APUNTES PROGRAMACIÓN PYTHON

1. VARIABLES, CONSTANTES Y TIPOS DE DATOS

Una variable es un elemento cuyo valor puede "variar" a los largo de un programa. El caso contrario lo tenemos en las constantes, las cuales mantienen su valor a lo largo de un programa.

Ambas representan datos que pertenecen a algún tipo: números enteros, reales, caracteres, cadenas de caracteres, etc.

Cada tipo de dato tiene unos límites definidos y debidos al nº de bits empleados para albergarlos. Por ejemplo, si quisiéramos almacenar un número con un bit sólo podría tener dos valores 0 ó 1; si utilizáramos 2 bits tendríamos 4 posibilidades: 00, 01, 10 y 11. Con tres bits, 8 posibilidades y así sucesivamente. Vemos que las posibilidades siempre son de 2^n, siendo n el nº de bits. Si tenemos en cuenta que uno de los bits puede ser utilizado para representar el signo, positivo o negativo, se reducen las posibilidades a la mitad.

En Python no hay que declarar el tipo de dato al asignarlo a una variable o constante sino que el compilador interpreta el tipo de datos según el formato y lo aplica.

Tipos numéricos

Los números se dividen en tres tipos de datos de Python:

- **int / Integer**: Int puede almacenar todos los valores enteros. Este tipo de dato puede ser de cualquier tamaño. No hay límite de tamaño.
- **float**: el **flotante** incluye todos los valores de punto flotante. Tampoco hay restricciones sobre el tamaño de un número de punto flotante.
- **complejo**: como sugiere el nombre, Python almacena los números en forma compleja. Por ejemplo, x + yj, donde x es la parte real e y es la parte imaginaria.

En el siguiente ejemplo vemos cómo se utilizan estas tres variables y además podemos operar entre ellas, eligiendo el compilador el tipo de dato más apropiado al resultado para no perder precisión.

```
variable_entera = 1234
variable_flotante = 3.0275
variable_compleja = 3 - 5j

print (variable_entera + variable_flotante + variable_compleja)
```

El resultado es:

(1240.0275-5j)

Tipos cadena

Una cadena es una secuencia de caracteres. La cadena de caracteres o *string*, queda acotada entre el carácter “ (doble comilla).

Veamos algún ejemplo en el que en la instrucción *print()* concatenamos dos cadenas mediante el operador +:

```
string1 = "Hola"
string2 = "Mundo"
print (string1 + " " + string2)
```

El resultado del programa es:

Hola Mundo

Tipos Booleanos

- **boolean:** tiene la finalidad de facilitar el trabajo con valores "verdadero/falso" (booleanos), resultantes por regla general de evaluar expresiones. Los dos valores posibles de este tipo son true y false.

Veamos el siguiente ejemplo;

```
condicion = True # asigna a la variable condicion el valor True
print (type(condicion)) # imprimimos el tipo de variable que ha quedado definida

if (condicion): # Si la condicion es True imprimirá el mensaje siguiente
    print ("La condición es verdadera")
```

Tipos Range

Un rango es una lista de números en sucesión aritmética. Es decir cada elemento de la lista es igual al anterior más una cantidad fija denominado distancia:

range (valor inicial, valor máximo, distancia)

```
list(range(5, 21, 3))
[5, 8, 11, 14, 17, 20]
```

Donde 5 es el elemento inicial, 21 es el valor máximo y 3 es la distancia entre términos

```
list(range(10, 20, 3))
[10, 13, 16, 19]
```

Si el valor máximo es menor que el inicial la lista quedará vacía

```
list(range(5, 1))
[]
```

Si no se indica la distancia, ésta por defecto es 1

```
list(range(5, 10))
[5, 6, 7, 8, 9]
```

Se pueden concatenar rangos si previamente se convierten en listas:

```
>>> list(range(3)) + list(range(5))
[0, 1, 2, 0, 1, 2, 3, 4]
```

La función len() devuelve la longitud de una cadena, lista o rango:

```
>>> len("mensaje secreto")
15
>>> len(["a","b","c"])
3
>>> len(range(1, 100, 7))
15
```

2. ENTRADA DE DATOS POR TECLADO: LA FUNCIÓN input()

La función input() permite introducir datos por el teclado, Veámoslo con los siguientes ejemplos:

```
print ("¿Cómo te llamas?")
nombre = input()
print("Me alegro de saludarte", nombre)
```

La salida del programa será:

```
¿Cómo te llamas?
Pedro
Me alegro de saludarte Pedro
```

La función input() admite pasarle argumentos con lo cual el programa queda más compacto:

```
nombre = input ("¿Cómo te llamas? ")
print("Me alegro de saludarte", nombre)
```

El resultado será:

```
¿Cómo te llamas? Pedro
Me alegro de saludarte Pedro
```

La función input convierte el dato recibido en cadena de caracteres. Por esta razón si queremos operar con valores numéricos hay que hacer una conversión de tipos mediante las funciones *int ó float.*

Veamos los siguientes ejemplos:

```
edad = int(input("¿Cuántos años tienes? "))
print("Te faltan ", 100-edad, " años para llegar a los 100 años")
```

El resultado del programa será:

```
¿Cuántos años tienes? 60
Te faltan  40  años para llegar a los 100 años
```

3. SALIDA DE DATOS POR PANTALLA: LA FUNCIÓN print()

La función print() permite mostrar información por pantalla.

Permite imprimir datos, variables y constantes. Las cadenas de caracteres se delimitan por dobles o simples comillas.

Por defecto la función print() salta de línea al terminar. Si queremos que el terminador no sea el retorno de carro lo podemos especificar mediante el parámetro *end*.

Veamos los siguientes ejemplos:

```
print("Hola")
print ("Pedro")
```

El resultado será:

```
Hola
Pedro
```

Sin embargo si realizamos el siguiente programa haciendo uso del parámetro "end":

```
print("Hola", end=" ")
print ("Pedro")
```

El resultado es: *Hola Pedro*

La función print permite combinar todo tipo de datos:

```
edad=18
print("Hola Pedro tienes", edad, "años")
```

El resultado es: *Hola Pedro tienes 18 años*

4. CONTROL DE FLUJO: EL CONDICIONAL “IF” y operadores

A la hora de tomar decisiones condicionadas dentro de un programa utilizamos la función **if** cuyo formato es:

```
if condición_1:
   bloque 1
elif condición_2:
   bloque 2
else:
```

Las condiciones de decisión arrojan como resultado un boolean (True/False) y en función de la veracidad de la condición se elegirá un camino u otro.

Es muy importante definir correctamente la sangría: Cada bloque de instrucciones ha de estar en el mismo nivel de sangrado.
Se recomienda utilizar siempre el mismo número de espacios en el sangrado.

Si quisiéramos que en un bloque no se hiciera ninguna acción utilizaríamos la instrucción **pass**

```
edad=int(input("¿Cuántos años tiene? ")) #preguntamos la edad y convertimos el string a número entero
if (edad<18): # podemos poner la condición entre paréntesis
   pass
else:
   print ("Es usted mayor de edad")
   print ("Es usted",(edad-18),"años mayor de edad") #comienza la tercera sangría
```

Operadores:

Para calcular si la (condición) se cumple o no, es decir si es verdadera o falsa podemos utilizar los siguientes operadores:

- **==**, Igual a: devuelve verdadero si el valor del lado izquierdo es igual al lado derecho.

- **! =**, No igual a: devuelve verdadero si el valor del lado izquierdo no es igual al lado derecho.
- **<**, menos que: el resultado verdadero si el valor del lado izquierdo es inferior al del lado derecho.
- **<=**, menor o igual que: devuelve verdadero si el valor del lado izquierdo es menor o igual que el lado derecho.
- **>**, Mayor que: devuelve verdadero si el valor del lado izquierdo es mayor que el lado derecho.
- **> =**, Mayor que o igual a: regresa verdadero si el valor del lado izquierdo es mayor o igual que el lado derecho.

Además podemos utilizar operadores lógicos para chequear varias condiciones:

- **&&, AND lógico:** devuelve verdadero cuando ambas condiciones son verdaderas.
- **|| , O lógico:** devuelve verdadero si al menos una condición es verdadera.

Ej. (3<5) && (2>1) devolverá un valor true, puesto que ambas condiciones son ciertas.

(3<5) || (2>5) devolverá un valor true, puesto que al menos una de ellas es cierta.

```
valor1 = 15;
valor2 = 8;
if ( (valor1 > 10) && (valor2 >10) ):
        .print(“Ambos valores son mayores de 10”);
```

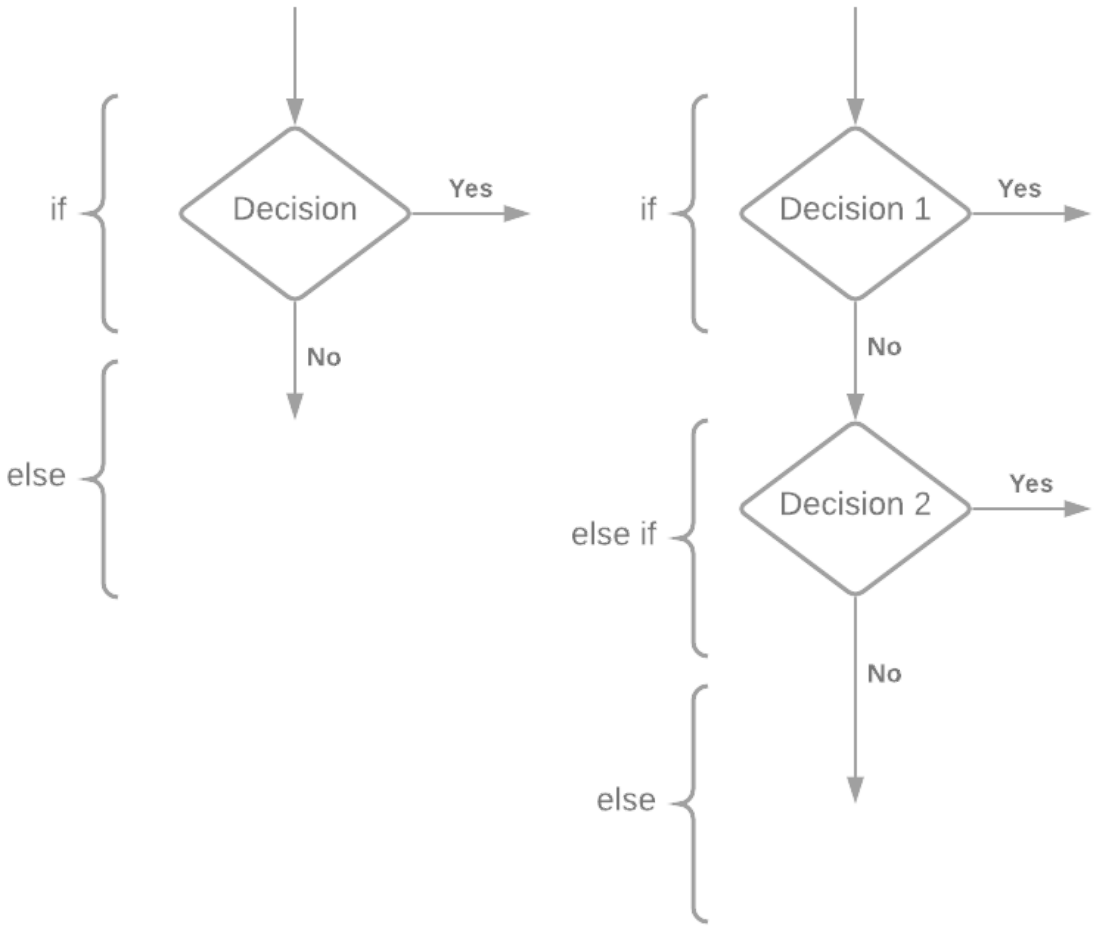
Veamos un ejemplo:

```
edad=int(input("¿Cuántos años tiene? ")) #preguntamos la edad y convertimos el string a número entero
if (edad<18): # podemos poner la condición entre paréntesis
   print ("Es usted menor de edad") #Comienza la primera sangría
   print ("Le faltan a usted",(18-edad),"años para ser mayor de edad")
elif edad==18:
    print ("Acaba de cumplir usted la mayoría de edad") # comienza la segunda sangría
else:
   print ("Es usted mayor de edad")
   print ("Es usted",(edad-18),"años mayor de edad") #comienza la tercera sangría
```

Diagrama de flujo:

A la hora de programar es necesario realizar previamente el diagrama de flujo del programa a realizar.

El formato de diagrama de flujo para el condicional if es el siguiente:



5. OPERADORES

- ## Operadores aritméticos:

  Se utilizan para realizar operaciones aritméticas simples.

  - *: Multiplicación
  - /: División
  - %: Módulo
  - +: Adición
  - –: Resta
  - **: Potencia
  - //: División entera

- ## Operadores unarios:

  Los operadores unarios solo necesitan un operando. Se usan para incrementar, disminuir o negar un valor.

  - –: Unario menos, utilizado para negar los valores.
  - +: Unario más, usado para dar valores positivos. Solo se usa cuando se convierte deliberadamente un valor negativo en positivo.
  - ++: Operador de incremento, utilizado para incrementar el valor en 1. Hay dos variedades de operador de incremento.
    - Pre-Incremento: el valor se incrementa primero y luego se calcula el resultado.
    - Post-Incremento: el valor se usa por primera vez para calcular el resultado y luego se incrementa.
  - —: Operador de decremento , usado para incrementar el valor en 1. Hay dos variedades de operador de incremento.
    - Pre-Decremento: el valor se disminuye primero y luego se calcula el resultado.
    - Post-Decremento: el valor se usa por primera vez para calcular el resultado y luego se disminuye.
  - ! : Operador lógico “no”, utilizado para invertir un valor booleano.

- ## Operador de Asignación:

  El operador de asignación se usa para asignar un valor a cualquier variable. Tiene una asociación de derecha a izquierda, es decir, el valor dado en el lado derecho del operador se asigna a la variable de la izquierda y, por lo tanto, el valor del lado derecho debe declararse antes de usarlo o debe ser una constante.

  El formato general del operador de asignación es, a = b;

  En muchos casos, el operador de asignación se puede combinar con otros operadores para construir una versión más corta de la declaración llamada Declaración Compuesta (Compound Statement). Por ejemplo, en lugar de a = a + 5, podemos escribir a + = 5.

  int a = 5;

  a += 5; // a = a + 5;

  - + = , para sumar el operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.
  - – = , para restar el operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.
  - * = , para multiplicar el operando izquierdo con el operando derecho y luego asignándolo a la variable de la izquierda.
  - / = , para dividir el operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.
  - ^ = , para aumentar la potencia del operando izquierdo al operando derecho y asignarlo a la variable de la izquierda.
  - % = , para asignar el módulo del operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.

6. CONTROL DE FLUJO. EL BUCLE FOR

El comando **for** permite la creación de un bucle con un cierto número de repeticiones o iteraciones. Para ello se define:

```
for variable in elemento iterable (lista, cadena, range, etc.):
    cuerpo del bucle
```

- Utilizando un rango

```
for i in range(5):
   print (i)
```

El resultado del programa es la repetición del bucle 5 veces, pero en un rango el primer elemento es cero.:

```
0
1
2
3
4
```

```
for i in range(5):
   print (“Hola”)
```

El resultado del programa es la repetición del bucle cinco veces.:

```
Hola
Hola
Hola
Hola
Hola
```

- Utilizando una lista

Para definir una lista basta con: lista = [“Hola”, “cinco”, 3, 2.27]

Una lista puede almacenar datos de diferente tipo como podemos ver.

```
for i in[1,2,3]:
   print (“Hola”)
```

El resultado del programa es la repetición del bucle tres veces.:

```
Hola
Hola
Hola
```

Podemos hacer uso de la variable de control y de los valores que va tomando:

```
for i in[1,2,3]:
   print (“Hola”,i)
```

El resultado del programa es:

```
Hola 1
Hola 2
Hola 3
```

Otro ejemplo:

```
for i in[1,1,1]:
   print (“Hola”)
```

El resultado del programa es la repetición del bucle tres veces, aunque los elementos de la cadena sean los mismos. Lo que importa es el nº de términos en este caso.:

```
Hola
Hola
Hola
```

Cuando la variable de control “i” no va a ser utilizada se puede sustituir por un guión bajo

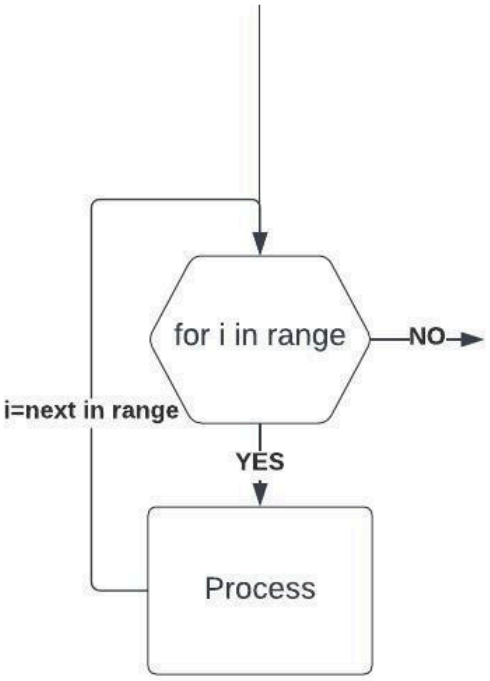
```
for _ in[1,1,1]:
   print (“Hola”)
```

El resultado del programa es la repetición del bucle tres veces, aunque los elementos de la cadena sean los mismos. Lo que importa es el nº de términos en este caso.:

```
Hola
Hola
```

Hola

El diagrama de flujo del bucle for sería de la siguiente manera:



7. MODIFICACIONES EN LOS BUCLES: CONTINUE, BREAK

Hay varias formas de cambiar el flujo de un bucle en función de las condiciones que se vayan produciendo, condiciones que normalmente son comprobadas mediante el uso del ***if***

- ### Comando Continue:

  El comando continue permite saltar toda la iteración actual del bucle para pasar a la siguiente, pero sin llegar a romper el bucle.

  Por ejemplo, el siguiente programa permite eliminar de una palabra la vocal “e”

```
cadena = “Abecedario”
for letra in cadena:
```

```
    if letra == “e”:
        continue
    print(letra, end= “ “)
```

Salida:

```
# A
# b
# c
# d
# a
# r
# i
# o
```

- Comando Break:

El comando break, a diferencia del ***continue***, permite romper el bucle, saltándolo por completo y siguiendo con el resto del programa.

El programa anterior cambia completamente su comportamiento mediante el uso de ***break***

```
cadena = “Abecedario”
for letra in cadena:
    if letra == “e”:
        break
    print(letra)
```

Salida:

```
# A
# b
```

El comando break sólo afecta al bucle en el que se encuentra. Esto conlleva que si tenemos dos bucles anidados, como en el siguiente ejemplo, sólo afectará al interior.

```
for i in range(5):
    for j in range(5):
        if j == 2:
         break
        print(i,j)
```

```
0 0
0 1
1 0
1 1
2 0
2 1
3 0
3 1
4 0
4 1
```

8. BUCLE WHILE

Otra forma de controlar el flujo de un programa es mediante la instrucción while. Esta instrucción permite que se ejecute un bucle **mientras (while)**, la condición que evalúa el while sea true.

```
x = 10
while x > 0:
   x -=1
   print(x, end=" ")
```

```
9 8 7 6 5 4 3 2 1 0
```

Podemos hacer uso de la función **else**, la cual se cumplirá si el bucle termina de forma normal, llegando hasta el final (y no si termina con un break).

```
x = 10
while x > 0:
   x -=1
   print(x, end=" ")
else:
   print ("El bucle ha llegado al final")
```

```
9 8 7 6 5 4 3 2 1 El bucle ha llegado al final
```

Si quisiéramos hacer un bucle infinito bastaría con:

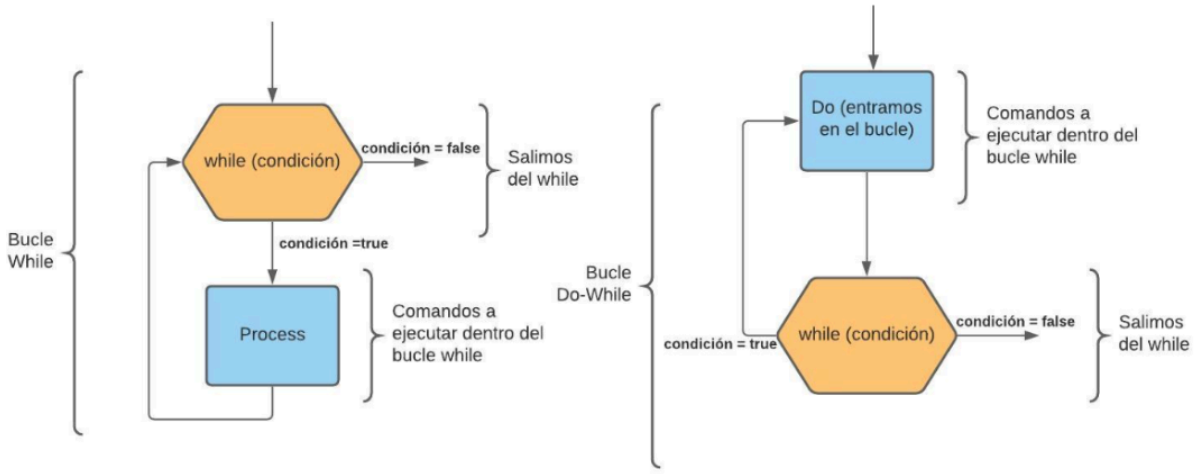
```
while True:
   print("x", end=" ")
```

Espero que sepáis disculpar que no llene el documento de “x”

¿Sabrías decir qué hará el siguiente programa?

```
text = "Escolapios"
i = 0
while i < len(text):
    print(text[:i + 1])
    i += 1
```

El diagrama de flujo del bucle while y do-while es el siguiente:



9. FUNCIONES

Las funciones permiten automatizar tareas que requerimos con frecuencia y que además se pueden generalizar por medio de parámetros o argumentos.
Son un conjunto de líneas de código (instrucciones), encapsulados en un bloque, usualmente reciben parámetros, cuyos valores se utilizan para efectuar operaciones y adicionalmente pueden retornar un valor. En otras palabras una función puede recibir parámetros o argumentos (algunas no reciben nada), hace uso de dichos valores recibidos como sea necesario y retorna un valor usando la instrucción *return*.
La sintaxis para declarar una función es muy simple, veamos:

```
def nombreFuncion(argumento1, argumento2, ...):
        Bloque de instrucciones
```

Veamos algunos ejemplos:

```
def suma (a,b):
```

```
    return a+b
```

Cuando invoco a la función (la función tiene que estar declarada antes de su uso).:

`print (suma(3,2))` imprime 5

Una función no tiene porqué devolver un valor o recibir un parámetro:

```
def escribe_HolaMundo():
  print("HolaMundo")
```

`escribe_HolaMundo()` escribe por pantalla HolaMundo

10. OPERACIONES CON STRINGS (LISTAS)[1]

En ocasiones es necesario operar con los elementos de un string o una lista de modo que podamos ordenar, añadir, quitar o insertar elementos en un lugar determinado.

Los operadores más utilizados son los siguientes:

1. Operaciones de definición

Estas operaciones nos permiten definir o crear una lista.

a. [ ]

Crea una lista vacía.

```
y = []
```

Para copiar una lista hemos de llevar sumo cuidado porque la sintaxis es diferente a otros lenguajes de programación.

Si por ejemplo tengo una lista que es

```
lista1 = [1, 2,3,4]
```

Si ejecuto lista2=lista1 lo que realmente es asignar un nombre diferente a lista1, pero no he creado una lista diferente. Es decir, si modifico el elemento N de lista2 también queda modificado el de lista1

```
lista2[0] = 6
```

Entonces lista1=[6,2,3,4]

Para copiar una lista en otra diferente he de hacer:

[1] Fuente: “Operaciones sobre listas en Python” de Hanzel Godinez H.

```
lista3=lista1[:]
```

Si ahora modifico un elemento de lista3 no se modifica ninguno en lista1

2. Extracción de partes de la cadena

trozo[inicio:fin:paso]

Esta estructura permite extraer una serie de elementos del string, comenzando por el numerado como inicio y terminando por el numerado como fin-1, aumentando de paso en paso.

Si omites inicio se empezará en el primer elemento del iterable, si omites fin se terminará en el último elemento del iterable.

Si el paso es negativo, el iterable se recorre "hacia atrás", y en ese caso los valores por defecto cuando se omite inicio y fin se invierten.

Así pues trozo[::-1] devuelve los elementos del iterable, comenzando por el último y terminando por el primero, en orden inverso a como estaban.

```
Ej
frase="Hola Mundo"
print (frase[2:9:1])
print (frase)

Output:
la Mund
Hola Mundo
```

3. Operaciones mutables

Estas operaciones nos permiten trabajar con listas alterando o modificando su definición previa.

a. append

Añade un único elemento al final de la lista.

```
x = [1, 2]
x.append('h')
print(x)
Output:
[1, 2, 'h']
```

b. extend

Añade otra lista al final de una lista.

```
x = [1, 2]
x.extend([3, 4])
print(x)
Output:
[1, 2, 3, 4]
```

c. insert

Inserta un nuevo elemento en una posición determinada de la lista, este método recibe un primer argumento que equivale a la posición, y un segundo argumento que corresponde al elemento por agregar.

```
x = [1, 2]
x.insert(0, 'y')
print(x)
Output:
['y', 1, 2]
```

d. del

Elimina el elemento ubicado en el índice descrito, a su vez este método tiene la capacidad de eliminar una sección de elementos de la lista, mediante el operador ":" que permite definir un punto de [inicio:fin] sin considerar el punto final, esto puntos pueden ser obviados, con lo cual como punto de inicio se tomará la posición 0 y punto de fin la última posición de la lista.

```
x = [1, 2, 3]
del x[1]
print(x)
Output:
[1, 3]
y = [1, 2, 3, 4, 5]
del y[:2]
print(y)
Output:
[3, 4, 5]
```

e. remove

Remueve la primer coincidencia del elemento especificado.

```
x = [1, 2, 'h', 3, 'h']
x.remove('h')
print(x)
Output:
[1, 2, 3, 'h']
```

f. reverse

Invierte el orden de los elementos de la lista, dando como resultado una lista con los elementos del final al inicio y del inicio al final.

```
x = [1, 2, 'h', 3, 'h']
x.reverse()
print(x)
Output:
['h', 3, 'h', 2, 1]
```

g. sort

Por defecto este método ordena los elementos de la lista de menor a mayor, este comportamiento puede modificarse mediante el parámetro reverse=True.

```
x = [3, 2, 1, 4]
x.sort()
print(x)
Output:
[1, 2, 3, 4]
y = ['R', 'C', 'Python', 'Java', 'R']
y.sort(reverse=True)
print(y)
Output:
['R', 'R', 'Python', 'Java', 'C']
```

Es importante considerar que al aplicar el método sort, debes realizarlo sobre listas que contengan elementos del mismo tipo de dato, de lo contrario te enfrentarás a la excepción TypeError.

Otra de las bondades del método sort, es que permite el ordenamiento personalizado de una lista, con lo cual bastaría con definir una función que nos permita ordenar bajo algunos criterios, para posteriormente utilizar la misma mediante el argumento key. Debes tener en cuenta que por lo general este ordenamiento es menos eficiente que el orden default.

Ejemplo de ordenamiento customizado:

```
def ordenar_por_longitud(str):
  return len(str)

x = ['Python', 'es', 'lo', 'mejor']
x.sort(key=ordenar_por_longitud)
print(x)
```

4. Operaciones inmutables

Estas operaciones nos permiten trabajar con listas sin alterar o modificar su definición previa.

a. sorted

Este método nos permite ordenar los elementos de una lista de menor a mayor, al igual que el método sort, este comportamiento puede modificarse mediante el parámetro reverse=True.

```
x = [5, 2, 9, 0]
print(sorted(x))
print(x)
Output:
[0, 2, 5, 9]
[5, 2, 9, 0]
```

Algo interesante del método sorted, es que este no se encuentra limitado a las listas, sino que a su vez es aplicable a las tuplas y los diccionarios, algo que en otro artículo estaremos abordando.

b. +

Esta operación nos permite concatenar o unir dos listas diferentes en nueva lista.

```
x = [1, 2, 3]
y = [4, 5, 6]
print(x + y)
print(x)
print(y)
Output:
[1, 2, 3, 4, 5, 6]
[1, 2, 3]
[4, 5, 6]
```

c. *

Esta operación nos permite replicar una lista hasta la cantidad de veces indicada.

```
x = [1, 2, 3]
print(x * 3)
print(x)
Output:
[1, 2, 3, 1, 2, 3, 1, 2, 3]
[1, 2, 3]
```

d. min

Este método retorna el elemento más pequeño dentro de una lista.

```
x = [40, 100, 3, 9, 4]
print(min(x))
print(x)
Output:
3
[40, 100, 3, 9, 4]
```

e. max

Al contrario del metodo min, este método retorna el elemento más grande dentro de una lista.

```
x = [40, 100, 3, 9, 4]
print(max(x))
print(x)
Output:
100
[40, 100, 3, 9, 4]
```

f. index

Retorna la posición en la lista del elemento especificado.

```
x = [10, 30, 20]
print(x.index(30))
print(x)
Output:
1
[10, 30, 20]
```

g. count

Retorna la cantidad de veces que el elemento especificado se encuentra en la lista.

```
x = [10, 30, 20, 30, 30]
print(x.count(30))
print(x)
Output:
3
[10, 30, 20, 30, 30]
```

h. sum

Este método realiza la suma de los elementos de la lista, esto siempre que los mismos puedan ser sumados. Sum es un método muy utilizado con listas de tipo numéricas.

```
x = [2.5, 3, 3.5]
print(sum(x))
print(x)
Output:
9.0
[2.5, 3, 3.5]
```

i. in

El método nos permite determinar si un elemento especifico se encuentra en una lista, este retorna únicamente dos posibles valores True cuando el elemento se encuentre en la lista, y False cuando este no lo este. Este método es ampliamente utilizado para evitar excepciones en métodos tales como: index y remove, ya que en caso que el elemento buscado no se encuentre en la lista, dará como resultado una excepción.

```
x = ['h', 2, 'a', 6, 9]
print('a' in x)
print(x)
Output:
True
['h', 2, 'a', 6, 9]
```

11. OPERACIONES CON ASCII

El código ASCII es un código internacional que asigna un código concreto a cada uno de los caracteres:

Binario	Dec	Hex	Representación
0010 0000	32	20	espacio ( )
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?

Binario	Dec	Hex	Representación
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_

Binario	Dec	Hex	Representación
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h

0110 1001	105	69	i	0111 0100	116	74	t
0110 1010	106	6A	j	0111 0101	117	75	u
0110 1011	107	6B	k	0111 0110	118	76	v
0110 1100	108	6C	l	0111 0111	119	77	w
0110 1101	109	6D	m	0111 1000	120	78	x
0110 1110	110	6E	n	0111 1001	121	79	y
0110 1111	111	6F	o	0111 1010	122	7A	z
0111 0000	112	70	p	0111 1011	123	7B	{
0111 0001	113	71	q	0111 1100	124	7C	|
0111 0010	114	72	r	0111 1101	125	7D	}
0111 0011	115	73	s	0111 1110	126	7E	~

Esta codificación universal permite programar de forma flexible con dichos caracteres.

Los comandos básicos son chr y ord.

chr(n) me devuelve el carácter cuyo código decimal es n y ord("CHAR") me devuelve el código decimal del carácter CHAR.

Ej:

```
print (chr(75))
print (ord("K"))
```

Salida:

```
K
75
```

Un ejemplo clásico es el de la conversión de mayúsculas a minúsculas y viceversa pues entre una letra minúscula y su correspondiente mayúscula tan sólo hay una diferencia de 32 entre sus códigos decimales.

12. DICCIONARIOS

En Python podemos crear un diccionario con una serie de elementos a los que corresponde una definición dada. De esta forma podemos hacer uso del diccionario en programas más complejos, por ejemplo para una traducción o codificación.

El formato es:

```
diccionario ={'gato':'cat', 'perro':'dog', 'caballo':'horse'}
```

```
print(diccionario.get('gato'))
>>> cat
```

Es similar a las listas, con el matiz de que dado que los diccionarios no tienen orden, no tienen funciones en las que se tenga en cuenta la posición.

- diccionario.get('key'): Devuelve el valor que corresponde con la key introducida.
- diccionario.pop('key'): Devuelve el valor que corresponde con la key introducida, y luego borra la key y el valor.
- diccionario.update({'key':'valor'}): Inserta una determinada key o actualiza su valor si ya existiera.
- «key» in diccionario: Devuelve verdadero (True) o falso (False) si la key (no los valores) existe en el diccionario.
- «definicion» in diccionario.values(): Devuelve verdadero (True) o falso (False) si definición existe en el diccionario (no como key).

13. MÉTODOS PYTHON ÚTILES PARA TRABAJAR CON CADENAS

En Python podemos encontrar los siguientes métodos o funciones para trabajar con strings. El formato de uso es: string.metodo()

Veamos un ejemplo de uso con el método capitalize, que permite poner en mayúsculas el inicio de la frase.

```
frase = "buenos días"

result = frase.capitalize()

print (result)

>> Buenos días
```

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and

	returns the position of where it was found. Returns -1 if not found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isascii()	Returns True if all characters in the string are ascii characters
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Converts the elements of an iterable into a string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and

	returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

14. MATRICES

Una matriz de 2 dimensiones no es más que una lista de elementos donde cada elemento es una nueva lista.

Ej:

```
linea0=[0,2,0,4,0]
linea1=[1,0,3,5,0]
linea2=[0,0,0,9,0]
linea3=[0,2,0,0,0]
linea4=[8,0,1,1,0]

matriz=[linea0,linea1,linea2,linea3,linea4]
```

El siguiente programa crea una matriz de datos aleatorios (hacemos uso de la librería random; la función randrange elige un número aleatorio dentro del rango seleccioando), busca los ceros y los sustituye por 'X'

```
import random
```

```
linea0=[0,0,0,0,0]
linea1=[0,0,0,0,0]
linea2=[0,0,0,0,0]
linea3=[0,0,0,0,0]
linea4=[0,0,0,0,0]

matriz=[linea0,linea1,linea2,linea3,linea4]

for i in range(5):
    for j in range(5):
        matriz[i][j]= random.randrange(0,10)

print("Esta es la matriz sin sustituir 0: ")

for i in range(5):
    print(matriz[i])

for i in range(5):
    for j in range(5):
        if matriz[i][j] == 0:
            matriz[i][j] = "x"

print("Esta es la matriz sustituyendo 0 por x: ")
for i in range(5):
    print(matriz[i])
```

Salida:

```
Esta es la matriz sin sustituir 0:
[9, 9, 3, 2, 0]
[6, 0, 9, 3, 1]
[5, 9, 5, 6, 9]
[3, 1, 0, 2, 4]
[0, 8, 6, 2, 8]
Esta es la matriz sustituyendo 0 por x:
[9, 9, 3, 2, 'x']
[6, 'x', 9, 3, 1]
[5, 9, 5, 6, 9]
[3, 1, 'x', 2, 4]
['x', 8, 6, 2, 8]


** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

15. GRÁFICOS CON MATPLOTLIB[2]

Para generar gráficos haremos uso de las funciones contenidas en la librería Matplotlib:

Seguimos nuestro viaje en la exploración de las librerías para análisis de datos que el ecosistema de Python nos ofrece. Hasta ahora hemos visto cómo empezar a trabajar con Python y las librerías de Numpy y Pandas. A esta lista agregamos la librería de visualización Matplotlib, la cual exploramos en detalle en este post y que nos va a permitir crear gráficos de calidad que podemos usar para comunicar nuestros descubrimientos, hipótesis y conclusiones.

Para empezar a trabajar con Matplotlib , tenemos que importarlo.

```
1 import matplotlib
```

💡 Puedes instalar la librería con el comando pip install matplotlib o conda install matplotlib dependiendo de tu instalación de Python .

Al usar Matplotlib en la línea de comandos o en un script de Python , las gráficas nos aparecerán en una ventana separada. Sin embargo, al trabajar con notebooks, podemos visualizar nuestros gráficos directamente en el documento (lo cual es ideal a la hora de generar informes con código y visualizaciones en un solo documento). Ésto lo indicamos con el siguiente comando.

```
%matplotlib inline
```

a. Nuestro primer gráfico

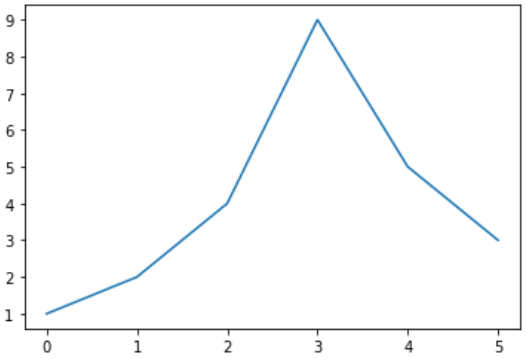
En el siguiente ejemplo puedes ver como genera un sencillo gráfico. Para ello

[2] https://colab.research.google.com/github/juansensio/blog/blob/master/022_matplotlib/matplotlib.ipynb

simplemente tenemos que llamar a la función plot

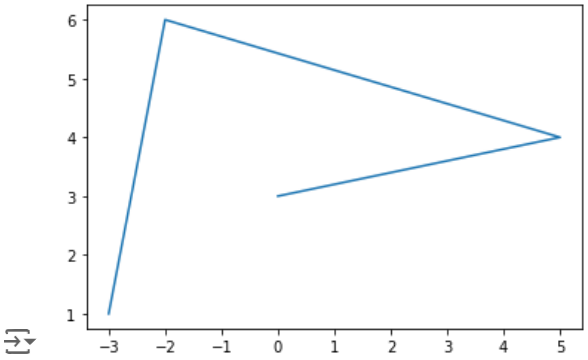
con una lista de valores para generar la imagen y show para visualizarla.

```
1 import matplotlib.pyplot as plt 2
3 plt.plot([1, 2, 4, 9, 5, 3])
4 plt.show()
```



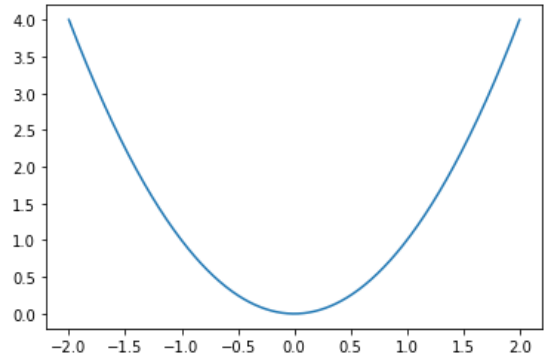
Como puedes ver, los valores de la lista se usan para el eje y mientras que en el eje x se utiliza el índice de cada valor en la lista. Podemos pasar dos listas a la función plot para indicar los valores en ambos ejes.

```
plt.plot([-3, -2, 5, 0], [1, 6, 4, 3])
plt.show()
```



Puedes visualizar cualquier función matemática generando una lista con los valores de las variables independientes y calculando los valore de las variables dependientes con Numpy de la siguiente manera

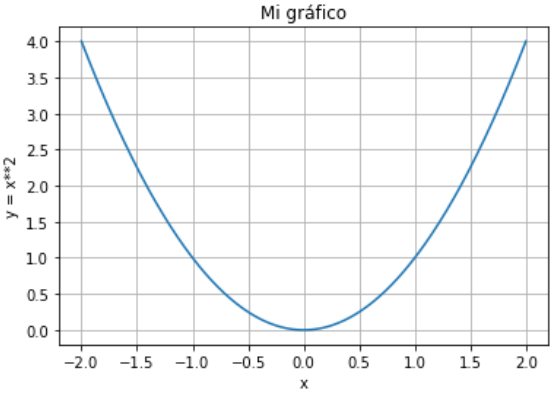
```
import numpy as np 2
x = np.linspace(-2, 2, 500)
y = x**2
plt.plot(x, y)
plt.show()
```



b. Customizando un gráfico

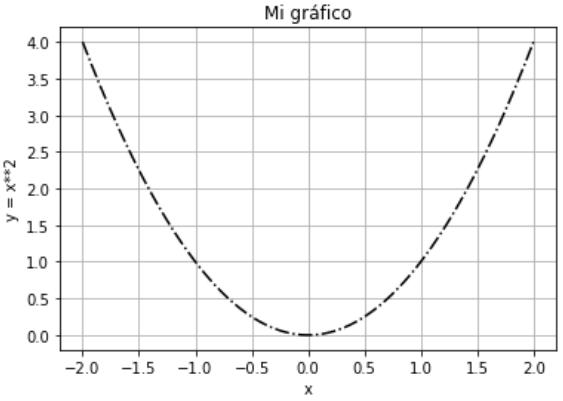
Ahora que conocemos la sintaxis básica para crear y visualizar un gráfico con Matplotlib vamos a ver diferente funcionalidad que nos ofrece para tunear nuestro gráfico y hacerlo más bonito. Podemos empezar añadiendo un título y etiquetas a nuestros ejes. En ocasiones también es interesante añadir una cuadrícula en el fondo para mejorar la lectura de datos.

```
plt.plot(x, y)
plt.title("Mi gráfico")
plt.xlabel("x")
plt.ylabel("y = x**2")
plt.grid(True)
plt.show()
```



Podemos cambiar el estilo de la línea de manera sencilla pasando diferentes opciones en la función plot , como un string que incluye el formato y color. Puedes encontrar una lista con los diferentes estilos en la documentación.

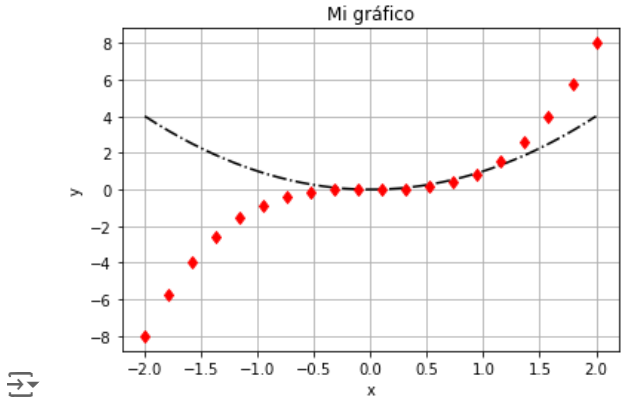
```
# línea negra punteada
plt.plot(x, y, '-.k')
plt.title("Mi gráfico")
plt.xlabel("x")
plt.ylabel("y = x**2")
plt.grid(True)
plt.show()
```




```
# diamantes rojos
x2 = np.linspace(-2, 2, 20)
y2 = x2**3
plt.plot(x2, y2, 'dr')
plt.title("Mi gráfico")
plt.xlabel("x")
plt.ylabel("y = x**3")
plt.grid(True)
plt.show()
```

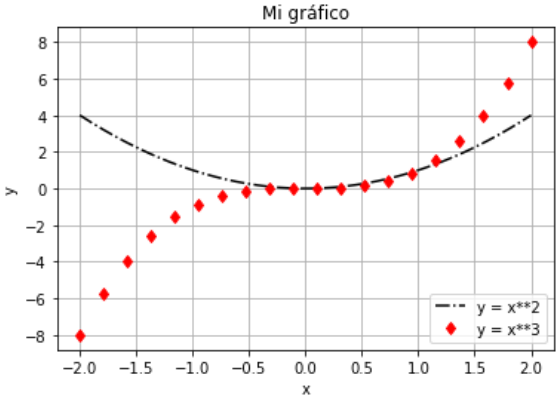
Podemos añadir varias curvas en la misma gráfica simplemente llamando a la función plot varias veces.

```
plt.plot(x, y, '-.k')
plt.plot(x2, y2, 'dr')
plt.title("Mi gráfico")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.show()
```



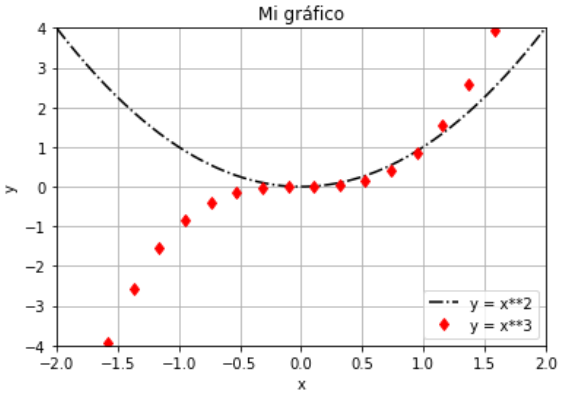
Añadiendo una etiqueta a cada curva podemos generar una leyenda, la cual podemos situar en diferentes posiciones.

```
plt.plot(x, y, '-.k', label="y = x**2")
plt.plot(x2, y2, 'dr', label="y = x**3")
plt.title("Mi gráfico")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.legend(loc='lower right')
plt.show()
```



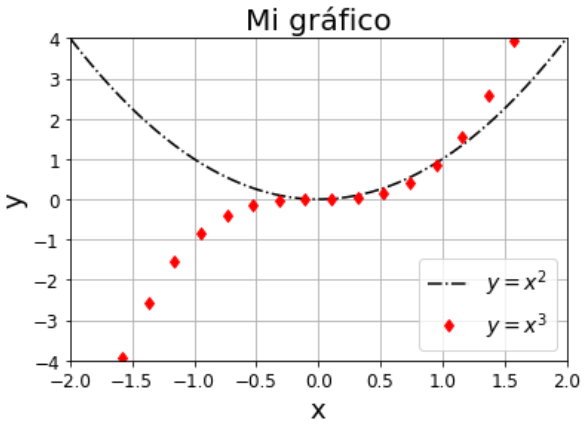
Con la función axis podemos delimitar la extensión de los ejes.

```
plt.plot(x, y, '-.k', label="y = x**2")
plt.plot(x2, y2, 'dr', label="y = x**3")
plt.title("Mi gráfico")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.legend(loc='lower right')
plt.axis([-2,2,-4,4])
plt.show()
```



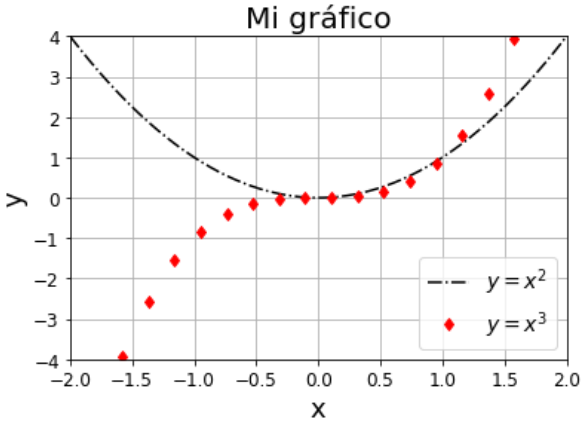
Por último vamos a ver cómo cambiar el tamaño del texto para que sea más legible. También podemos usar notación Latex para introducir ecuaciones y símbolos en nuestros títulos, ejes y leyendas.

```
plt.plot(x, y, '-.k', label="$y = x^2$")
plt.plot(x2, y2, 'dr', label="$y = x^3$")
plt.title("Mi gráfico", fontsize=20)
plt.xlabel("x", fontsize=18)
plt.ylabel("y", fontsize=18)
plt.grid(True)
plt.legend(loc='lower right', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.axis([-2,2,-4,4])
plt.show()
```



Ahora que tenemos nuestro gráfico listo, podemos guardarlo en un archivo simplemente sustituyendo la función show por savefig .

```
plt.plot(x, y, '-.k', label="$y = x^2$")
plt.plot(x2, y2, 'dr', label="$y = x^3$")
plt.title("Mi gráfico", fontsize=20)
plt.xlabel("x", fontsize=18)
plt.ylabel("y", fontsize=18)
plt.grid(True)
plt.legend(loc='lower right', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.axis([-2,2,-4,4])
plt.savefig("mi_grafico.png", transparent=True)
```



Además de las opciones de customización que hemos visto existen muchas otras que podrás encontrar en la documentación, la cual incluye multitud de ejemplos útiles.

c. Múltiples gráficos

Una funcionalidad muy útil que Matplotlib nos ofrece es la de generar varias gráficos en una misma figura. Vamos a ver cómo visualizar varias imágenes del dataset MNIST a la vez en una solo figura.

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist["data"], mnist["target"]
X.shape, y.shape
```

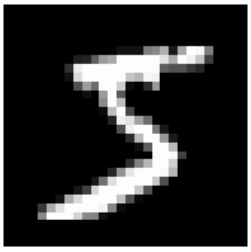
```
((70000, 784), (70000,))
```

```
X = X.reshape(-1, 28,28)
X.shape
```

```
(70000, 28, 28)
```

Empezamos definiendo una figura vacía con un tamaño determinado con la función figure . Después, podemos añadir sub-figuras a la figure con la función subplot indicando su posición en la figura global.

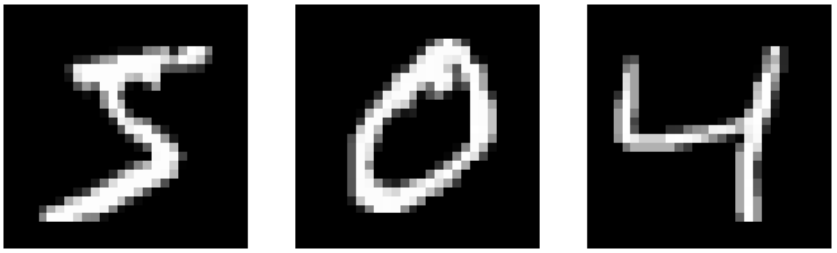
```
fig = plt.figure(figsize=(3,3))
ax = plt.subplot(1,1,1)
ax.imshow(X[0], cmap="gray")
plt.axis('off')
plt.show()
```



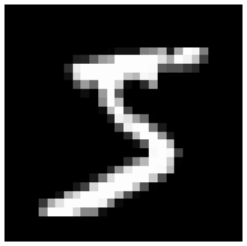
⚡ La función imshow nos permite visualizar imágenes. Puedes aprender más aquí.

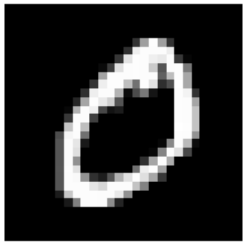
En la función subplot verás 3 números: número de filas, número de columnas y la posición de la sub-figura (empezando por 1). Podemos añadir más imágenes de la siguiente manera.

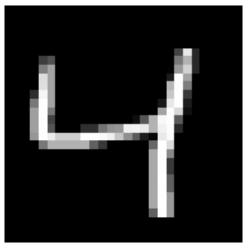
```
fig = plt.figure(figsize=(10,5))
ax = plt.subplot(1,3,1)
ax.imshow(X[0], cmap="gray")
ax.axis('off')
ax = plt.subplot(1,3,2)
ax.imshow(X[1], cmap="gray")
ax.axis('off')
ax = plt.subplot(1,3,3)
ax.imshow(X[2], cmap="gray")
ax.axis('off')
plt.show()
```



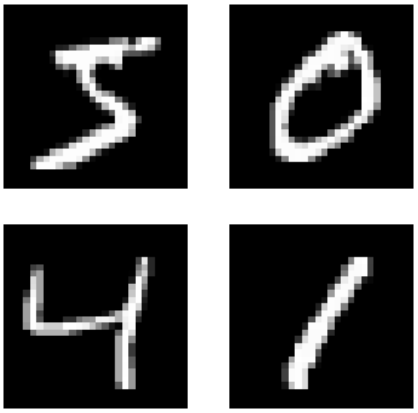
```
fig = plt.figure(figsize=(5,10))
ax = plt.subplot(3,1,1)
ax.imshow(X[0], cmap="gray")
ax.axis('off')
ax = plt.subplot(3,1,2)
ax.imshow(X[1], cmap="gray")
ax.axis('off')
ax = plt.subplot(3,1,3)
ax.imshow(X[2], cmap="gray")
ax.axis('off')
plt.show()
```





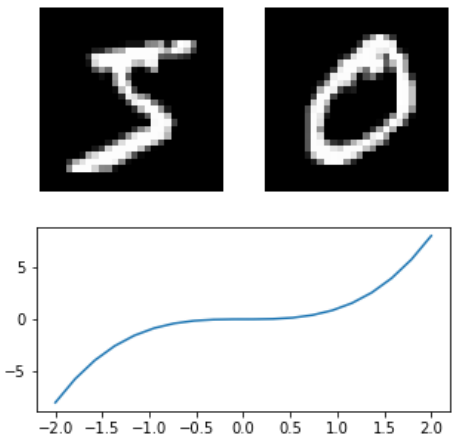


```
fig = plt.figure(figsize=(5,5))
ax = plt.subplot(2,2,1)
ax.imshow(X[0], cmap="gray")
ax.axis('off')
ax = plt.subplot(2,2,2)
ax.imshow(X[1], cmap="gray")
ax.axis('off')
ax = plt.subplot(2,2,3)
ax.imshow(X[2], cmap="gray")
ax.axis('off')
ax = plt.subplot(2,2,4)
ax.imshow(X[3], cmap="gray")
ax.axis('off')
plt.show()
```



También podemos combinar diferentes tipos de gráficos, con diferente extensión dentro de la gráfica global, para generar una figura como la siguiente.

```
fig = plt.figure(figsize=(5,5))
ax = plt.subplot(2,2,1)
ax.imshow(X[0], cmap="gray")
ax.axis('off')
ax = plt.subplot(2,2,2)
ax.imshow(X[1], cmap="gray")
ax.axis('off')
ax = plt.subplot(2,1,2)
ax.plot(x2, y2)
plt.show()
```



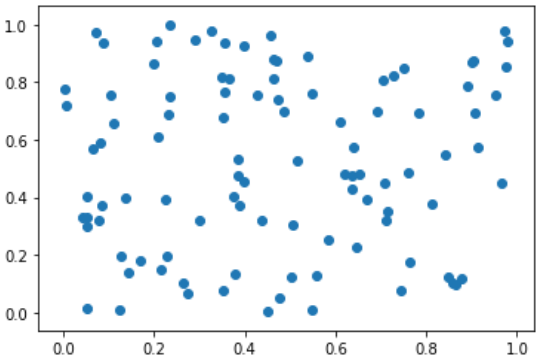
d. Otros tipos de gráficos

Además de la función plot , Matplotlib nos ofrece otras funciones para generar gráficos de otros tipos. Vamos a ver algunos ejemplos.

i. Scatter

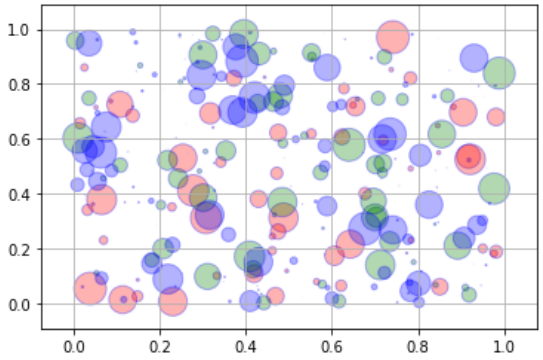
La función scatter nos permite visualizar un conjunto de puntos.

```
from numpy.random import rand
x, y = rand(2, 100)
plt.scatter(x, y)
plt.show()
```



Una opción muy interesante en este tipo de gráficos es visualizar cada punto con un color o tamaño diferente en función de otra variable, que puede representar una densidad, temperatura en una localización, etc.

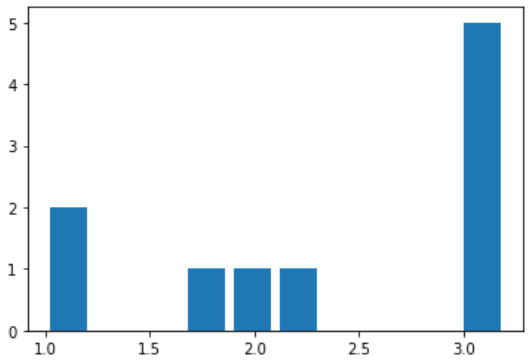
```
1for color in ['red', 'green', 'blue']:
    n = 100
    x, y = rand(2, n)
    scale = 500.0 * rand(n) ** 5
    plt.scatter(x, y, s=scale, c=color, alpha=0.3, edgecolors='blue')

plt.grid(True)
plt.show()
```



ii. Histogramas

Otro tipo de gráfico muy útil es el histograma de barras. Usamos la función hist a la cual podemos indicarle el número de barras con la opción bins .

```
1 data = [1, 1.1, 1.8, 2, 2.1, 3.2, 3, 3, 3, 3]
2 plt.hist(data, bins = 10, rwidth=0.8)
3 plt.show()
```

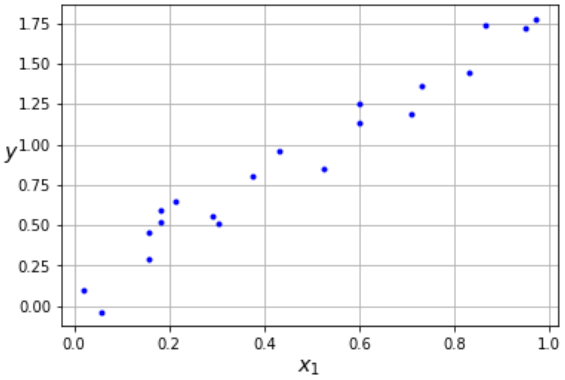


iii. Animaciones

Vamos a terminar viendo cómo generar animaciones, una funcionalidad muy útil que nos permite generar vídeos o gifs que pueden quedar muy bien en presentaciones. En el siguiente ejemplo veremos cómo generar una animación similar a la usada en el primer post del Perceptrón en el que vimos la evolución de nuestro modelo a medida que lo entrenamos.

```
1 # datos para entrenar el perceptrón 2
3 np.random.seed(42)
4
5 x = np.random.rand(20)
```

```
6 y = 2*x + (np.random.rand(20)-0.5)*0.5 7
plt.plot(x, y, "b.")
plt.xlabel("$x_1$", fontsize=14)
plt.ylabel("$y$", rotation=0, fontsize=14)
plt.grid(True)
plt.show()
```




```
1
```

```
# funciones necesarias para entrenar el perceptrón 2
def gradient(w, x, y):
# calculamos la derivada de la función de pérdida
# con respecto a los parámteros `w`
dldw = x*w - y
dydw = x
dldw = dldw*dydw
return np.mean(2*dldw)
10
def cost(y, y_hat):
# calculamos la función de pérdida
return ((y_hat - y)**2).mean() 14
def solve(epochs = 29, w = 1.2, lr = 0.2):
# iteramos un número determinado de `epochs`
# por cada epoch, calculamos gradientes y
# actualizamos los pesos
weights = [(w, gradient(w, x, y), cost(x*w, y))]
for i in range(1,epochs+1):
dw = gradient(w, x, y)
w = w - lr*dw
weights.append((w, dw, cost(x*w, y)))
return weights
```

Para generar la animación usamos el objeto animation y su función FuncAnimation , a la cual le podemos pasar el número de frames totales en la animación así como la figura sobre la que queremos aplicar la animación y la función con la lógica para actualizarla en cada frame. También configuramos Matplotlib para visualizar la animación como un vídeo en HTML .

```
# animación
2
3 from matplotlib import animation, rc
```

```
4 rc('animation', html='html5')
5
def update(i):
xs = np.linspace(0, 1, num=100)
(w, dw, cost) = weights[i]
ax.clear()
ax.plot(x, y, "b.")
ax.plot(xs, w*xs, "-k")
ax.set_xlabel("$x_1$", fontsize=14)
ax.set_ylabel("$y$", rotation=0, fontsize=14)
ax.set_title(f"Iteración: {i}")
ax.grid(True)
ax.axis([0,1,0,2])
return ax
18
19 weights = solve()
20 fig = plt.figure(figsize=(10,6))
21 ax = plt.subplot(1,1,1)
22 anim = animation.FuncAnimation(fig, update, frames=len(weights),
interval=100)
23 plt.close()
24
25 anim
```

0:00 / 0:03

Una vez generada la animación puedes descargarla desde el widget creado.

⚡ Para poder generar animaciones tendrás que instalar la librería ffmpeg , la cual no forma parte de Python . Puedes