



Las funciones con python™

Carmen Gandía

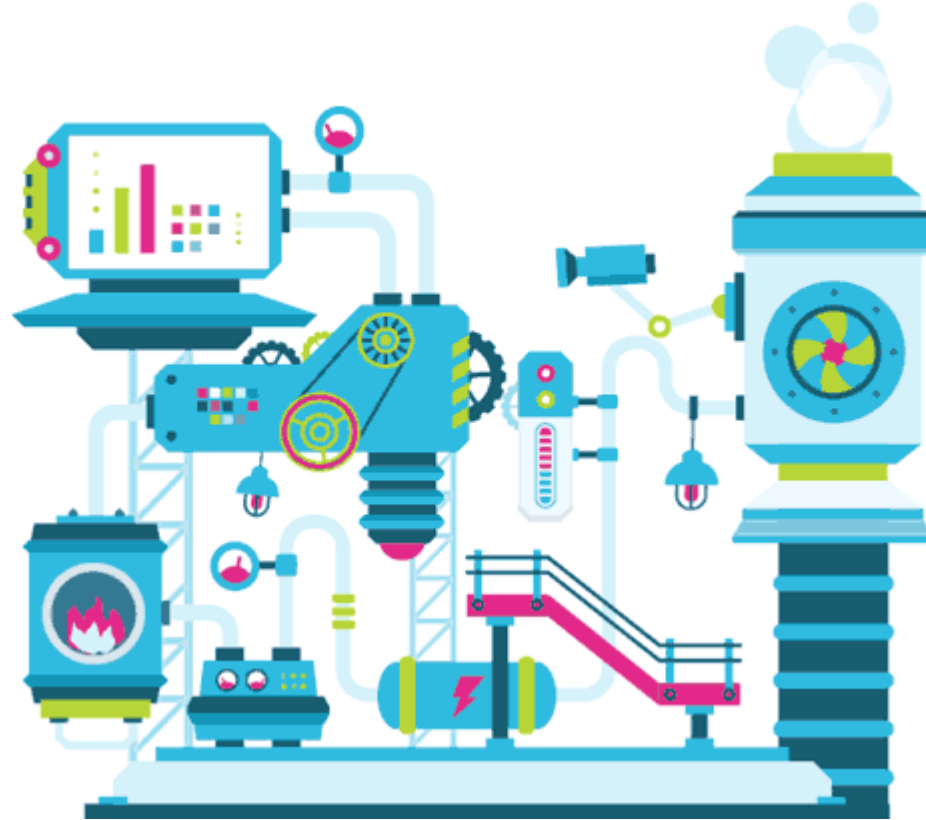
carmen.gandia@ua.es

Julio Mulero

julio.mulero@ua.es



Las funciones con Python





Las funciones

Operaciones con funciones

Las representaciones gráficas

Otros tipos de gráficos



Las funciones



Las funciones

Escribir las mismas instrucciones varias veces puede resultar **tedioso**. Una solución es juntar las **instrucciones** en una función que se comporta como las funciones en matemáticas: dando argumentos de entrada se obtiene una respuesta o salida.


 `def función(argumento/s):` 
 ` instrucciones`
`4 espacios` 




Las funciones

Definimos una función `hola1` que, dado un argumento, `imprime` «Hola» seguido del argumento. Por ejemplo, queremos que `hola1('Mateo')` imprima «Hola Mateo».

`def`
`print`



```
def hola1(nombre):  
    print('Hola', nombre)
```





Las funciones

Definimos una función `hola2` que pregunte su nombre al usuario de **forma interactiva** e imprima «Encantado de conocerte» seguido de su nombre. En cálculo numérico **no solemos abusar de esta vía** para solicitar los datos de forma interactiva. De esta forma, si no se indica lo contrario, los parámetros serán introducidos directamente como argumentos de la función.

input

```
def hola2():
```

```
    """
```

```
    Ejemplo de función sin argumento.  
    Imprime el dato ingresado.
```

```
    """
```

```
    print('Hola, soy la compu')
```

```
    nombre = input('¿Cuál es tu nombre? ')
```

```
    print('Encantada de conocerte,', nombre)
```





Las funciones

```
def hola2():  
    """  
    Ejemplo de función sin argumento.  
    Imprime el dato ingresado.  
    """  
    print('Hola, soy la compu')  
    nombre = input('¿Cuál es tu nombre? ')  
    print('Encantada de conocerte,', nombre)
```

```
In [10]: hola2()  
Hola, soy la compu
```

```
¿Cuál es tu nombre? Julio
```

```
Encantada de conocerte, Julio
```




Las funciones

```
def hola2():  
    """  
    Ejemplo de función sin argumento.  
    Imprime el dato ingresado.  
    """  
    print('Hola, soy la compu')  
    nombre = input('¿Cuál es tu nombre? ')  
    print('Encantada de conocerte,', nombre)
```



```
In [11]: help(hola2)  
Help on function hola2 in module __main__:
```

```
hola2()  
    Ejemplo de función sin argumento.  
    Imprime el dato ingresado.
```



Las funciones

Definimos una función `sumar2` que, dados dos argumentos, `devuelva` su suma. Observa el uso de la instrucción `return`.

return

```
def sumar2(a, b):  
    """Suma los argumentos."""  
    return a + b # resultado de la función
```





Las funciones

A diferencia de print, la orden return permite asignar el valor resultante a un nuevo objeto.

```
In [1]: def sumar2(a, b):  
...:     """Suma los argumentos."""  
...:     return a + b # resultado de la función
```

```
In [2]: a = sumar2(1,2)
```

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

```
Out[3]: 3
```

```
In [4]: a+1
```

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





Las funciones

Obviamente, a fin de obtener un resultado satisfactorio, los objetos introducidos como argumentos deben poder “sumarse”. Es decir, *las operaciones que se realicen con los objetos deben tener sentido*. Por ejemplo, no podemos sumar un número y una cadena de caracteres. Observa cómo se suman dos cadenas de caracteres.

```
In [2]: sumar2("a",2)
Traceback (most recent call last):

File "<ipython-input-2-be30d13a4a78>", line 1, in <module>
    sumar2("a",2)

File "<ipython-input-1-dc5ab59ec9d5>", line 3, in sumar2
    return a + b # resultado de la función

TypeError: can only concatenate str (not "int") to str
```

```
In [3]:
In [3]: sumar2("a","b")
Out[3]: 'ab'
```



```
In [4]: sumar2(1,2)
Out[4]: 3
```



Las funciones

Las funciones definidas, y compiladas previamente, pueden ser utilizadas a lo largo de la sesión de trabajo.

```
In [1]: def f(x):  
...:     """Multiplicar por 2."""  
...:     return 2*x
```

```
In [2]: def g(x):  
...:     """Multiplicar por 4."""  
...:     return f(f(x))
```

```
In [3]: g(3)  
Out[3]: 12
```



Las funciones

Las funciones también son **objetos**, y cuando definimos una función se fabrica un objeto de tipo function (función), con su propio contexto, y se construye una variable que tiene por identificador el de la función y hace referencia a ella.

```
In [1]: def f(x):  
...:     """Multiplicar por 2."""  
...:     return 2*x
```

```
In [2]: def g(x):  
...:     """Multiplicar por 4."""  
...:     return f(f(x))
```

```
In [3]: g(3)  
Out[3]: 12
```

```
In [4]: type(g)  
Out[4]: function
```



Unos ejercicios

Implementa una función que, dada una temperatura f en grados Fahrenheit, devuelva la temperatura en grados centígrados c , es decir, $c = 5(f - 32)/9$.

Implementa una función que, dada una temperatura c en grados centígrados, devuelva la temperatura en grados Fahrenheit f .

Implementa una función llamada `area_rectangulo(base, altura)` que devuelva el área del rectángulo a partir de una base y una altura. Calcula el área de un rectángulo de 15 de base y 10 de altura.

Implementa una función llamada `laboral` que pregunte al usuario cuál es su ocupación e imprima "Ok, tu trabajo es ____".



Unos ejercicios

Construya una función que devuelva el área y la longitud de una circunferencia de radio r que se introducirá como parámetro. Si no se especifica ningún parámetro se entenderá que el radio es la unidad.

Construya una función que devuelva el cociente y el resto de una división entera. Las entradas serán dividendo y divisor.

Construya la función $f(x) = 2e^x \log(2x) \sin(x)$ para $x \in (10,20)$.



Errores típicos

Los errores detienen la ejecución del programa y tienen varias causas. Entre los errores más usuales podemos citar los errores de **sintaxis**, de nombre y de semántica. Veamos algunos ejemplos:

SyntaxError

```
In [1]: print("hola"  
File "<ipython-input-1-9751fbfdc060>", line 1  
    print("hola"  
           ^
```



```
SyntaxError: unexpected EOF while parsing
```



Errores típicos

Los errores detienen la ejecución del programa y tienen varias causas. Entre los errores más usuales podemos citar los errores de sintaxis, de **nombre** y de semántica. Veamos algunos ejemplos:

NameError

```
In [1]: pint("hola")
```

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

```
File "<ipython-input-1-22de51f03de3>", line 1, in <module>  
    pint("hola")
```



```
NameError: name 'pint' is not defined
```



Errores típicos

Los errores detienen la ejecución del programa y tienen varias causas. Entre los errores más usuales podemos citar los errores de sintaxis, de nombre y de **semántica**. Veamos algunos ejemplos:

IndexError

```
In [1]: l = []
```

```
In [2]: l.pop()
```

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

```
File "<ipython-input-2-33a5ac1f2384>", line 1, in <module>
```

```
l.pop()
```



```
IndexError: pop from empty list
```



Errores típicos

Los errores detienen la ejecución del programa y tienen varias causas. Entre los errores más usuales podemos citar los errores de sintaxis, de nombre y de **semántica**. Veamos algunos ejemplos:

TypeError

```
In [1]: n = input("Introduce un número: ")
```

```
Introduce un número: 6
```

```
In [2]: n/2
```

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

```
File "<ipython-input-2-806318527c7e>", line 1, in <module>  
    n/2
```



```
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```



Unos ejercicios

¿Qué errores se pueden detectar en los siguientes códigos?

```
DiasSemana = ["LU", "MA", "MI", "JU", "VI", "SA", "DO"]  
DiaSemana.append(2)
```

```
Tupla = ("a", 2, "b", 3, 2)  
Tupla[5]
```

```
saludo = "hola"  
saludo/2
```



Las funciones matemáticas



Las funciones matemáticas

Lógicamente las funciones son grandes protagonistas del cálculo numérico. Atendiendo a todas las consideraciones anteriores, y a fin de poder trabajar con ellas de forma adecuada, utilizaremos la orden `return`.



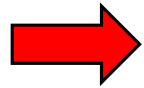
```
def f(x):  
    resultado = 2*x  
    return resultado
```



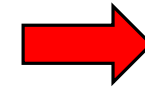
Las funciones matemáticas

Las funciones se pueden escribir de muchas formas distintas. De hecho, la programación es un ejercicio muy creativo.

```
def f(x):  
    resultado = 2*x  
    return resultado
```



```
def f(x):  
    return 2*x
```



```
f = lambda x: 2*x
```




Las funciones matemáticas

La función `quad` de `scipy.integrate` devuelve una lista con dos valores. El primero (20.25) es el valor aproximado de la integral definida en (0,4.5), mientras que el segundo, es una aproximación del error cometido.

quad

```
In [1]: import scipy.integrate as si  
In [2]: def f(x):  
...:     return 2*x  
In [3]: si.quad(f, 0, 4.5)  
Out[3]: (20.25, 2.248201624865942e-13)
```



No en vano, los métodos numéricos sirven para aproximar determinados valores como el valor de esta integral.

A veces no nos interesa demasiado tener definida una función, sino solamente su integral. La función `lambda` permite definir funciones anónimas que pueden ser usadas en contextos determinados como el cálculo de su integral definida mediante la función `quad` de `scipy.integrate`:

lambda

```
In [1]: import scipy.integrate as si  
In [2]: si.quad(lambda x: 2*x, 0, 4.5)  
Out[2]: (20.25, 2.248201624865942e-13)
```





Las funciones matemáticas

Obviamente, también podemos definir funciones matemáticas con varios parámetros. A veces estos argumentos adicionales son simples parámetros y otras veces ayudan a implementar funciones de varias variables:

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

$f: \mathbb{R} \rightarrow \mathbb{R}$ con $a \in \mathbb{R}$
 $x \mapsto ax$

```
In [2]: def f(x,a):  
...:     return a*x
```

$f: \mathbb{R}^2 \rightarrow \mathbb{R}$
 $(x,y) \mapsto x \sin(y)$

```
In [3]: def f(x,y):  
...:     return x*np.sin(y)
```



Las representaciones gráficas



Las representaciones gráficas

Existe una gran variedad de módulos para hacer gráficos de todo tipo con Python, pero el estándar de facto en ciencia es `matplotlib`. Se trata de un paquete grande y relativamente complejo que entre otros contiene dos módulos principales, `pyplot` y `pylab`.

Nosotros/as trabajaremos con `pyplot`, que ofrece una interfaz para crear gráficos fácilmente, automatizando la creación de figuras y ejes automáticamente cuando hace un gráfico:

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



Las representaciones gráficas

La instrucción básica para la construcción de representaciones gráficas es `plot`.

La sintaxis de `plot()` es simplemente `plot(x, y)`, pero si no se incluye la lista `x`, ésta se reemplaza por el número de elementos o índice de la lista `y`, por lo que es equivalente a hacer `plot(range(len(y)), y)`.

`plt.plot(x,y)`



Las representaciones gráficas

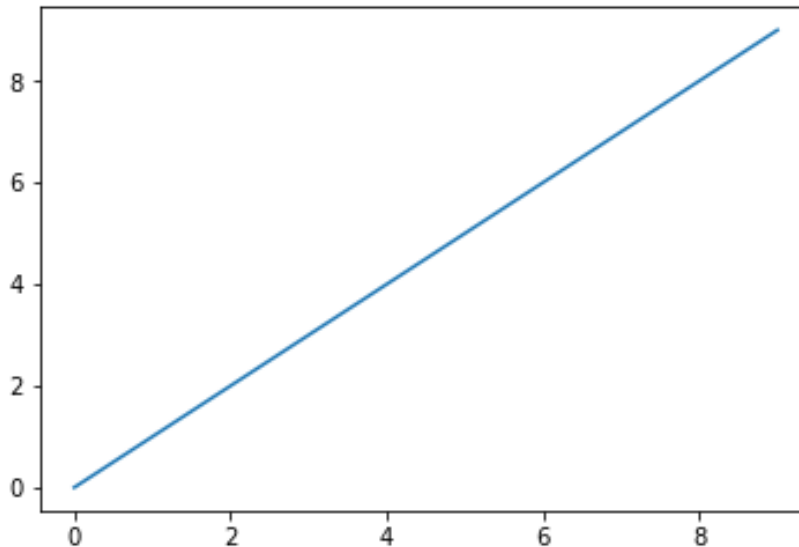
```
In [1]: import numpy as np  
...: import matplotlib.pyplot as plt
```

```
In [2]: x = np.arange(10.)
```

```
In [3]: plt.plot(x)
```

```
Out[3]: [<matplotlib.lines.Line2D at 0x102c40f6518>]
```

Generalmente, `plt.plot` necesita dos argumentos `x` e `y` (que serán arrays) para pintar los puntos (x_i, y_i) . Cuando solo le damos un argumento `x` pinta los puntos (i, x_i) para $i=0, 1, \dots$



En la consola `Ipython` aparece automáticamente el gráfico construido. Si trabajamos en otros entornos, puede ser necesario añadir la orden `plt.show()` para visualizar el gráfico.



Las representaciones gráficas

```
In [2]: x = np.arange(10.)
```

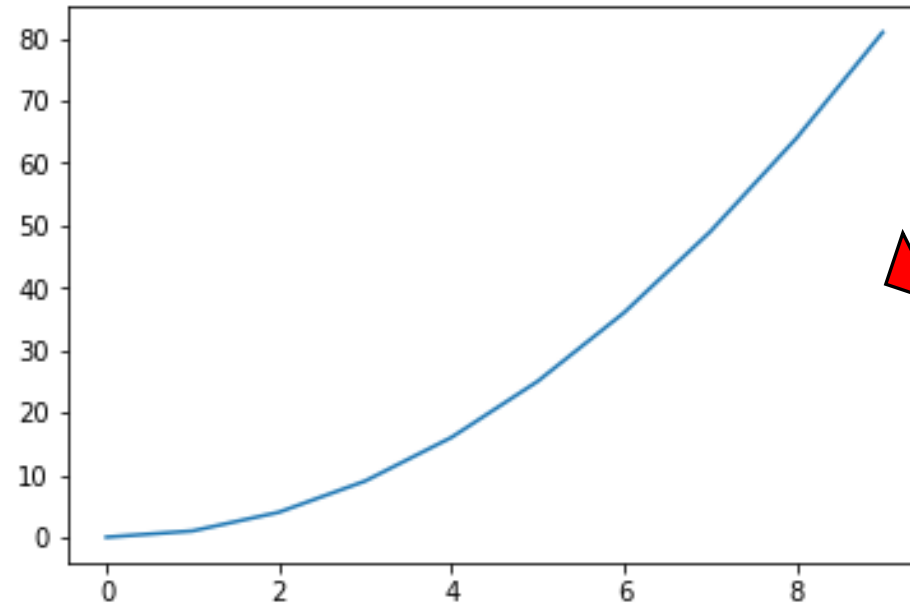
```
....:
```

```
....: y = x**2
```

```
....:
```

```
....: plt.plot(x,y)
```

```
Out[2]: [<matplotlib.lines.Line2D at 0x1a8dc2b46a0>]
```



Fijaos que esto no es la gráfica de la función x^2 , sino que realmente es una poligonal uniendo los puntos (x_i, y_i) con segmentos.



Las representaciones gráficas

```
In [2]: x = np.arange(10.)
```

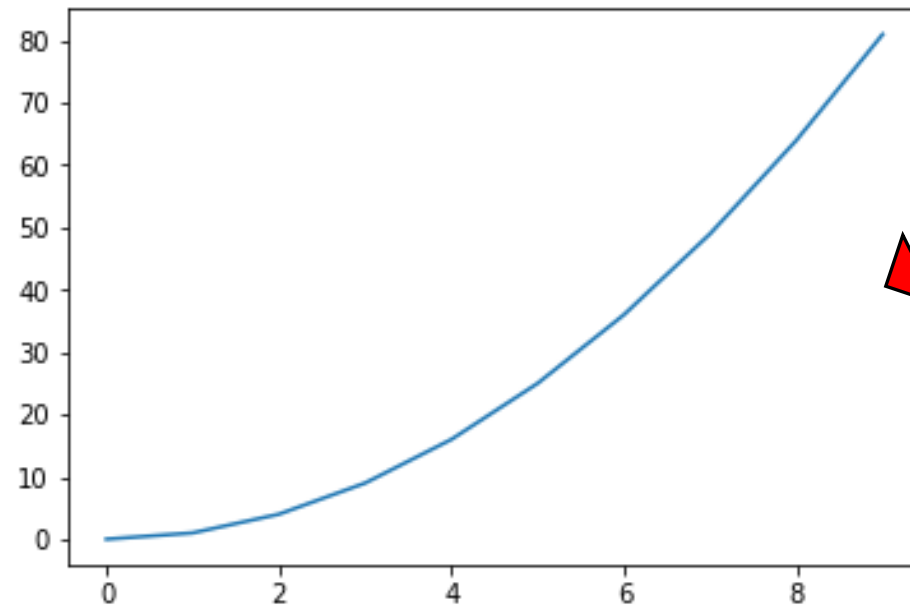
```
....:
```

```
....: y = x**2
```

```
....:
```

```
....: plt.plot(x,y)
```

```
Out[2]: [<matplotlib.lines.Line2D at 0x1a8dc2b46a0>]
```



Fijaos que esto no es exactamente la gráfica de la función x^2 , sino que es una poligonal uniendo los puntos (x_i, y_i) con segmentos.

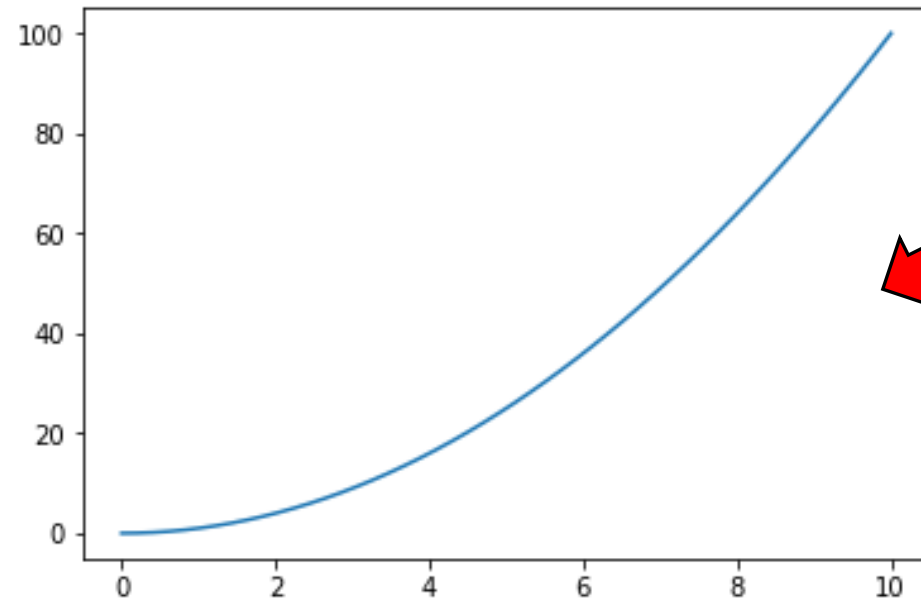


Las representaciones gráficas

```
In [2]: x = np.linspace(0,10.)  
.....:  
.....: y = x**2  
.....:  
.....: plt.plot(x,y)  
Out[2]: [<matplotlib.lines.Line2D at 0x1ec069d35c0>]
```

`len(x)`
50

Por defecto, `np.linspace` devuelve 50 puntos equiespaciados, aunque podríamos especificar la cantidad de puntos, si es necesario.



Usando un `np.linspace` obtenemos una gráfica más "suave".



Las representaciones gráficas

Podemos compilar dos `plt.plot` a la vez (seleccionando todas las líneas) y se representarán las dos funciones en la misma gráfica. Python asigna por defecto diferentes colores.



```
In [2]: x = np.linspace(0,10.)
```

```
...:
```

```
...: y = x**2
```

```
...:
```

```
...: plt.plot(x,y)
```

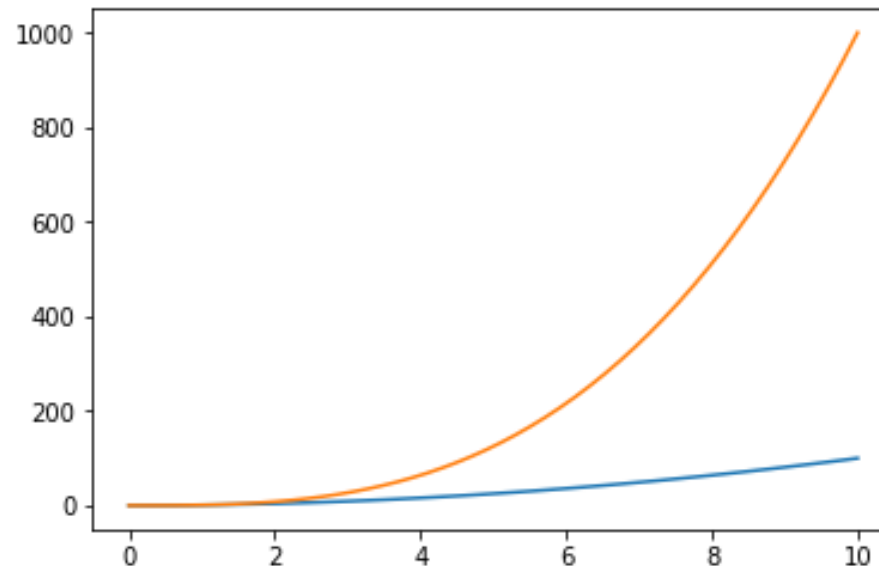
```
...:
```

```
...: z = x**3
```

```
...:
```

```
...: plt.plot(x,z)
```

```
Out[2]: [<matplotlib.lines.Line2D at 0x23b4e4e5198>]
```



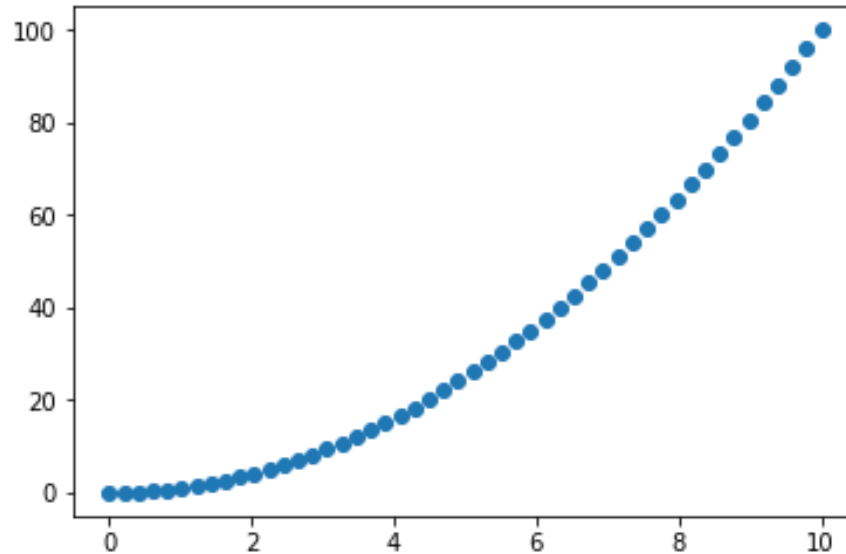


Las representaciones gráficas

```
In [2]: x = np.linspace(0,10.)  
...:  
...: y = x**2  
...:  
...: plt.plot(x,y,"o")  
Out[2]: [<matplotlib.lines.Line2D at 0x1b750a0d9e8>]
```



Por defecto, plot construye poligonales. Si queremos pintar los puntos, debemos añadir un parámetro adicional.





Las representaciones gráficas

Símbolo	Descripción
"_"	Línea continua
"_:"	Línea a trazos
"_."	Línea a puntos y rayas
"_."	Línea punteada
"."	Símbolo punto
"."	Símbolo pixel
"o"	Símbolo círculo relleno
"v"	Símbolo triángulo hacia abajo
"^"	Símbolo triángulo hacia arriba
"<"	Símbolo triángulo hacia la izquierda
">"	Símbolo triángulo hacia la derecha
"s"	Símbolo cuadrado
"p"	Símbolo pentágono
"*"	Símbolo estrella
"+"	Símbolo cruz
"x"	Símbolo X
"D"	Símbolo diamante
"d"	Símbolo diamante delgado



Las representaciones gráficas

```
In [2]: x = np.linspace(0,10.)
```

```
...:
```

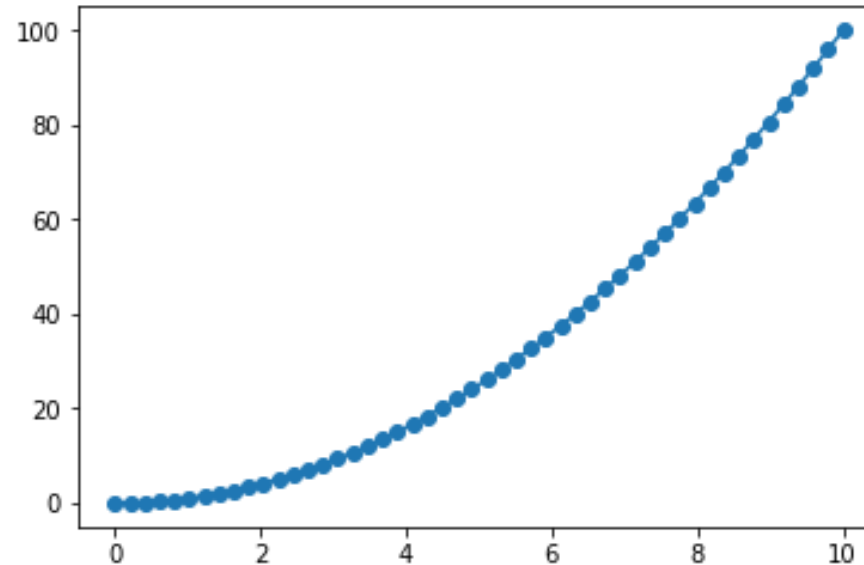
```
...: y = x**2
```

```
...:
```

```
...: plt.plot(x,y,"o-")
```

```
Out[2]: [<matplotlib.lines.Line2D at 0x24949072780>]
```

También podemos
combinar símbolos y
líneas.





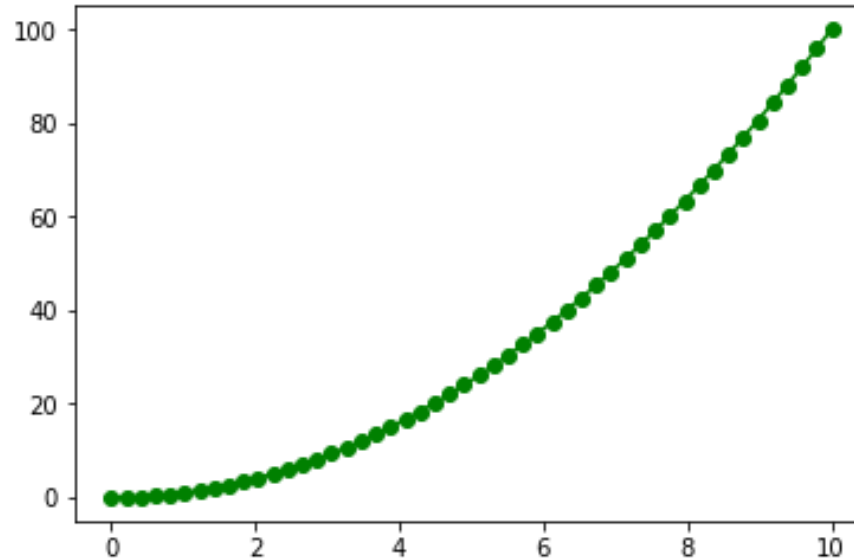
Las representaciones gráficas

```
In [2]: x = np.linspace(0,10.)  
...:  
...: y = x**2  
...:  
...: plt.plot(x,y,"go-")  
Out[2]: [<matplotlib.lines.Line2D at 0x235f592b7f0>]
```



Y para cambiar los colores debemos añadir una letra adicional.

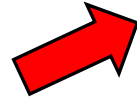
Símbolo	Color
"b"	Azul
"g"	Verde
"r"	Rojo
"c"	Cian
"m"	Magenta
"y"	Amarillo
"k"	Negro
"w"	Blanco





Las representaciones gráficas

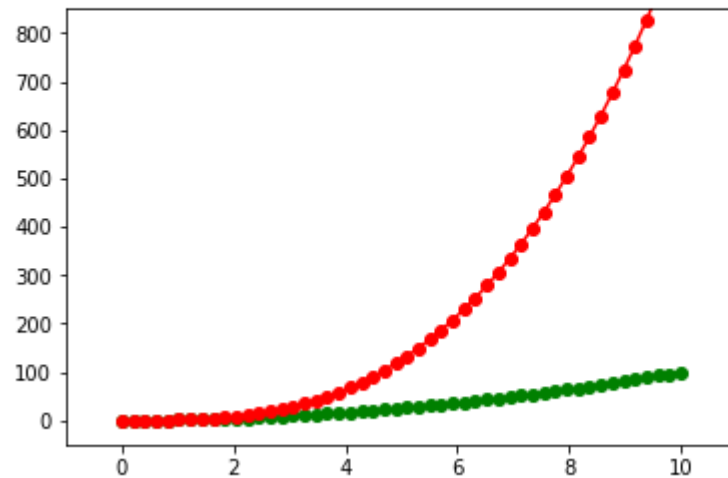
Es posible cambiar el intervalo mostrado en los ejes con `xlim()` e `ylim()`.



```
In [2]: x = np.linspace(0,10.)
...:
...: y = x**2
...:
...: z = x**3

In [3]: plt.xlim(-1, 11)      # nuevos límites para el eje 0X >]
...:
...: plt.ylim(-50, 850)
...:
...: plt.plot(x,y,"go-",x,z,"ro-")

Out[3]:
[<matplotlib.lines.Line2D at 0x1db5a3759b0>,
 <matplotlib.lines.Line2D at 0x1db5a375b00>]
```





Las representaciones gráficas

Además del marcador y el color indicado de la manera anterior, se pueden cambiar muchas otras propiedades de la gráfica como parámetros de plot() independientes:

Parámetro	Significado y valores
alpha	grado de transparencia, float (0.0=transparente a 1.0=opaco)
color o c	Color de matplotlib
label	Etiqueta con cadena de texto, string
markeredgecolor o mec	Color del borde del símbolo
markeredgewidth o mew	Ancho del borde del símbolo, float (en número de puntos)
markerfacecolor o mfc	Color del símbolo
markersize o ms	Tamaño del símbolo, float (en número de puntos)
linestyle o ls	Tipo de línea, “_” “_” “_.” “.” “None”
linewidth o lw	Ancho de la línea, float (en número de puntos)
marker	Tipo de símbolo, “+” “*” “,” “.” “1” “2” “3” “4” “<” “>” “D” “H” “^” “_” “d” “h” “o” “p” “s” “v” “x” “ ” TICKUP TICKDOWN TICKLEFT TICKRIGHT

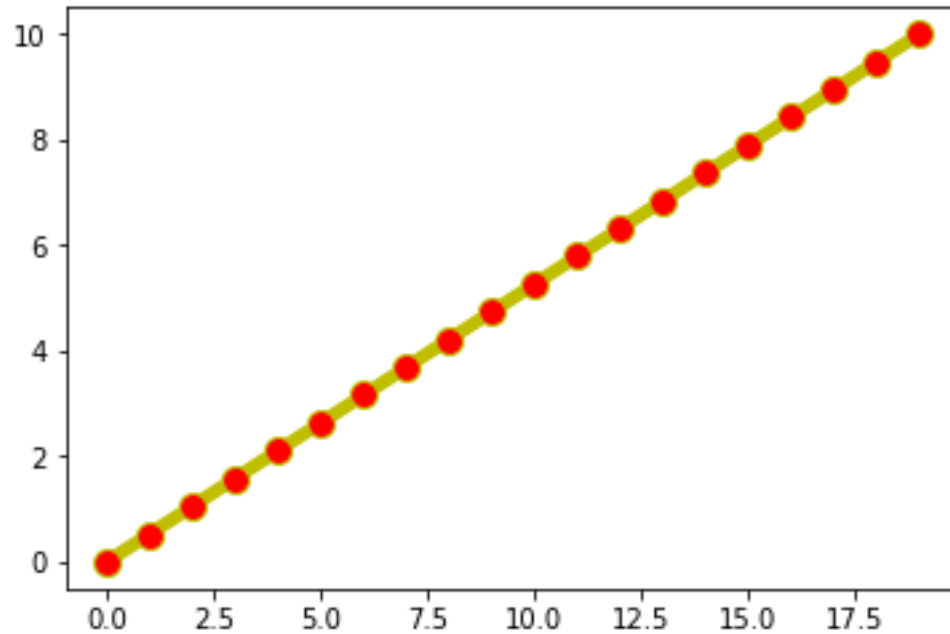


Las representaciones gráficas

```
In [6]: x = np.linspace(0,10.,20)
```

```
In [7]: plt.plot(x, lw=5, c='y', marker='o', ms=10, mfc='red')
```

```
Out[7]: [<matplotlib.lines.Line2D at 0x1db5a0031d0>]
```



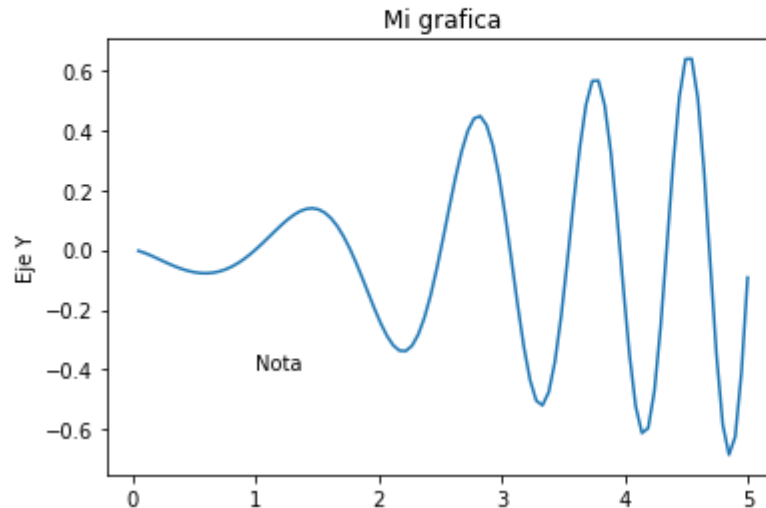


Las representaciones gráficas

Existen funciones para añadir texto (etiquetas) a los ejes de la gráfica y a la gráfica en sí.



```
In [2]: x = np.linspace(0, 5, 100)
....:
....: plt.xlabel('Eje X')          # Etiqueta del eje OX
....:
....: plt.ylabel('Eje Y')          # Etiqueta del eje OY
....:
....: plt.title('Mi grafica')      # Título del gráfico
....:
....: plt.text(1, -0.4, 'Nota')    # Texto en coordenadas (1, -0.4)
....:
....: plt.plot(x, np.log10(x)*np.sin(x**2))
__main__:11: RuntimeWarning: divide by zero encountered in log10
__main__:11: RuntimeWarning: invalid value encountered in multiply
Out[2]: [<matplotlib.lines.Line2D at 0x222c2613f28>]
```





Las representaciones gráficas

```
t = np.arange(0.1, 20, 0.1)
```

```
y1 = np.sin(t)/t
```

```
y2 = np.sin(t)*exp(-t)
```

```
plt.plot(t, y1, t, y2)
```

```
# Texto en la gráfica en coordenadas (x,y)
```

```
plt.text(2, 0.6, r'$\frac{\sin(x)}{x}$', fontsize=20)
```

```
plt.text(13, 0.2, r'$\sin(x) \cdot e^{-x}$', fontsize=16)
```

```
# Añado una malla al gráfico
```

```
plt.grid()
```

```
plt.title('Representacion de dos funciones')
```

```
plt.xlabel('Tiempo / s')
```

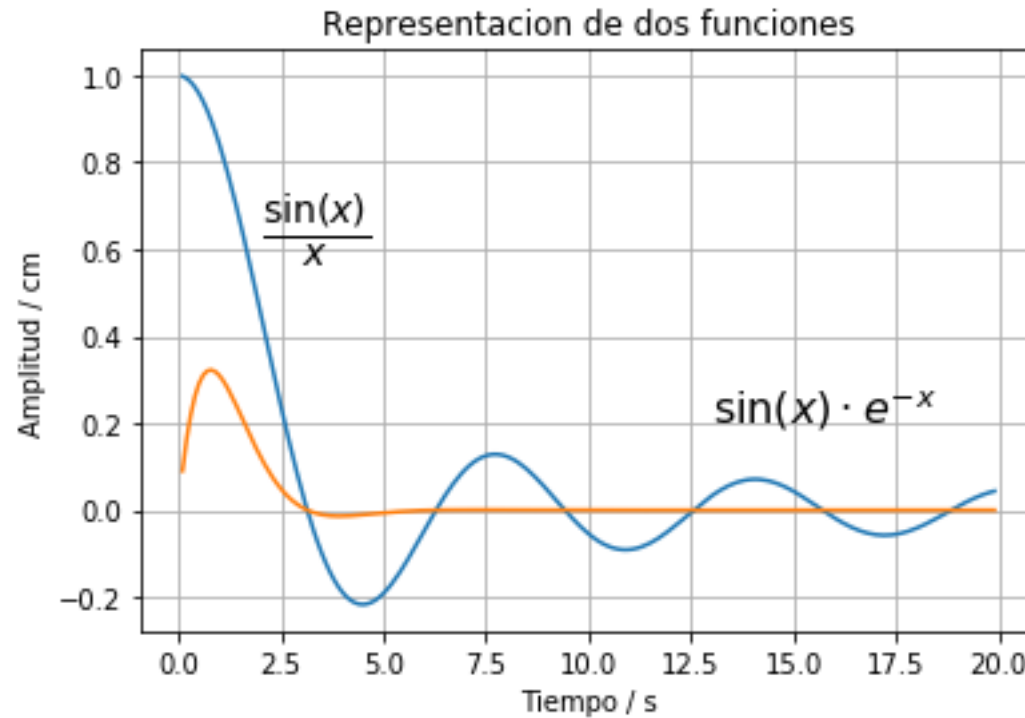
```
plt.ylabel('Amplitud / cm')
```

Las especificaciones técnicas para una gráfica se pueden escribir antes o después del plot, pero debe compilarse todo junto.

Se puede añadir código Latex para insertar fórmulas matemáticas como texto.



Las representaciones gráficas





Las representaciones gráficas

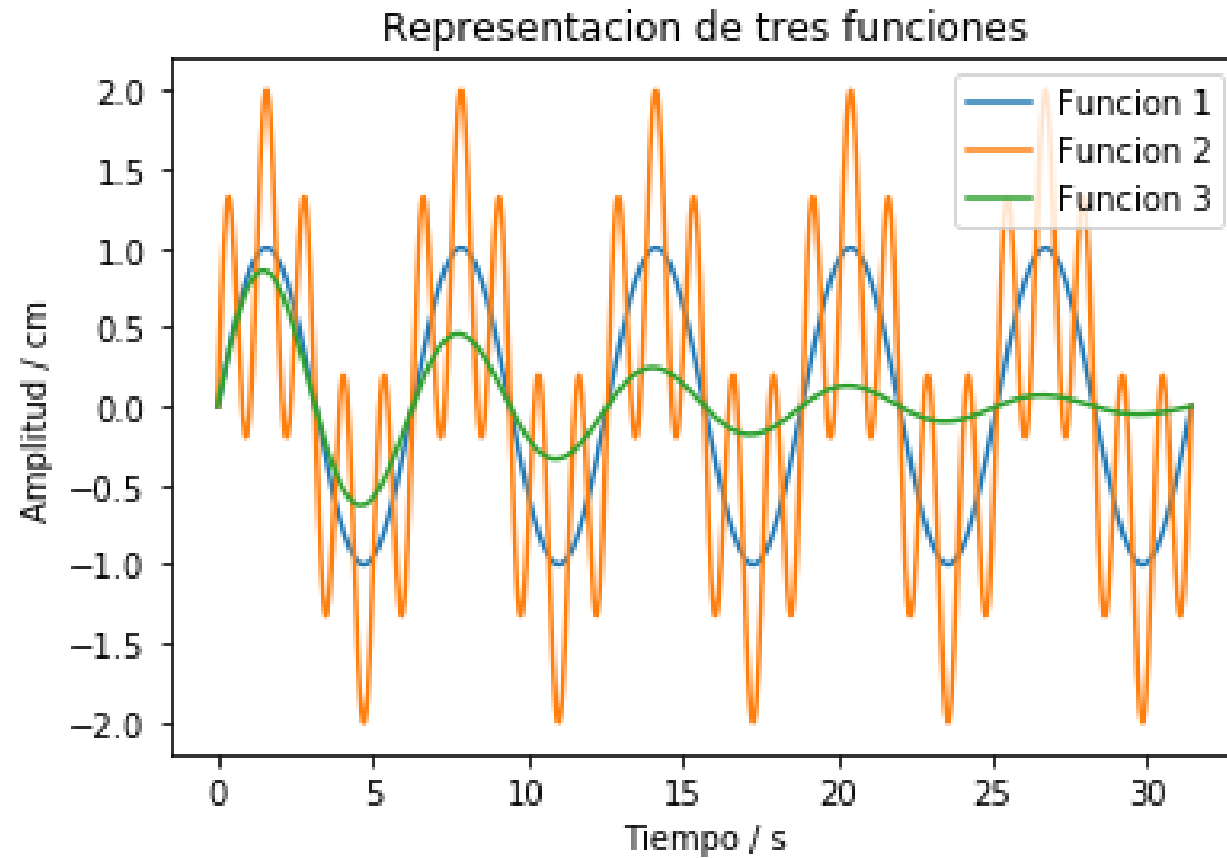
```
def f1(x):  
    y = np.sin(x)  
    return y  
  
def f2(x):  
    y = np.sin(x) + np.sin(5.0*x)  
    return y  
  
def f3(x):  
    y = np.sin(x) * np.exp(-x/10.)  
    return y  
  
# array de valores a representar  
x = np.linspace(0, 10*np.pi, 800)  
  
plt.plot(x, f1(x), x, f2(x), x, f3(x))  
# Añado Leyenda, tamaño de letra 10, en esquina superior  
plt.legend(['Funcion 1', 'Funcion 2', 'Funcion 3'],  
           prop = {'size': 10}, loc='upper right')  
  
plt.xlabel('Tiempo / s')  
plt.ylabel('Amplitud / cm')  
plt.title('Representacion de tres funciones')
```

Cuando representamos tres funciones en el mismo gráfico, es recomendable añadir una leyenda.





Las representaciones gráficas





Las representaciones gráficas

En ocasiones nos interesa mostrar varios gráficos diferentes en una misma figura o ventana.

A fin de obtener una correcta disposición de las gráficas, podemos usar la función `subplot()`, indicando entre paréntesis un número con tres dígitos. El primer dígito indica el número de filas en las que se dividirá la figura, el segundo el número de columnas y el tercero se refiere al gráfico con el que estamos trabajando en ese momento.

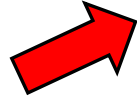
Supongamos que queremos representar estas tres funciones usando tres gráficas en la misma figura, una al lado de la otra y, por lo tanto, con una fila y tres columnas...

```
def f1(x):  
    y = np.sin(x)  
    return y  
  
def f2(x):  
    y = np.sin(x) + np.sin(5.0*x)  
    return y  
  
def f3(x):  
    y = np.sin(x) * np.exp(-x/10.)  
    return y
```



Las representaciones gráficas

Podemos modificar el tamaño para que las gráficas no queden demasiado pequeñas (y no se solapen)...



```
plt.figure(figsize=(10,5))
```

```
plt.subplot(131)
```

```
plt.plot(x,f1(x),'r-')
```

```
# Etiqueta del eje Y, que es común para todas
```

```
plt.ylabel('Amplitud / cm')
```

```
plt.title('Función 1')
```

```
# Figura con una fila y tres columnas, activo segundo subgráfico
```

```
plt.subplot(132)
```

```
plt.plot(x,f2(x),'b-')
```

```
# Etiqueta del eje X, que es común para todas
```

```
plt.xlabel('Tiempo / s')
```

```
plt.title('Función 2')
```

```
# Figura con una fila y tres columnas, activo tercer subgráfico
```

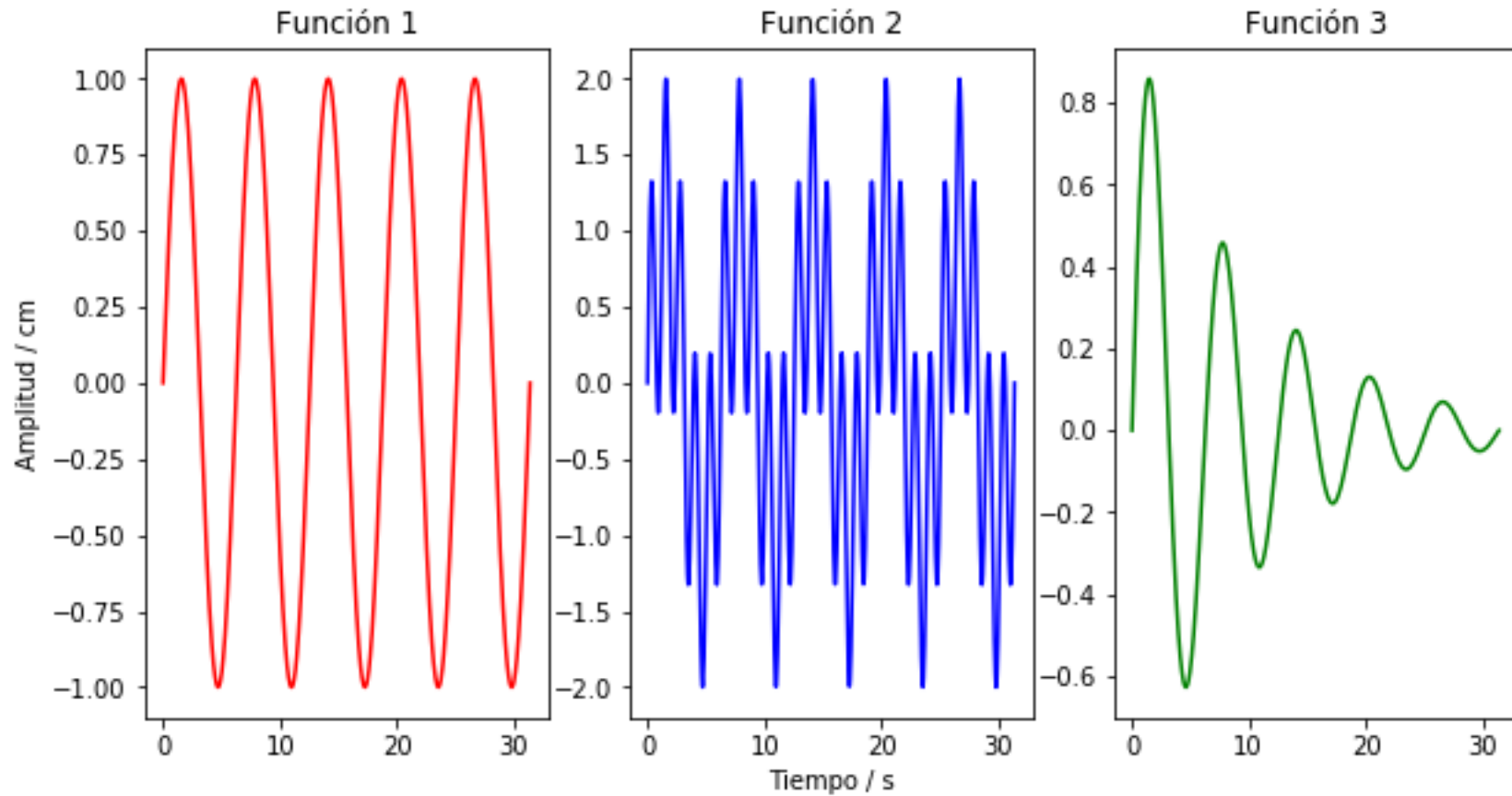
```
plt.subplot(133)
```

```
plt.plot(x, f3(x),'g-')
```

```
plt.title('Función 3')
```




Las representaciones gráficas





Un ejercicio

Escribe una función que permita dibujar cualquier elipse, de semiejes a y b y con centro en el punto (x_0, y_0) . Las entradas de esta función serán las coordenadas del centro y los semiejes.



Otros tipos de gráficos



Gráficos bidimensionales

Aunque `matplotlib` está especializado en gráficos 2D, incluye un `toolkit` para hacer gráficos 3D de muchos tipos usando OpenGL, que nos resolverá casi todas las necesidades para gráficos de este tipo.

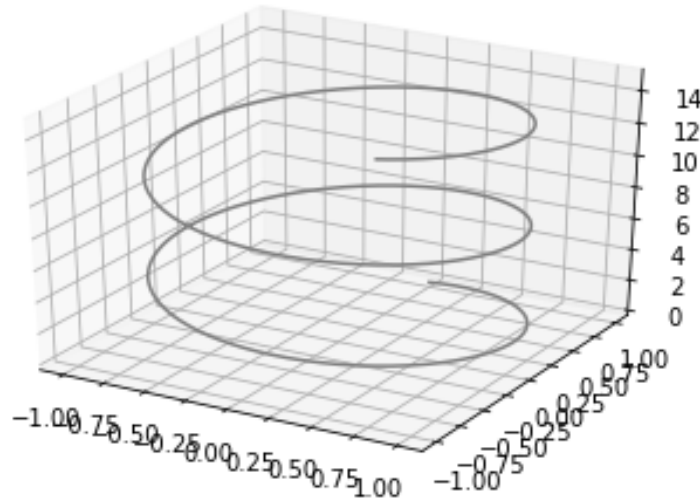
El dibujo de líneas en el espacio mediante el comando `plot3D` no dista mucho de lo que hacemos en el plano con el comando `plot`. En el plano se dan pares de valores que representan puntos en el plano y el comando `plot` traza la línea resultante de unir dichos puntos. En el espacio se darán ternas de números, cada una de las cuales representa un punto y obtendremos la gráfica que resulta de unir dichas ternas.



Gráficos bidimensionales

```
In [1]: import numpy as np
...: import matplotlib.pyplot as plt
...: from mpl_toolkits import mplot3d
```

```
In [2]: fig = plt.figure()
...: ax = plt.axes(projection='3d')
...:
...: zline = np.linspace(0, 15, 1000)
...: xline = np.sin(zline)
...: yline = np.cos(zline)
...: ax.plot3D(xline, yline, zline, 'gray')
Out[2]: [<mpl_toolkits.mplot3d.art3d.Line3D at 0x28bd8a76d30>]
```





Gráficos bidimensionales

Por otra parte, con `plot_surface` podemos dibujar superficies.

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
# Axes3D import has side effects, it enables using projection='3d'
import matplotlib.pyplot as plt
import random

def fun(x, y):
    return x**2 + y

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = y = np.arange(-3.0, 3.0, 0.05)
X, Y = np.meshgrid(x, y)
zs = np.array([fun(np.ravel(X), np.ravel(Y))])
Z = zs.reshape(X.shape)

ax.plot_surface(X, Y, Z)

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

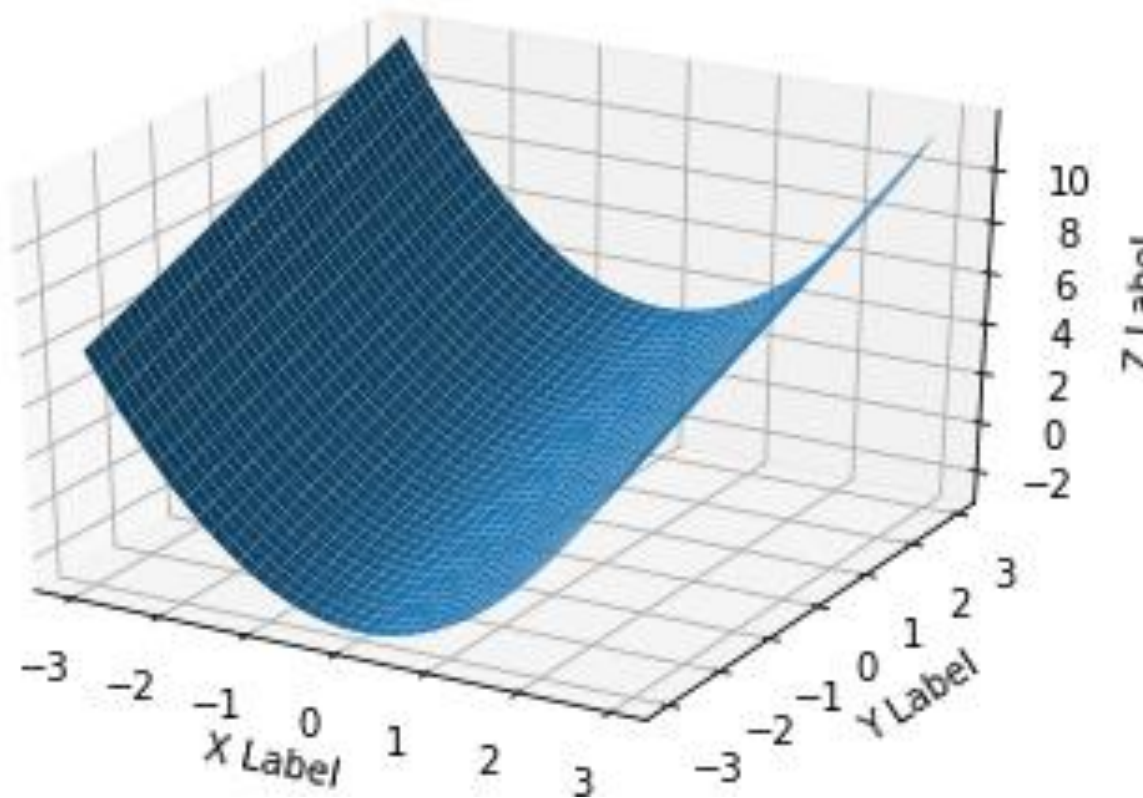
plt.show()
```





Gráficos bidimensionales

Aunque matplotlib está especializado en gráficos 2D, incluye un toolkit para hacer gráficos 3D de muchos tipos usando OpenGL, que nos resolverá casi todas las necesidades para gráficos de este tipo.





Gráficos estadísticos

Uno de los primeros gráficos (que todos/as conocemos) que se usan para representar un conjunto de datos cualitativos (como el género) o cuantitativos discretos (como el número de hijos) es el **diagrama de barras**. Podemos construirlo fácilmente usando la función **bar**.

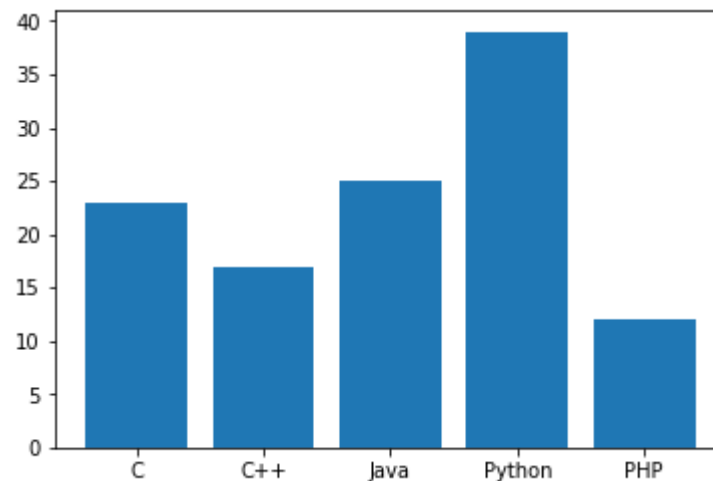
```
In [1]: import matplotlib.pyplot as plt
```



```
In [2]: lenguajes = ['C', 'C++', 'Java', 'Python', 'PHP']  
...: estudiantes = [23,17,25,39,12]
```

```
In [3]: plt.bar(lenguajes,estudiantes)
```

```
Out[3]: <BarContainer object of 5 artists>
```





Gráficos estadísticos

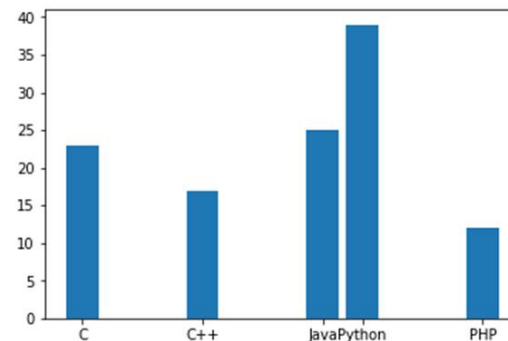
Mediante bar podemos indicar las posiciones mediante una tupla (en orden creciente) y las alturas de las barras. Adicionalmente, podemos cambiar las etiquetas de las barras.



```
In [1]: import matplotlib.pyplot as plt  
  
In [2]: posiciones = [-1, 2, 5, 6, 9]  
...: alturas = [23,17,25,39,12]  
...: etiquetas = ['C', 'C++', 'Java', 'Python', 'PHP']
```

```
In [3]: plt.bar(posiciones,alturas)  
...: plt.xticks(posiciones, etiquetas)
```

```
Out[3]:  
([<matplotlib.axis.XTick at 0x2b72b4e8d30>,  
 <matplotlib.axis.XTick at 0x2b72b498550>,  
 <matplotlib.axis.XTick at 0x2b72b3f09e8>,  
 <matplotlib.axis.XTick at 0x2b72b787b70>,  
 <matplotlib.axis.XTick at 0x2b72b7950b8>],  
 <a list of 5 Text xticklabel objects>)
```





Gráficos estadísticos

Muy similar al gráfico de barras es el de escaleras. No tenemos más que usar la función `stairs` y todo lo dicho sirve.

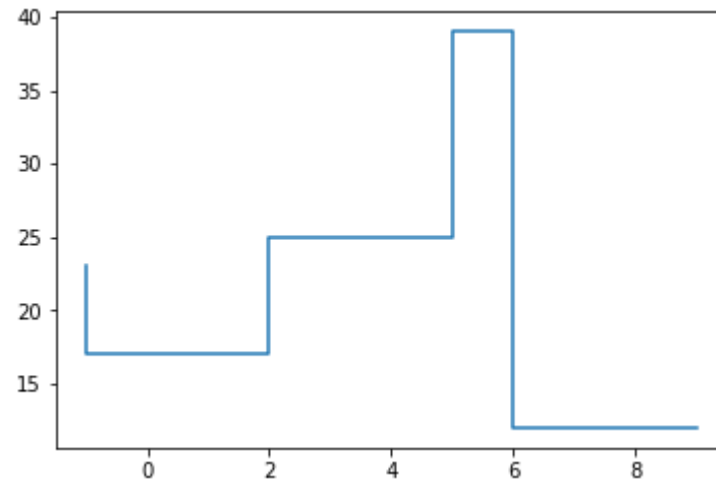
```
In [1]: import matplotlib.pyplot as plt
```

```
In [2]: posiciones = [-1, 2, 5, 6, 9]  
...: alturas = [23,17,25,39,12]
```



```
In [3]: plt.step(posiciones,alturas)
```

```
Out[3]: [<matplotlib.lines.Line2D at 0x2302f567320>]
```





Gráficos estadísticos

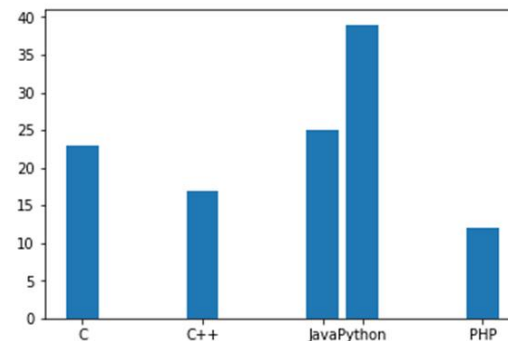
Mediante bar podemos indicar las posiciones mediante una tupla (en orden creciente) y las alturas de las barras. Adicionalmente, podemos cambiar las etiquetas de las barras.



```
In [1]: import matplotlib.pyplot as plt  
  
In [2]: posiciones = [-1, 2, 5, 6, 9]  
...: alturas = [23,17,25,39,12]  
...: etiquetas = ['C', 'C++', 'Java', 'Python', 'PHP']
```

```
In [3]: plt.bar(posiciones,alturas)  
...: plt.xticks(posiciones, etiquetas)
```

```
Out[3]:  
([<matplotlib.axis.XTick at 0x2b72b4e8d30>,  
  <matplotlib.axis.XTick at 0x2b72b498550>,  
  <matplotlib.axis.XTick at 0x2b72b3f09e8>,  
  <matplotlib.axis.XTick at 0x2b72b787b70>,  
  <matplotlib.axis.XTick at 0x2b72b7950b8>],  
  <a list of 5 Text xticklabel objects>)
```





Gráficos estadísticos

Cuando tenemos un conjunto de datos numéricos, por ejemplo como consecuencia de la medida de una cierta magnitud, y queremos representarlos gráficamente para ver la distribución subyacente de los mismos se suelen usar los gráficos llamados **histogramas**. Los histogramas son los equivalentes de los diagramas de barras para variables cuantitativas continuas.

Los histogramas representan el número de veces que los valores del conjunto caen dentro de un intervalo dado, frente a los diferentes intervalos en los que queramos dividir el conjunto de valores.

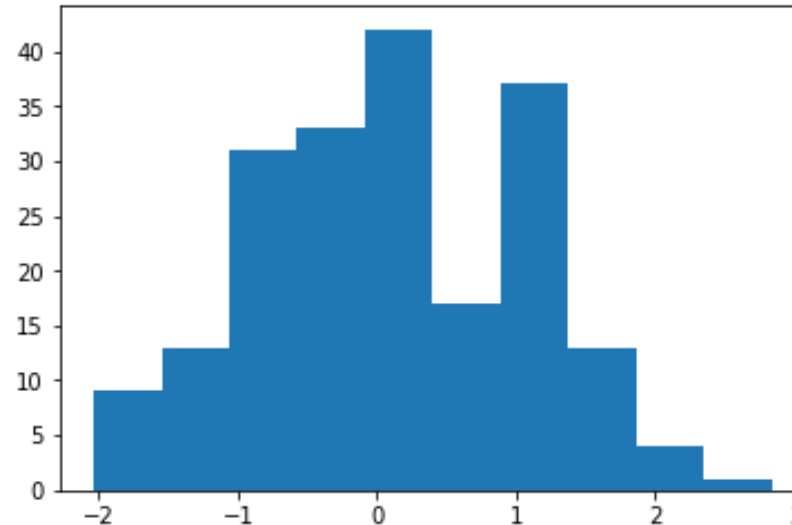
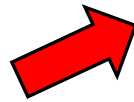
En Python podemos hacer histogramas muy fácilmente con la función **hist** indicando como parámetro un array con los números del conjunto.



Gráficos estadísticos

```
In [2]: nums = np.random.randn(200) # array con 200 números
        aleatorios
        ....:
        ....: # generamos el histograma
        ....: plt.hist(nums)
Out[2]:
(array([ 9., 13., 31., 33., 42., 17., 37., 13.,  4.,  1.]),
 array([-2.03729892, -1.55018975, -1.06308059, -0.57597142,
        -0.08886226,
         0.39824691,  0.88535607,  1.37246523,  1.8595744 ,
         2.34668356,
         2.83379273])),
<a list of 10 Patch objects>)
```

Si no se indica nada más, se generará un histograma con 10 intervalos (llamados bins, en inglés) en los que se divide la diferencia entre el máximo y el mínimo valor del conjunto.

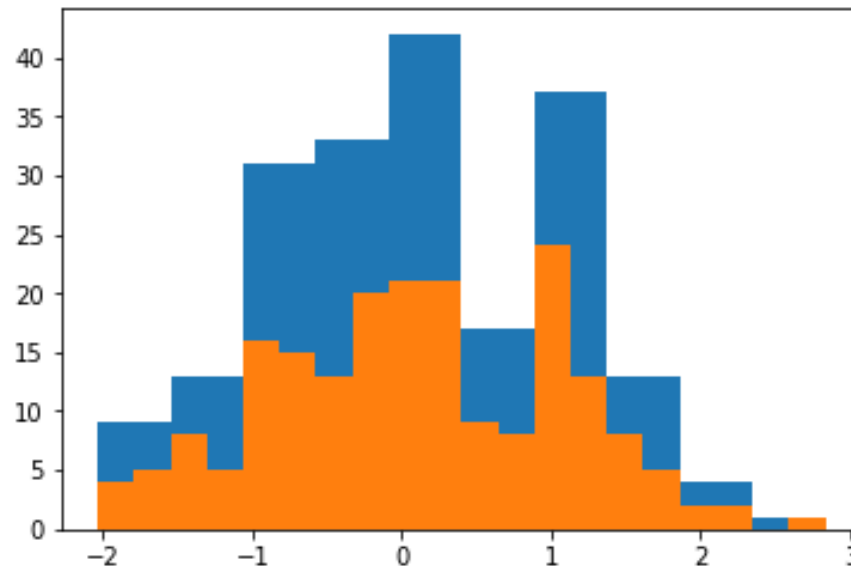




Gráficos estadísticos

```
In [4]: plt.hist(nums)
....:
....: plt.hist(nums,bins=20)
Out[4]:
(array([ 4.,  5.,  8.,  5., 16., 15., 13., 20., 21., 21.,  9.,
        8., 24.,
        13.,  8.,  5.,  2.,  2.,  0.,  1.]),
 array([-2.03729892, -1.79374433, -1.55018975, -1.30663517,
        -1.06308059,
        -0.819526   , -0.57597142, -0.33241684, -0.08886226,
         0.15469232,
         0.39824691,  0.64180149,  0.88535607,  1.12891065,
         1.37246523,
         1.61601982,  1.8595744 ,  2.10312898,  2.34668356,
         2.59023814,
```

Pero se le puede indicar el número de intervalos a dividir el rango e, incluso, pintar los dos histogramas en el mismo gráfico.





Gráficos estadísticos

```
In [2]: nums = np.random.randn(200) # array con 200 números aleatorios
```

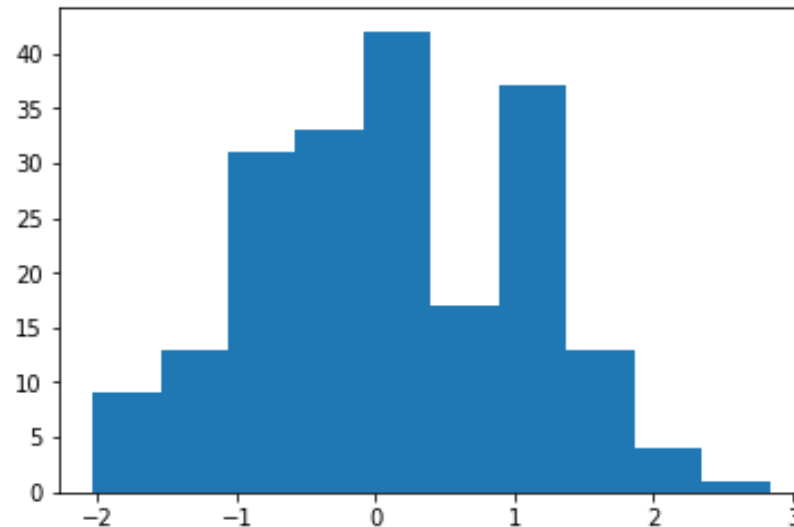
```
....:
```

```
....: # generamos el histograma
```

```
....: plt.hist(nums)
```

```
Out[2]:
```

```
(array([ 9., 13., 31., 33., 42., 17., 37., 13.,  4.,  1.]),  
 array([-2.03729892, -1.55018975, -1.06308059, -0.57597142,  
        -0.08886226,  
         0.39824691,  0.88535607,  1.37246523,  1.8595744 ,  
         2.34668356,  
         2.83379273])),  
<a list of 10 Patch objects>)
```





Gráficos estadísticos

Para variables cualitativas y cuantitativas discretas son también útiles los gráficos llamados tartas o **gráficos de sectores**. Cada sector de la figura tendrá un área proporcional a la frecuencia del cada dato. Los datos se normalizan dividiendo cada uno de ellos por la suma de todos. La tarta estará incompleta si la suma de los datos es inferior a 1.

```
In [1]: import matplotlib.pyplot as plt
```

```
In [2]: etiquetas = ['Perros', 'Gatos', 'Pájaros', 'Caballos']  
...: frecuencias = [15, 30, 45, 10]
```

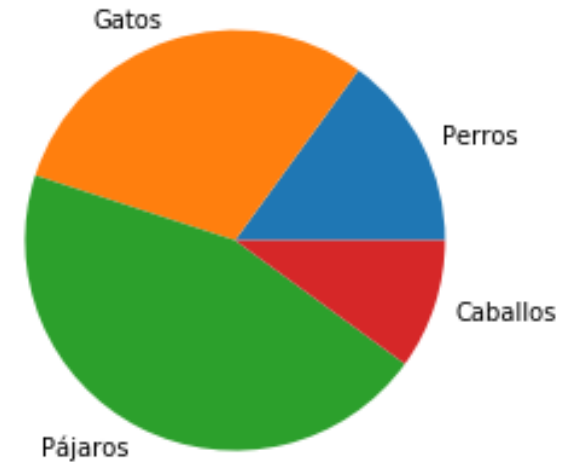
```
In [3]: plt.pie(frecuencias, labels=etiquetas)
```

```
Out[3]:
```

```
In [3]: plt.pie(frecuencias, labels=etiquetas)
```

```
Out[3]:
```

```
([<matplotlib.patches.Wedge at 0x22c622a5550>,  
 <matplotlib.patches.Wedge at 0x22c622a5a90>,  
 <matplotlib.patches.Wedge at 0x22c622a5f28>,  
 <matplotlib.patches.Wedge at 0x22c622b5470>],  
 [Text(0.9801071672559598, 0.4993895680663527, 'Perros'),  
  Text(-0.33991877217145816, 1.046162142464278, 'Gatos'),  
  Text(-0.49938947630209474, -0.9801072140121813, 'Pájaros'),  
  Text(1.0461621822461364, -0.3399186497354948, 'Caballos')])
```





Gráficos estadísticos

```
In [2]: nums = np.random.randn(200) # array con 200 números aleatorios
```

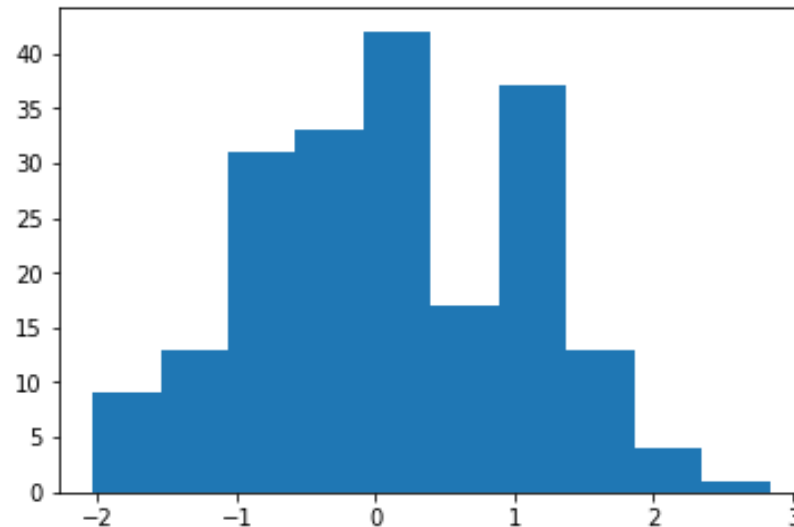
```
....:
```

```
....: # generamos el histograma
```

```
....: plt.hist(nums)
```

```
Out[2]:
```

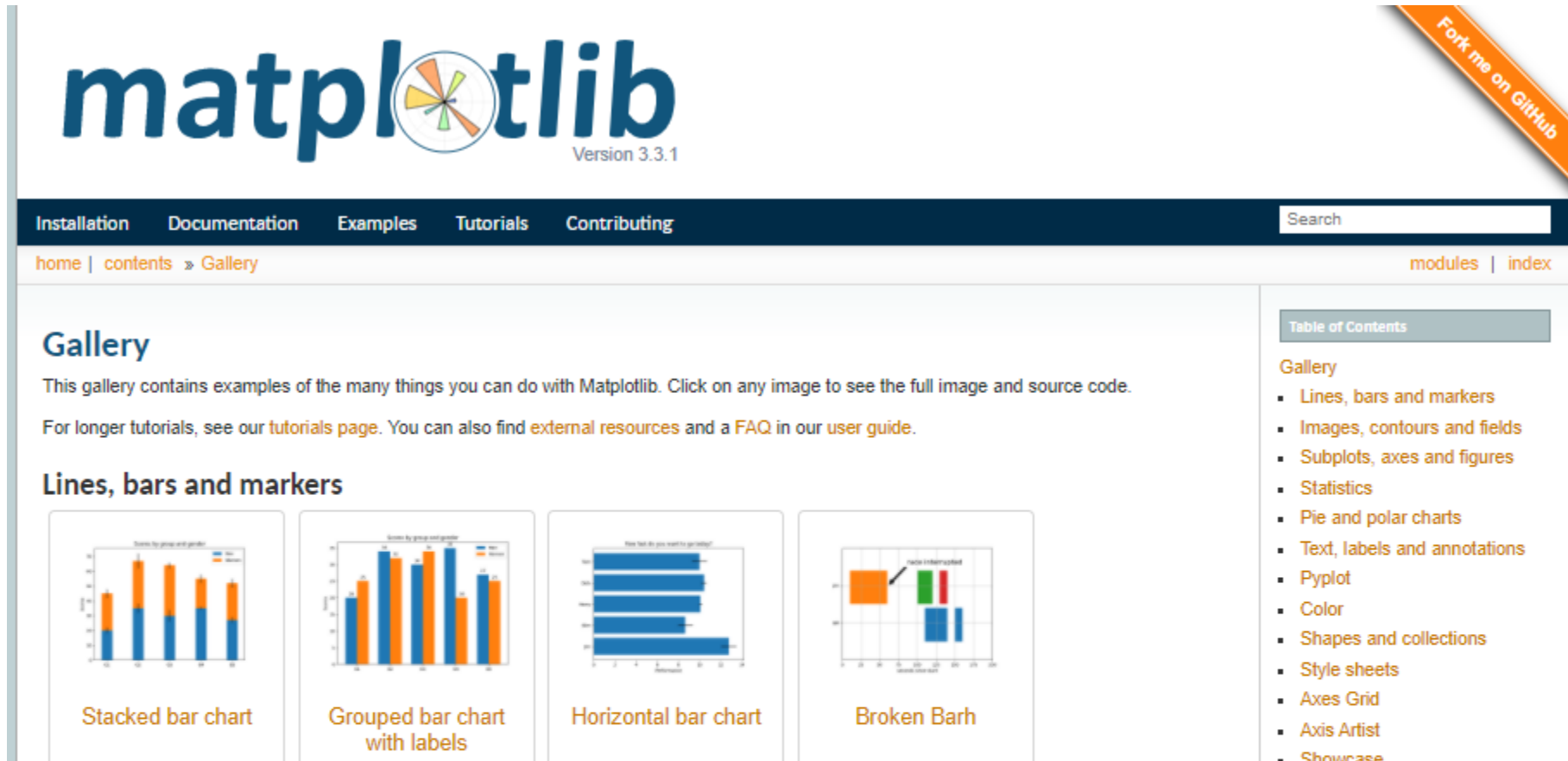
```
(array([ 9., 13., 31., 33., 42., 17., 37., 13.,  4.,  1.]),  
 array([-2.03729892, -1.55018975, -1.06308059, -0.57597142,  
        -0.08886226,  
         0.39824691,  0.88535607,  1.37246523,  1.8595744 ,  
         2.34668356,  
         2.83379273])),  
<a list of 10 Patch objects>)
```





Otros cientos de miles de gráficos

<https://matplotlib.org/gallery/index.html>



The screenshot shows the Matplotlib gallery website. At the top, there's a navigation bar with links for Installation, Documentation, Examples, Tutorials, and Contributing. A search bar is also present. Below the navigation bar, the page title "Gallery" is displayed, followed by a description: "This gallery contains examples of the many things you can do with Matplotlib. Click on any image to see the full image and source code. For longer tutorials, see our [tutorials page](#). You can also find [external resources](#) and a [FAQ](#) in our [user guide](#)."

The main content area features a section titled "Lines, bars and markers" with four example plots:

- Stacked bar chart
- Grouped bar chart with labels
- Horizontal bar chart
- Broken Barh

On the right side, there's a "Table of Contents" section with a list of topics:

- Gallery
 - Lines, bars and markers
 - Images, contours and fields
 - Subplots, axes and figures
 - Statistics
 - Pie and polar charts
 - Text, labels and annotations
 - Pyplot
 - Color
 - Shapes and collections
 - Style sheets
 - Axes Grid
 - Axis Artist
 - Showcase



Las funciones con python™

Carmen Gandía

carmen.gandia@ua.es

Julio Mulero

julio.mulero@ua.es