😎 Variables y Objetos

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]: id(5) # Este es el identificador del objeto número entero "5"
Out[1]: 2080026487216

In [2]: id(3.14) # Este es el identificador del objeto número decimal "3.14"
Out[2]: 2080114435504

In [3]: id("hola") # Este es el identificador del objeto cadena "hola"
Out[3]: 2080114410224

In [4]: id([3, 4]) # Este es el identificador del objeto lista "[3, 4]"
Out[4]: 2080114497216
```

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**, pero podría ser cualquier otro valor, lo único importante es que cada objeto tenga un identificador distinto.

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

```
In [5]: type(5)
Out[5]: int

In [6]: type(3.14)
Out[6]: float

In [7]: type("hola")
Out[7]: str

In [8]: type([3, 4])
Out[8]: list

In [9]: # valor: El valor que contiene el objeto es el propio dato.
```

Creación y destrucción de objetos

Python crea los objetos a medida que se necesitan y los destruye cuando ya no necesitan (y ninguna variable hace referencia a ellos).

Cuando se destruye y crea un nuevo objeto, el identificador puede o no coincidir con identificadores utilizados anteriormente.

En Windows, en el caso de los números enteros pequeños Python parece no destruir los objetos (o puede que les asigne el mismo identificador). En el ejemplo siguiente, se observa como los objetos "5" y "6" tienen siempre el mismo identificador.

```
In [10]: id(5)
Out[10]: 2080026487216

In [11]: id(6)
Out[11]: 2080026487248

In [12]: id(5)
Out[12]: 2080026487216

In [13]: id(6)
Out[13]: 2080026487248
```

Pero normalmente, los objetos se crean y se destruyen a medida que se usan y dejan de usar. En los ejemplos siguientes, se observa como los objetos (números grandes o cadenas) tienen cada vez un identificador distinto:

```
In [14]: id(12345)
Out[14]: 2080114436912

In [15]: id(12346)
Out[15]: 2080114437040

In [16]: id(12345)
Out[16]: 2080114437872

In [17]: id(12346)
Out[17]: 2080114437264

In [18]: id("hola")
Out[18]: 2080114659632

In [19]: id("adios")
Out[19]: 2080114676336

In [20]: id("hola")
Out[20]: 2080114675568

In [21]: id("adios")
Out[21]: 2080114673776
```

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

```
In [22]: a = "hola"
         id("hola")
Out[22]: 2080114724784

In [23]: id("adios")
Out[23]: 2080114836656

In [24]: id("hola")
Out[24]: 2080114724784

In [25]: id("adios")
Out[25]: 2080114643568
```

In [26]:
```
id("hola")
```

Out[26]: 2080114724784

Las **variables** en PYthon son simples referencias a los objetos. Una variable es una especie de etiqueta o de alias para referirnos al objeto. En Python las variables también tienen identificador, tipo y valor, pero esas tres características son las del objeto al que hace referencia.

Cuando asignamos un valor a una variable, lo que estamos creando es una etiqueta para referirnos al objeto que contiene el dato. Por ejemplo, si a una variable se le asigna el número 5, Python crea el objeto número 5 y la variable hace referencia a ese objeto, como se comprueba en el ejemplo siguiente:

In [27]:
```
id(5) # Este es el identificador del objeto número entero "5"
```

Out[27]: 2080026487216

In [28]:
```
a = 5
id(a) # El identificador de la variable a es el mismo que el del número entero "5"
```

Out[28]: 2080026487216

Python crea el objeto "número 5" cuando ve que en el programa se va a utilizar un 5.

En el ejemplo anterior se crearía el objeto y después 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.

Lógicamente, 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 [29]:
```
type(5) # El tipo del objeto número entero "5" es int
```

Out[29]: int

In [30]:
```
a = 5
type(a) # El tipo de la variable a el mismo que el del objeto número entero "5"
```

Out[30]: 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, como se comprueba en el ejemplo siguiente:

In [31]:
```
id(1)
# Este es el identificador del objeto "1"
```

Out[31]: 2080026487088

In [32]:
```
id(2)
 # Este es el identificador del objeto "2"
```

Out[32]: 2080026487120

In [33]:
```
a = 1 # Si la variable "a" se asigna al objeto "1" ...
id(a) # ... la variable "a" tiene el mismo identificador que el objeto "1"
```

Out[33]: 2080026487088

In [34]:
```
a = 2 # Pero si la variable "a" se asigna al objeto "2" ...
id(a) # ... la variable "a" tiene el mismo identificador que el objeto "2"
```

Out[34]: 2080026487120

In [35]:
```
b = a # Si la variable "b" se asigna al mismo objeto que la variable "a" ...
id(b) # ... la variable "b" tiene el mismo identificador que el objeto "2"
```

Out[35]: 2080026487120

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 [36]:
```
id(5), id(7)
# Estos son los identificadores de los objetos "5" y "7"
```

Out[36]: (2080026487216, 2080026487280)

In [37]:
```
a = 5 # La etiqueta "a" se asigna al objeto "5" y ...
b = a # ... la etiqueta "b" se asigna al mismo objeto que "a"
a, b # Tanto "a" como "b" valen 5 ...
```

Out[37]: (5, 5)

In [38]:
```
id(a), id(b) # ... porque a y b tienen el mismo identificador que "5"
```

Out[38]: (2080026487216, 2080026487216)

In [39]:
```
b = b + 2 # Pero si aumentamos el valor de b en dos unidades ...
a, b # ... "a" sigue valiendo 5 pero "b" vale 7
```

Out[39]: (5, 7)

In [40]:
```
id(a), id(b) # La variable "b" hace ahora referencia al objeto "7"
```

Out[40]: (2080026487216, 2080026487280)

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

In [41]:
```
b = b + 2
```

Pero 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 [42]:
```
id("hola"), id("hola y adios")
```

Out[42]: (2080114916784, 2080114637552)

In [43]:
```
a = "hola"
b = a
a, b
```

Out[43]: ('hola', 'hola')

In [44]:
```
id(a), id(b)
```

Out[44]: (2080114496688, 2080114496688)

In [45]:
```
b = b + " y adios"
a, b
```

Out[45]:
```
('hola', 'hola y adios')
```

In [46]:
```
id(a), id(b)
```

Out[46]:
```
(2080114496688, 2080114987632)
```

En este ejemplo podemos observar que el id de la primera 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"**), pero eso 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 [47]:
```
a = [5] # La variable "a" se asigna al objeto "lista [5]" y ...
b = a
```

In [48]:
```
id(a), id(b) # ... por eso "a" y "b" tienen el mismo identificador
```

Out[48]:
```
(2080114916096, 2080114916096)
```

In [49]:
```
b = b + [6] # Pero si añadimos un elemento a "b" ...
b
```

Out[49]:
```
[5, 6]
```

In [50]:
```
b = a # ... la variable "b" se asigna al mismo objeto que "a" ...
b
```

Out[50]:
```
[5]
```

In [51]:
```
id(a), id(b) # ... la variable "b" hace ahora referencia al objeto "[5, 6]" ...
```

Out[51]:
```
(2080114916096, 2080114916096)
```

In [52]:
```
a, b # ... y por eso "a" y "b" son distintos.
```

Out[52]:
```
([5], [5])
```

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

b = b + [6] # Pero si agregamos un elemento a "b" ... ... en realidad Python no añade 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]**

En otros casos sí que se modifican los objetos mutables a los que hacen referencia las variables:

In [53]:
```
a = [5] # La variable "a" identifica a la lista [5] ...
```

In [54]:
```
b = a # ... la variable "b" identifica al mismo valor que a...
```

In [55]:
```
id(a), id(b) # ... por eso "a" y "b" tienen el mismo identificador
```

Out[55]:
```
(2080114839232, 2080114839232)
```

In [56]:
```
b += [6] # Pero si añadimos un elemento a b usando += ...
```

In [57]:
```
id(a), id(b) # ... tanto "a" como "b" siguen haciendo referencia a la misma lista ...
```

Out[57]:
```
(2080114839232, 2080114839232)
```

In [58]:
```
a, b # ... que ha cambiado con respecto a su valor inicial
```

Out[58]:
```
([5, 6], [5, 6])
```

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

b += [6] # Pero si agregamos un elemento a b usando += ... ... en realidad 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 [59]:
```
a = [5] # La variable "a" identifica a la lista [5] ...
```

In [60]:
```
b = a # ... la variable "b" identifica al mismo valor que a...
```

In [61]:
```
id(a), id(b) # ... por eso "a" y "b" tienen el mismo identificador
```

Out[61]:
```
(2080114988544, 2080114988544)
```

In [62]:
```
b.append(6) # Pero si añadimos un elemento a b usando append() ...
```

In [63]:
```
id(a), id(b) # ... tanto "a" como "b" siguen haciendo referencia a la misma lista ...
```

Out[63]:
```
(2080114988544, 2080114988544)
```

In [64]:
```
a, b # ... que ha cambiado con respecto a su valor inicial
```

Out[64]:
```
([5, 6], [5, 6])
```

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 [65]:
```
a = [5] # Al crear dos variables con la misma lista ...
b = [5]
id(a), id(b) # ... Python crea dos objetos y cada variable hace referencia a uno de ellos.
```

Out[65]:
```
(2080115060160, 2080114987968)
```

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

In [66]:
```
a = 5 # Al crear dos variables que hacen referencia al mismo valor ...
b = 5
id(a), id(b) # ... Python crea un sólo objeto
```

Out[66]:
```
(2080026487216, 2080026487216)
```

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

In [67]:
```
a = 3.14 # Al crear dos variables que hacen referencia al mismo valor ...
b = 3.14
id(a), id(b) # ... Python crea dos objetos
```

Out[67]:
```
(2080115040656, 2080115041648)
```

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 [68]:
```
id(3), id(4) # Al crear una tupla de dos identificadores de números ...
# ... Python crea los dos objetos.
```

Out[68]:
```
(2080026487152, 2080026487184)
```

In [69]:
```
id([3]), id([4]) # Pero al crear una tupla de dos identificadores de listas ...
# ... Python parece crear un único objeto.
```

Out[69]:
```
(2080115030464, 2080115030464)
```

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

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

In [71]:
```
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]:
```
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]:
```
a
```

Out[73]:
```
(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 [74]:
```
a = ([3], [5])
a
```

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

In [75]:
```
a[0] += [4]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [75], in <cell line: 1>()
----> 1 a[0] += [4]

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

In [76]:
```
a[0] += [3]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [76], in <cell line: 1>()
----> 1 a[0] += [3]

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

In [77]:
```
a
```

Out[77]:
```
([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 [78]:
```
a = ([3], [5])
a
```

Out[78]:
```
([3], [5])
```

In [79]:
```
a[0].append(4)
a
```

Out[79]:
```
([3, 4], [5])
```

In [ ]:
```

```