

Prácticas de Aprendizaje Automático

Grupo 3

Clase 2: Introducción a Scikit-learn, Numpy y Matplotlib

Pablo Mesejo

Universidad de Granada

Departamento de Ciencias de la Computación e Inteligencia Artificial



UNIVERSIDAD
DE GRANADA



Índice

1. Retomando los fundamentos de Python
2. Arrays en Numpy. Funciones básicas
3. Indexado Numpy
4. Datos
 - a) Terminología
 - b) Generación de datos
 - c) Lectura de datos
5. Representación con Matplotlib

Retomando los fundamentos de Python

¿Alguien sabe qué hace esta función?

```
def someGreatFunction(arr):  
    if len(arr) <= 1:  
        return arr  
    pivot = arr[len(arr) // 2]  
    left = [x for x in arr if x < pivot]  
    middle = [x for x in arr if x == pivot]  
    right = [x for x in arr if x > pivot]  
    return someGreatFunction(left) + middle + someGreatFunction(right)  
  
print(someGreatFunction([3,6,8,10,1,2,1]))
```

Retomando los fundamentos de Python (y 2)

```
def QuickSort(arr):  
    if len(arr) <= 1:  
        return arr  
    pivot = arr[len(arr) // 2]  
    left = [x for x in arr if x < pivot]  
    middle = [x for x in arr if x == pivot]  
    right = [x for x in arr if x > pivot]  
    return QuickSort(left)+middle+QuickSort(right)  
  
print(QuickSort([3,6,8,10,1,2,1]))
```

```
pivot: 10  
left: [3, 6, 8, 1, 2, 1]  
middle: [10]  
right: []
```

```
-----  
pivot: 1  
left: []  
middle: [1, 1]  
right: [3, 6, 8, 2]
```

```
-----  
pivot: 8  
left: [3, 6, 2]  
middle: [8]  
right: []
```

```
-----  
pivot: 6  
left: [3, 2]  
middle: [6]  
right: []
```

```
-----  
pivot: 2  
left: []  
middle: [2]  
right: [3]
```

```
-----  
[1, 1, 2, 3, 6, 8, 10]
```

Ordenando las ideas

- **NumPy**

- Biblioteca que da soporte para **vectores y matrices**
- Incluye numerosas funciones para operar sobre dichos arrays



- **Matplotlib**

- Biblioteca que permite la **visualización** de funciones matemáticas



- **SciPy**

- Biblioteca para **cálculo científico y técnico**
- Más completo que NumPy (a nivel de funciones de álgebra lineal, p.ej.; e incluye módulos de integración, optimización, ecuaciones diferenciales, etc.)
- Utiliza NumPy



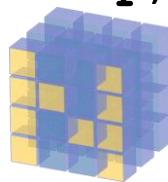
- **Scikit-learn**

- Biblioteca de **aprendizaje automático**
- Está construida sobre SciPy



Arrays en Numpy. Funciones básicas.

- Numpy es el paquete principal en python para manejar arrays de N dimensiones de forma eficiente
 - Proporciona herramientas para integrar código C/C++/Fortran
- Incluye funciones para realizar operaciones matriciales, álgebra lineal, transformaciones de Fourier, y generación de números aleatorios.
- Importamos el paquete con `import numpy` (se recomienda usar `import numpy as np` para poder llamarla con `np`).



Arrays en Numpy. Funciones básicas.

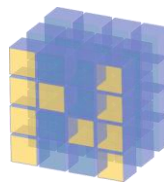
- ¿Por qué no usar simplemente listas de Python?
 - En términos generales, NumPy presenta
 - una mayor rapidez de ejecución
 - y un menor consumo de memoria
 - Muchas más funciones disponibles

```
In [1]: import numpy as np
...: x = np.array([3, 6, 9, 12])
...: x/3.0
Out[1]: array([1., 2., 3., 4.])
```

```
In [2]: y = [3, 6, 9, 12]
...: y/3.0
Traceback (most recent call last):
```

```
File "<ipython-input-2-d16e9e874e49>", line 2, in <module>
    y/3.0
```

```
TypeError: unsupported operand type(s) for /: 'list' and 'float'
```



NumPy

Crear Arrays en Numpy (1)

```
no_initialized = np.empty(<shape>, <type>)
```

```
zeros = np.zeros(<shape>, <type>)
```

```
ones = np.ones(<shape>, <type>)
```

- `<shape>` → Tupla con el tamaño por dimensión. Ejemplo: Matriz 5x5 → (5, 5).
- `<type>` → Tipo de numpy (np.int32, np.float32, bool,...).
- También podemos crear un array nuevo con la misma forma y tipo que otro usando `np.empty_like`, `np.zeros_like` o `np.ones_like`

```
array1 = np.ones((1,5),np.float)
```

```
array1  
array([[1., 1., 1., 1., 1.]])
```

```
new_array1 = np.zeros_like(array1)
```

```
new_array1  
array([[0., 0., 0., 0., 0.]])
```

Y, por supuesto, se puede inicializar con los valores que uno quiera: `x = np.array([2,3,1,0])`

Crear Arrays en Numpy (2)

Crear array aleatorio:

- Uniforme: `np.random.uniform(low=<min_val>, high=<max_val>, size=<shape>)`
 - Este array es de números reales entre <min_val> y <max_val>.
- Uniforme (enteros): `np.random.randint(low=<min_val>, high=<max_val>, size=<shape>)`
 - Este array es de números enteros entre <min_val> y <max_val>.

```
In [67]: print(np.random.uniform(-10, 10, (1,5)))  
[[-2.52124033 -7.22533183 -2.98541844  8.92440291 -3.91384372]]
```

```
In [68]: print(np.random.uniform(-10, 10, (1,5)))  
[[-9.17158651  4.04813184 -1.69356788  3.75857095 -3.21201073]]
```

```
In [69]: print(np.random.uniform(-10, 10, (1,5)))  
[[-0.97694229 -7.08639096 -9.07903603  0.10606073  7.36018195]]
```

Crear Arrays en Numpy (y 3)

Crear array en un rango determinado de valores:

- `np.arange([start,] stop[, step,], dtype=None)`

```
In [22]: np.arange(1,20,1)
```

```
Out[22]:
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,  
       18, 19])
```

```
In [23]: np.arange(20)
```

```
Out[23]:
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,  
       17, 18, 19])
```

Arrays en Numpy. Funciones básicas.

Obtener tamaño del array: **array.shape**

Obtener tipo array: **array.dtype**

Cambiar tipo array: **array.astype(<new_numpy_type>)**

Cambiar forma del array (tienen que mantenerse el mismo número de elementos):
array.reshape(<new_shape>). Se puede poner una dimensión como -1 (es decir, desconocida) → el nuevo tamaño se calculará en función del tamaño del resto de dimensiones y del número de elementos.

```
In [78]: array2 = np.zeros((4,), np.int)
```

```
In [79]: array2
```

```
Out[79]: array([0, 0, 0, 0])
```

```
In [80]: array2 = array2.reshape((-1,2))
```

```
In [81]: array2
```

```
Out[81]:  
array([[0, 0],  
       [0, 0]])
```

```
In [82]: array2.shape
```

```
Out[82]: (2, 2)
```

Arrays en Numpy. Funciones básicas.

Trasponer un array: `array.transpose()` o `array.T`

```
In [9]: a = np.array([(0.1, 1.), (0.1, 2.), (0.1, 3.), (0.1, 4.), (0.1, 5.)])  
...:
```

```
In [10]: a.transpose()
```

```
Out[10]:  
array([[0.1, 0.1, 0.1, 0.1, 0.1],  
       [1. , 2. , 3. , 4. , 5. ]])
```

```
In [11]: a
```

```
Out[11]:  
array([[0.1, 1. ],  
       [0.1, 2. ],  
       [0.1, 3. ],  
       [0.1, 4. ],  
       [0.1, 5. ]])
```

```
In [12]: a.T
```

```
Out[12]:  
array([[0.1, 0.1, 0.1, 0.1, 0.1],  
       [1. , 2. , 3. , 4. , 5. ]])
```

Arrays en Numpy. Funciones básicas.

- Diferenciando entre vectores fila y columna...

```
In [1]: import numpy as np
...: a = np.array([1, 2, 3])
...: a
```

```
Out[1]: array([1, 2, 3])
```

```
In [2]: a.transpose()
Out[2]: array([1, 2, 3])
```

```
In [3]: a.shape = (3,1)
```

```
In [4]: a
Out[4]:
array([[1],
       [2],
       [3]])
```

```
In [5]: a.transpose()
Out[5]: array([[1, 2, 3]])
```

Es necesario emplear una
dimensión extra para
diferenciar ambos casos

Otra opción:
usar **np.newaxis**

```
In [6]: a = np.array([1, 2, 3])
...: a
Out[6]: array([1, 2, 3])
```

```
In [7]: a[:, np.newaxis]
Out[7]:
array([[1],
       [2],
       [3]])
```

```
In [8]: a[np.newaxis, :]
Out[8]: array([[1, 2, 3]])
```

Arrays en Numpy. Estadísticas (1)

- Mínimo: `array.min(axis=<dim>)`
- Máximo: `array.max(axis=<dim>)`
- Índice del mínimo: `array.argmin(axis=<dim>)`
- Índice del máximo: `array.argmax(axis=<dim>)`
- Media: `array.mean(axis=<dim>)`
- Media ponderada: `np.average(array, axis=<dim>, weights=<pesos>)`
- Desviación estándar: `array.std(axis=<dim>)`
- Varianza: `array.var(axis=<dim>)`

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

```
x.mean()  
1.5
```

```
x.argmin()  
3
```

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

```
x.mean(0)  
array([1. , 1.5, 0.5, 0. ])
```

```
x.mean(1)  
array([1.5, 0. ])
```

Arrays en Numpy. Estadísticas (2)

- Mediana: `np.median(array, axis=<dim>)`
- Percentiles: `np.percentile(array, q <lista percentiles>, axis=<dim>)`
- Suma: `array.sum(axis=<dim>)`
- Multiplicación: `array.prod(axis=<dim>)`
- Algún elemento es verdad: `array.any(axis=<dim>)`
- Todos son verdad: `array.all(axis=<dim>)`

```
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

```
In [29]: X.sum()  
Out[29]: 45
```

<dim>: dimensión a lo largo de la cual se calcula. Por defecto, como si hubiese una sola dimensión.

En 2D: axis-0: recorre filas | axis-1: recorre columnas

Todas estas funciones, salvo **any** y **all**, tienen otra versión **np.nan<función>** que realiza dicha función tratando los valores NaN como zeros (o ignorándolos).

```
In [144]: x = np.array([10, np.nan, 10, np.nan, 10, np.nan, 10, np.nan, np.nan])
```

```
In [145]: np.nanmedian(x)  
Out[145]: 10.0
```

```
In [146]: np.median(x)  
C:\Users\Pablo.DESKTOP-UVM4BCE\Anaconda3\lib\site-packages\numpy\lib\function_base.py:3250: RuntimeWarning: Invalid value encountered in median  
  r = func(a, **kwargs)  
Out[146]: nan
```

Arrays en Numpy. Funciones básicas.

Permutar dimensiones: `array.swapaxes(<dim1>, <dim2>)`

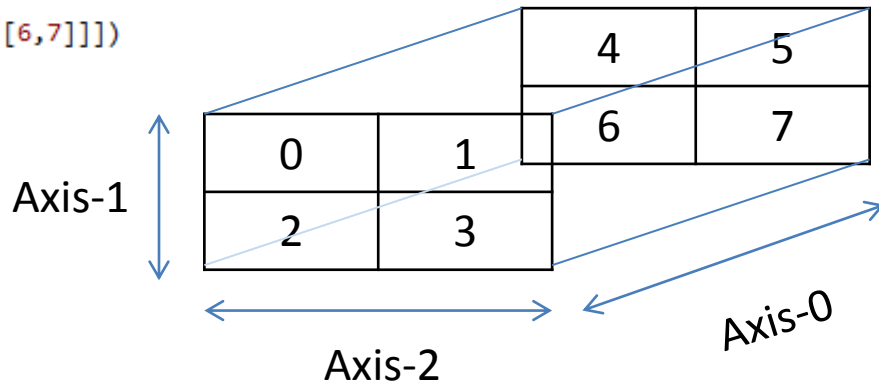
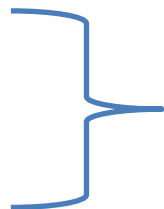
```
In [1]: import numpy as np
```

```
In [2]: x = np.array([[[0,1],[2,3]], [[4,5],[6,7]]])  
...: x
```

```
Out[2]:  
array([[[0, 1],  
        [2, 3]],  
       [[4, 5],  
        [6, 7]]])
```

```
In [3]: x.swapaxes(0,2)
```

```
Out[3]:  
array([[[0, 4],  
        [2, 6]],  
       [[1, 5],  
        [3, 7]]])
```



Arrays en Numpy. Funciones básicas.

Permutar dimensiones: `array.swapaxes(<dim1>, <dim2>)`

```
In [22]: a = np.arange(18)
```

```
In [23]: a.reshape(2,3,3)
```

```
Out[23]:
```

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

```
       [[ 9, 10, 11],  
        [12, 13, 14],  
        [15, 16, 17]])])
```

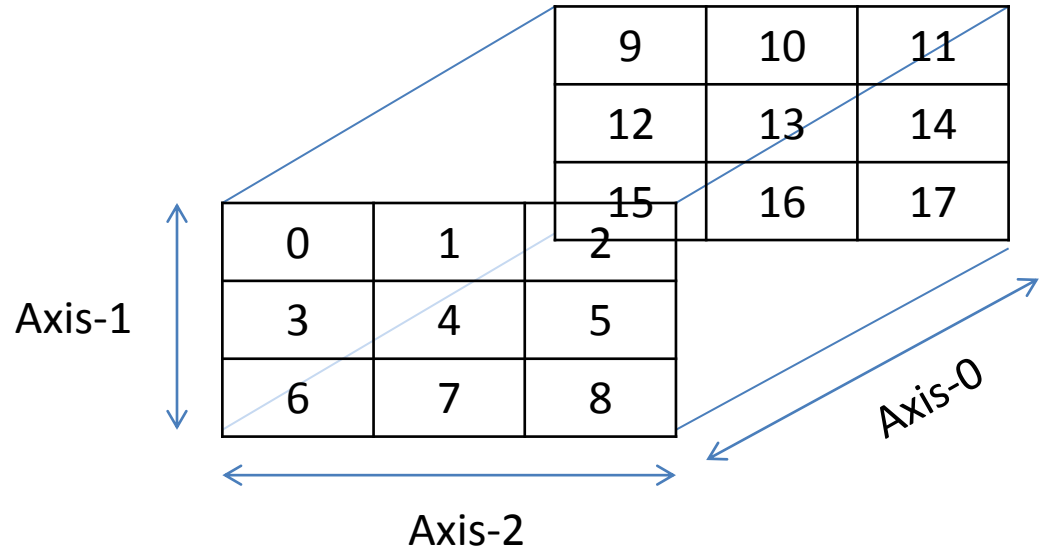
```
In [24]: a.reshape(2,3,3).swapaxes(0,2)
```

```
Out[24]:
```

```
array([[[ 0,  9],  
        [ 3, 12],  
        [ 6, 15]],
```

```
       [[ 1, 10],  
        [ 4, 13],  
        [ 7, 16]],
```

```
       [[ 2, 11],  
        [ 5, 14],  
        [ 8, 17]])])
```



Arrays en Numpy. Funciones básicas.

- Varias formas de hacer lo mismo:
 - `array.swapaxes(0,1)`
 - `array.T`
 - `array.transpose()`

```
In [36]: c
Out[36]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [37]: c.swapaxes(0,1)
Out[37]:
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

```
In [38]: c.T
Out[38]:
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

```
In [39]: c.transpose()
Out[39]:
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Arrays en Numpy. Axis.

```
In [2]: new_array = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

```
In [3]: new_array.shape
```

```
Out[3]: (3, 3)
```

Array con 3 filas y 3 columnas

```
In [4]: new_array
```

```
Out[4]:
```

```
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

```
In [5]: new_array.sum(axis=0)
```

```
Out[5]: array([12, 15, 18])
```

Sumamos los elementos de cada columna

```
In [6]: new_array.sum(axis=1)
```

```
Out[6]: array([ 6, 15, 24])
```

Sumamos los elementos de cada fila

```
In [7]: new_array = np.array([[[1,2,3],[4,5,6],[7,8,9]],[[10,10,10],[20,20,20],[30,30,30]]])
```

```
...: new_array.shape
```

```
Out[7]: (2, 3, 3)
```

Array tridimensional: 2 matrices de 3x3

```
In [8]: new_array
```

```
Out[8]:
```

```
array([[[ 1,  2,  3],  
        [ 4,  5,  6],  
        [ 7,  8,  9]],
```

```
       [[10, 10, 10],  
        [20, 20, 20],  
        [30, 30, 30]])
```

```
In [9]: new_array.sum(axis=0)
```

```
Out[9]:
```

```
array([[11, 12, 13],  
       [24, 25, 26],  
       [37, 38, 39]])
```

Sumamos con respecto a la 3era dimensión

```
In [10]: new_array.sum(axis=1)
```

```
Out[10]:
```

```
array([[12, 15, 18],  
       [60, 60, 60]])
```

Sumamos los elementos de cada columna

```
In [11]: new_array.sum(axis=2)
```

```
Out[11]:
```

```
array([[ 6, 15, 24],  
       [30, 60, 90]])
```

Sumamos los elementos de cada fila

2D

3D

Arrays en Numpy. Funciones básicas.

Copiar array: `array2 = array.copy()`

```
In [1]: import numpy as np
```

```
In [2]: x = np.array([1, 2, 3])
```

```
...: y = x
```

```
...: z = np.copy(x)
```

```
...: y[0] = 10
```

```
In [3]: print(x, y, z)
```

```
[10  2  3] [10  2  3] [1 2 3]
```

```
In [4]: x is y
```

```
Out[4]: True
```

```
In [5]: z is x
```

```
Out[5]: False
```

**Si no se usa esta función, array2
tendría una referencia**

Unlike some other languages, creating a new variable with an assignment statement in Python such as `x = some_numpy_array` does not make a copy of `some_numpy_array`.

Instead, the assignment statement makes `x` and `some_numpy_array` both point to the same `numpy` array in memory. Because `x` and `some_numpy_array` are both refer (or pointer) to the same `numpy` array in memory, the `numpy` array can be changed by operations on either `x` or `some_numpy_array`. If you aren't aware of this behavior then you may run into very difficult to identify bugs in your calculations!

Arrays en Numpy. Funciones básicas.

Ordenar array de menor a mayor: `array.sort(axis=<dim>)`

Índices que ordenan array de menor a mayor: `np.argsort(array, axis=<dim>)`

```
In [1]: import numpy as np
```

```
In [2]: a=np.array([5,3,7,8,1,2,3])
```

```
In [3]: a.sort()
```

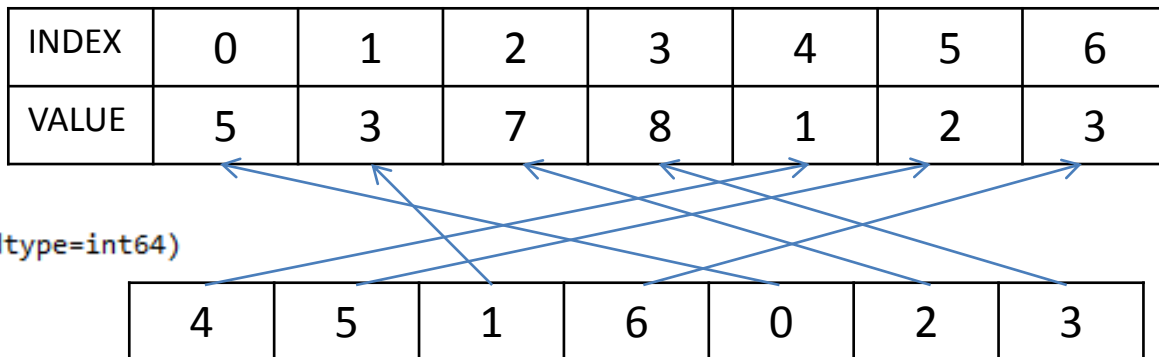
```
In [4]: a
```

```
Out[4]: array([1, 2, 3, 3, 5, 7, 8])
```

```
In [5]: a=np.array([5,3,7,8,1,2,3])
```

```
In [6]: np.argsort(a)
```

```
Out[6]: array([4, 5, 1, 6, 0, 2, 3], dtype=int64)
```



Arrays en Numpy. Funciones básicas.

¿¿Y... ordenar array de mayor a menor??

`a[::-1].sort()` ordena el array *in place*
`np.sort(a)[::-1]` crea un nuevo array

```
In [1]: import numpy as np

In [2]: a=np.array([5,3,7,8,1,2,3])
...: a[::-1].sort()
...: a
Out[2]: array([8, 7, 5, 3, 3, 2, 1])

In [3]: a=np.array([5,3,7,8,1,2,3])
...: np.sort(a)[::-1]
Out[3]: array([8, 7, 5, 3, 3, 2, 1])

In [4]: a
Out[4]: array([5, 3, 7, 8, 1, 2, 3])
```

Arrays en Numpy. Operaciones elemento a elemento

- Array (a1) con array (a2): Suma ($\mathbf{a1+a2}$), producto($\mathbf{a1*a2}$), resta($\mathbf{a1-a2}$), división($\mathbf{a1/a2}$), división entera($\mathbf{a1//a2}$), potencia ($\mathbf{a1**a2}$), mayor/mayor igual ($\mathbf{a1>a2}$ / $\mathbf{a1>=a2}$), menor / menor igual ($\mathbf{a1<a2}$ / $\mathbf{a1<=a2}$), igual ($\mathbf{a1==a2}$) y no igual ($\mathbf{a1!=a2}$)
- Escalar (c) con array (a): Suma ($\mathbf{c+a}$), producto($\mathbf{c*a}$), resta($\mathbf{a-c}$ ó $\mathbf{c-a}$), división($\mathbf{a/c}$ ó $\mathbf{c/a}$), división entera ($\mathbf{a//c}$ ó $\mathbf{c//a}$), potencia ($\mathbf{a**c}$ ó $\mathbf{c**a}$), mayor/mayor igual ($\mathbf{a>c}$ / $\mathbf{a>=c}$), menor/menor igual ($\mathbf{a<c}$ / $\mathbf{a<=c}$), igual ($\mathbf{a==c}$) y no igual ($\mathbf{a!=c}$)
- Resto: **`np.mod(a1, a2)`** / **`np.mod(a, c)`** / **`np.mod(c, a)`**
- Valor absoluto: **`np.abs(a)`**
- Raíz cuadrada: **`np.sqrt(a)`**
- Exponencial ($\mathbf{e**a}$): **`np.exp(a)`**
- Logaritmo natural / Logaritmo 2 / Logaritmo 10: **`np.log(a)`** / **`np.log2(a)`** / **`np.log10(a)`**

Arrays en Numpy. Operaciones elemento a elemento

- Funciones trigonométricas: `np.cos(a)`, `np.sin(a)`, `np.tan(a)`, `np.arccos(a)`, `np.arcsin(a)`, `np.arctan(a)`
- Signo: `np.sign(a)`
- Mínimo elemento a elemento: `np.minimum(a1, a2)`
- Máximo elemento a elemento: `np.maximum(a1, a2)`
- ceil, floor, redondear al entero más cercano: `np.ceil(a)`, `np.floor(a)`, `np rint(a)`
- Obtener los valores únicos: `np.unique(array)`
- ¿Están los elementos de un array en otro? `np.in1d(a1, a2)`
- Unión, intersección, diferencia (de conjuntos) y diferencia simétrica: `np.union1d(a1, a2)`, `np.intersect1d(a1, a2)`, `np.setdiff1d(a1, a2)`, `setxor1d(a1, a2)`

Arrays en Numpy. Operaciones con matrices

- Producto: **array1.dot(array2)**
- Transpuesta: **array1.transpose()**
- Diagonal (como array de 1d): **np.diagonal(array1)**
- Traza (suma de la diagonal de la matriz):
np.trace(array1)
- Determinante: **np.linalg.det(array1)**
- Inversa: **np.linalg.inv(array1)**

```
In [1]: import numpy as np

In [2]: array2 = np.array([[0,1,0],[1,0,1]])
...:
...: array1 = np.copy(array2)

In [3]: array2
Out[3]:
array([[0, 1, 0],
       [1, 0, 1]])

In [4]: array1
Out[4]:
array([[0, 1, 0],
       [1, 0, 1]])

In [5]: array1.dot(array2.transpose())
Out[5]:
array([[1, 0],
       [0, 2]])

In [6]: np.trace(array1.dot(array2.transpose()))
Out[6]: 3
```

Arrays en Numpy. Vectorización.

If-else vectorizado: `np.where(<condición array>, <valor cond true>, <valor cond false>)`

np.where: “dime dónde en este array, las entradas satisfacen una condición dada”

```
In [1]: import numpy as np
```

```
In [2]: a = np.arange(5,10)
```

```
In [3]: a
```

```
Out[3]: array([5, 6, 7, 8, 9])
```

```
In [4]: np.where(a < 8)
```

```
Out[4]: (array([0, 1, 2], dtype=int64),)
```

```
In [5]: a = np.arange(4,10).reshape(2,3)
```

```
In [6]: a
```

```
Out[6]: array([[4, 5, 6],  
               [7, 8, 9]])
```

```
In [7]: idxs = np.where(a > 5)
```

```
...: idxs
```

```
Out[7]: (array([0, 1, 1, 1], dtype=int64), array([2, 0, 1, 2], dtype=int64))
```

```
In [8]: result = a[idxs]
```

```
...: result
```

```
Out[8]: array([6, 7, 8, 9])
```

Posiciones (0,2), (1,0),

(1,1), (1,2)

Filas

Columnas

Arrays en Numpy. Vectorización.

- Repetir elementos de un array: **np.repeat(array, <nº repeticiones>, axis=<dim>)**

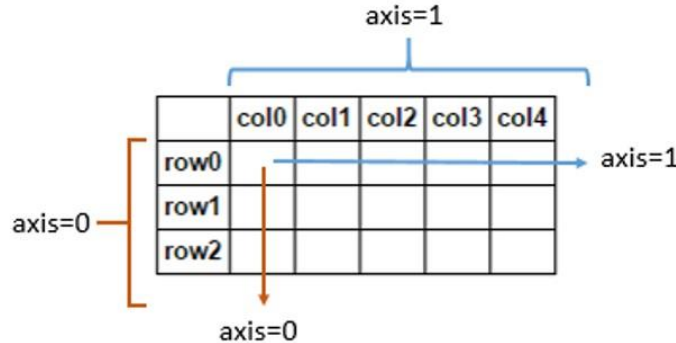
```
In [14]: np.repeat(3, 4)
Out[14]: array([3, 3, 3, 3])
```

```
In [15]: x = np.array([[1,2],[3,4]])
```

```
In [16]: x
Out[16]:
array([[1, 2],
       [3, 4]])
```

```
In [17]: np.repeat(x, 3, axis=0)
Out[17]:
array([[1, 2],
       [1, 2],
       [1, 2],
       [3, 4],
       [3, 4],
       [3, 4]])
```

```
In [18]: np.repeat(x, 3, axis=1)
Out[18]:
array([[1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4]])
```



```
In [13]: np.tile(x,(2,1))
...:
Out[13]:
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

Repetir array:
np.tile(array, <nº repeticiones>)

Arrays en Numpy. Vectorización.

- Podemos usar la función `np.apply_over_axes(f, array, axes=(<dim1>, <dim2>, ...))` para aplicar la función `f` sobre las dimensiones de array indicadas.

El orden en el que se indican las dims es el que se seguirá a la hora de realizar el cálculo.

```
In [1]: import numpy as np
```

```
In [2]: a = np.arange(24).reshape(2,3,4)
```

```
...: a
```

```
Out[2]:
```

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
```

```
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
```

```
In [3]: np.apply_over_axes(np.sum, a, [0,2])
```

```
Out[3]:
```

```
array([[[ 60],
        [ 92],
        [124]])])
```

12	14	16	18		60
20	22	24	26	→	92
28	30	32	34		124

Arrays en Numpy. Vectorización.

- Del mismo modo, **np.apply_along_axis(f, axis=<dim>, array)** aplica la función solo sobre la dimensión indicada.

```
In [6]: a
Out[6]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
```

```
In [7]: np.apply_along_axis(np.sum, 0, a)
Out[7]:
array([[12, 14, 16, 18],
       [20, 22, 24, 26],
       [28, 30, 32, 34]])
```

Código Numpy eficiente

```
In [1]: import numpy as np
...: ini = 1
...: end = 4
...: x = np.tile(np.arange(ini, end+1), (end+1,1))
...: x
```

```
Out[1]:
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]])
```

Tile repite (end+1,1) veces el vector del primer argumento

```
In [2]: y = x.copy()
...: for i in range(x.shape[0]):
...:     for j in range(x.shape[1]):
...:         x[i,j] *= 2
...: 
```

```
Out[2]:
array([[ 1,  4,  9, 16],
       [ 1,  4,  9, 16],
       [ 1,  4,  9, 16],
       [ 1,  4,  9, 16],
       [ 1,  4,  9, 16]])
```

```
In [4]: for i in range(end-1, end+1):
...:     for j in range(x.shape[1]):
...:         x[i, j] += 5
...: 
```

```
Out[4]:
array([[ 1,  4,  9, 16],
       [ 1,  4,  9, 16],
       [ 1,  4,  9, 16],
       [ 6,  9, 14, 21],
       [ 6,  9, 14, 21]])
```

```
In [3]: y *= 2
```

Equivalent to previous two nested for-loops

```
Out[3]:
array([[ 1,  4,  9, 16],
       [ 1,  4,  9, 16],
       [ 1,  4,  9, 16],
       [ 1,  4,  9, 16],
       [ 1,  4,  9, 16]])
```

```
In [5]: y[np.arange(end-1, end+1), :] += 5
```

```
Out[5]:
array([[ 1,  4,  9, 16],
       [ 1,  4,  9, 16],
       [ 1,  4,  9, 16],
       [ 6,  9, 14, 21],
       [ 6,  9, 14, 21]])
```

Evitar bucles **for** anidados:
For-loops ralentizarán dramáticamente vuestro código (~10-100x)

Hay muchas más funciones en el paquete, se recomienda echarle un vistazo a la documentación disponible en:
<http://www.numpy.org/>

Hablando de comandos útiles y eficientes

- **Zip**: permite iterar sobre varias listas/arrays en paralelo

```
In [1]: list_a = [1,2,3,4]
...: list_b = [2,3,4,5]
...:
...: list_a*list_b
Traceback (most recent call last):

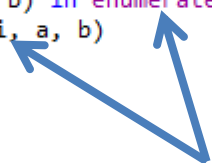
File "<ipython-input-1-89f73911f1be>", line 4, in <module>
    list_a*list_b

TypeError: can't multiply sequence by non-int of type 'list'
```

In [2]:

```
In [2]: [a*b for a,b in zip(list_a,list_b)]
Out[2]: [2, 6, 12, 20]
```

```
In [1]: alist = ['a1', 'a2', 'a3']
...: blist = ['b1', 'b2', 'b3']
...:
...: for i, (a, b) in enumerate(zip(alist, blist)):
...:     print(i, a, b)
0 a1 b1
1 a2 b2
2 a3 b3
```



Enumerate permite iterar sobre índices y elementos de una lista

Indexado Numpy (1)

Permite hacer lo mismo que las listas de python y mucho más.

```
x = [5]
```

```
type(x)  
list
```

```
x = np.array([5])
```

```
type(x)  
numpy.ndarray
```

Además, podemos indexar usando arrays de enteros o un array con booleanos (True, False):

```
In [1]: import numpy as np  
...:  
...: m = np.arange(0,6).reshape(2,3)
```

```
In [2]: m  
Out[2]:  
array([[0, 1, 2],  
       [3, 4, 5]])
```

```
In [3]: print('Mostrar la primera fila')  
...: print(m[0,:])  
Mostrar la primera fila  
[0 1 2]
```

```
In [4]: print('Mostrar las columnas pares')  
...: print(m[:,::2])  
Mostrar las columnas pares  
[[0 2]  
 [3 5]]
```

```
In [5]: print('Mostrar la esquina inferior derecha')  
...: print(m[-1,-1])  
Mostrar la esquina inferior derecha  
5
```

```
In [6]: m[m<3]=0  
...: print('Todos los elementos menores a 3 son 0 ahora')  
...: print(m)  
Todos los elementos menores a 3 son 0 ahora  
[[0 0 0]  
 [3 4 5]]
```


Indexado Numpy (y 2)

```
In [24]: data = np.array([[11, 22, 33,44],  
...:                     [55, 66,77,88],  
...:                     [99, 111,222,333]])
```

```
In [25]: X, y = data[:, :-1], data[:, -1]
```

```
In [26]: X
```

```
Out[26]:
```

```
array([[ 11,  22,  33],  
       [ 55,  66,  77],  
       [ 99, 111, 222]])
```

```
In [27]: y
```

```
Out[27]: array([ 44,  88, 333])
```

Coge todas las filas de la última columna

Coge todas las filas que van de la primera columna (columna 0) a la última (sin incluirla)

Datos. Terminología.

- **Variables:** Características de interés.
 - Se puede representar como una matriz.
Ej.: estimar el precio de un piso a partir de la zona, su tamaño, etc.
- **Muestra Observada:** Conjunto de valores de la variable obtenidos de manera homogénea.
 - Sería una fila de la matriz.
Ej.: en el ejemplo anterior, un piso concreto
- **Tamaño muestral:** Número de datos observados.
 - Sería el número de columnas de la matriz
Ej.: el número de pisos que tenemos en nuestra base de datos
- **Tipos de atributos:**
 - **Cualitativo** : Intrínsecamente no tiene carácter numérico (categórica).
Ej.: la calificación energética de la vivienda (A-G)
 - **Cuantitativo** : Intrínsecamente numérico:
 - Discreto (cantidad finita o numerable de valores). Ej.: el nº de tiendas en el barrio
 - Continuo (valores reales). Ej.: la superficie de piso

Datos. Generación de datos (1)

Numpy cuenta con varias funciones para la generación de datos pseudo-aleatorios (Random sampling) dentro de `numpy.random`.

Podemos fijar la semilla del generador de números pseudo-aleatorios para tener resultados reproducibles con **`np.random.seed(seed)`** (seed es un entero).

```
np.random.uniform()  
0.2209062316173077
```

```
np.random.uniform()  
0.08972395853794723
```

```
np.random.seed(37)
```

```
np.random.uniform()  
0.9444966028573069
```

```
np.random.uniform()  
0.4640981743044076
```

```
np.random.seed(37)
```

```
np.random.uniform()  
0.9444966028573069
```

```
np.random.uniform()  
0.4640981743044076
```

Datos. Generación de datos (y 2)

Además, incluye funciones para alterar aleatoriamente el orden de un array:

- **`np.random.shuffle(x)`**: Modifica `x` cambiando el orden de los elementos aleatoriamente. (Función in-place, no devuelve nada).
- **`np.random.permutation(x)`**: Devuelve el array `x` con sus elementos desordenados (de forma aleatoria).

Aparte de por no ser una función In-Place, *permutation* se diferencia en que, si le pasas un entero, te devuelve un *shuffled range*: shuffled range i.e.

```
np.random.shuffle(np.arange(n))
```

```
In [33]: np.random.seed(37)
```

```
In [34]: x = np.array([1,2,3,4,5])
```

```
In [35]: np.random.shuffle(x)
```

```
In [36]: x
```

```
Out[36]: array([3, 2, 5, 1, 4])
```

```
In [37]: x = np.array([1,2,3,4,5])
```

```
In [38]: np.random.seed(37)
```

```
In [39]: np.random.permutation(x)
```

```
Out[39]: array([3, 2, 5, 1, 4])
```

```
In [40]: x = 5
```

```
In [41]: np.random.permutation(x)
```

```
Out[41]: array([0, 1, 4, 3, 2])
```

```
In [42]: np.random.shuffle(x)
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-42-3ca5a81e845c>", line 1, in <module>  
    np.random.shuffle(x)
```

```
File "mtrand.pyx", line 4815, in mtrand.RandomState.shuffle
```

```
TypeError: object of type 'int' has no len()
```

Generación de datos con Random sampling (`numpy.random`)

`beta` (a, b[, size])

`binomial` (n, p[, size])

`chisquare` (df[, size])

`dirichlet` (alpha[, size])

`exponential` ([scale, size])

`f` (dfnum, dfden[, size])

`gamma` (shape[, scale, size])

`geometric` (p[, size])

`gumbel` ([loc, scale, size])

`hypergeometric` (ngood, nbad, nsample[, size])

`laplace` ([loc, scale, size])

`logistic` ([loc, scale, size])

`lognormal` ([mean, sigma, size])

`logseries` (p[, size])

`multinomial` (n, pvals[, size])

`multivariate_normal` (mean, cov[, size, ...])

`negative_binomial` (n, p[, size])

`noncentral_chisquare` (df, nonc[, size])

`noncentral_f` (dfnum, dfden, nonc[, size])

`normal` ([loc, scale, size])

`pareto` (a[, size])

`poisson` ([lam, size])

`power` (a[, size])

`rayleigh` ([scale, size])

`standard_cauchy` ([size])

`standard_exponential` ([size])

`standard_gamma` (shape[, size])

`standard_normal` ([size])

`standard_t` (df[, size])

`triangular` (left, mode, right[, size])

`uniform` ([low, high, size])

`vonmises` (mu, kappa[, size])

`wald` (mean, scale[, size])

`weibull` (a[, size])

`zipf` (a[, size])

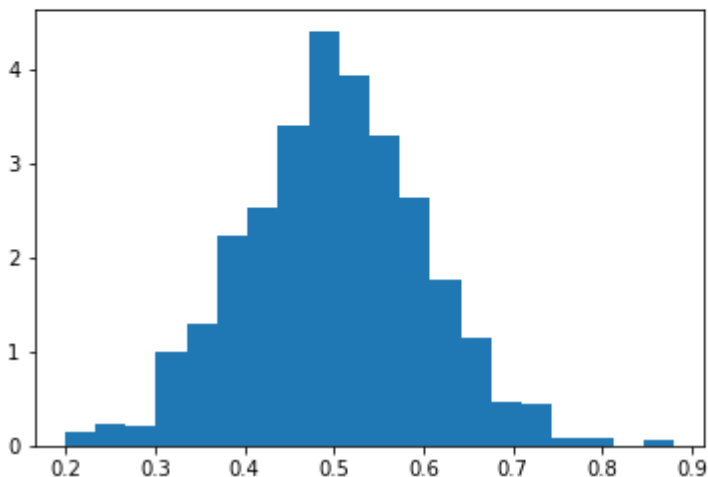
Generación de datos.

Distribución	Comando (<code>np.random.</code>)
Normal	<code>normal(loc, scale, size)</code>
Exponencial	<code>exponential(scale, size)</code>
Gamma	<code>gamma(shape, scale, size)</code>
Weibull	<code>weibull(a, size)</code>
Beta	<code>beta(a, b, size)</code>
t de Student (estandarizada)	<code>standard_t(df, size)</code>
F	<code>f(dfnum, dfden, size)</code>
Chi cuadrado	<code>chisquare(df, size)</code>
Binomial	<code>binomial(n, p, size=None)</code>
Poisson	<code>poisson(lam, size)</code>

Generación de datos. Ejemplos.

Distribución Normal

```
In [1]: import matplotlib.pyplot as plt
...: import numpy as np
...:
...: mu, sigma = 0.5, 0.1
...: s = np.random.normal(mu, sigma, 1000)
...:
...: # Create the bins and histogram
...: count, bins, ignored = plt.hist(s, 20, density=True)
```



Distribución Binomial

```
In [1]: import numpy as np

In [2]: #Draw samples from the distribution:
...: n, p = 10, .5 # number of trials, probability of each trial
...: s = np.random.binomial(n, p, 100)
...: # result of flipping a coin 10 times, tested 25 times.
...: s
Out[2]:
array([5, 5, 3, 3, 4, 6, 5, 4, 6, 4, 5, 4, 8, 5, 6, 5, 6, 6, 2, 2, 3, 5,
       3, 6, 8, 7, 6, 6, 7, 5, 4, 6, 6, 7, 6, 7, 4, 4, 2, 5, 4, 6, 6, 7,
       6, 7, 3, 5, 6, 7, 3, 6, 4, 6, 5, 4, 7, 4, 6, 4, 5, 5, 5, 4, 5, 4,
       4, 6, 3, 9, 5, 3, 5, 3, 7, 3, 4, 4, 2, 6, 5, 3, 4, 5, 5, 2, 6, 4,
       8, 6, 5, 6, 4, 5, 4, 4, 3, 4, 4, 4])

In [3]: #What is the probability of getting 3 or less heads
...: sum(s <= 3)/100
Out[3]: 0.17
```

Probabilidad del 17% de obtener 3 o menos caras al repetir 100 veces el experimento de lanzar 10 veces una moneda

Generación de datos discretos.

A parte de `randint` (`random_integers` está obsoleta), podemos generar datos discretos muestreando aleatoriamente de un conjunto. Para ello, podemos usar la función `np.random.choice(a, size, replace=True, p=None)`:

- **a**: Lista o array con los posible valores.
- **size**: Forma del array a generar (tupla o lista con el tamaño de cada dimensión).
- **replace**: Indica si los elementos se sacarán de la muestra, de forma que no se repitan.
- **p**: Array con la probabilidad de cada elemento. Es opcional.

```
In [2]: np.random.randint(1,5)
Out[2]: 2
```

```
In [3]: type(np.random.randint(5))
Out[3]: int
```

```
In [4]: np.random.randint(1,5, size=(3,2))
Out[4]:
array([[2, 4],
       [3, 4],
       [2, 4]])
```

```
In [5]: np.random.choice(5, 3)
Out[5]: array([1, 4, 3])
```

```
In [6]: np.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])
Out[6]: array([3, 2, 2], dtype=int64)
```

```
In [7]: np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
Out[7]: array([3, 2, 0])
```

3 enteros
aleatorios
entre 0 y 4

Probabilidad de seleccionar
cada elemento

Lectura de datos (1)

Hay muchas formas de leer datos de disco.

P. ej., podemos leer una matriz de un fichero de texto con cada línea siendo una fila y los elementos separados por espacios

```
# -*- coding: utf-8 -*-  
import numpy as np
```

```
matrix = []
```

```
f = open('mat.txt', 'r')
```

```
for l in f:
```

```
    row_matrix = []
```

```
    l = l.rstrip()
```

```
    for e in l.split(' '):
```

```
        row_matrix.append(float(e))
```

```
    if len(row_matrix) > 0:
```

```
        matrix.append(row_matrix)
```

```
f.close()
```

```
matrix = np.array(matrix, np.float64)
```

Eliminamos espacios en blanco al final de cada string

Individualizamos cada número (sabemos que están separados por espacios)

```
In [5]: matrix  
Out[5]:  
array([[ 1.,  2.,  3.,  4.],  
       [ 5.,  6.,  7.,  8.],  
       [ 9., 10., 11., 12.]])
```

Lectura de datos (y 2)

Podemos hacer lo mismo usando la función

```
np.loadtxt(<path fichero>, delimiter=<delimitador>,  
dtype=<tipo>, skiprows=<saltar filas al inicio>)
```

Con esa función se puede leer también un csv en una matriz de numpy y guardarlo con

```
np.savetxt(<path fichero>, <array>, delimiter=<delimitador>,  
header=<cabecera>)
```

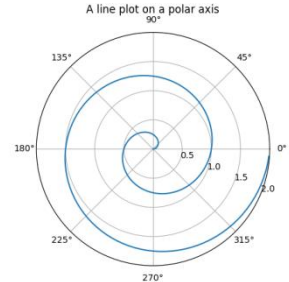
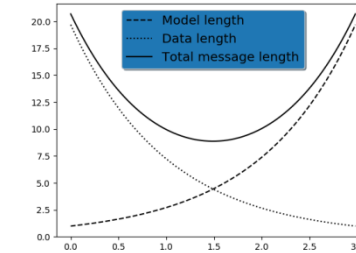
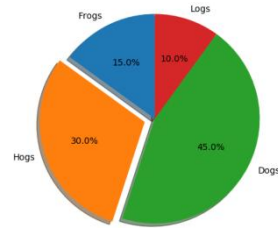
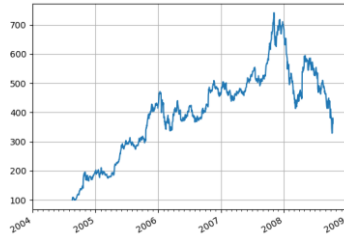
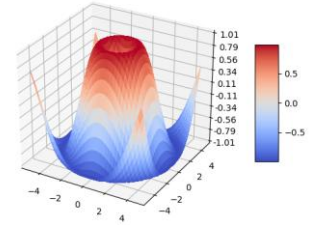
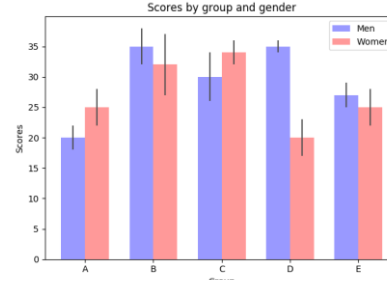
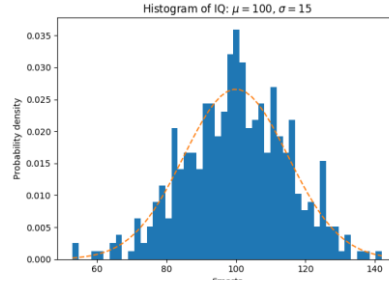
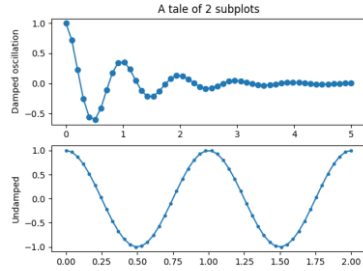
Si vamos a guardar o leer un array de numpy es preferible usar el formato binario .npy.

Para guardar usamos `np.save(<path fichero>, <array>)`

Para leer usamos `np.load(<path fichero>)`

Representación con Matplotlib

- Nos permite emplear distintos tipos de gráficos para visualizar los datos de formas sencilla y rápida usando listas o vectores de numpy



matplotlib

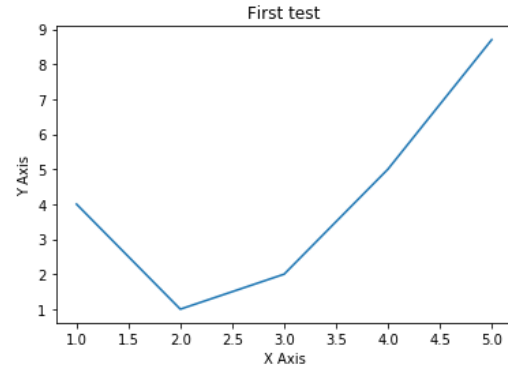
Representación con Matplotlib

- Algunas funciones básicas:

```
import matplotlib.pyplot as plt
```

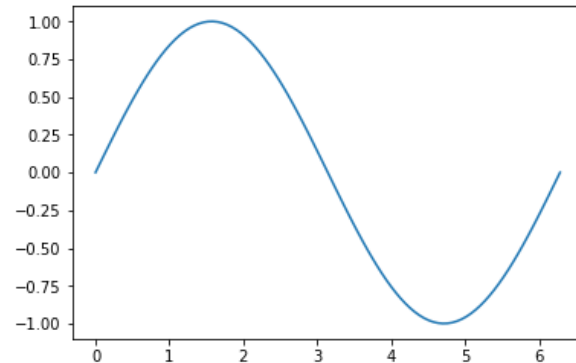
```
y = [4, 1, 2, 5, 8.7]  
x = range(1, len(y)+1)
```

```
plt.plot(x, y)  
plt.xlabel('X Axis')  
plt.ylabel('Y Axis')  
plt.title('First test')  
plt.show()
```



```
import numpy as np  
import matplotlib.pyplot as plt
```

```
x = np.linspace(0, 2*np.pi, 100)  
y = np.sin(x)  
plt.plot(x, y)  
plt.show()
```



Tipos de gráficos

Líneas: `plt.plot(<x>, <y>, <color y forma>)`

Barras: `plt.bar(<x>,<y>, color=<color>, orientation={ 'vertical' ,
'horizontal' })`

Puntos: `plt.scatter(<x>, <y>, c=<color index>, cmap=<map del
color; puede ser alguno de los declarados en plt.cm >,
s=<tamaño>)`

Diagrama de cajas: `plt.boxplot(<x>, notch=<pintar los intervalos de
confianza>, sym=<Str con el símbolo para los outliers. Si no se
pone nada no los pintará>, vert=<pintar en vertical u
horizontal>)`

Para más información acerca del color, forma y otros parámetros, consultar documentación.

Representación con Matplotlib

Podemos alterar muchos aspectos del gráfico mostrado: el rango de los ejes, incluir una leyenda, un grid...

```
import numpy as np
import matplotlib.pyplot as plt
```

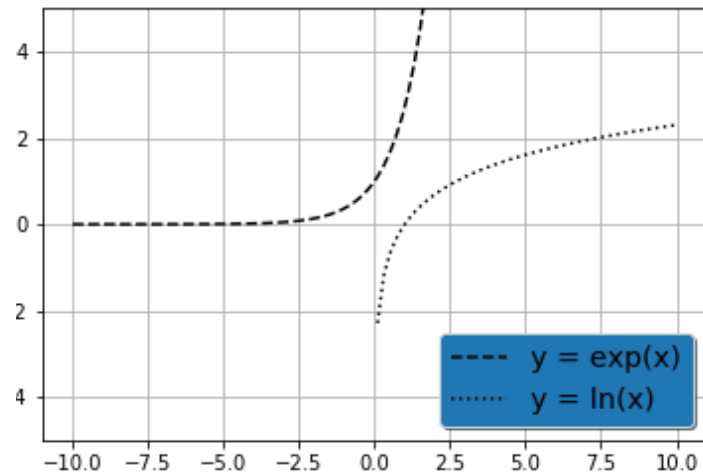
```
# Make some fake data.
x = np.linspace(-10, 10, 100)
y1 = np.exp(x)
y2 = np.log(x)
```

```
# Create plots with pre-defined labels.
fig, ax = plt.subplots()
ax.plot(x, y1, 'k--', label='y = exp(x)')
ax.plot(x, y2, 'k:', label='y = ln(x)')
```

```
legend = ax.legend(loc='lower right', shadow=True, fontsize='x-large')
```

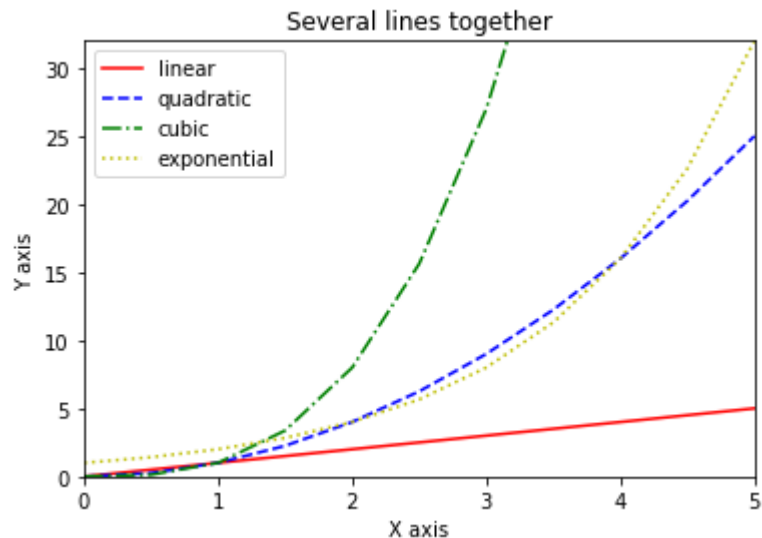
```
ax.set_ylim((-5, 5))
```

```
# Put a nicer background color on the legend.
legend.get_frame().set_facecolor('C0')
plt.grid()
plt.show()
```



Más ejemplos...

```
max_val = 5.  
t = np.arange(0., max_val+0.5, 0.5)  
plt.plot(t, t, 'r-', label='linear')  
plt.plot(t, t**2, 'b--', label='quadratic')  
plt.plot(t, t**3, 'g-.', label='cubic')  
plt.plot(t, 2**t, 'y:', label='exponential')  
  
plt.xlabel('X axis')  
plt.ylabel('Y axis')  
  
plt.title('Several lines together')  
plt.legend()  
plt.axis([0, max_val, 0, 2**max_val])  
plt.show()
```



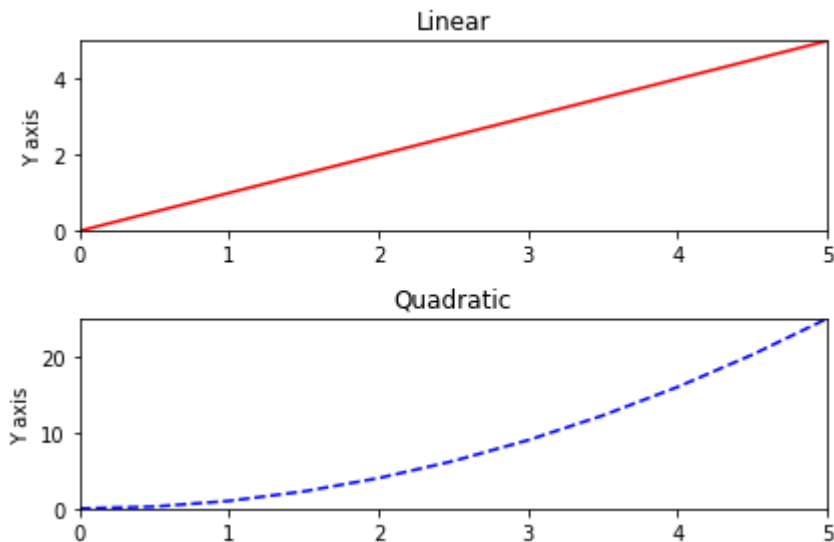
Más ejemplos...

- Podemos pintar varios gráficos en una misma figura:

```
max_val = 5.  
t = np.arange(0., max_val+0.5, 0.5)  
ax = plt.subplot('211') #Crear dos figuras (2 filas): una encima de la otra (1 col)  
ax.set_title('Linear')  
ax.plot(t, t, 'r-')  
ax.set_ylabel('Y axis')  
ax.axis([0, max_val, 0, max_val])  
  
ax = plt.subplot('212') #Crear segunda figura  
ax.set_title('Quadratic')  
ax.plot(t, t**2, 'b--')  
ax.set_ylabel('Y axis')  
ax.axis([0, max_val, 0, max_val**2])  
  
plt.tight_layout() #Dejar espacio entre figuras  
plt.show()
```

`ax = plt.subplot("ijk"):`

- i: Número de filas de figuras.
- j: Número de columnas de figuras.
- k: Identificador de la figura a pintar (para determinar la posición).



Referencias recomendables

- <https://cs231n.github.io/python-numpy-tutorial/>
- <https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/lectures/python-review.pdf>
- <https://docs.scipy.org/doc/numpy/user/quickstart.html>
- <https://stanfordpython.com/#labs>