, 18]], [[19, 2
0, 21],[22, 23, 24]] ])

arr[0, 1, 2]
```

In [ ]:

```
arr[1,:,:]
```

In [ ]:

```
arr[:,1,:]
```

In [ ]:

```
arr[:,:,1]
```

¡Haz las pruebes que consideres necesarias para convencerte!

In [ ]:

```

```

¿Admite dimensiones de distintos tamaños?

¿Y distintos tamaños dentro de una dimensión?

In [ ]:

```

```

Y para casi terminar... como describir las dimensiones de un array!

In [ ]:

```
arr.shape
```

In [ ]:

```

```

Algo que no podíamos hacer con una lista que sí que podemos hacer con un array es crear objetos vacíos!!!

In [ ]:

```
arr  = np.zeros(10)
arr
```

Pero aún mejor...

In [ ]:

```
arr  = np.empty((3,4))
arr
```

Otras funciones útiles que permiten generar arrays son:

- **np.empty(dimensiones)**: Crea y devuelve una referencia a un array vacío con las dimensiones especificadas en la tupla dimensiones.
- **np.zeros(dimensiones)**: Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones. Rellena el array con ceros.

- **np.zeros(dimensiones)**: Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son todos ceros.
- **np.ones(dimensiones)**: Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son todos unos.
- **np.full(dimensiones, valor)**: Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son todos valor.
- **np.identity(n)**: Crea y devuelve una referencia a la matriz identidad de dimensión n.
- **np.arange(inicio, fin, salto)**: Crea y devuelve una referencia a un array de una dimensión cuyos elementos son la secuencia desde inicio hasta fin tomando valores cada salto.
- **np.linspace(inicio, fin, n)**: Crea y devuelve una referencia a un array de una dimensión cuyos elementos son la secuencia de n valores equidistantes desde inicio hasta fin.
- **np.random.random(dimensiones)**: Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son aleatorios.

In [ ]:

```
```

Atributos de un array

Existen varios atributos y funciones que describen las características de un array.

- **a.ndim**: Devuelve el número de dimensiones del array a.
- **a.shape**: Devuelve una tupla con las dimensiones del array a.
- **a.size**: Devuelve el número de elementos del array a.
- **a.dtype**: Devuelve el tipo de datos de los elementos del array a.

In [ ]:

```
```

Tipos de datos en Numpy

- **i - integer**
- **b - boolean**
- **u - unsigned integer**
- **f - float**
- **c - complex float**
- **m - timedelta**
- **M - datetime**
- **O - object**
- **S - string**
- **U - unicode string**
- **V - fixed chunk of memory for other type (void)**

In [ ]:

```
arr = np.array(['platano', 'pera', 'kiwi'])

arr.dtype
```

Y convertirlo es parecido a las funciones de python básico:

In [ ]:

```
arr = np.array([1.1, 7.1, 3.9])

arr.astype('i')
```

Cuando tenemos un array de tipo numérico (o textual), también vamos a poder reordenarlo usando la función sort, que funciona de un modo particular para arrays de más de una dimensión

In [ ]:

```
arr = np.array([6, 8, 3, 1, -2])

print(np.sort(arr))
```

In [ ]:

```
arr = np.array([[4, 3, 2, 0, 1],[6, 8, 3, 1, -2]])

print(np.sort(arr))
```

¡Tenemos que ir con cuidado, ya que si los datos estaban "apareados", acabamos de romperlos!

Funciones estadísticas sencillas

Numpy también tiene algunas (muchas) funciones estadísticas y de procesamiento implementadas. Comentalas en función de su uso

In [ ]:

```
arr = np.array([2,4,5,0,6,1,5,4,1,-3])

print(np.min(arr))
print(np.max(arr))
print(np.mean(arr))
print(np.median(arr))
print(np.std(arr))
print(np.absolute(arr))
print(np.cumsum(arr))
print(np.unique(arr)) # Mención especial
print(np.exp(arr))
print(np.prod(arr))
```

Esto es solamente una pequeña muestra de todo lo que se puede hacer con Numpy. Si usas el tabulador comprobar la cantidad de opciones que ofrece

In [ ]:

```

```

Bucles en Arrays

Primero vemos un bucle por valores

In [ ]:

```
arr = np.array([5, 3, 1, 2, 3])

for x in arr:
    print(x)
```

Y por posiciones...

In [ ]:

```
for i in range(len(arr)):
    print(arr[i])
```

Pero las iteraciones se complican cuando empezamos a tener más de una dimensión!

In [ ]:

```
arr = np.array([[3,2,1],[6,7,9]])

for x in arr:
    print(x)
```

In [ ]:

```
arr = np.array([[3,2,1],[6,7,9]])

for x in arr:
    for y in x:
        print(y)
```

Define un array de 3 dimensiones y crea una estructura iterativa que te muestre los valores uno a uno

In [ ]:

Como no podía ser de otra manera, hay un modo sencillo de obtener los elementos por separado

In [ ]:

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

for x in np.nditer(arr):
    print(x)
```

In [ ]:

```
np.nditer(arr)
```

Si queremos iterar de un modo un poco más complejo podemos usar estructuras del tipo "enumerate", que permiten iterar simultaneamente por valor y posición. No son exclusivas de numpy, así que las veremos en otros tipos de objeto.

In [ ]:

```
arr = np.array([[1, 3, 3, 2], [11, 5, 7, 3]])

for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

Los objetos de tipos "enumerate" pueden procesarse por sí mismos, lo que permite ser mucho más flexible a la hora de definir qué queremos hacer en cada caso

In [ ]:

```
arr[idx]
```

In [ ]:

```
idx[0]
```

De manera muy parecida, podemos encontrar posiciones y elementos usando el método where(), que podremos usar para filtrar

In [ ]:

```
arr = np.array([[1, 3, 3, 2], [11, 5, 7, 3]])

np.where(arr == 11)
```

In [ ]:

```
np.where(arr > 5)
```

Filtrando arrays

Una manera directa y sencilla de aplicar lo que estamos viendo para seleccionar elementos concretos de un array es definir filtros dentro de [ ]

In [ ]:

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

filtro = arr > 3

arr[filtro]
```

In [ ]:

```
filtro
```

Uniendo arrays

Lo que veremos a continuación es como juntar arrays, y evidentemente vamos a tener que específicar cómo queremos hacerlo, ya que estos tienen distintas dimensiones.

Un primer ejemplo sencillo utiliza la instrucción concatenate

In [ ]:

```
arr1 = np.array([5, 2, 1])
arr2 = np.array([0, 5, 1])

arr = np.concatenate((arr1, arr2))

arr
```

Esencialmente estamos haciendo lo mismo que con extend() al usar una lista.

¿Qué pasa cuando tenemos más de una dimensión?

In [ ]:

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2))
print(arr)
```

Podemos ejercer mucho más control con el parámetro axis! Por defecto, 0 es filas, 1 es columnas, etc.

In [ ]:

```
np.concatenate((arr1, arr2),axis = 0)
```

In [ ]:

```
np.concatenate((arr1, arr2),axis = 1)
```

Las funciones de tipo "stack" son parecidas, pero no idénticas al concatenate... ¿Cuál es la diferencia en el siguiente ejemplo?

In [ ]:

```
arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

print("Concatenate")
print(np.concatenate((arr1, arr2)))

print("Stack")
print(np.stack((arr1, arr2), axis=0))
```

Prueba de cambiar el "axis", a ver cómo reacciona

In [ ]:

```

```

Partiendo arrays

La operación opuesta a juntar arrays se llama "split", y funciona de una manera un poco particular, ya que le especificaremos cuántos trozos queremos

In [ ]:

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

splitarray = np.array_split(arr, 3)
splitarray
```

In [ ]:

```
for x in splitarray:
    print(x)
```

Un ejemplo un poco más complejo es el siguiente, donde podemos controlar el tipo de cortes que realizamos

In [ ]:

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])
splitarray = np.array_split(arr, 3, axis=0)

print(splitarray)
```

```
for x in splitarray:
    print(x)
```

In [ ]:

```
splitarray = np.array_split(arr, 3, axis=1)

for x in splitarray:
    print(x)
```

Datos faltantes

A la hora de trabajar con datos, es muy habitual encontrar "celdas vacías". Es lo que en python (y numpy) se denominan NaN.

Para realizar los ejemplos vamos a forzar la presencia de estos casos, pero en el mundo real pasa de manera natural

In [ ]:

```
arr = np.array([1, 2, np.nan, 7, 4, 0])

sum(arr)
```

Para eliminarlos, podemos hacer varias cosas, algunas más elegantes que otras

In [ ]:

```
lista = []
for x in arr:
    if not np.isnan(x):
        lista.append(x)
np.array(lista)
```

In [ ]:

```
np.array([x for x in arr if not np.isnan(x)])
```

In [ ]:

```
arr[np.logical_not(np.isnan(arr))]
```

In [ ]:

```
arr[np.isfinite(arr)]
```

Copy vs. View

¿Cuales son las diferencias entre crear una copia y crear una vista de un array?

Experimenta un poco para ver los detalles y qué pasa cuando alteramos uno de los dos

In [ ]:

```
arr = np.array([1, 2, 3, 4, 5])
copia = arr.copy()
vista = arr.view()

arr[0] = 42


print("Original",arr)
print("Copia",copia)
print("Vista",vista)
```

¿Y si alteramos la vista? ¿Y la copia?

In [ ]:

```
vista2 = arr.view(dtype=np.int16)

print("Original",arr.dtype, arr)
print("Vista2",vista2.dtype, vista2)
```

Transformando Arrays

Una manera de dar forma a arrays (y a listas), es utilizando el método reshape, que permite definir sus dimensiones a posteriori

In [ ]:

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

arr = arr.reshape(4, 3)
arr
```

In [ ]:

```
arr = arr.reshape(3, 2, 2)
arr
```

Operaciones matemáticas con arrays

Existen dos formas de realizar operaciones matemáticas con arrays: *a nivel de elemento* y *a nivel de array*.

- **Las operaciones a nivel de elemento operan los elementos que ocupan la misma posición en dos arrays.**
  - **Se necesitan, por tanto, dos arrays con las mismas dimensiones y el resultado es una array de la misma dimensión.**
- **Los operadores mamemáticos +, -, *, /, %, ** se utilizan para la realizar suma, resta, producto, cociente, resto y potencia a nivel de elemento.**

In [ ]:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[1, 1, 1], [2, 2, 2]])

print(a + b )
print(a / b)
print(a ** 2)
```

Álgebra matricial

Numpy incorpora funciones para realizar las principales operaciones algebraicas con vectores y matrices.

La mayoría de los métodos algebráicos se agrupan en el submódulo linalg.

Producto escalar de dos vectores

Para realizar el producto escalar de dos vectores se utiliza el operador @

o el método u.dot(v): Devuelve el producto escalar de los vectores u y v.

In [ ]:

```
a = np.array([1, 2, 3])
b = np.array([1, 0, 1])

print(a @ b)
print(a.dot(b))
```

Módulo de un vector

Para calcular el módulo de un vector se utiliza el método norm(v): Devuelve el módulo del vector v.

In [ ]:

```
a = np.array([3, 4])
print(np.linalg.norm(a))
```

Producto de dos matrices

Para realizar el producto matricial se utiliza el mismo operador @

y método que para el producto escalar de vectores a.dot(b): Devuelve el producto matricial de las matrices a y b siempre y cuando sus dimensiones sean compatibles.

In [ ]:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[1, 1], [2, 2], [3, 3]])

print(a @ b)
print(a.dot(b))
```

Matriz traspuesta

a.T: Devuelve la matriz traspuesta de la matriz a.

In [ ]:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a.T)
```

Traza de una matriz

a.trace(): Devuelve la traza (suma de la diagonal principal) de la matriz cuadrada a.

In [ ]:

```
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(a.trace())
```

Determinante de una matriz

det(a): Devuelve el determinante de la matriz cuadrada a.

In [ ]:

```
a = np.array([[1, 2], [3, 4]])
print(np.linalg.det(a))
```

Matriz inversa

inv(a): Devuelve la matriz inversa de la matriz cuadrada a.

In [ ]:

```
a = np.array([[1, 2], [3, 4]])
print(np.linalg.inv(a))
```

Autovalores de una matriz

eigvals(a): Devuelve los autovalores de la matriz cuadrada a.

In [ ]:

```
a = np.array([[1, 1, 0], [1, 2, 1], [0, 1, 1]])
print(np.linalg.eigvals(a))
```

Autovectores de una matriz

eig(a): Devuelve los autovalores y los autovectores asociados de la matriz cuadrada a.

In [ ]:

```
a = np.array([[1, 1, 0], [1, 2, 1], [0, 1, 1]])
print(np.linalg.eig(a))
```

Solución de un sistema de ecuaciones

solve(a, b): Devuelve la solución del sistema de ecuaciones lineales con los coeficientes de la matriz a y los términos independientes de la matriz b.

In [ ]:

```
a = np.array([[1, 2], [3, 5]])
b = np.array([1, 2])
```

```
print(np.linalg.solve(a, b))
```