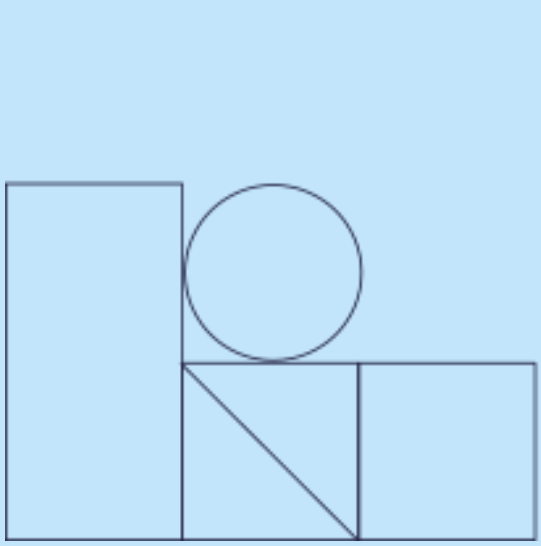
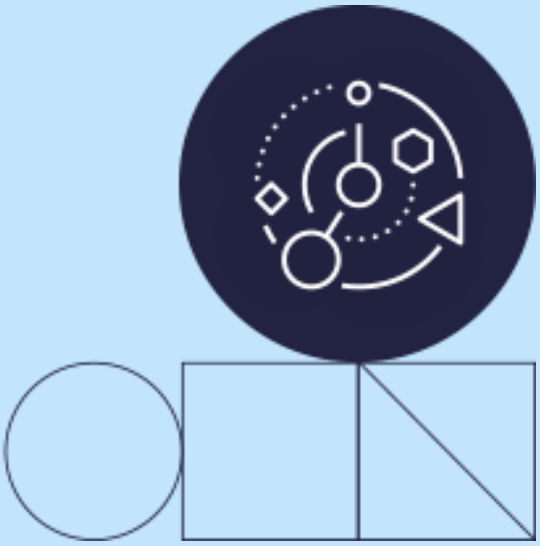
IBM SkillsBuild

IBM

Conceptos básicos y sintaxis de Python

Sentencias condicionales en Python





Índice

Introducción

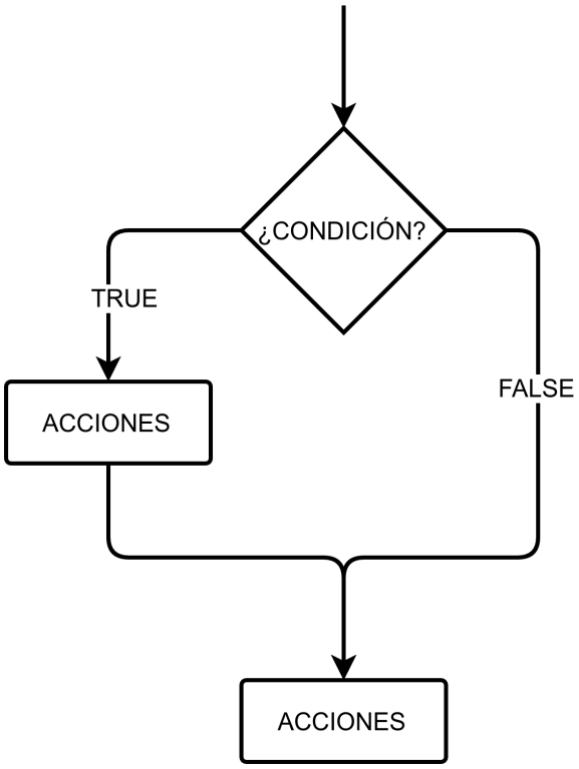
Python también dispone de modificadores de flujo de ejecución como condicionales y bucles con los que poder tomar decisiones o repetir código un número de veces.

Control de Flujo en Python

En esta unidad vamos a conocer las sentencias condicionales.

La sentencia if

La primera sentencia que nos permite controlar el flujo de nuestro programa es la sentencia **if**. Esta sentencia nos permite ejecutar trozos de código condicionalmente, tal y como vemos en el siguiente diagrama de flujo:



Síntaxis:

```
>>> if CONDICION:
...     print('Se ejecuta si CONDICION es TRUE')
...     print('También se ejecuta')
>>> print('Ya estamos fuera del if')
```

Veamos un ejemplo:

```
>>> a = 10
>>> b = 3

>>> if a > b:
...     print ('SI se cumple la condición')
# Bloque identado 4 espacios
...
>>> print ('Ya estamos fuera del if')

SI se cumple la condición
Ya estamos fuera del if
```

Como vemos la sentencia **if** tiene una sintaxis que nos resultará familiar. Básicamente se trata de realizar una evaluación de una expresión que devuelve un booleano (en este caso ***a > b***). Si la expresión devuelve ***TRUE*** entonces entramos en un bloque de código específico (el primer **print** del ejemplo). Si la expresión devuelve ***FALSE***, no entramos en ese bloque de código.

Sintaxis y anidamiento en Python

A diferencia de otros lenguajes cercanos a C, Python elimina bastantes elementos para aumentar la legibilidad del código. Por ejemplo, la misma sentencia en C++ sería lo siguiente:

```
>>> if (a > b){
        printf (“a es mayor que b”);
}
```

Mientras que, en Python, el mismo código sería:

```
>>> if a > b:              # if en Python
...     print ('a es mayor que b')
```

Las principales diferencias consisten en que Python añade los dos puntos (:) al final de la evaluación condicional para indicar que va a empezar un bloque nuevo.

A su vez elimina:

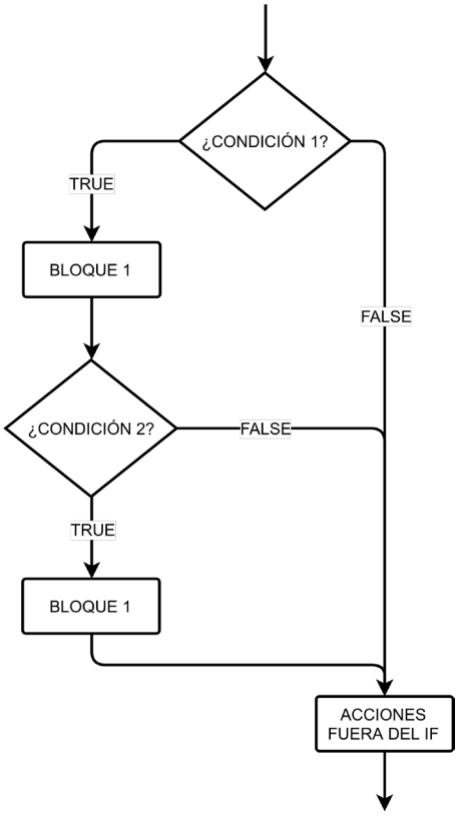
- **Paréntesis en la evaluación**. En Python son opcionales.
- **Llaves que encierran el bloque**. Python no utiliza llaves para separar bloques de código. En su lugar, la anidación se indica mediante el uso de bloques indentados, normalmente por cuatro (4) espacios.
- **Punto y coma**. En Python es opcional y se usa normalmente para delimitar varias sentencias en una misma línea (una práctica nada recomendada).

Cuando se diseñó Python, se hizo pensando en la legibilidad de su código, por lo que se apostó por simplificar el número de elementos que pueblan cada sentencia. Además, el hecho de eliminar llaves para separar bloques de código, implicó que se utilizara la indentación como medio de separar bloques de código. Esto fuerza a que los programadores tengan que tener cuidado de la indentación al escribir, lo que produce que el código sea más ordenado y coherente, aumentando así su legibilidad. Esto fuerza a que siempre seamos coherentes con las indentaciones, cosa que no pasa en otros lenguajes de programación donde el hecho de usar llaves lleva a muchos programadores a ser descuidados con las indentaciones, por lo que muchas veces vemos distintas estrategias de indentación en un mismo programa. A pesar de ser una de las características que más molestan a los recién llegados, esta es una de las muchas decisiones de diseño del lenguaje donde se puede apreciar su elegancia.

La anidación de bloques se hace en grupos de 4 espacios. Así, si tenemos múltiples anidaciones, iremos aumentando 4 espacios en cada bloque.

```
>>> if ___:                  # Raíz. 1er nivel de indentación (0 espacios)
...     ___                      # Bloque 1: 4 espacios
...     if ___:                  # Seguimos en Bloque 1: 4 Espacios
...         ___                      # Bloque 2: 4 + 4 espacios
...         ___                      # Seguimos en bloque 2. 4 + 4 espacios
...     ___                      # Hemos vuelto al Bloque 1.
>>> ___                      # Raíz. Ya hemos salido del 'if' 1er nivel
```

En un diagrama de flujo esto correspondería con:



Múltiples ramas condicionales

Finalizado ya el interludio sobre sintaxis, veamos cómo podemos crear múltiples ramas condicionales. Para ello, introducimos las palabras clave ***else*** y ***elif*** (else if).

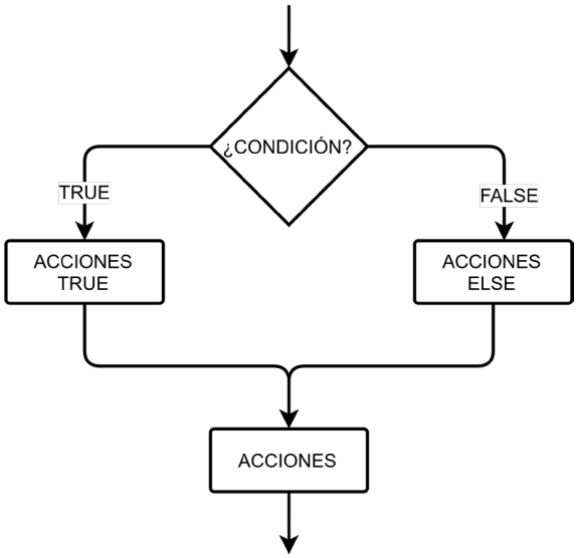
```
>>> a = 10
>>> b = 3

>>> if a > b:
...     print ('Se ha cumplido la condición')
... else:
...     print ('No se ha cumplido la
condición')
...
>>> print ('Ya estamos fuera del if')

No se ha cumplido la condición
Ya estamos fuera del if
```

En este primer caso vemos como, mediante la palabra clave ***else*** nos permite ejecutar ramas de código en el caso que no se haya cumplido la condición evaluada en la expresión del ***if***. Fijémonos que la palabra clave ***else*** se encuentra al mismo nivel que el ***if*** al que enlaza.

En este caso el diagrama de flujo sería el siguiente:



En el caso en el que quisiéramos evaluar más condiciones utilizamos la palabra clave ***elif*** (else if).

```
>>> a = 10
>>> b = 10

>>> if a > b:
...     print ('A es menor que B')
... elif a == b
...     print ('A es igual a B')
... else:
...     print ('A es mayor que B')

...
>>> print ('Ya hemos salido del
condicional')


A es igual a B
Ya hemos salido del condicional
```

La palabra clave ***elif*** nos permite hacer evaluaciones alternativas a la del ***if***. En el caso en el que el ***if*** haya devuelto ***False***, entonces entraríamos a evaluar el bloque del ***elif*** y ejecutar su código asociado en el caso en el que devuelve ***True***. Al igual que en otros lenguajes de programación, podemos utilizar el número de ***elif*** que necesitemos. Así mismo, si ningún ***elif*** se cumple, tenemos un bloque ***else*** para ejecutar un determinado código en el caso en el que ninguna de las evaluaciones se haya cumplido (todas hayan devuelto ***False***).

Expresiones condicionales

Si hemos programado en C/C++ o javascript es muy probable que conozcamos el operador ternario:

(a > b) ? 20: 30.

Este operador devuelve 20 si **a** es mayor que ***b*** o 30 si no lo es. En Python tenemos un operador equivalente, pero con una sintaxis mucho más legible:

```
>>> a = 10
>>> b = 3
>>> 20 if a > b else 30
20
```

Esta expresión se le llama expresión condicional u operador ternario. A diferencia que la sentencia ***if***, este operador es una expresión en sí misma y no una sentencia. La notación de la misma es la siguiente:

```
x = true_value if condición else
false_value
```

A pesar de que la expresión se lea de izquierda a derecha, realmente se evalúa en el siguiente orden. Primero se evalúa ***condición***. Si ésta es verdadera, entonces se devuelve ***true_value***. Si no lo es, se devuelve ***false_value***.

Notad que es una manera muy reducida de escribir condicionales ya que es equivalente al siguiente bloque de código:

```
>>> if condición:
...     x = true_value
... else:
...    x = false_value
```

No obstante, es conveniente no abusar de estas expresiones y utilizarlas sólo cuando estemos escribiendo evaluaciones pequeñas que no requieran mucho texto ni condiciones complejas de entender.

Además, aunque no sea necesario, en muchas ocasiones es conveniente envolver entre paréntesis esta expresión para mejorar su legibilidad:

```
x = (true_value if condición else
false_value)
```

Nosotros os recomendamos que siempre que uséis esta expresión la encerréis entre paréntesis. Muchos os preguntaréis por qué no tratamos las sentencias **switch-case** en esta unidad. La respuesta es sencilla, Python no dispone de sentencias **switch-case** por lo que este tipo de condicionales múltiples deberá resolverse utilizando **if-elif**.

Vamos a realizar algunos ejemplos prácticos del uso del condicional en Python:

Ejemplo 1:

Un usuario introduce texto desde teclado y queremos averiguar si es un número entero. Si es entero lo añadiremos a una variable tipo lista de números enteros.

Para resolver este ejercicio hay varias cosas que aún no hemos visto:

Cómo introducir texto desde teclado Se realiza con la sentencia:

```
a = input("Introduce un número: ")
```

Lo que introduzcamos por teclado se almacenará en la variable a y será siempre de tipo string, como podemos comprobar con el siguiente Código:

```
>>> texto = input ("Introduce algo por teclado: ")

Introduce algo por teclado: 5

>>> type(texto)
Out: str

>>> print(texto)
5
```

¿Cómo saber si un texto se corresponde a un número?. Para ello utilizamos la siguiente instrucción:

ejemplo6.1.py

```
l = list()                              #Creamos una lista vacia
texto = input("Introduce un número entero por teclado: ")
if texto.isnumeric():                   # Comprobamos si son numeros
        num = int(texto)
        l.append(num)
        print("Número " + str(num) + " añadido a la lista")
else:
        print("No has introducido un número entero")
```

ejemplo6.2.py

```
d = { "50862634" : 37 , "43394932" : 32}      #Creamos diccionario

texto = input("Introduce un documento de indentidad ")

if texto in d:      #Comprobamos si esta en el diccionario
        print("La edad de " + texto + " es " + str(d[texto]))
else:
        edad = input("Documento no existente. Introduce edad: ")
        if edad.isnumeric():
                num = int(edad)
                d[texto] = num
                print("Añadido al diccionario")

print(d)   #Mostramos por pantalla el diccionario
```

Cuando veamos los bucles, veremos que podemos utilizar este método para crear una lista de números sin correr el riesgo de generar errores cuando el usuario introduce algún carácter no numérico.

Ejemplo 2:

Tenemos un diccionario en el que asociamos los números de los documentos de identidad de ciertas personas con su edad. Queremos realizar un programa en el que el usuario introduzca un el número de un documento de identidad. Si dicho número ya está en el diccionario debe mostrar la edad, en caso contrario debe solicitarnos que introduzcamos la edad, que posteriormente añadiremos al diccionario.

Aquí tenemos la solución:

ejemplo6.2.py

```
d = { "50862634" : 37 , "43394932" : 32}      #Creamos diccionario

texto = input("Introduce un documento de indentidad ")

if texto in d:      #Comprobamos si esta en el diccionario
       print("La edad de " + texto + " es " + str(d[texto]))
else:
       edad = input("Documento no existente. Introduce edad: ")
       if edad.isnumeric():
              num = int(edad)
              d[texto] = num
              print("Añadido al diccionario")

print(d)   #Mostramos por pantalla el diccionario
```

Ejemplo 3:

¿Cómo haríamos si quisiéramos guardar este diccionario en un archivo y posteriormente abrirlo siempre que queramos consultarlo?

Para ello usamos el paquete Path y Pickle (los veremos más detalladamente en otro momento). Pickle nos ofrece procedimientos para poder leer y escribir diccionarios en archivos. El paquete Path lo utilizamos para comprobar si el archivo existe.

Veamos un ejemplo sencillo para leer un archivo y guardarlo en un diccionario:

```
# read python dict from a file
pkl_file = open('myfile.pkl', 'rb')
mydict2 = pickle.load(pkl_file)
pkl_file.close()
```

Y ahora veamos cómo escribir un diccionario en un archivo:

```
# write python dict to a file
mydict = {'a': 1, 'b': 2, 'c': 3}
output = open('myfile.pkl', 'wb')
pickle.dump(mydict, output)
output.close()
```

(Como verás hemos puesto los nombres de variables y comentarios en inglés, esta es la práctica habitual en programación)

ejemplo6.3.py

```
import pickle
from pathlib import Path

#Create an empty dictionary
d = dict()

#Ask for file name to load dictionary from
file_name = input("Introduce el nombre del archivo con los datos: ")

#Comprobamos que existe
path = Path(file_name)
if path.is_file():
	# Open file in reading mode
	input_file = open(file_name, 'rb')
	d = pickle.load(input_file)
	#Close de file
	input_file.close()
else:
	print("El file no existe, creamos diccionario vacio")


#Check for values or add new ones
document_number = input("Introduce un documento de indentidad ")

if document_number in d:  #Check whether it is on dict or not
	print("La edad de " + document_number + " es " + str(d[document_number]))
else:
	age = input("Documento no existente. Introduce edad: ")
	if age.isnumeric():

		num = int(age)
		d[document_number] = num
		print("Añadido al diccionario")

# Save dict on file and close
output_file = open(file_name, 'wb')
pickle.dump(d, output_file)
output_file.close()
```

Bucle while

Con estos tres ejemplos hemos podido ver el uso de los condicionales y además hemos podido repasar los temas anteriores. A continuación, veremos los bucles y los combinaremos con sentencias condicionales.

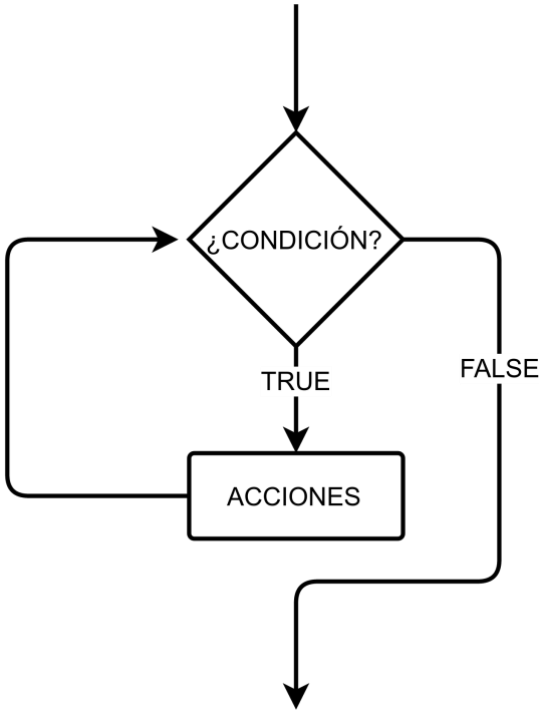
Otra de las sentencias más utilizadas en un programa son los bucles **while**. Al igual que en otros lenguajes de programación, el bucle **while** repite un trozo de código iterativamente mientras se cumpla una determinada condición.

```
>>> a = 0
>>> while a<3:
...       print (a, end=' ')   # Acabamos
con espacios en lugar de salto de línea
...       a += 1               # Equivalente
a: a = a + 1
>>> print (a)                  # Estamos
fuera del while
>>> print('Hemos salido fuera del while')

0 1 2 3
Hemos salido fuera del while
```

Al comenzar cada iteración la expresión de inicio del **while** es evaluada. Si la expresión se cumple (devuelve **True**), se vuelve a entrar en el **while**. Si no se cumple (devuelve **False**), el **while** deja de ejecutarse y pasamos a la siguiente sentencia tras el **while**.

Así es como se expresaría en un diagrama de flujo:



Bucles y las sentencias continue, break, pass y los bloques else

Ahora que sabemos cómo realizar bucles en **Python**, vamos a ver algunas sentencias que nos permiten alterar el flujo natural de los mismos. Éstas son las siguientes sentencias:

- ***break*:** Interrumpe el flujo y sale fuera de bucle.
- ***continue*:** Salta al comienzo de la siguiente iteración del bucle.
- ***pass*:** No hace nada. Es una sentencia vacía.
- ***Else:*** Al finalizar un bucle: Se ejecuta sólo si el bucle ha finalizado con normalidad. Es decir, se ejecuta si el bucle finaliza sin haber ejecutado un ***break***.

Sentencia break

Veamos algunos ejemplos de estas sentencias, empezando por la sentencia ***break***.

```
>>> a = 5
>>> while a:              # Utilizamos la
propia variable como condición
```

```
...       print (a, end=' ')
...       a == 2:
...           break
...       a -= 1
...
>>> print ('\nFuera del Bucle.')
>>> print('Valor de "a": {}'.format(a))

5 4 3
Fuera del Bucle.
```

Como vemos, esta sentencia interrumpe inmediatamente el bucle y sale a la siguiente sentencia del código.

Notemos que en este caso la expresión que estamos evaluando es la propia variable ***a***. Esto puede parecer raro al principio, pero es una forma muy elegante de evaluar expresiones en este tipo de sentencias. Tened en cuenta que en el ***while*** siempre se evalúa una expresión booleana. Es decir, que en realidad estamos evaluando ***bool(a)*** que, al ser a un entero, devuelve siempre ***True*** hasta que ***a*** sea 0. Veremos este tipo de expresiones muy frecuentemente en **Python**.

Sentencia continue

La sentencia **continue** nos permite interrumpir la iteración actual:

```
>>> a = 7
>>> while bool(a):
...       a -= 1
...       if a % 2 == 0
...            continue   # Saltamos a la
siguiente iteracción si es a es par.
...       print (a, end=' ')
...
>>> print ('\nFuera del Bucle.')

5 3 1
Fuera del Bucle.
```

La sentencia pass

La sentencia **pass** no hace absolutamente nada. Es un marcador de posición para dejar bucles vacíos.

```
>>> a = 5
>>> while a:
...       pass    # Presiona Ctrl-C para
abortar la ejecución
...
```

Notad que este bucle se queda bloqueado debido a que no hacemos nada dentro de él. Si queremos abortar la ejecución tendremos que presionar ***Ctrl-C.***

Bloques else al finalizar bucles

Como hemos visto, tenemos dos maneras de finalizar bucles. La primera es una finalización limpia, sin interrupciones, y la segunda es mediante el uso de la sentencia **break**. Para saber si bucle ha finalizado de cierta manera, en muchos lenguajes de programación se utilizan flags (banderas) en el código que luego hay que evaluar. En **Python**, esto no es necesario ya que podemos utilizar bloques **else** al final de un bucle.

Por ejemplo, veamos un ejemplo donde hacemos evaluamos si un número es primo o no dentro de un bucle **while**.

```
>>> a = 13
>>> b = a // 2           # División entera.
P. ej. 13 // 2 == 6
>>> while b > 1:
...       if a % b == 0:  # % es el operador
resto. P. ej. 10 % 5 == 0
...           print('{b} es factor de
{a}'.format(b,a))
...           break
...       b -= 1
... else:
...           print('{} es primo'.format(a))
...
>>> print ('\nFuera del Bucle.')

13 es primo
Fuera del Bucle.
```

En este caso, hemos realizado la búsqueda de factores del número 13 y, al no haber encontrado ninguno, no hemos ejecutado **break** que teníamos dentro del condicional. Por ello, el bucle ha finalizado limpiamente por lo que se ha ejecutado el bloque **else** al final del **while**.

El bucle for

La sintaxis del bucle **for** es muy sencilla y parecida a la del bucle **while**, con la excepción que en la cabecera del **for**, en lugar de evaluar una expresión en cada iteración, vamos asignando elementos de un iterador a una variable.

```
>>> for elem in objeto:  # Vamos asignado elementos de objeto en elem
...     sentencias
...     if test:        # Podemos usar breaks
...         break
...     if test:        # Podemos usar continues
...         continue
... else:               # Si no hemos salido a través de break
...     sentencias
```

Empecemos con un ejemplo sencillo:

```
>>> for s in ['Me', 'gusta', 'Python!']:
...    print(s, end=' ')
Me gusta Python!
```

En este ejemplo vamos recorriendo elementos de una lista de **strings** y vamos imprimiéndolos en pantalla. Veamos otro ejemplo, esta vez sumando números:

```
>>> a = 0
>>> for x in [1, 2, 3, 4]:
...     a += x
...
>>> print(a)
10
```

Y ahora otro ejemplo, esta vez con **strings**:

```
>>> for c in 'Me gusta Python!':
...    print(c, end=' ')
...
M e g u s t a P y t h o n !
```

Este bucle recorre un **string** letra a letra y las va mostrando por pantalla, añadiendo un espacio tras cada impresión en pantalla.

Finalmente, veamos un ejemplo recorriendo un diccionario, ya que uno de los usos más frecuentes del bucle **for** es recorrer objetos iterables, diccionarios, tuplas...etc:

```
>>> for k in d:
...    print(k, end=' ')
Apellidos edad nombre
```

Por defecto, al pasarle un diccionario a un ***for***, lo recorremos por sus claves. Hay varias maneras de extraer los valores de un diccionario. Más adelante, aprenderemos cómo recorrer simultáneamente las claves y valores, pero de momento veamos una manera muy sencilla de hacerlo:

```
>>> keys = ['nombre', 'apellidos', 'edad']
>>> values = ['Guido', 'van Rossum', '60']
>>> d = dict(zip(keys, values))   # Creamos el diccionario

>>> for k in d:
...     info = '{}: {}'.format(k, d[k])
# Texto formateado con claves y valores

...    print(info)
Apellidos: van Rossum
edad: 60
nombre: Guido
```

El método **str.format** sustituye las llaves de la cadena de texto por los parámetros que le pasamos al llamarlo. En cada iteración del ejemplo le estamos pasando la clave **(k)** del diccionario y el valor del diccionario correspondiente a esa clave **(d[k])**.

Asignación en tuplas

Un concepto que aún no habíamos visto es que Python permite la **asignación en tuplas**. Es decir, que podemos asignar los elementos de una tupla a varias variables a la vez.

```
>>> a, b = (3, 4)     # Asignamos los
elementos de la tupla a cada variable
>>> print(a, b)
3 4
```

Más adelante exploraremos más detalladamente estas capacidades de Python. Pero de momento nos basta con conocer que esto es muy conveniente en bucles **for**. Por ejemplo:

```
>>> t = [(1, 2), (3, 4), (5, 6)]
>>> for x, y in t:
...     print (x + y, end= '  ')
3 7 11
```

En este ejemplo estamos recorriendo una lista de tuplas y las vamos asignando a dos variables simultáneamente (**x**, e **y**) que luego utilizamos dentro del **for** para ir sumándolas. Este proceso se llama **desempaquetado en tuplas**.

Veréis que es muy útil en muchas situaciones como, por ejemplo, cuando queremos recorrer dos listas en paralelo:

```
>>> 1a = [10, 20, 30, 40]
>>> 1b = [5, 25, 50, 10]
>>> for a, b in zip(1a, 1b):
...     m = max(a, b)   # max(a, b)
                        devuelve el máximo
                        entre a y b
...     print(m , end=’ ‘)

10 25 50 40
```

En este ejemplo recorremos dos listas usando la función ***zip*** (cremallera) que, en cada iteración, nos devuelve una tupla cogiendo un elemento de cada una de las listas. Luego, dentro del ***for***, comparamos cuál de los dos elementos es mayor con la función ***max*** y lo mostramos por pantalla.

Otro uso muy común del desempaquetado en tuplas es navegar por los objetos de un diccionario. Anteriormente vimos un ejemplo donde recorríamos un diccionario a través de sus claves. Ahora vamos a ver cómo recorrer todos sus elementos a la vez:

```
>>> keys = ['nombre', 'apellidos', 'edad']
>>> values = ['Guido', 'van Rossum', 60]

>>> d = dict(zip(keys, values))

>>> for k, v in d.items():    # d.items
devuelve tupla con clave, valor
...     print('{}: {}'.format(k, v))

apellidos: van Rossum
edad: 60
nombre: Guido
```

Vemos que el método ***dict.items*** nos devuelve una tupla (clave, valor) correspondiente a una entrada del diccionario en cada iteración del ***for***.

Vistas en diccionarios

Tened en cuenta que a partir de Python 3.5, el método ***dict.items*** devuelve una vista de los elementos del diccionario, es decir, no devuelve los objeto en sí hasta que no los recorramos o los convirtamos explícitamente en listas. Otros métodos tienen el mismo comportamiento son ***dict.keys*** y ***dict.values***, que devuelven vistas de las claves y los valores respectivamente.

```
>>> for k in d.keys():
...     print(k, end= ' ')
apellidos edad nombre

>>> for v in d.values():
...     print(v, end= ' ')
Van Rossum 60 Guido
```

Si en lugar de recorrerlos, intentamos extraer todos las claves o elementos vemos que no devuelven una lista, sino que son vistas del diccionario:

```
>>> d.keys()
dict_keys(['nombre', 'apellidos', 'edad'])

>>> d.keys()
dict_keys

>>> d.values()
dict_values(['van Rossum', 60, 'Guido']

>>> type(d.values())
dict_values

>>> d.items()

dict_items([('apellidos', 'van Rossum'),
('edad', 60, ('nombre', 'Guido')])


>>> type(d.items())
dict_items
```

Tened en cuenta que no podemos acceder directamente a estas listas, sino que tenemos que, o bien recorrerlas como hemos visto antes, o bien envolverlas en listas para poder indexarlas, trocearlas, etc.:

```
>>> d.keys()[1]      # No podemos acceder a
una vista de diccionario
-----------------------------------
Error. Texto omitido por simplicidad
TypeError: 'dict_keys' object does not
support indexing

>>> list (d.keys())[1]
'edad'
```

Bucles for y contadores

A estas alturas seguramente ya os habréis extrañado de las diferencias entre el **for** de otros lenguajes como C/C++ o Java con los de Python. Los bucles **for** de **Python** son más parecidos a los **for-each** de otros lenguajes de programación.

Es decir, están destinados a recorrer los elementos de secuencias o iterables.
Primero de todo, veamos cómo es un **for** al estilo C y al estilo Pythónico. La vía Pythónica es, en casi todos los casos, más legible y más rápida de ejecutar.

```
>>> letras = list('abcdefghijklmnopqrstuvwxyz')

>>> for i in range(len(letras)):     # Versión C/C++. No lo uséis!
...     print(letras[i], end=' ')
...
a b c d e f g h i j k l m n o p q r s t u v w x y z


>>> for c in letras:                 # Versión Pythónica. Más legible y rápido
...     print(c, end=' ')
...
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Como podemos ver se puede hacer un **for** al estilo C/C++. La función **range** nos devuelve un listado de números consecutivos de la longitud que le pasemos. En este caso, le hemos pasado la longitud de la lista de letras.

```
# Ejemplos de uso de range
# Hay que envolverlo en lista o recorrerlo
en for

>>> list(range(5))   # Devuelve 5 elementos
empezando en 0
[0, 1, 2, 3, 4]

>>> list(range(-5, 5))     # Devuelve
elementos en el rango -5, 5
[-4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(range(-5, 5, 2))  # Elementos -5 a
5 en saltos de 2
[-5, -3, -1, 1, 3]
```

Por lo general, es siempre recomendable utilizar el bucle **for** al estilo de Python. Cuando empecemos a utilizar bucles nos veréis tentados a utilizar los **for** de la manera típica de otros lenguajes. Salvo algunas excepciones, siempre es mejor hacerlo de la manera Pythónica.

A continuación, exponemos algunos de los esos casos típicos donde los recién llegados a **Python** tienen tendencia a programar bucles **for** no Pythónicos. También damos la alternativa que se debe utilizar.

El primero de estos ejemplos ya lo hemos visto anteriormente. Se trata de recorrer varias listas simultáneamente.

```
>>> import random

>>> l1 = letras[:8]                    # Creamos 3 sub-listas a partir de trozos
>>> l2 = letras[8:16]
>>> l3 = letras[16:]

>>> random.shuffle(l1)                 # Barajamos cada trozo
>>> random.shuffle(l2)
>>> random.shuffle(l3)

>>> for i in range(len(l1)):           # Versión NO Pythónica. NO RECOMENDADA
...     print(l1[i] + l2[i] + l3[i], end=' ')
...
cpv emq dlt hnr fis gow akz bjx

>>> for a, b, c in zip(l1, l2, l3):    # Versión Pythónica.
...     print(a + b + c, end=' ')
...
cpv emq dlt hnr fis gow akz bjx
```

En este ejemplo estamos troceando la lista letras en tres partes, luego barajamos cada una de las partes y recorremos las tres listas simultáneamente para ir mostrando en pantalla una letra de cada lista en cada iteración. La versión Pythónica utiliza la función **zip**, que ya habíamos visto antes. La única diferencia es que, en este caso, en lugar de unir dos listas estamos haciéndolo con tres. La gran ventaja de zip (además de ser más legible) es que no tenemos que preocuparnos de que todas las listas sean de igual longitud. Cuando una de las listas se termina, **zip** detiene la ejecución. En la alternativa no Pythónica, tendríamos que consultar las longitudes de cada lista y recorrer el bucle siguiendo los índices de la lista más corta.

Otro ejemplo típico en el que los programadores noveles de Python tienden a hacer uso de **for** no Pythónicos es cuando estamos haciendo búsquedas de índices de los elementos de una lista. Por ejemplo:

```
>>> letras = list('abcdefghijklmnopqrstuvwxyz')
>>> vocales = 'aeiou'

>>> random.shuffle(letras)
>>> print(''.join(letras))
rqkmjgvzblsaoicfntxewduphy

>>> for i in range(len(letras)):                    # Manera NO Pythónica
...     if letras[i] in vocales:
...         print('{} en la posición {}'.format(letras[i], i))
...
a en la posición 11
o en la posición 12
i en la posición 13
e en la posición 19
u en la posición 22

>>> for i, letra in enumerate(letras):              # Manera Pythónica
...     if letra in vocales:
...         print('{} en la posición {}'.format(letra, i))
...
a en la posición 11
o en la posición 12
i en la posición 13
e en la posición 19
u en la posición 22
```

En este ejemplo buscamos las posiciones de las vocales en un abecedario desordenado. Para ello usamos la función **enumerate**, que nos pide una secuencia y nos devuelve la misma secuencia asociada a sus índices:

```
# Ejemplos de enumerate. Hay que envolverlo en list() o recorrerlo en un for
>>> abcde = sorted(letras)[:5]

>>> list(enumerate(abcde))              # Devuelve secuencia con sus índices
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]

>>> list(enumerate(abcde, 10))          # Podemos decirle en qué índice empieza
[(10, 'a'), (11, 'b'), (12, 'c'), (13, 'd'), (14, 'e')]
```

Iteradores

A continuación, vamos a comprender que son los *iteradores*, y cuál es el protocolo que permite que cualquier secuencia de Python pueda ser recorrida en un ***for***. Además, veremos que, sobrecargando un par de funciones, podemos dotar a nuestras propias clases de esas capacidades de iteración.

Objetos Iterables

Como bien sabemos, podemos recorrer cualquier secuencia en un bucle ***for:***

```
>>> for num in [1, 2, 3, 4, 5, 6]:
...     print(num ** 2, end= ' ')
1 4 9 16 25 36

>>> for num in [12, 38, 99, 1]:
...     print(num / 2, end= ' ')
6.0 19.0 49.5 0.5

>>> for letra in 'Python':
...     print(letra.upper(), end=' ')
P Y T H O N
```

De hecho, es posible recorrer estas secuencias porque son **iterables** El concepto de **iterable** es una generalización del término secuencia. Un objeto iterable es un objeto que cumple una de estas condiciones:

- Está almacenado físicamente como una secuencia
- Produce un resultado detrás de otro en el contexto de una herramienta de iteración como un bucle for, una lista por comprensión, etc.

Otra manera de comprender el concepto de objeto **iterable** es entender que un **iterable** es, o bien una secuencia ordenada físicamente (como una lista, tupla, etc.) o bien un objeto que se comporta virtualmente como una secuencia.

El Protocolo de Iteración de Python

Pero, ¿qué significa que un objeto se comporta como una secuencia virtual? Básicamente significa que implementa un protocolo, llamado protocolo de iteración, que define cómo tiene que comportarse un objeto de manera que sea capaz de devolver un elemento detrás de otro a demanda cuando queramos recorrerlo.

La mejor manera de entender el protocolo de iteración es verlo funcionar en los objetos builtin de Python. Nosotros lo vamos a ver con el tipo **file**, que lo usaremos para ir recorriendo un fichero línea a línea.

Primero, vamos a crear el fichero, que contendrá una versión recortada del Zen de Python:

```
>>> zen = '''\
... Bello es mejor que feo.
... Explícito es mejor que implícito.
... Simple es mejor que complejo.
... Complejo es mejor que complicado.
... '''
...
>>> f = open('short.zen.txt', 'w')
>>> f.writelines(zen)   # Escribe el
fichero
>>> f.close().          # Cierra el fichero
```

Ahora que lo tenemos, vamos a leer ese fichero línea a línea, a mano:

```
>>> f = open('short_zen.txt', 'r')
>>> f.readline()
'Bello es mejor que feo.\n'

>>> f.readline()
'Explícito es mejor que implícito.\n'

>>> f.readline()
'Simple es mejor que complejo.\n'

>>> f.readline()
'Complejo es mejor que complicado.\n'

>>> f.readline()          # Devuelve una cadena vacía cuando termina el fichero
''
```

El método ***readline*** nos permite ir leyendo el fichero línea a línea hasta que nos encontremos una cadena vacía. Sin embargo, ¿cómo funciona este método?

De hecho, los ficheros en Python implementan un método que tiene un comportamiento muy parecido. El método **__next__** también lee una línea del fichero cada vez que lo llamamos. Sin embargo, cuando finaliza el fichero, nos genera una excepción de tipo ***StopIteration***.

```
>>> f = open('short_zen.txt', 'r')
>>> f.__next__()
'Bello es mejor que feo.\n'

>>> f.__next__()
'Explícito es mejor que implícito.\n'

>>> f.__next__()
'Simple es mejor que complejo.\n'

>>> f.__next__()
'Complejo es mejor que complicado.\n'

>>> f.__next__()
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-55-39ec527346a9> in <module>()
----> 1 f.__next__()

StopIteration:
```

Esta función es lo que llamamos el *protocolo de iteración* de Python. Cualquier objeto que implemente esta función para avanzar al siguiente resultado y que lance la excepción ***StopIteration*** tras el último resultado es considerado como un **iterador**.

Esto implica que cualquier objeto que implementa estas dos reglas, puede ser incluido en un bucle ***for***, en una lista por comprensión, etc. ya que lo que hacen en realidad estos mecanismos es ir llamando a la función **__next__** en cada iteración.

De hecho, esto es lo que pasa con el tipo builtin de ficheros de Python: podemos recorrerlo en un bucle ***for*** para ir procesando sus líneas una a una:

```
>>> for linea in open('short_zen.txt'):

...     print(linea.upper(), end='')
BELLO ES MEJOR QUE FEO.
EXPLÍCITO ES MEJOR QUE IMPLÍCITO.
SIMPLE ES MEJOR QUE COMPLEJO.
COMPLEJO ES MEJOR QUE COMPLICADO.
```

La función next

Para simplificar la iteración manual con ***__next__***, Python ofrece la función builtin **next** que nos permite acceder a la iteración manual de ***__next__*** más fácilmente. Lo que hace realmente la función ***next(obj)*** es llamar directamente a ***obj.__next()__:***

```
>>> f = open('short.zen.txt', 'r')
>>> f.__next__()
'Bello es mejor que feo.\n'

>>> next(f)
'Explícito es mejor que implícito.\n'

>>> next(f)
'Complejo es mejor que complicado.\n'

>>> next(f)
----------------------------------------
StopIteration             Traceback(most recent
call last)
<ipython-input-77-468f0afdf1b9> in
<module>()
----> 1 next(f)

StopIteration
```

Iteradores e iterables

Ahora que empezamos a entender cómo funciona el protocolo de iteración en Python, vamos a profundizar un poco más para entender como las diferencias entre objetos **iterables** como las secuencias que hemos visto al principio de la unidad y los **iteradores** que acabamos de conocer.

La función iter

Acabamos de ver como los ficheros en Python son **iteradores** ya que implementan la función ***__next__***. Veamos si ocurre lo mismo con otros tipos de datos como las listas:

```
>>> lista = [1, 2, 3]
>>> next(lista)
----------------------------------------
TypeError         Traceback(most recent
call last)
<ipython-input-79-b6aa4a0df19a> in
<module>()
      1 lista = [1, 2, 3]
----> 2 next(lista)


TypeError: 'list' object is not an
interator
```

¿Qué es lo que ha pasado? La función ***next*** nos está produciendo una excepción de tipo ***TypeError*** diciéndonos que las listas nos son **iteradores**. ¿Pero, entonces, como es posible que podamos recorrer una lista en un bucle ***for***?

Si recordáis, al principio de la unidad hemos dicho que las listas son **iterables**, y con el error que acabamos de obtener está claro que un **iterable** no es lo mismo que un **iterador**. Por ello, es el momento de entender qué es un **iterable** y en qué se diferencia de un **iterador:**

- **Iterable:** Un objeto iterable es un objeto que devuelve un iterador. Para ello implementa el método ***__iter__***.
- **Iterador:** Un objeto iterador implementa ***__next__***, lo que le permite ser iterado en bucles ***for***, etc.

Como vemos, los objetos iterables no son iteradores, sino que devuelven iteradores cuando se les pide. Veamos un ejemplo con nuestra lista:

```
>>> li = lista.__iter__()
>>> li.__next__()
1

>>> next(li)
2

>>> next(li)
3

>>> next(li)
----------------------------------------
StopIteration         Traceback(most recent
call last)
<ipython-input-91-deb767b63ff8> in
<module>()
----> 1 next(li)

StopIteration:
```

Para simplificar la iteración manual, Python implementa la función ***iter***. De esta manera ***iter(obj_iterable)*** es equivalente a ***obj_iterable.__iter()__:***

```
>>> li = iter(lista)
>>> next(li)
1

>>> next(li)
2

>>> next(li)
3

>>> next(li)
----------------------------------------
StopIteration         Traceback(most recent
call last)
<ipython-input-95-deb767b63ff8> in
<module>()
----> 1 next(li)

StopIteration:
```

De hecho, lo que hacen los bucles ***for***, las listas por comprensión, etc. es llamar a la función ***__iter__*** del objeto **iterable** que van a recorrer antes de empezar a hacerlo. Con esto obtienen el **iterador** que es lo que recorren de verdad.

Notad también que a medida que avanzamos por un objeto **iterador**, vamos consumiéndolo y ya no podemos volver atrás. Cuando llegamos al final del iterador y obtenemos la excepción ***StopIteration***, decimos que hemos *consumido el iterador*. Para volver a recorrerlo de nuevo, tendríamos que pedirle al ***iterable*** que ha creado el **iterador** que lo genere de nuevo.

```
>>> li = iter(lista)
>>> next(li)
1

>>> next(li)
2

>>> next(li)
3

>>> next(li)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-100-deb767b63ff8> in <module>()
----> 1 next(li)

StopIteration:


>>> next(li)             # Hemos consumido el iterador: no podemos iterar más
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-101-deb767b63ff8> in <module>()
----> 1 next(li)

StopIteration:


>>> li = iter(lista)      # Generamos otro iterador para volver a recorrerlo
>>> next(li)
1
```

Creando nuestros propios iteradores

Ahora que entendemos completamente cómo funciona el *protocolo de iteración* de Python y qué son los **iterables** y los **iteradores,** estamos preparados para crear nuestros propios objetos iterables e iteradores.

Como hemos visto, un objeto iterador es aquel que implementa el método ***__next__*** especificando cuál va a ser el siguiente elemento a devolver tras cada llamada.

Sabiendo esto, vamos a crear nuestro primer objeto iterador:

```
>>> rep = Repetidor(3, ):
...
...     def __init__(self, veces, item):
...         self.veces = veces
...         self.item = item
...         self.counter = 0
...
...     def __next__(self):
...         if self.counter == self.veces:
...            raise StopIteration('Iterador
consumido')
...         self.counter += 1
...         return self.item
```

En este ejemplo estamos creando una clase que permite ir generando repeticiones del parámetro que le hayamos pasado al constructor. Con esto hemos creado nuestro primer **iterador**.

Vamos a recorrerlo manualmente:

```
>>> class Repetidor(3, 'Python', )

>>> next(rep)
'Python'

>>> next(rep)
'Python'

>>> next(rep)
'Python'

>>> next(rep)
----------------------------------------
StopIteration         Traceback(most recent
call last)
<ipython-input-150-000231fd72d3> in
<module>()

----> 1 next(rep)

<ipython-input-146-6c109aef3369> in
__next__(self)
     8   def __next__(self):
     9       if self.counter == self.veces:
----> 10        raise
StopIteration('Iterador consumido')
     11  self.counter += 1
     12  return self.item

StopIteration: Iterador consumido
```

Pero fijaos que aún no puede ser recorrido por un bucle ***for*** porque nuestra clase no es ***iterable:***

```
>>> rep r in Repetidor(3, 'Python!'):
...     print(r, end=' ')
----------------------------------------
TypeError         Traceback(most recent
call last)
<ipython-input-151-e97e3e194464> in
<module>()
----> 1 for r in Repetidor(3, 'Python!'):
      2     print(r, end=' ')

TypeError: 'Repetidor' object is not
iterable
```

Para convertirlo en un **iterable** simplemente hay que implementar el método ***__iter__***:

```
>>> class Repetidor():
...
...     def __init__(self, veces, item):
...         self.veces = veces
...         self.item = item
...         self.counter = 0
...
...     def __next__(self):
...         if self.counter == self.veces:
...           raise StopIteration('Iterador
consumido')
...         self.counter += 1
...         return self.item
...
...     def __iter__(self):
...         return self

>>> for r in Repetidor(3, 'Python!'):
...     print(r, end=' ')

Python! Python! Python!
```

Notad que, como ***Repetidor*** ya era un **iterador**, puede devolverse a sí mismo usando la referencia a ***self***.

Al haber definido los métodos ***__iter__*** y ***__next__***, nuestra clase ahora es un **iterable** y un **iterador** a la vez.

Tened en cuenta que es posible crear iterables que no son iteradores: con que devuelvan un objeto iterador, que no tiene por qué ser uno mismo, ya basta:

```
>>> class Repetidor():
...
...      def __init__(self, veces, *items):
...          self.veces = veces
...          self.items = items
...
...      def __iter__(self):
...          return.items(self.items *
self.veces)     # Devuelve iterador de la
lista

>>> for r in Repetidor(3, 'a', 'b', 'c'):
...     print(r, end=' ')
a b c a b c a b c
```

En este ejemplo hemos modificado nuestra clase ***Repetidor*** para que acepte varios elementos y los repita de manera concatenada un número determinado de veces.

Ahora ***Repetidor*** es sólo un **iterable** ya que no implementa el método ***__next__***. Esto hace que con ***__iter__*** ya no se devuelve a sí misma, si no que devuelve un iterador formado por la lista resultante de la combinación de los parámetros entrada por el número de repeticiones.

```
>>> rep =  Repetidor(3, 'a', 'b', 'c')
>>> it = iter(rep)
>>> isinstance(it,Repetidor)
False
```

La principal ventaja de esto es que ahora el objeto no se consume a sí mismo, ya que genera un iterador que es el que sí se consume. Dependiendo de vuestras necesidades de diseño podéis necesitar una u otra opción.