Tipos de datos mutables e inmutables

Un concepto preliminar importante: nosotros nombramos cada objeto mediante su identificador, pero en la memoria, lo que identifica un objeto es la dirección de memoria en que se almacena. En Python, la dirección de memoria de un objeto se puede obtener con la función "id":

In [1]:

```
lista = [1, 2, 3]
print(lista)
print(id(lista))
n = 7
print(n)
print(id(n))
```

```
[1, 2, 3]
2236636309696
7
140735390031808
```

Vamos a cambiar el contenido de una lista. ¿Afecta esto a la dirección de memoria en que se almacena?

In [2]:

```
lista = [1, 2, 3]
print(lista)
print(id(lista))
lista[0] = 666
print(lista)
print(id(lista))
```

```
[1, 2, 3]
2236636311104
[666, 2, 3]
2236636311104
```

Vamos a cambiar el contenido de un entero. ¿Afecta esto a la dirección de memoria en que se almacena?

In [3]:

```
n = 100
print(n)
print(id(n))
n = 666
print(n)
print(id(n))
```

```
100
140735390034784
666
2236636228976
```

Vemos que las modificaciones efectuadas sobre una lista pueden alterarla in situ, y la lista ha mutado. Si se modifica un entero, deja de existir y se crea otro entero: el entero no ha podido mutar. En resumen, ahora podemos afirmar lo siguiente:

- Las listas son objetos mutables.
- Los enteros no son objetos mutables.

Consecuencia: datos mutables e inmutables como parámetros

Una lista y un entero pueden ser parámetros de una función. La función puede cambiar internamente su valor. ¿Cómo afecta esto a los parámetros reales?

In [4]:

```
def cambio(numeros, a):
    numeros[0] = numeros[0] * 100
    a = a + 1000

lista = [1, 2, 3]
n = 5
print(id(lista), lista)
print(id(n), n)
cambio(lista, n)
print(id(lista), lista)
print(id(n), n)
```

```
2236636301120 [1, 2, 3]
140735390031744 5
2236636301120 [100, 2, 3]
140735390031744 5
```

En efecto: los parámetros de una función son siempre la referencia. Si la función modifica un objeto mutable, el parámetro real se ve afectado, porque la función opera sobre dicho objeto allí donde está almacenado. En cambio, si un objeto inmutable cambia en la función, el nuevo valor se coloca en un lugar distinto de la memoria, y no afecta al parámetro real.

Para simplificar, podríamos decir que los objetos mutables se pasan como parámetros por referencia o por variable, y que los inmutables pasan como parámetros por valor... aunque en realidad, todos pasan como referencias...

Otra consecuencia: la asignación de datos mutables...

Es mejor que lo veas con un ejemplo.

In [5]:

```
lista_a = [1, 2, 3]
lista_b = lista_a
print(lista_a, lista_b)
lista_a[0] = 666
print(lista_a, lista_b)
```

```
[1, 2, 3] [1, 2, 3]
[666, 2, 3] [666, 2, 3]
```

Observa el cambio que se ha realizado sobre la `lista_b` "sin que la hayamos cambiado".

Hagamos lo mismo con enteros...

In [6]:

```
a = 7
b = a
print(a, b)
a = 666
print(a, b)
```

```
7 7
666 7
```

Seguramente, ahora puedes explicar la diferencia en el comportamiento de listas y enteros.

¿Y si queremos copiar una lista en otra variable, para que se comporte con independencia?

In [7]:

```
lista_a = [1, 2, 3]
lista_b = lista_a.copy()
print(lista_a, lista_b)
lista_a[0] = 666
print(lista_a, lista_b)
```

```
[1, 2, 3] [1, 2, 3]
[666, 2, 3] [1, 2, 3]
```

Lamentablemente, copy únicamente clona el primer nivel de referencias internas.

Si queremos que una estructura se clone al completo, esto es, con todos los niveles de profundidad...

In [8]:

```
a = [0, [1, 2]]
b = a
c = a.copy()

import copy
d = copy.deepcopy(a)

print("a = ", a)
print("b = ", b)
print("c = ", c)
print("d = ", d)
print()

a[0] = "cambio"
a[1][1] = "cambio"

print("a = ", a)
print("b = ", b)
print("c = ", c)
print("d = ", d)
```

```
a =  [0, [1, 2]]
b =  [0, [1, 2]]
c =  [0, [1, 2]]
d =  [0, [1, 2]]

a =  ['cambio', [1, 'cambio']]
b =  ['cambio', [1, 'cambio']]
c =  [0, [1, 'cambio']]
d =  [0, [1, 2]]
```

¿Y si queremos que un entero se copie en otro compartiendo su referencia? ¿O que un entero sea un parámetro por variable de una función?

La respuesta es la siguiente: podemos *envolverlo* en una lista. Veámoslo:

In [9]:

```
a = [7]
b = a
print(a, b)
a = [666]
print(a, b)
print("------------------------------")
def cambio(nn):
    nn[0] = nn[0] + 1000
cambio(a)
print(a, b)
```

```
[7] [7]
[666] [7]
------------------------------
[1666] [7]
```

Otra forma de replicar los elementos de una lista: ´lista[:]´.

In [10]:

```
a = [1, 2, 3]
b = a[:]
print(a, b)
a[0] = 666
print(a, b)
```

```
[1, 2, 3] [1, 2, 3]
[666, 2, 3] [1, 2, 3]
```

Al igual que `copy()`, únicamente replica el primer nivel:

In [11]:

```
a = [1, [2, 3, 4], 5]
b = a[:]
print(a, b)
a[0] = 666
a[1][1] = 777
print(a, b)
```

```
[1, [2, 3, 4], 5] [1, [2, 3, 4], 5]
[666, [2, 777, 4], 5] [1, [2, 777, 4], 5]
```

Catálogo de objetos mutables e inmutables

- *Inmutables*: int, float, decimal, complex, bool, string, tuple, range, frozenset, bytes
- *Mutables*: list, dict, set, bytearray, user-defined classes (salvo que se declaren específicamente como inmutables

En efecto, las cadenas de caracteres son inmutables, al contrario que las listas. Tú mismo puedes idear algún ejemplo que lo muestre.

Pequeño apéndice

Las ventajas de los objetos mutables son obvias: eficiencia. Los peligros, también. A manera de ejemplo final, observa el siguiente comportamiento:

In [12]:

```
matriz = [[1, 2, 3]] * 3
print(matriz)
matriz[0][0] = 666
print(matriz)
```

```
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
[[666, 2, 3], [666, 2, 3], [666, 2, 3]]
```

In [13]:

```
print(matriz[0], id(matriz[0]))
print(matriz[1], id(matriz[1]))
print(matriz[2], id(matriz[2]))
```

```
[666, 2, 3] 2236636253696
[666, 2, 3] 2236636253696
[666, 2, 3] 2236636253696
```