😎 Variables y tipos booleanos

La información que se guarda en una variable puede ser de muchos tipos:

- **Números (enteros, decimales, imaginarios, en notación científica, con precisión arbitraria, en base decimal o en otras bases, etc.)**
- **Cadenas de texto (una sola letra o más letras, del juego de caracteres ASCII occidental o del juego de caracteres Unicode, etc)**
- **Conjuntos de números o texto (matrices, listas, tuplas, etc.)**
- **Estructuras más complicadas (punteros, diccionarios, etc.)**

Cada tipo de información se almacena de forma distinta, por lo que existen diferentes tipos de variables para cada tipo de información.

Algunos lenguajes de programación como C, C++, Java, etc, exigen que antes de utilizar una variable se defina el tipo de información que se va a guardar en esa variable. Otros lenguajes de programación como Python, PHP, etc, no lo exigen y es el intérprete del lenguaje el que decide el tipo de variable a utilizar en el momento que se guarda la información.
Los lenguajes que requieren definir los tipos de las variables se denominan lenguajes *tipificados* y los que no se denominan lenguajes *no tipificados*.

Variables en Python

En Python **las variables son "etiquetas"** que permiten hacer referencia a los datos (que se guardan en unas "cajas" llamadas objetos). Python es un lenguaje de programación orientado a objetos y su modelo de datos también está basado en objetos. Para cada dato que aparece en un programa, Python crea un objeto que lo contiene.

Cada objeto tiene:

- **Un identificador único (un número entero, distinto para cada objeto). El identificador permite a Python referirse al objeto sin ambigüedades.**
- **Un tipo de datos (entero, decimal, cadena de caracteres, etc.). El tipo de datos permite saber a Python qué operaciones pueden hacerse con el dato.**
- **Un valor (el propio dato).**

Así, las variables en Python no guardan los datos, sino que son simples nombres para poder hacer referencia a esos objetos.

Si escribimos la instrucción:

In [1]:
```
1 x = 5
```

... Python:

- **Crea el objeto "5". Ese objeto tendrá un identificador único que se asigna en el momento de la creación y se conserva a lo largo del programa. En este caso, el objeto creado será de tipo número entero y guardará el valor 5.**
- **Asocia el nombre x al objeto número entero 5 creado.**

Así, al describir la instrucción anterior no habría que decir **'la variable x almacena el número entero 5'**, sino que habría que decir **'podemos llamar x al objeto número entero 5'.**

La variable x es como una etiqueta que nos permite hacer referencia al objeto "5", más cómoda de recordar y utilizar que el identificador del objeto.

Por otro lado, en Python se distingue entre objetos mutables y objetos inmutables:

- **Los objetos inmutables son objetos que no se pueden modificar. Por ejemplo, los números, las cadenas y las tuplas son objetos inmutables.**
- **Los objetos mutables son objetos que se pueden modificar. Por ejemplo, las listas y los diccionarios son objetos mutables.**

En el caso de los objetos inmutables (como los números) no hay mucha diferencia entre considerar la variable como una caja o como una etiqueta, pero en el caso de los objetos mutables (como las listas) pensar en las variables como cajas puede llevar a error.

Definir una variable

Las variables en Python se crean cuando se definen por primera vez, es decir, cuando se les asigna un valor por primera vez.

Para asignar un valor a una variable se utiliza el operador de igualdad **(=)**. A la izquierda de la igualdad se escribe el nombre de la variable y a la derecha el valor que se quiere dar a la variable.

En el ejemplo siguiente se almacena el número decimal 2.5 en una variable de nombre x (realmente habría que decir que se crea la etiqueta x para hacer referencia al objeto número decimal 2.5). Fíjese en que los números decimales se escriben con punto (.) y no con coma (,).

La variable se escribe siempre a la izquierda de la igualdad:

In [2]:
```
1 x = 2.5
```

Si se escribe al revés, Python genera un mensaje de error:

In [3]:
```
1 2.5 = x
```

```
  Input In [3]
    2.5 = x
    ^
SyntaxError: cannot assign to literal
```

Para que se muestre el valor de una variable, basta con escribir su nombre:

In [4]:
```
1 x = 2.5
2 x
```

Out[4]:
```
2.5
```

Si una variable no se ha definido previamente, escribir su nombre genera un mensaje de error:

In [5]:
```
1 x = -10
2 y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [5], in <cell line: 2>()
      1 x = -10
----> 2 y

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

- **Una variable puede almacenar números, texto o estructuras más complicadas (que se verán más adelante).**
- **Si se va a almacenar texto, el texto debe escribirse entre comillas simples (') o dobles ("), que son equivalentes.**

A las variables que almacenan texto se les llama cadenas (de texto).

In [6]:
```
1 nombre = "Juana Juanita"
2 nombre
```

Out[6]:
```
'Juana Juanita'
```

Si no se escriben comillas, Python supone que estamos haciendo referencia a otra variable (que, si no está definida, genera un mensaje de error)

In [7]:
```
1 nombre = Pepe
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [7], in <cell line: 1>()
----> 1 nombre = Pepe

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

In [8]:
```
1 nombre = Pepito Pepe
```

```
  Input In [8]
    nombre = Pepito Pepe
                    ^
SyntaxError: invalid syntax
```

Eliminar una variable

La instrucción **del** elimina completamente una variable.

In [9]:
```
1 nombre = "Pepito Pepe"
2 nombre
```

Out[9]:
```
'Pepito Pepe'
```

In [10]:
```
1 del nombre
2 nombre
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [10], in <cell line: 2>()
      1 del nombre
----> 2 nombre

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

Nombres de variables

Aunque no es obligatorio, se recomienda que el nombre de la variable esté relacionado con la información que se almacena en ella, para que sea más fácil entender el programa. Si el programa es sencillo o mientras se está escribiendo un programa, esto no parece muy importante, pero si se consulta un programa escrito por otra persona, o escrito por uno mismo pero hace tiempo, resultará mucho más fácil entender el programa si los nombres están bien elegidos.

> **El nombre de una variable debe empezar por una letra o por un guion bajo (_) y puede seguir con más letras, números o guiones bajos.**

In [11]:
```
1 _x = 34.5
2 _x
```

Out[11]: 34.5

In [12]:
```
1 x1 = 150
2 x1
```

Out[12]: 150

In [13]:
```
1 fecha_de_nacimiento = "2 de octubre de 1990"
2 fecha_de_nacimiento
```

Out[13]: '2 de octubre de 1990'

Los nombres de variables **NO** pueden incluir espacios en blanco.

In [14]:
```
1 fecha de nacimiento = "2 de octubre de 1990"
```

```
  Input In [14]
    fecha de nacimiento = "2 de octubre de 1990"
          ^
SyntaxError: invalid syntax
```

Los nombres de variables pueden contener cualquier carácter alfabético (los del alfabeto inglés, pero también ñ, ç o vocales acentuadas), aunque **se recomienda utilizar únicamente los caracteres del alfabeto inglés.**

In [15]:
```
1 año = 1988
2 año #anio
```

Out[15]: 1988

Los nombres de las variables pueden contener mayúsculas, pero tenga en cuenta que **Python distingue entre mayúsculas y minúsculas** (en inglés se dice que Python es case-sensitive).

In [20]:
```
1 nombre = "Pepito Matero"
2 nombre
```

Out[20]: 'Pepito Matero'

In [21]:
```
1 Nombre = "Evaristo Nambia"
2 Nombre
```

Out[21]: 'Pepito Matero'

In [22]:
```
1 nomBre = "Gervasio Lucero"
2 nomBre
```

Out[22]: 'Gervasio Lucero'

Cuando el nombre de una variable contiene varias palabras, se aconseja separarlas **con guiones bajos** para facilitar la legibilidad, aunque también se utiliza la notación **camelCase**, en las que las palabras no se separan pero empiezan con mayúsculas (salvo la primera palabra)

In [24]:
```
1 fecha_de_nacimiento = "2 de octubre de 1990"
2 fecha_de_nacimiento
```

Out[24]: '2 de octubre de 1990'

In [25]:
```
1 fechaDeNacimiento = "2 de octubre de 1990"
2 fechaDeNacimiento
```

Out[25]: '2 de octubre de 1990'

Las palabras reservadas del lenguaje están prohibidas como nombres de variables. Ejemplo:

In [26]:
```
1 lambda = 3
```

```
  Input In [26]
    lambda = 3
           ^
SyntaxError: invalid syntax
```

Como se mencionó anteriormente, los nombres de las funciones integradas sí que se pueden utilizar como nombres de variables, pero mejor no hacerlo porque a continuación ya no se puede utilizar la función como tal:

In [27]:
```
1 print = 3
2 print("Hola")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [27], in <cell line: 2>()
      1 print = 3
----> 2 print("Hola")

TypeError: 'int' object is not callable
```

Eliminando con **del** la variable con nombre de función, se recupera la función.

In [28]:
```
1 del print
2 print("Hola")
```

```
Hola
```

Tipos de variables

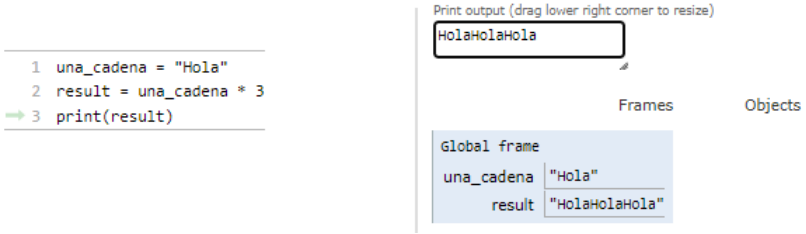
Aunque se definan de forma similar, para Python no es lo mismo un número entero, un número decimal o una cadena ya que, por ejemplo, dos números se pueden multiplicar pero dos cadenas no *(curiosamente, una cadena sí se puede multiplicar por un número)*. Por tanto, estas tres definiciones de variables no son equivalentes:

In [29]:
```
1 fecha = 1990 # fecha está almacenando un número entero
2 fecha = 1990.0 # fecha está almacenando un número decimal
3 fecha = "1990" # fecha está almacenando una cadena de 4 letras
4 fecha = [2, "octubre", 1990] # fecha está almacenando una lista (un tipo de variable que puede contener
5 # varios elementos ordenados)
```

Este ejemplo demuestra también que se puede volver a redefinir una variable. Python modifica el tipo de la variable automáticamente.

In [30]:
```
1 una_cadena = "Hola"
2 result = una_cadena * 3
3 result
```

Out[30]: 'HolaHolaHola'



Mostrar el valor de las variables

Para que se emita el valor de una variable, basta con escribir su nombre:

In [31]:
```
1 a = 2
2 a
```

Out[31]: 2

También se puede conocer el valor de varias variables a la vez escribiéndolas separadas por comas (se emitirán entre paréntesis porque Python las considera como un **conjunto ordenado** llamado **tupla**):

In [32]:
```
a = b = 2
c = "pepe"
a
```

Out[32]: 2

In [33]:
```
c,b
```

Out[33]: ('pepe', 2)

Utilizar o modificar variables ya definidas

Una vez se ha definido una variable, se puede utilizar para hacer cálculos o para definir nuevas variables, como muestran los siguientes ejemplos:

In [34]:
```
a = 2
a + 3
```

Out[34]: 5

In [35]:
```
horas = 5
minutos = 60 * horas
segundos = 60 * minutos
segundos
```

Out[35]: 18000

In [36]:
```
horas = 1
minutos = 2
segundos = 3
segundos + 60 * minutos + 3600 * horas
```

Out[36]: 3723

En caso de utilizar una variable no definida anteriormente, Python genera un mensaje de error:

In [37]:
```
dias = 7 * semanas
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [37], in <cell line: 1>()
----> 1 dias = 7 * semanas

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

Se puede redefinir el valor de una variable a partir de su propio valor. Por ejemplo:

In [38]:
```
a = 10
a = a + 5
a
```

Out[38]: 15

Es importante reiterar que los nombres de las variables no tienen sentido real para Python, son simples etiquetas para referirse al contenido.

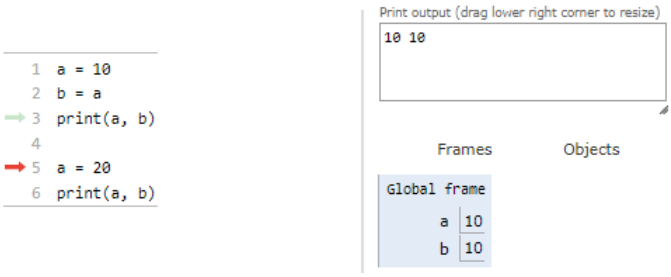
Cuando se modifica una variable, el valor anterior se pierde y no se puede recuperar (salvo si se realiza la operación contraria o hay otra variable que conserva el valor anterior).

Hay que tener cuidado al modificar una variable que se ha utilizado para definir otras variables, porque esto puede afectar al resto:

Si se trata objetos **inmutables**, el resto de variables no resultan afectadas, como muestra el siguiente ejemplo:

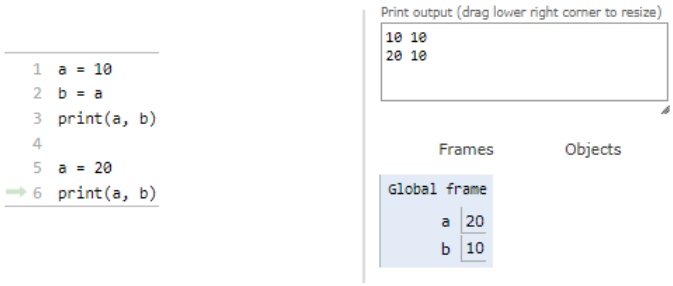
In [39]:
```
a = 10
b = a
a, b
```

Out[39]: (10, 10)



In [40]:
```
a = 20
a, b
```

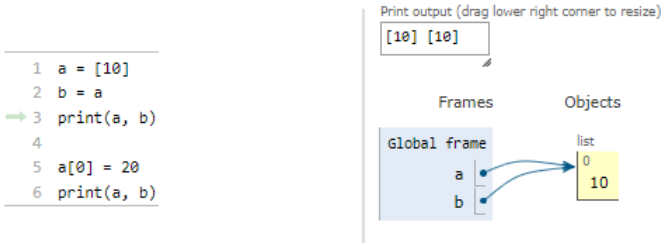
Out[40]: (20, 10)



Pero si se trata de objetos **mutables** y al modificar la variable se modifica el objeto, el resto de variables sí resultan afectadas, como muestra el siguiente ejemplo:

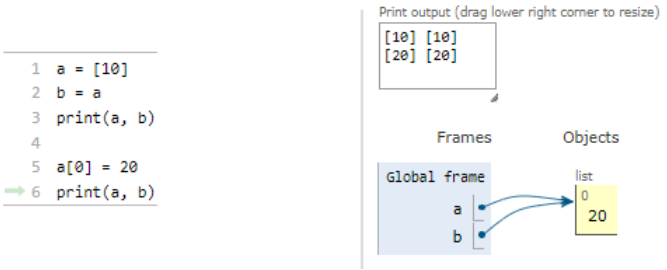
In [41]:
```
a = [10]
b = a
a, b
```

Out[41]: ([10], [10])



In [42]:
```
a[0] = 20
a, b
```

Out[42]: ([20], [20])



Asignaciones compactas

Cuando una variable se modifica a partir de su propio valor, se puede utilizar la denominada **"asignación compacta"**:

Por ejemplo:

In [43]:
```
a = 10
a += 5
a
```

Out[43]: 15

Es equivalente a:

In [44]:
```
a = 10
a = a + 5
a
```

Out[44]: 15

Asignación compacta	Es equivalente
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a / b
a **= b	a = a ** b

Asignación compacta	Es equivalente
a //= b	a = a // b

Nota: Las dos notaciones son completamente equivalentes en el caso de los objetos inmutables (números, cadenas, tuplas), pero en el caso de los objetos mutables (listas, diccionarios, etc.) existe una diferencia. En Python NO existen los operadores incremento (++) o decremento (- -) que sí existen en otros lenguajes de programación:

In [45]:
```
1 a = 5
2 a++
```

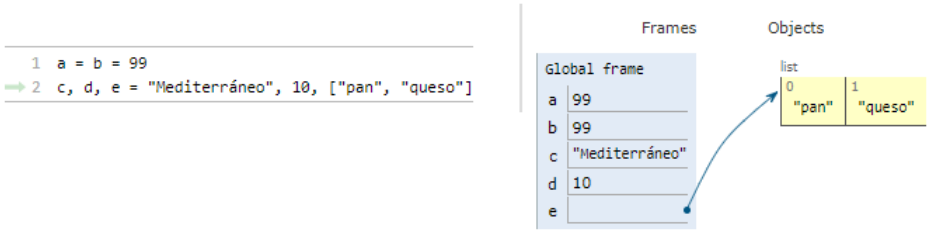
```
  Input In [45]
    a++
      ^
SyntaxError: invalid syntax
```

Definir y modificar varias variables a la vez

En una misma línea se pueden definir simultáneamente varias variables, con el mismo valor o con valores distintos, como muestra el siguiente ejemplo:

In [46]:
```
1 a = b = 99
2 c, d, e = "Mediterráneo", 10, ["pan", "queso"]
```

- En el primer caso las dos variables tendrán el mismo valor.
- En el segundo caso la primera variable tomará el primer valor y así sucesivamente.



In [47]:
```
1 a,b
```

Out[47]: (99, 99)

In [48]:
```
1 c, d, e
```

Out[48]: ('Mediterráneo', 10, ['pan', 'queso'])

Si el número de variables no coincide con el de valores, Python genera un mensaje de error.

In [49]:
```
1 a, b, c = 1, 2
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [49], in <cell line: 1>()
----> 1 a, b, c = 1, 2

ValueError: not enough values to unpack (expected 3, got 2)
```

In [50]:
```
1 a, b, c = 1, 2, 3, 4
```

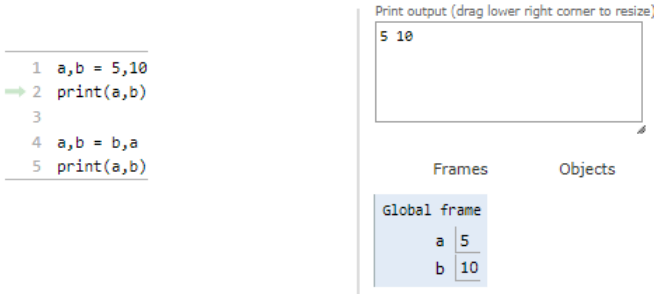
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [50], in <cell line: 1>()
----> 1 a, b, c = 1, 2, 3, 4

ValueError: too many values to unpack (expected 3)
```

Se pueden modificar varias variables en una sola instrucción y la modificación se realiza en un solo paso:

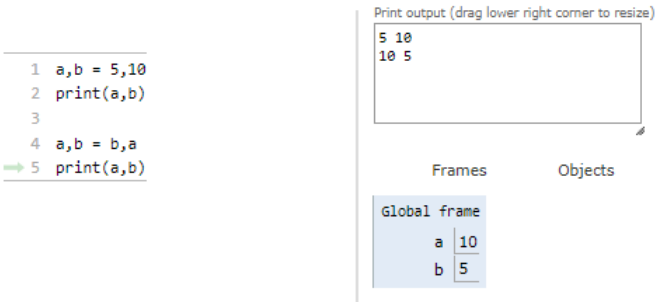
In [51]:
```
1 a,b = 5,10
2 a,b
```

Out[51]: (5, 10)



In [52]:
```
a,b = b,a
a,b
```

Out[52]: (10, 5)



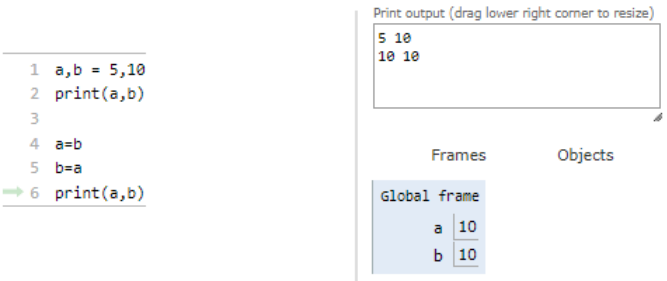
Obsérvese que si este procedimiento lo hubiéramos hecho paso a paso, el resultado hubiera sido distinto:

In [53]:
```
a,b = 5,10
a,b
```

Out[53]: (5, 10)

In [54]:
```
a=b
b=a
a,b
```

Out[54]: (10, 10)



El motivo de este resultado se entiende fácilmente mostrando los valores tras cada operación:

In [55]:
```
a,b=5,10
a,b
```

Out[55]: (5, 10)

In [56]:
```
a=b
a,b
```

Out[56]: (10, 10)

In [57]:
```
b=a
a,b
```

Out[57]: (10, 10)

Tipos booleanos: True y False

Una variable booleana es una variable que sólo puede tomar dos posibles valores: **True (verdadero) o False (falso)**.

En Python cualquier variable (en general, cualquier objeto) puede considerarse como una variable booleana.

En general los elementos nulos o vacíos se consideran False y el resto se consideran True.

Para comprobar si un elemento se considera True o False, se puede convertir a su valor booleano mediante la función bool().

Ejemplos:

```
In [58]: bool(0)
Out[58]: False

In [59]: bool(0.0)
Out[59]: False

In [60]: bool("")
Out[60]: False

In [61]: bool(None)
Out[61]: False

In [62]: bool(())
Out[62]: False

In [63]: bool([])
Out[63]: False

In [64]: bool({})
Out[64]: False

In [65]: bool(25)
Out[65]: True

In [66]: bool(-9.5)
Out[66]: True

In [67]: bool("abc")
Out[67]: True

In [68]: bool((1, 2, 3))
Out[68]: True

In [69]: bool([27, "octubre", 1997])
Out[69]: True

In [70]: bool({27, "octubre", 1997})
Out[70]: True
```

Operadores lógicos

Los operadores lógicos son unas operaciones que trabajan con valores booleanos. Ejemplos:

and: "y" lógico. Este operador da como resultado True si y sólo si sus dos operandos son True:

```
In [71]: True and True
Out[71]: True

In [72]: True and False
Out[72]: False

In [73]: False and True
Out[73]: False
```

In [74]:
```
1 False and False
```

Out[74]: False

or: "o" lógico. Este operador da como resultado True si algún operando es True:

In [75]:
```
1 True or True
```

Out[75]: True

In [76]:
```
1 True or False
```

Out[76]: True

In [77]:
```
1 False or True
```

Out[77]: True

In [78]:
```
1 False or False
```

Out[78]: False

> **Nota: En el lenguaje cotidiano, el "o" se utiliza a menudo en situaciones en las que sólo puede darse una de las dos alternativas. Por ejemplo, el medio de transporte -en un viaje- se puede elegir "tren o micro", pero no las dos cosas. En lógica, ese tipo de "o" se denomina "o exclusivo" (xor)**

not : negación. Este operador da como resultado True si y sólo si su argumento es False:

In [79]:
```
1 not True
```

Out[79]: False

In [80]:
```
1 not False
```

Out[80]: True

Expresiones compuestas

Al componer expresiones complejas Python evalúa primero los not, luego los and y por último los or, por lo cual se recomienda el uso de paréntesis. Ejemplos:

El operador not se evalúa antes que el operador and:

In [81]:
```
1 not True and False
```

Out[81]: False

In [82]:
```
1 (not True) and False
```

Out[82]: False

In [83]:
```
1 not (True and False)
```

Out[83]: True

El operador not se evalúa antes que el operador or:

In [84]:
```
1 not False or True
```

Out[84]: True

In [85]:
```
1 (not False) or True
```

Out[85]: True

In [86]:
```
1 not (False or True)
```

Out[86]: False

El operador and se evalúa antes que el operador or:

In [87]:
```
1 False and True or True
```

Out[87]: True

In [88]:
```
1 (False and True) or True
```

Out[88]: True

In [89]:
```
False and (True or True)
```

Out[89]: False

In [90]:
```
True or True and False
```

Out[90]: True

In [91]:
```
(True or True) and False
```

Out[91]: False

In [92]:
```
True or (True and False)
```

Out[92]: True

Si en las expresiones lógicas se utilizan valores distintos de True o False, Python utiliza esos valores en vez de True o False

In [93]:
```
3 or 4
```

Out[93]: 3

Si se quieren mostrar valores booleanos, se puede convertir el resultado a un valor booleano

In [94]:
```
3 or 4
bool(3 or 4) # Verdadero porque 3 es diferente de 0
```

Out[94]: True

Evaluación de expresiones lógicas

Python permite utilizar valores no booleanos en las expresiones lógicas. Para evaluar las expresiones, Python convierte esos valores en booleanos. En estos casos, la respuesta de Python no es un valor booleano, sino alguno de los valores de la expresión. Por ejemplo:

In [95]:
```
2 or 3
```

Out[95]: 2

In [96]:
```
3 and 5
```

Out[96]: 5

Teniendo en cuenta que los números distintos de 0 se consideran True, el resultado de las dos expresiones es True.

Python evalúa las expresiones y puede saber la respuesta final sin necesidad de terminar algunas expresiones.

Por ejemplo, al evaluar **2 or 3**, como se trata de un **or**, en cuanto Python evalúa el primer valor (2, o sea True), ya sabe que el valor final va a ser **True**.

Por eso emite 2. Sin embargo, el evaluar 3 and 5, como se trata de un and, tiene que evaluar los dos valores.

Como los dos son True, el resultado es True. Responde con el último valor evaluado, emite 5. Ejemplos:

In [97]:
```
0 or "" or "Pepe" or True
```

Out[97]: 'Pepe'

In [98]:
```
3 and "" and 5 and [1, 2, 3]
```

Out[98]: ''

In [99]:
```
4 or 3 and 2 or 5
```

Out[99]: 4

In [100]:
```
(4 or 3) and (2 or 5)
```

Out[100]: 2

Comparaciones

Las comparaciones también dan como resultado valores booleanos:

> Mayor que; < Menor que

In [101]:
```
1 3 > 2
```

Out[101]: True

In [102]:
```
1 3 < 2
```

Out[102]: False

>= Mayor o igual que; <= Menor o igual que

2 >= 1 + 1

In [103]:
```
1 4 - 2 <= 1
```

Out[103]: False

== Igual que; != Distinto de

In [104]:
```
1 2 == 1 + 1
```

Out[104]: True

In [105]:
```
1 6 / 2 != 3
```

Out[105]: False

En Python (y en otros muchos lenguajes de programación):

- Un signo igual (=) significa asignación, es decir, almacenar un valor en una variable.
- Mientras que dos signos iguales seguidos (==) significa comparación, es decir, decir si es verdad o mentira que dos expresiones son iguales.

Python permite encadenar varias comparaciones y el resultado será verdadero si y sólo si todas las comparaciones lo son.

In [106]:
```
1 4 == 3 + 1 > 2
```

Out[106]: True

In [107]:
```
1 2 != 1 + 1 > 0
```

Out[107]: False

También puede hacerse en Python:

In [108]:
```
1 4 == 3 + 1 and 3 + 1 > 2
```

Out[108]: True

In [109]:
```
1 2 != 1 + 1 and 1 + 1 > 0
```

Out[109]: False