

Objetos: identificador, tipo y valor

En Python, cada dato (número, cadena, lista, etc.) que aparece en un programa es un objeto. Cada objeto tiene:

- un identificador,
- un tipo y
- un valor

identificador: Cada objeto tiene un identificador único que se puede conocer mediante la función `id()`:

```
In [1]: 1 id(5) # Este es el identificador del objeto número entero "5"
```

```
Out[1]: 1476686277040
```

```
In [2]: 1 id(3.14) # Este es el identificador del objeto número decimal "3.14"
```

```
Out[2]: 1476770063440
```

```
In [3]: 1 id("hola") # Este es el identificador del objeto cadena "hola"
```

```
Out[3]: 1476773336240
```

```
In [4]: 1 id([3, 4]) # Este es el identificador del objeto lista "[3, 4]"
```

```
Out[4]: 1476773336128
```

En la versión de Python para Windows (programada en C, por lo que a veces se denomina CPython), **el identificador de un objeto es su posición en la memoria de la computadora**.

tipo: Cada objeto tiene un tipo que se puede # conocer mediante la función `type()`:

```
In [5]: 1 type(5)
```

```
Out[5]: int
```

```
In [6]: 1 type(3.14)
```

```
Out[6]: float
```

```
In [7]: 1 type("hola")
```

```
Out[7]: str
```

```
In [8]: 1 type([3, 4])
```

```
Out[8]: list
```

valor: El valor que contiene el objeto es el propio dato.

Creación y destrucción de objetos

Los objetos se crean y se destruyen a medida que se usan y dejan de usar. Cuando se destruye y crea un nuevo objeto, el identificador puede o no coincidir con identificadores utilizados anteriormente. En los ejemplos siguientes, se observa como los objetos (números grandes o cadenas) tienen cada vez un identificador distinto:

```
In [9]: 1 id(12345)
```

```
Out[9]: 1476773688016
```

```
In [10]: 1 id(12346)
```

```
Out[10]: 1476773687504
```

```
In [11]: 1 id(12345)
```

```
Out[11]: 1476773688272
```

```
In [12]: 1 id(12346)
```

```
Out[12]: 1476773688240
```

```
In [13]: 1 id("hola")
```

```
Out[13]: 1476773573296
```

```
In [14]: 1 id("adios")
```

```
Out[14]: 1476773574064
```

```
In [15]: 1 id("hola")
```

```
Out[15]: 1476773557488
```

```
In [16]: 1 id("adios")
```

```
Out[16]: 1476773573296
```

Sin embargo, en el caso de números pequeños, el identificador no cambia.

```
In [17]: 1 id(5)
```

```
Out[17]: 1476686277040
```

```
In [18]: 1 id(6)
```

```
Out[18]: 1476686277072
```

```
In [19]: 1 id(5)
```

```
Out[19]: 1476686277040
```

```
In [20]: 1 id(6)
```

```
Out[20]: 1476686277072
```

Python crea el objeto "número 5" cuando ve que en el programa se va a utilizar un 5. Luego se ejecutaría la función `id()` que obtiene el identificador del objeto.

Si la primera instrucción hubiera sido la asignación de la variable, de la misma forma se hubiera creado primero el objeto "número 5" y después se le hubiera asociado la etiqueta "a" a dicho objeto.

Al tener una etiqueta que hace referencia a un objeto, el objeto ya no se destruye. En el ejemplo siguiente, se observa como el objeto "hola" mantiene el mismo identificador debido a que la etiqueta "a" hace referencia a él.

```
In [21]: 1 a = "hola"
        2 id("hola")
```

```
Out[21]: 1476773577648
```

```
In [22]: 1 id("hola")
```

```
Out[22]: 1476773577648
```

```
In [23]: 1 id("adios")
```

```
Out[23]: 1476773577264
```

```
In [24]: 1 id("adios")
```

```
Out[24]: 1476773600816
```

El tipo de una variable coincide con el tipo del objeto al que hace referencia la variable, como se comprueba en el ejemplo siguiente:

```
In [25]: 1 type(5) # El tipo del objeto número entero "5" es int
```

```
Out[25]: int
```

```
In [26]: 1 a = 5
        2 type(a) # El tipo de la variable a el mismo que el del objeto número entero "5"
```

```
Out[26]: int
```

Cuando cambiamos el valor de una variable, en la mayoría de los casos, la variable pasa simplemente de hacer referencia a un objeto a hacer referencia a otro (aunque en algunos casos se puede estar modificando el valor del objeto), ejemplo:

```
In [27]: 1 id(1) # Este es el identificador del objeto "1"
```

```
Out[27]: 1476686276912
```

```
In [28]: 1 id(2) # Este es el identificador del objeto "2"
```

```
Out[28]: 1476686276944
```

```
In [29]: 1 a = 1 # Si la variable "a" se asigna al objeto "1" ...
        2 id(a) # ... La variable "a" tiene el mismo identificador que el objeto "1"
```

```
Out[29]: 1476686276912
```

```
In [30]: 1 a = 2 # Pero si la variable "a" se asigna al objeto "2" ...
        2 id(a) # ... La variable "a" tiene el mismo identificador que el objeto "2"
```

```
Out[30]: 1476686276944
```

```
In [31]: 1 b = a # Si La variable "b" se asigna al mismo objeto que la variable "a" ...
        2 id(b) # ... La variable "b" tiene el mismo identificador que el objeto "2"
```

Out[31]: 1476686276944

En Python algunos tipos de objetos son inmutables, pero otros objetos son mutables:

- 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 diccionarios son objetos mutables.

Variables y objetos inmutables

Como los objetos inmutables no se pueden modificar, al modificar las variables que hacen referencia a objetos inmutables, las variables pasan siempre a hacer referencia a otros objetos.

```
In [32]: 1 id(5), id(7) # Estos son Los identificadores de Los objetos "5" y "7"
```

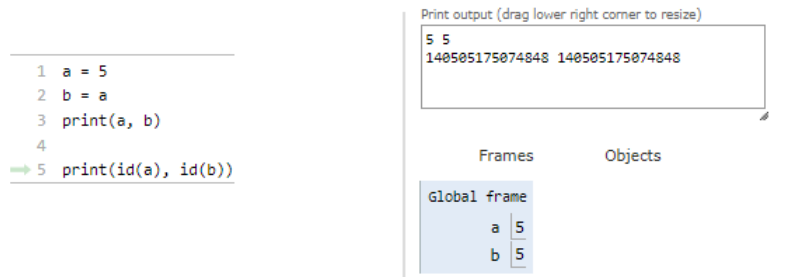
Out[32]: (1476686277040, 1476686277104)

```
In [33]: 1 a = 5 # La etiqueta "a" se asigna al objeto "5" y ...
        2 b = a # ... La etiqueta "b" se asigna al mismo objeto que "a"
        3 a, b # Tanto "a" como "b" valen 5 ...
```

Out[33]: (5, 5)

```
In [34]: 1 id(a), id(b) # ... porque a y b tienen el mismo identificador que "5"
```

Out[34]: (1476686277040, 1476686277040)

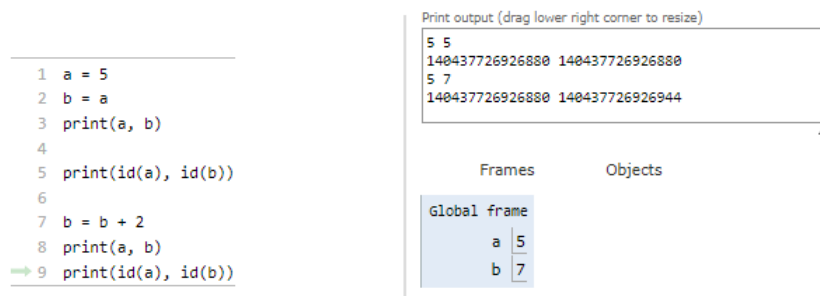


```
In [35]: 1 b = b + 2 # Pero si aumentamos el valor de b en dos unidades ...
        2 a, b # ... "a" sigue valiendo 5 pero "b" vale 7
```

Out[35]: (5, 7)

```
In [36]: 1 id(a), id(b) # La variable "b" hace ahora referencia al objeto "7"
```

Out[36]: (1476686277040, 1476686277104)



En este ejemplo, modificamos el valor de la variable b ...

```
In [68]: 1 b = 5
        2 b = b + 2
        3 b
```

Out[68]: 7

Si "aumentamos" el valor de b en 2 unidades ... en realidad Python no aumenta el valor de "b" sino que:

- Evalúa la expresión de la derecha de la igualdad, es decir, suma 5 (el valor del objeto al que hace referencia la variable "b" en ese momento) y 2, obteniendo el valor "7"
- Crea el objeto que contiene el resultado (el objeto "7") (en el programa del ejemplo, el objeto 7 ya existía porque habíamos pedido su identificador antes, pero si no hubiera existido todavía, se hubiera creado en ese momento)
- Asigna la variable "b" al objeto "7"

Lo mismo ocurriría en el caso de las cadenas de texto, otro tipo de objeto inmutable:

```
In [38]: 1 id("hola"), id("hola y adios")
```

```
Out[38]: (1476772597552, 1476772597232)
```

```
In [39]: 1 a = "hola"
2 b = a
3 a, b
```

```
Out[39]: ('hola', 'hola')
```

```
In [40]: 1 id(a), id(b)
```

```
Out[40]: (1476772610160, 1476772610160)
```

```
In [41]: 1 b = b + " y adios"
2 a, b
```

```
Out[41]: ('hola', 'hola y adios')
```

```
In [42]: 1 id(a), id(b)
```

```
Out[42]: (1476772610160, 1476773602608)
```

Podemos observar que el id de la cadena **"hola y adios"** (aquella cuyo identificador se solicita en la primera instrucción) no es el mismo que el de la que aparece más adelante (al agregarle **"y adios"** a **"hola"**), esto se debe a que Python ha destruido y vuelto a crear el objeto.

Variables y objetos mutables

Sin embargo, en el caso de los objetos **mutables** tenemos dos posibilidades:

- a) **modificar las variables o**
- b) **modificar los objetos mutables a los que hacen referencia.**

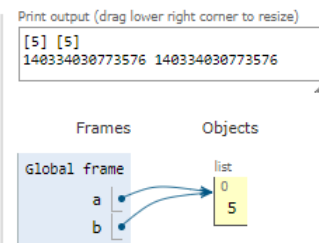
Por ejemplo, en el caso de las listas: En algunos casos las variables pasan a hacer referencia a otros objetos, como ocurre con los objetos inmutables.

```
In [43]: 1 a = [5] # La variable "a" se asigna al objeto "lista [5]" y ...
2 b = a
```

```
In [44]: 1 id(a), id(b) # ... por eso "a" y "b" tienen el mismo identificador
```

```
Out[44]: (1476772610880, 1476772610880)
```

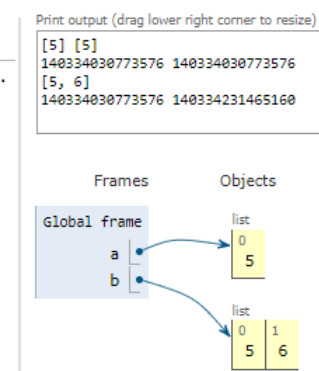
```
1 a = [5] # La variable "a" se asigna al objeto "lista [5]" y ...
2 b = a
3 print(a, b)
4 print(id(a), id(b))
5
6
7
8
```



```
In [45]: 1 b = b + [6] # Pero si agregamos un elemento a "b" ...
2 b
```

```
Out[45]: [5, 6]
```

```
1 a = [5] # La variable "a" se asigna al objeto "lista [5]" y ...
2 b = a
3 print(a, b)
4 print(id(a), id(b))
5
6
7 b = b + [6] # Pero si añadimos un elemento a "b" ...
8 print(b)
9 print(id(a), id(b))
10
11
12
13
```



Si volvemos a asignar el valor de a, a b, obtenemos el mismo valor e identificador anterior:

```
In [46]: 1 b = a # ... La variable "b" se asigna al mismo objeto que "a" ...
        2 b

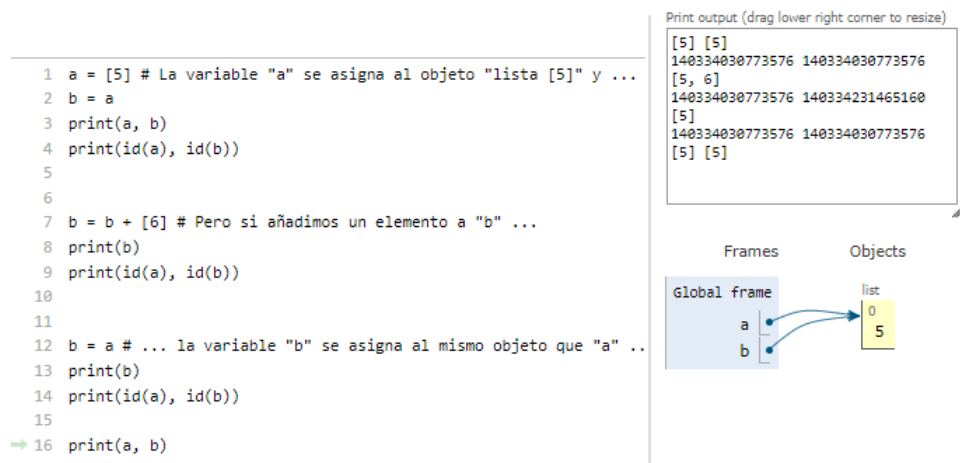
Out[46]: [5]
```

```
In [47]: 1 id(a), id(b) # ... La variable "b" hace ahora referencia al objeto "[5, 6]" ...

Out[47]: (1476772610880, 1476772610880)
```

```
In [48]: 1 a, b

Out[48]: ([5], [5])
```



b = b + [6] # agregamos un elemento a "b"

pero Python no agrega un elemento a "b" sino que:

- Evalúa la expresión de la derecha de la igualdad, es decir, añade el elemento "6" a una lista [5], obteniendo la lista [5, 6]
- Crea el objeto [5, 6]
- Asigna la variable "b" al objeto [5, 6]

Existen otros casos dónde se modifican los objetos mutables a los que hacen referencia las variables. Ejemplo:

```
In [49]: 1 a = [5] # La variable "a" identifica a La lista [5] ...

In [50]: 1 b = a # ... La variable "b" identifica al mismo valor que a...

In [51]: 1 id(a), id(b) # ... por eso "a" y "b" tienen el mismo identificador

Out[51]: (1476773634880, 1476773634880)
```

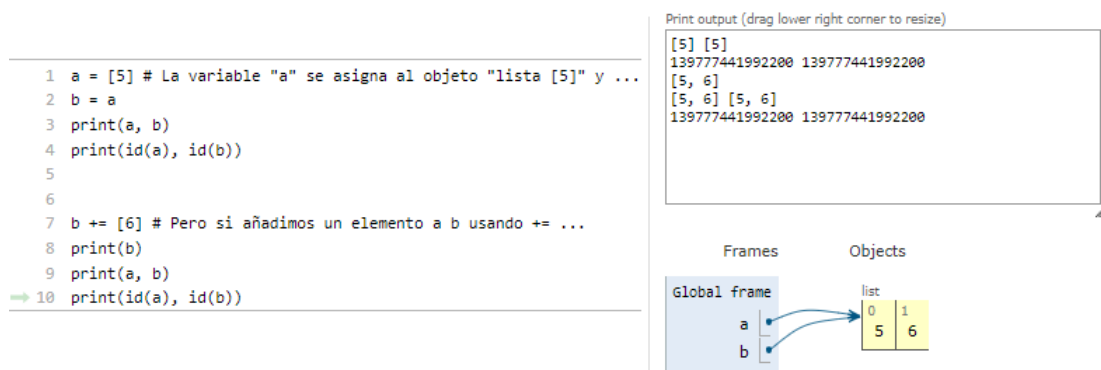
```
In [52]: 1 b += [6] # Pero si agregamos un elemento a b usando += ...

In [53]: 1 id(a), id(b) # ... tanto "a" como "b" siguen haciendo referencia a La misma lista ...

Out[53]: (1476773634880, 1476773634880)
```

```
In [54]: 1 a, b # ... que ha cambiado con respecto a su valor inicial

Out[54]: ([5, 6], [5, 6])
```



b += [6] # agregamos un elemento a b usando +=

Pero Python:

- Sí que agrega el elemento "6" al objeto lista [5], convirtiéndola en la lista [5, 6]
- Tanto "a" como "b" siguen haciendo referencia a la misma lista, que ahora es [5, 6]

Lo mismo pasaría si se utilizara el método `append()`:

```
In [55]: 1 a = [5] # La variable "a" identifica a la lista [5] ...
```

```
In [56]: 1 b = a # ... La variable "b" identifica al mismo valor que a...
```

```
In [57]: 1 id(a), id(b) # ... por eso "a" y "b" tienen el mismo identificador
```

```
Out[57]: (1476772502912, 1476772502912)
```

```
In [58]: 1 b.append(6) # Pero si añadimos un elemento a b usando append() ...
```

```
In [59]: 1 id(a), id(b) # ... tanto "a" como "b" siguen haciendo referencia a la misma lista ...
```

```
Out[59]: (1476772502912, 1476772502912)
```

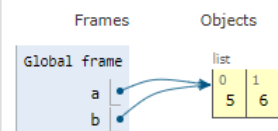
```
In [60]: 1 a, b # ... ha cambiado a respecto de su valor inicial.
```

```
Out[60]: ([5, 6], [5, 6])
```

```
1 a = [5] # La variable "a" se asigna al objeto "lista [5]" y ...
2 b = a
3 print(a, b)
4 print(id(a), id(b))
5
6
7 b.append(6) # Pero si añadimos un elemento a b usando append() ...
8 print(b)
9 print(a, b)
→ 10 print(id(a), id(b))
```

Print output (drag lower right corner to resize)

```
[5] [5]
139730711469512 139730711469512
[5, 6]
[5, 6] [5, 6]
139730711469512 139730711469512
```



Más sobre creación y destrucción de objetos

Los objetos **mutables** son todos distintos, es decir, Python crea objetos cada vez que utilizamos un dato. Por ejemplo, en el caso de las listas:

```
In [61]: 1 a = [5] # Al crear dos variables con la misma lista ...
```

```
2 b = [5]
```

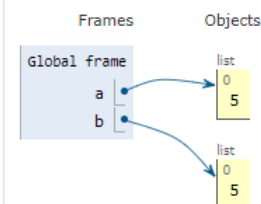
```
3 id(a), id(b) # ... Python crea dos objetos y cada variable hace referencia a uno de ellos.
```

```
Out[61]: (1476773787136, 1476773715840)
```

```
1 a = [5]
2 b = [5]
3
4 print(a, b)
→ 5 print(id(a), id(b))
6 # ... Python crea dos objetos y cada variable hace referencia
7 # a uno de ellos.
```

Print output (drag lower right corner to resize)

```
[5] [5]
139904309923400 139904330907016
```



En el caso de objetos **inmutables**, Python a veces crea objetos distintos y a veces no. Por ejemplo, como se vió anteriormente, en el caso de los números enteros pequeños (o cadenas) sólo crea un objeto:

```
In [62]: 1 a = 5 # Al crear dos variables que hacen referencia al mismo valor ...
```

```
2 b = 5
```

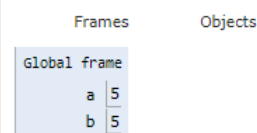
```
3 id(a), id(b) # ... Python crea un sólo objeto
```

```
Out[62]: (1476686277040, 1476686277040)
```

```
1 a = 5
2 b = 5
3 print(a, b)
→ 4 print(id(a), id(b))
5 # ... Python crea un sólo objeto
```

Print output (drag lower right corner to resize)

```
5 5
140414543104032 140414543104032
```



Pero en el caso de los decimales, crea objetos distintos.

```
In [63]: 1 a = 3.14 # Al crear dos variables que hacen referencia al mismo valor ...
        2 b = 3.14
        3 id(a), id(b) # ... Python crea dos objetos
```

Out[63]: (1476773909744, 1476773909392)

Al escribir tuplas de identificadores de valores distintos, en el caso de objetos inmutables Python crea varios objetos, pero en el caso de objetos mutables Python parece crear sólo uno:

```
In [64]: 1 id(3), id(4) # Al crear una tupla de dos identificadores de números ...
        2 # ... Python crea los dos objetos.
```

Out[64]: (1476686276976, 1476686277008)

```
In [65]: 1 id([3]), id([4]) # Pero al crear una tupla de dos identificadores de listas ...
        2 # ... Python parece crear un único objeto.
```

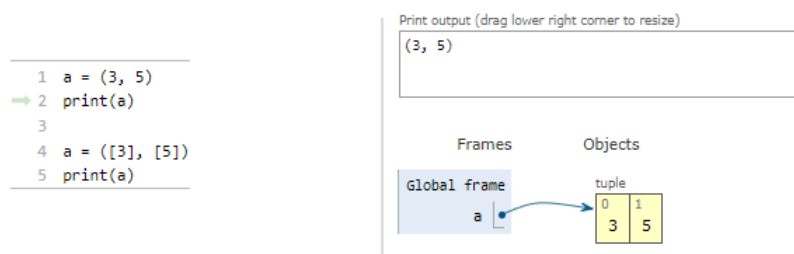
Out[65]: (1476773517568, 1476773517568)

Lo que ocurre es que como el objeto "lista [3]" no se utiliza para nada más (no hay ninguna variable haciendo referencia a él), nada más crearlo se destruye. Al crear el objeto "lista [4]", Python le da el mismo identificador que había utilizado antes.

Las tuplas son objetos inmutables, es decir, que no se pueden modificar:

```
In [66]: 1 a = (3, 5)
        2 a
```

Out[66]: (3, 5)



```
In [67]: 1 a[0] = 4
```

```
-----
TypeError                                 Traceback (most recent call last)
Input In [67], in <cell line: 1>()
----> 1 a[0] = 4

TypeError: 'tuple' object does not support item assignment
```

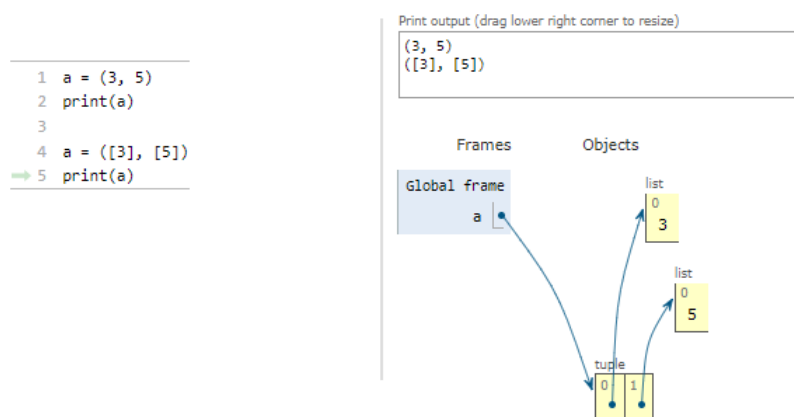
```
In [69]: 1 a
```

Out[69]: (3, 5)

Pero si una tupla incluye objetos mutables, aunque se produzca un error por intentar un objeto inmutable, el objeto mutable sí que se modifica:

```
In [70]: 1 a = ([3], [5])
        2 a
```

Out[70]: ([3], [5])



```
In [71]: 1 a[0] += [4]
```

TypeError Traceback (most recent call last)
Input In [71], in <cell line: 1>()
----> 1 a[0] += [4]

TypeError: 'tuple' object does not support item assignment

```
In [72]: 1 a[0] += [3]
```

TypeError Traceback (most recent call last)
Input In [72], in <cell line: 1>()
----> 1 a[0] += [3]

TypeError: 'tuple' object does not support item assignment

```
In [73]: 1 a
```

Out[73]: ([3, 4, 3], [5])

La explicación es que el objeto tupla está formado por dos objetos listas y Python primero modifica el objeto lista (lo que está permitido) pero después detecta que se ha intentado modificar el objeto tupla (lo que no está permitido). Para modificar el objeto mutable sin generar mensajes de error, debemos utilizar el método **append()**:

```
In [74]: 1 a = ([3], [5])  
2 a
```

Out[74]: ([3], [5])

```
In [75]: 1 a[0].append(4)  
2 a
```

Out[75]: ([3, 4], [5])

```
1 a = ([3], [5])  
2 print(a)  
3  
4  
5 a[0].append(4)  
⇒ 6 print(a)
```

