OBJETOS MUTABLES E INMUTABLES EN PYTHON

Tipos de datos Mutables objetos que pueden modificar su valor	

Lista	>>> lista = [1,2,3]
	>>> type(lista)
	<class 'list'>

Diccionario	>>> diccionario = {"nombre": "Pepe", "edad": 35 }
	>>> type(diccionario)
	<class 'dict'>

Conjunto	>>> conjunto = {"mesa" , "silla", "armario"}
	>>> type(conjunto)
	<class 'set'>

Bytearray	>>> vehiculo = bytearray("camion", "cp1252")
	>>> type(vehiculo)
	<class 'bytearray'>

MemoryView	>>> memv = memoryview(vehiculo)
	>>> type(memv)
	<class 'memoryview'>

Tipos de datos Inmutables objetos que no pueden modificar su valor	

Cadena	>>> nombre = "Pepe"
	>>> type(nombre)
	<class 'str'>

Entero	>>> numero = 20
	>>> type(numero)
	<class 'int'>

Flotante	>>> decimal = 30.5
	>>> type(decimal)
	<class 'float'>

Imaginario	>>> imaginario = 4 + 4J
	>>> type(imaginario)
	<class 'complex'>

Tupla	>>> tupla = ('a', 'b', 'c')
	>>> type(tupla)
	<class 'tuple'>

Rango	>>> rango = range(5)
	>>> type(rango)
	<class 'range'>

Frozenset	>>> frozen = frozenset({"mesa" , "silla", "armario"})
	>>> type(frozen)
	<class 'frozenset'>

Booleano	>>> booleano = True
	>>> type(booleano)
	<class 'bool'>

Byte	>>> serie = b'abc'
	>>> type(serie)
	<class 'bytes'>

None	>>> a = None
	>>> type(a)
	<class 'NoneType'>

Para tener en cuenta:

Cada elemento de datos en un programa Python es un objeto de un tipo o clase específica. La vida de un objeto comienza cuando se crea, en cuyo momento también se crea al menos una referencia a él. Durante la vida útil de un objeto, se pueden crear y eliminar referencias adicionales, el objeto permanece activo, por así decirlo, siempre que haya al menos una referencia a él. Cuando la referencia a un objeto se desactiva, el objeto ya no es accesible. Python eventualmente notará que es inaccesible y reclamará la memoria asignada para que pueda usarse para otra cosa, este proceso se conoce como recolección de basura (**garbage collector**)

La función incorporada **id** () devuelve la identidad de un objeto como un número entero, la identificación se asigna al objeto cuando se crea y es único y constante para ese objeto mientras tenga vida útil. La identificación puede ser diferente cada vez que ejecute el programa a excepción de algún objeto que tiene una identificación única constante, como los enteros de -5 a 256. Python almacena en caché el valor id () de los tipos de datos de uso común.

La función incorporada **type ()** devuelve el tipo de dato de un objeto.

El operador **is** tiene un fin muy específico y es ver si dos identificadores "apuntan" al mismo objeto.

El **==** compara valores de objetos

En Python, cuando en una función, se pasa como parámetro un objeto inmutable lo que ocurriría en realidad es que se crearía una nueva instancia, entonces los cambios no se verían reflejados fuera de la función. Lo que se hace en realidad es *pasar por valor* la referencia al objeto. El caso de los objetos mutables se comportan como *paso por referencia.*

Cuando necesitemos ayuda para conocer que función o métodos podemos usar, en determinados tipos de datos, help() y dir() desde la terminal de Python

dir(<identificador>): *devuelve la lista de atributos y métodos del objeto que le pasemos.*	>>> nombre = 'Python'
	>>> dir(nombre)
	['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', **'split'**, 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

help(<identificador>: *devuelve documentación de lo que le pasemos.*	>>> help(nombre.**split**)
	Help on built-in function split:
	
	split(sep=None, maxsplit=-1) method of builtins.str instance
	Return a list of the words in the string, using sep as the delimiter string.
	
	sep
	The delimiter according which to split the string.
	None (the default value) means split according to any whitespace,
	and discard empty strings from the result.
	maxsplit
	Maximum number of splits to do.
	-1 (the default value) means no limit.
	>>> help("keywords") ….lista de palabras reservadas

Formas de escribir nombre de variables

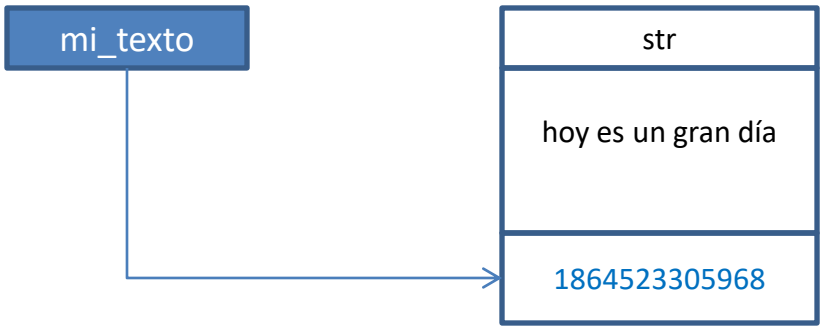
Pascal Case: En esta forma, si un nombre de una variable tiene varias palabras, cada palabra empieza con mayúscula. Por ejemplo: NumeroDeEstudiantes y se recomienda para nombre de clases.

Snake Case: En esta forma se separa cada palabra del nombre de la variable con guión bajo (_) y toda la palabra va en minúsculas. Se recomienda por PEP8 para funciones y nombre de variables. Por ejemplo: numero_de_estudiantes.

Camel Case: En esta forma a partir de la segunda palabra del nombre de la variable comienza en mayúscula. Por ejemplo: numeroDeEstudiantes.

Ejemplo 1 - inmutablidad de cadenas

```
>>> mi_texto = "hoy es un gran día"
>>> print(mi_texto)
hoy es un gran día
>>> mi_texto[0] = "H"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support
item assignment
>>> id(mi_texto)
1864523305968
>>> type(mi_texto)
<class 'str'>
>>>
```

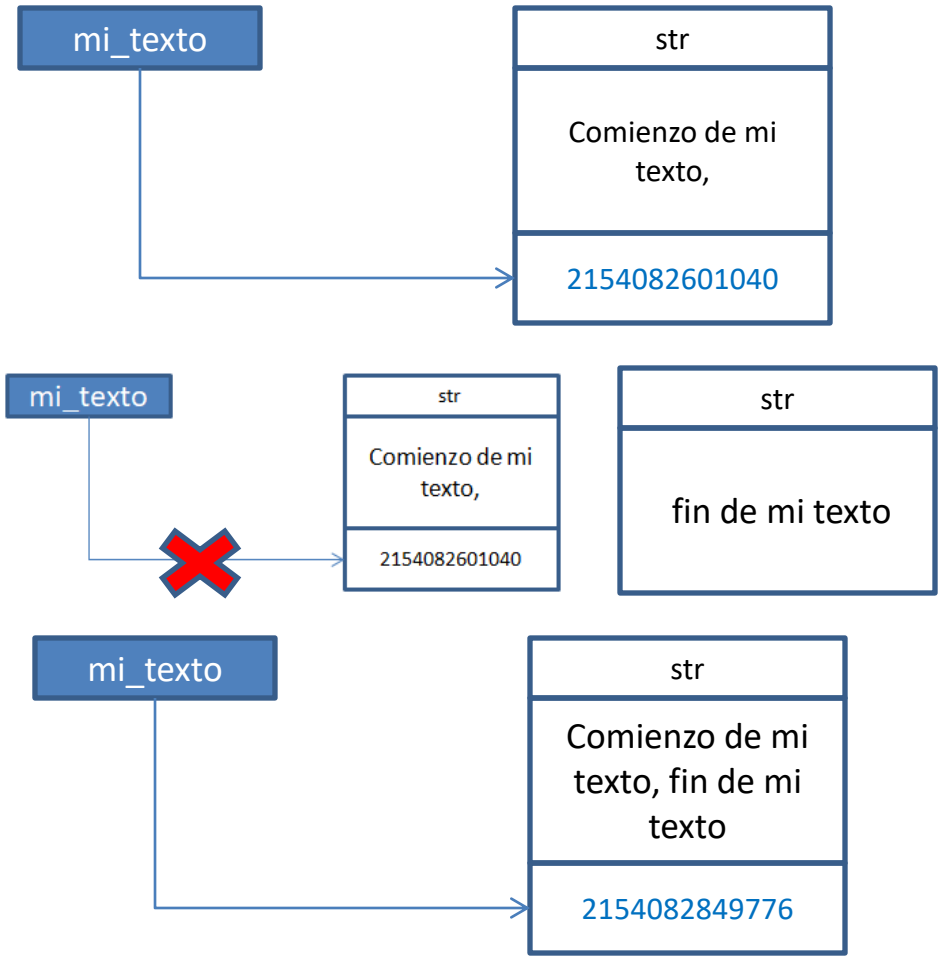


Un objeto inmutable no se puede modificar

Ejemplo 2 - inmutablidad de cadenas

```
>>> mi_texto = "Comienzo de mi texto, "
>>> print(mi_texto)
Comienzo de mi texto,
>>> id(mi_texto)
2154082601040
```

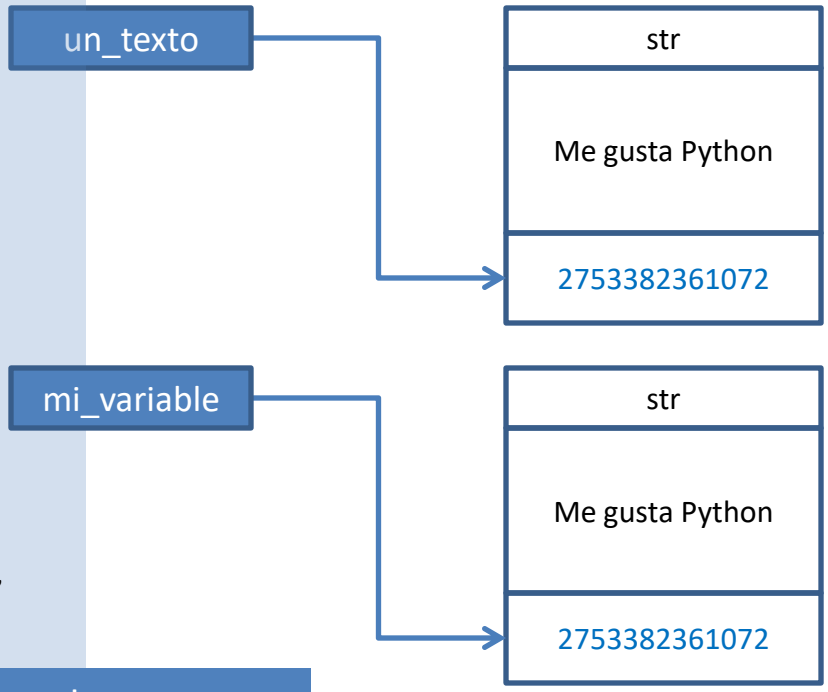
```
>>> mi_texto = mi_texto  + " fin de mi texto"
>>> print(mi_texto)
Comienzo de mi texto,  fin de mi texto
>>> id(mi_texto)
2154082849776
```



Un **string es inmutable** porque, como muestra el ejemplo, en la memoria no se ha ampliado "Comienzo de mi texto, " -guardado previamente- sino que se ha copiado junto con el agregado de " fin de mi texto", en otro lugar de la memoria, para guardarlo completo y el identificador indicará el último valor guardado. Es decir, un string siempre se va a crear de nuevo (inmutable) aunque nosotros creamos que se modifica (falsa creencia de mutabilidad)

Ejemplo 3 - inmutablidad de cadenas

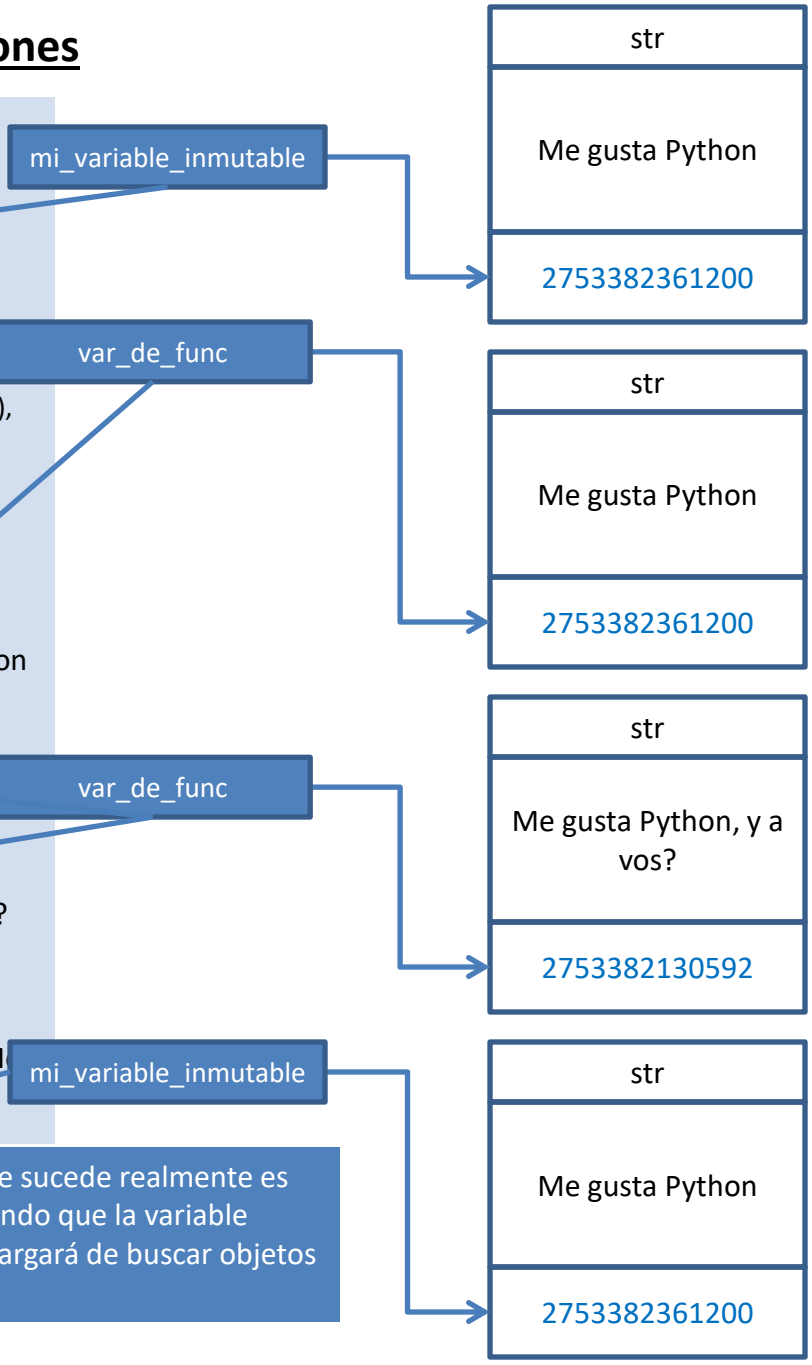
```
>>> un_texto = "Me gusta Python"
>>> id(un_texto)
2753382361072
>>> print(un_texto)
Me gusta Python
>>>

>>> def mi_funcion(mi_variable):
...     print("Id de <mi_variable> dentro de la función: {},
mi_variable: {}\n".format(id(mi_variable),mi_variable))
...
>>> mi_funcion(un_texto)
Id de <mi_variable> dentro de la función: 2753382361072,
mi_variable: Me gusta Python
```



Cuando se pasa como parámetro el identificador de un objeto inmutable a una función, el valor del identificador se copia.

Ejemplo 4 - inmutablidad de cadenas - Conclusiones

```
>>> mi_variable_inmutable = "Me gusta Python"
>>> print("mi_variable_inmutable (id: {}):
{}\n".format(id(mi_variable_inmutable), mi_variable_inmutable))

mi_variable_inmutable (id: 2753382361200): Me gusta Python

>>>
>>> def mi_funcion(var_de_func):
...     print("var_de_func antes de retornar (id: {}): {}\n".format(id(var_de_func),
var_de_func))
...     var_de_func += ", y a vos?"
...     print("var_de_func (id: {}): {}\n".format(id(var_de_func), var_de_func))
...     return var_de_func
...
>>> retorno_de_funcion = mi_funcion(mi_variable_inmutable)
var_de_func antes de retornar (id: 2753382361200): Me gusta Python

var_de_func (id: 2753382130592): Me gusta Python, y a vos?

>>> print("retorno_de_funcion (id: {}):
{}\n".format(id(retorno_de_funcion),retorno_de_funcion))

retorno_de_funcion (id: 2753382130592): Me gusta Python, y a vos?

>>> print("mi_variable_inmutable después de función (id: {}):
{}\n".format(id(mi_variable_inmutable), mi_variable_inmutable))
mi_variable_inmutable después de función (id: 2753382361200): M
gusta Python
```

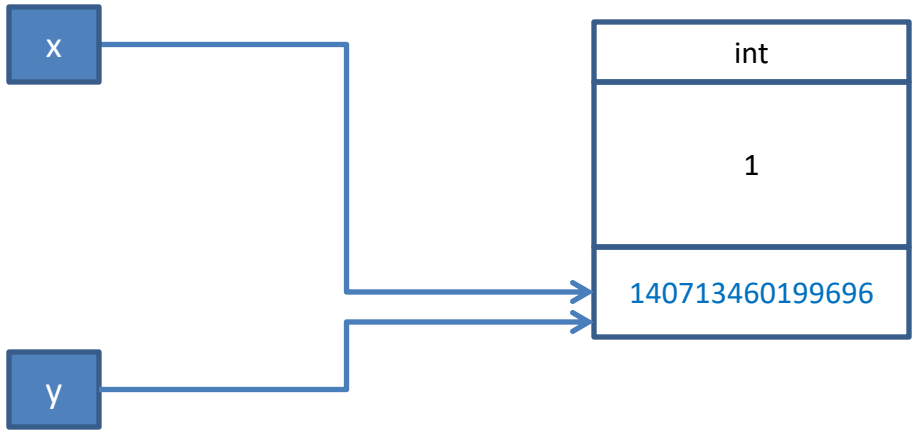


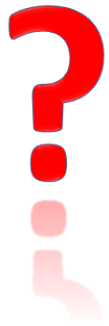
Al modificarse un objeto de un valor inmutable dentro de la función, lo que sucede realmente es que se copia y guarda la modificación en otra dirección de memoria haciendo que la variable identificador "apunte" a este nuevo objeto. El "garbage collector", se encargará de buscar objetos a los que nadie "apunte" para liberar esa memoria.

Números enteros

```
>>> x=1
>>> id(x)
140713460199696
>>> y=1
>>> id(x)
140713460199696
>>> print(x is y)
True
>>> print(x == y)
True
```

```
>>> x = 1000
>>> id(x)
1430654133456
>>> y = 1000
>>> id(y)
1430651652144
>>> print(x is y)
False
>>> print(x == y)
True
```





Números enteros

Python, para mejorar sus prestaciones, tiene *creados de antemano* los enteros entre -5 y 256, [referencia], porque estos enteros se usan mucho.

Cuando un identificador -en un programa- es asignado a uno de estos enteros, *se le hace "apuntar" al dato pre-creado*. Por eso los identificadores del programa "apuntarán" al mismo objeto.

```
>>> a = 1
>>> id(a)
140713460199696
>>> a=1000
>>> id(a)
1465919382448
>>> b=1
>>> id(b)
140713460199696
>>> id(1)
140713460199696
```

Recordar:

is para ver si dos identificadores señalan al mismo dato

== para comparar si los datos señalados son iguales

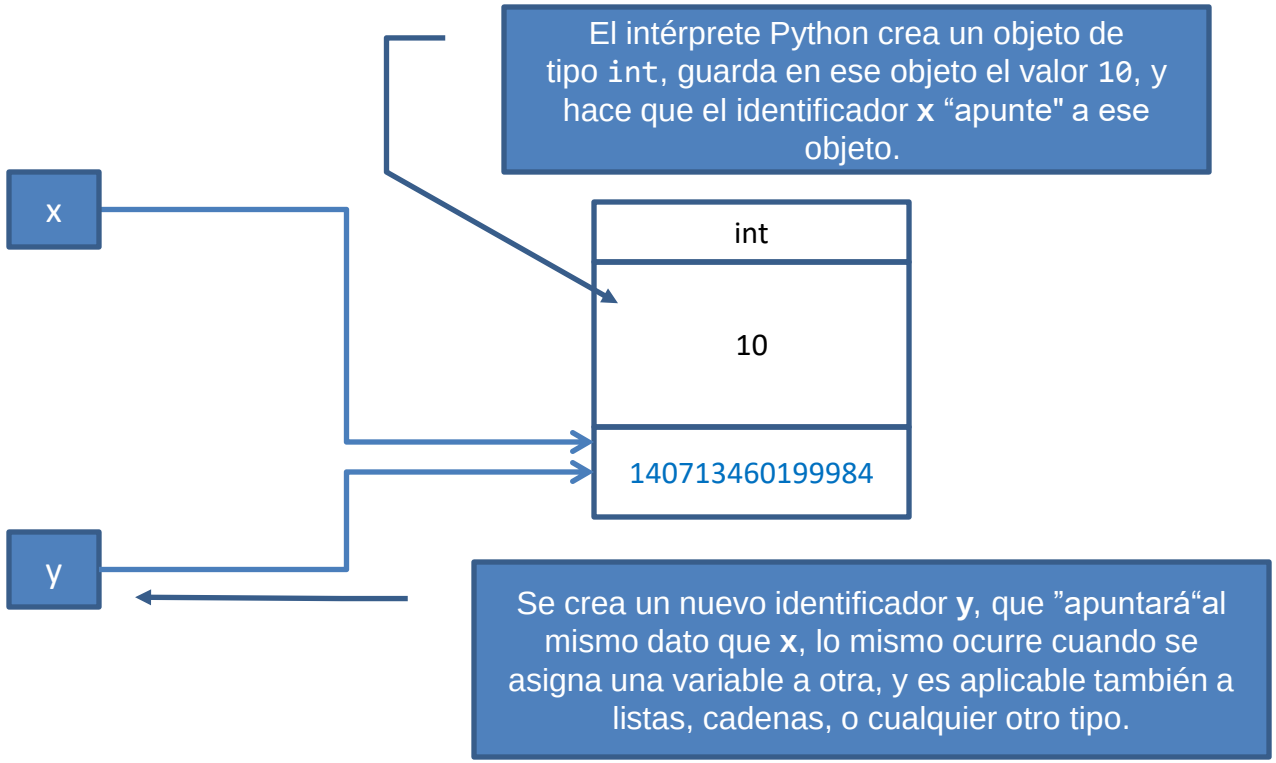
Números enteros

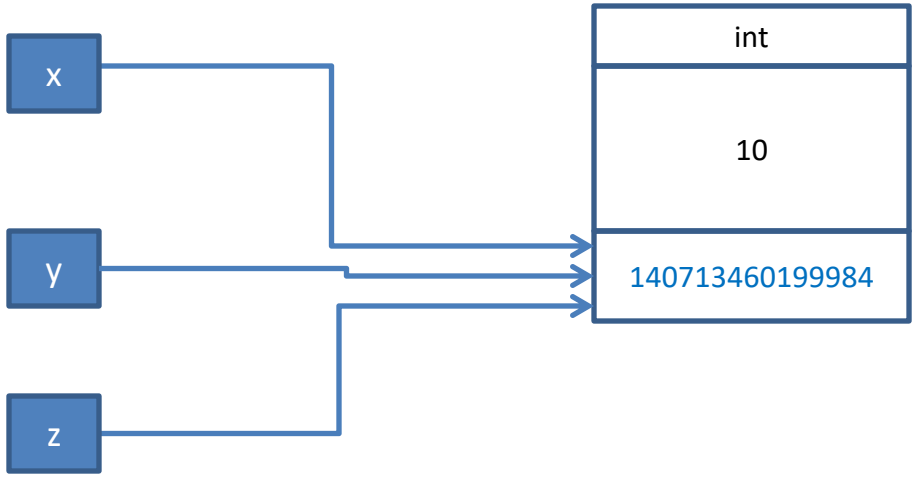
```
>>> x = 10
>>> id(x)
>>> type(x)
<class 'int'>
140713460199984
```

```
>>> y = 10
>>> id(y)
140713460199984
>>> type(y)
<class 'int'>
```

```
>>> print(x is y)
True
>>> print(x == y)
True
```

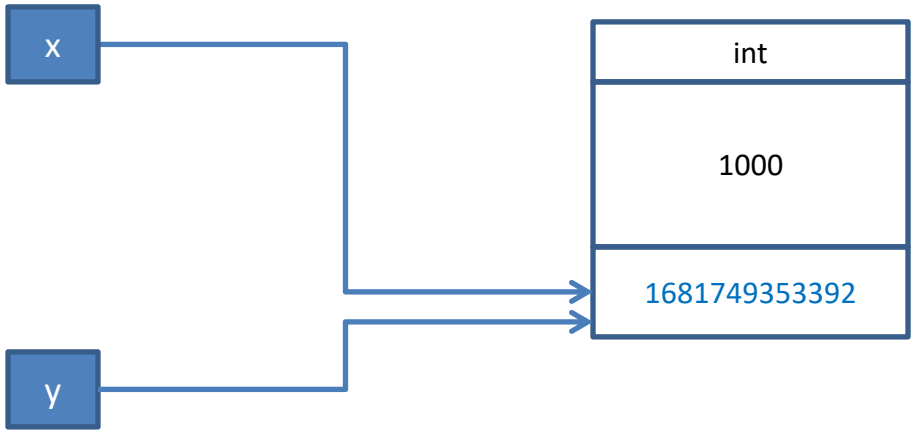
```
>>> z = x
>>> id(z)
140713460199984
>>> type(z)
<class 'int'>
>>>
```





Ejemplo 1 - Números enteros , inmutabilidad y asignaciones

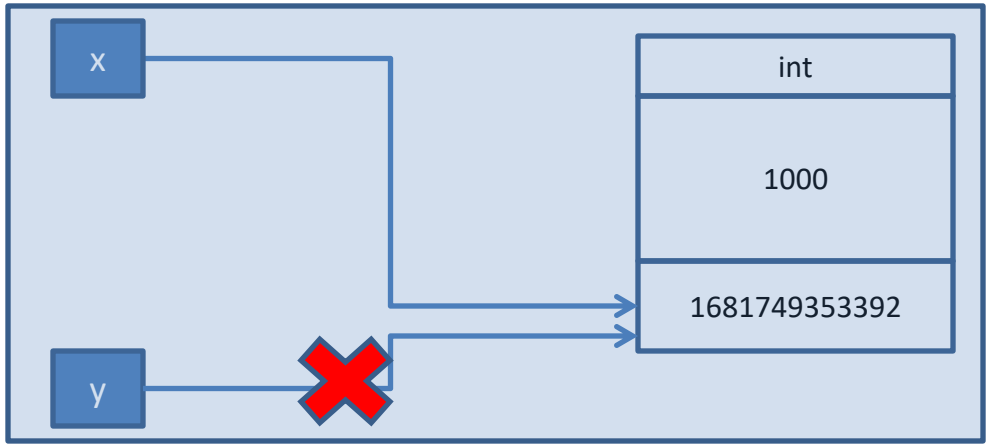
```
>>> x = 1000
>>> id(x)
1681749353392
>>> type(x)
<class 'int'>

>>> y = x
>>> id(y)
1681749353392
>>> type(y)
<class 'int'>

>>> print(x == y)
True
>>> print(x is y)
True
>>>
```



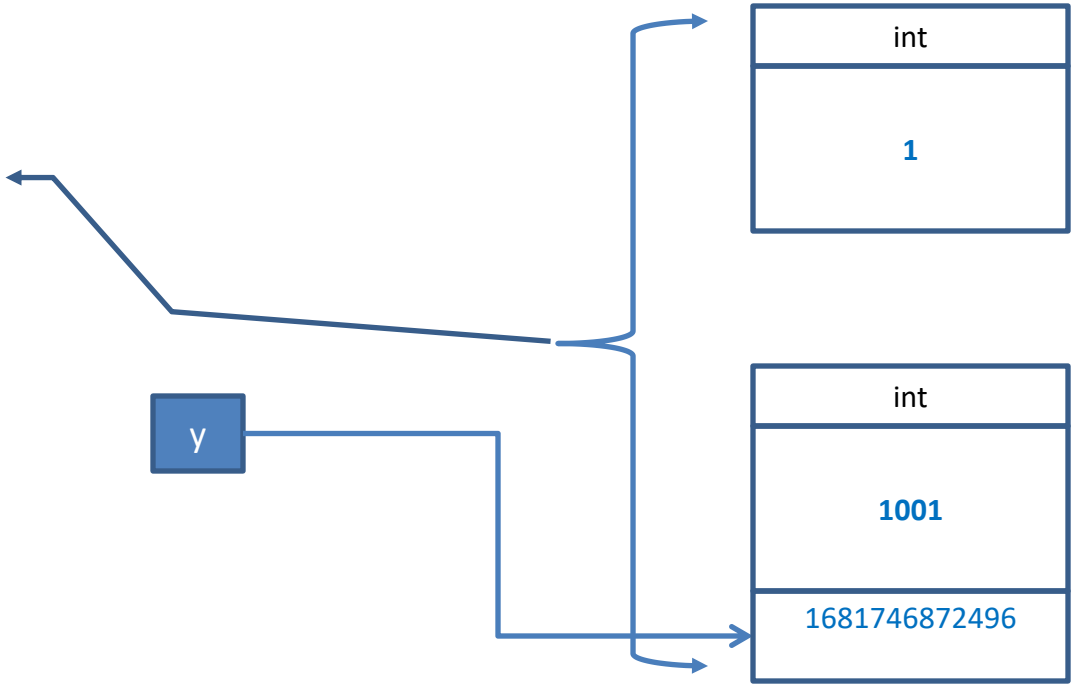
Ejemplo 1- Números enteros , inmutabilidad y asignaciones

*si ahora cambio el valor de **y** haciendo por ejemplo **y = y + 1** también cambiará el de **x**, ¿no?.....*

```
>>> y = y + 1
>>> id(y)
1681746872496
>>>
```



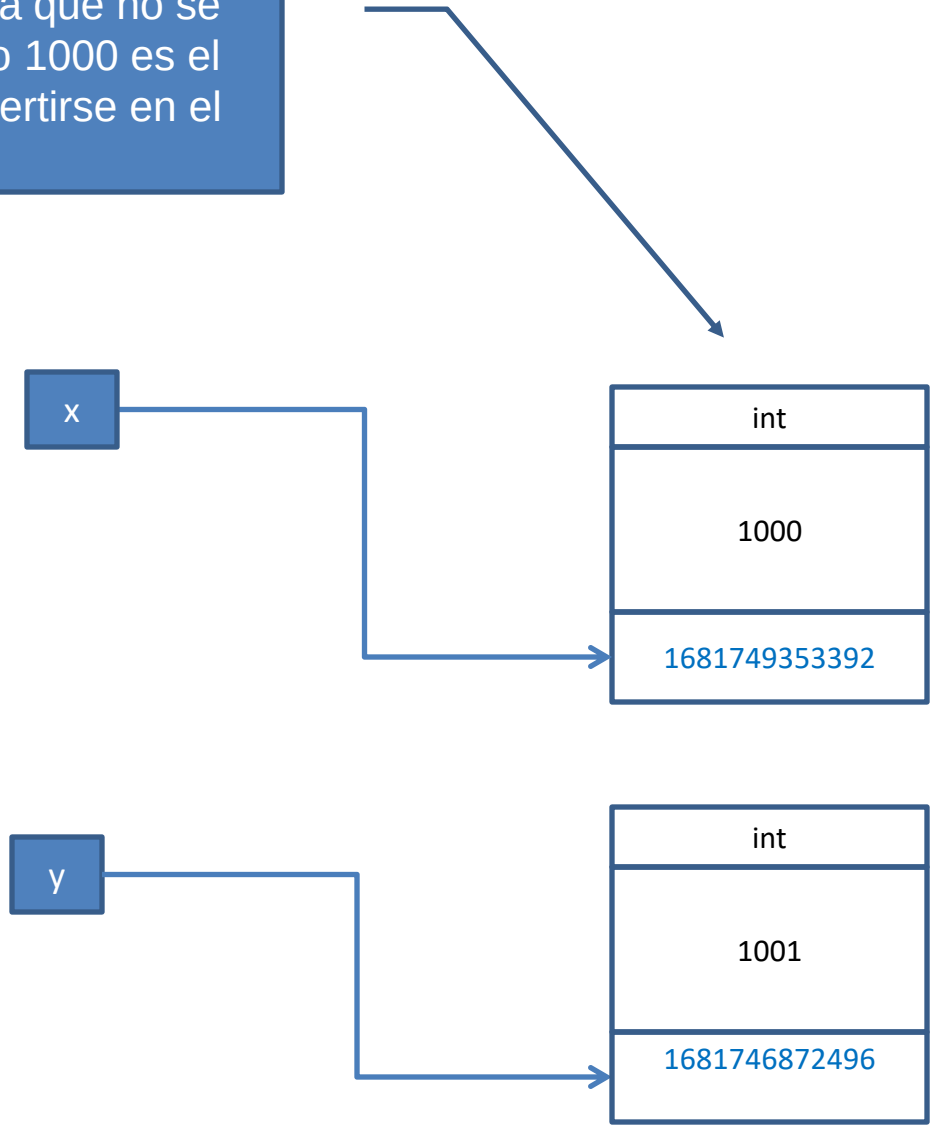
El intérprete Python evaluará el lado derecho de la asignación, para lo cual creará otro entero con valor 1 y lo sumará al entero “indicado" por **y**, o sea 1000, el resultado de esa suma es 1001, por lo tanto Python **creará un nuevo dato** para el resultado, con valor 1001, y **reasignará** el identificador **y** para que apunte a este nuevo dato. El 1000 original no se ha modificado (no podría porque es inmutable).



Ejemplo 1 - Números enteros , inmutabilidad y asignaciones - Conclusiones

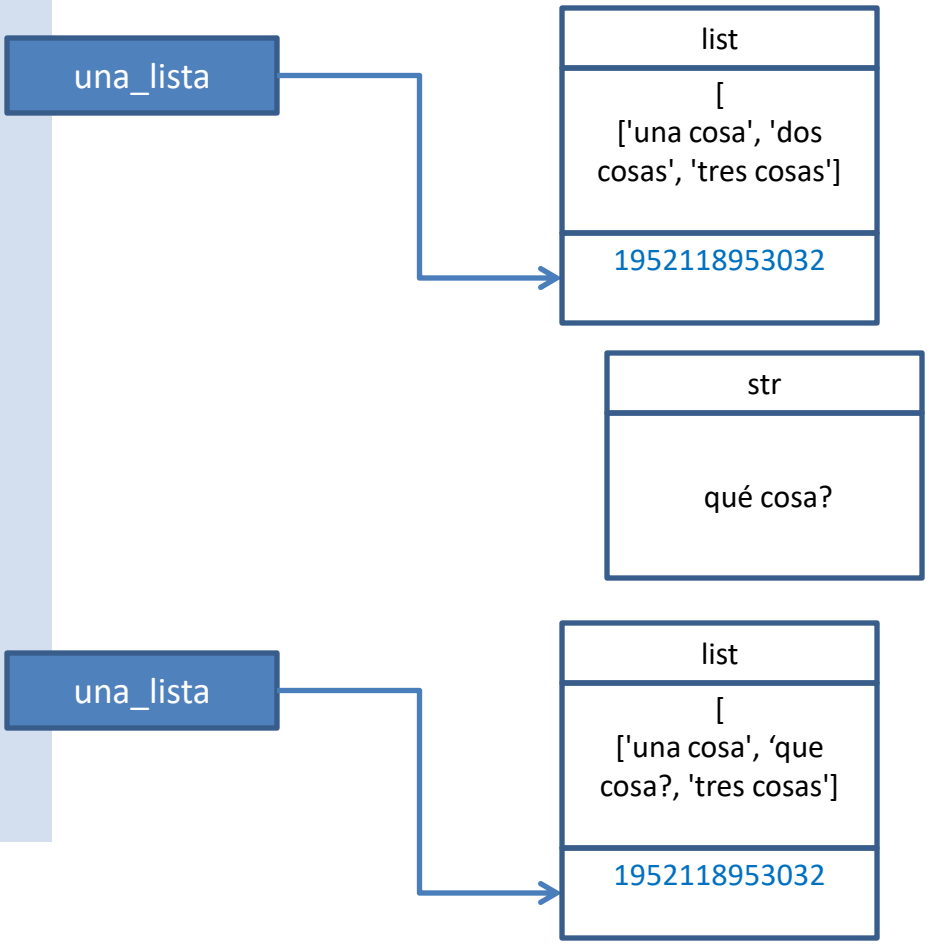
Un entero es ***inmutable***, y significa que no se puede cambiar por otro. El número 1000 es el número 1000, y nunca podrá convertirse en el número 1001

```
>>> print (x is y)
False
>>> print (x == y)
False
>>>
```



Ejemplo 1 – Listas y mutabilidad

```
>> una_lista = ["una cosa","dos cosas","tres cosas"]
>>> print(una_lista)
['una cosa', 'dos cosas', 'tres cosas']
>>> id(una_lista)
1952118953032
```



Modifico un elemento de la lista

```
>>> una_lista[1]= "que cosa?"
>>> print(una_lista)
['una cosa', 'que cosa?', 'tres cosas']
>>> id(una_lista)
1952118953032
>>> type(una_lista)
<class 'list'>
```

Ejemplo 2 – Listas y mutabilidad

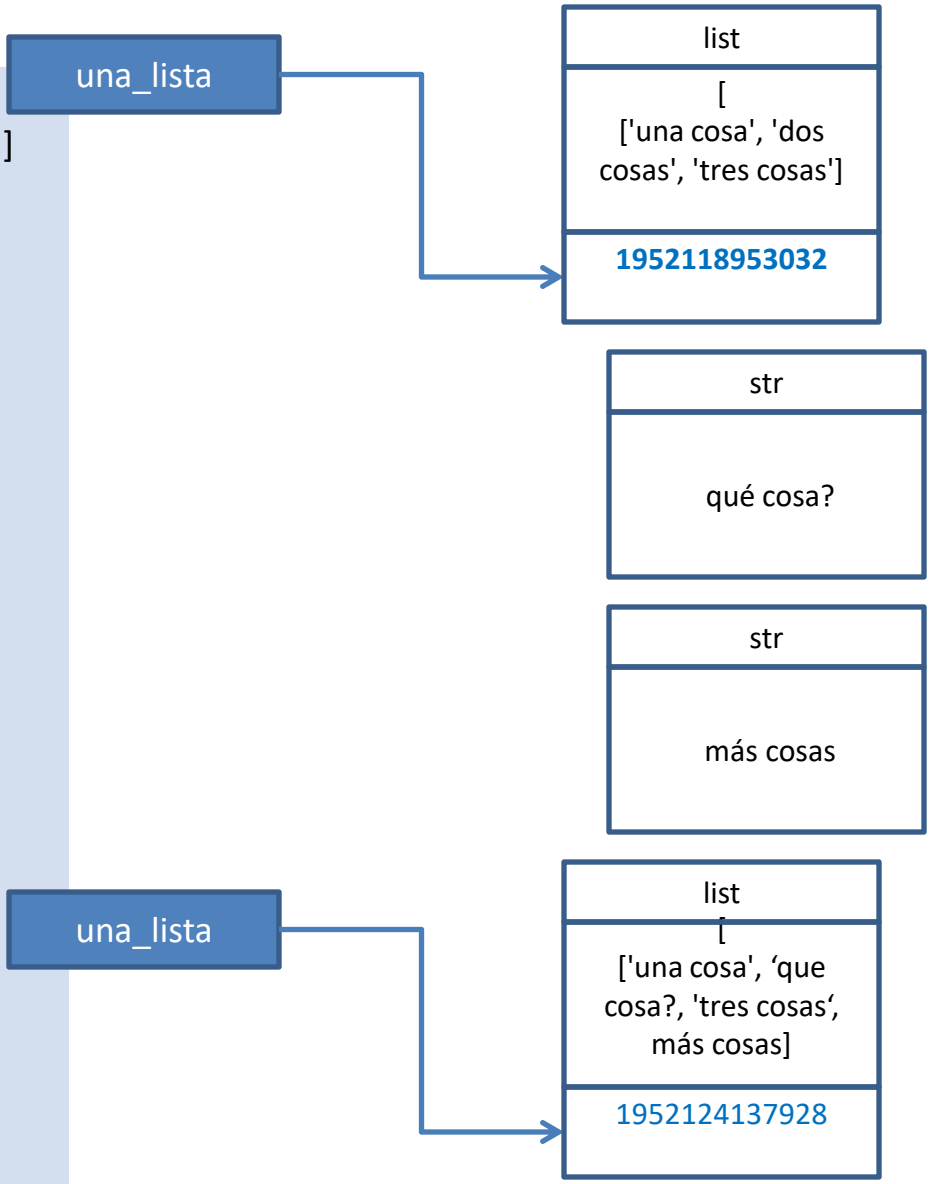
```
>>> una_lista = ["una cosa","dos cosas","tres cosas"]
>>> print(una_lista)
['una cosa', 'dos cosas', 'tres cosas']
>>> id(una_lista)
1952118953032
```

```
>>> una_lista[1]= "que cosa?"
>>> print(una_lista)
['una cosa', 'que cosa?', 'tres cosas']
>>> id(una_lista)
1952118953032
```

```
>>> una_lista = una_lista + ["más cosas"]
```

La concatenación crea una nueva lista copiando los elementos de las listas que participan como operandos

```
>>> print(una_lista)
['una cosa', 'que cosa?', 'tres cosas', 'más cosas']
>>>id(una_lista)
1952124137928
```



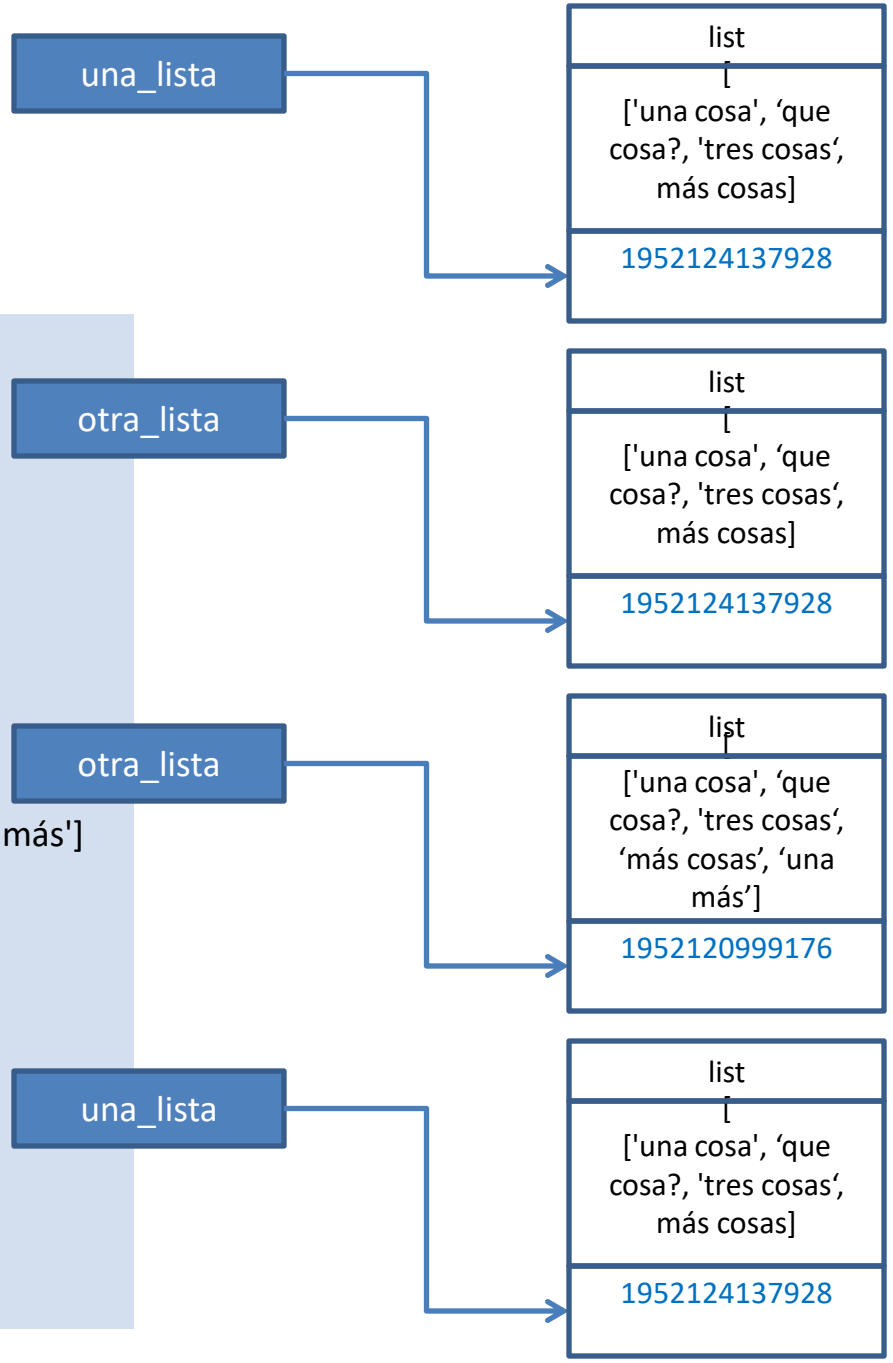
Ejemplo 3 – Listas y mutabilidad

Al asignar una lista a otra nueva tiene el mismo identificador

```
>>> otra_lista = una_lista
>>> print(otra_lista)
['una cosa', 'que cosa?', 'tres cosas', 'más cosas']
>>> id(otra_lista)
1952124137928
```

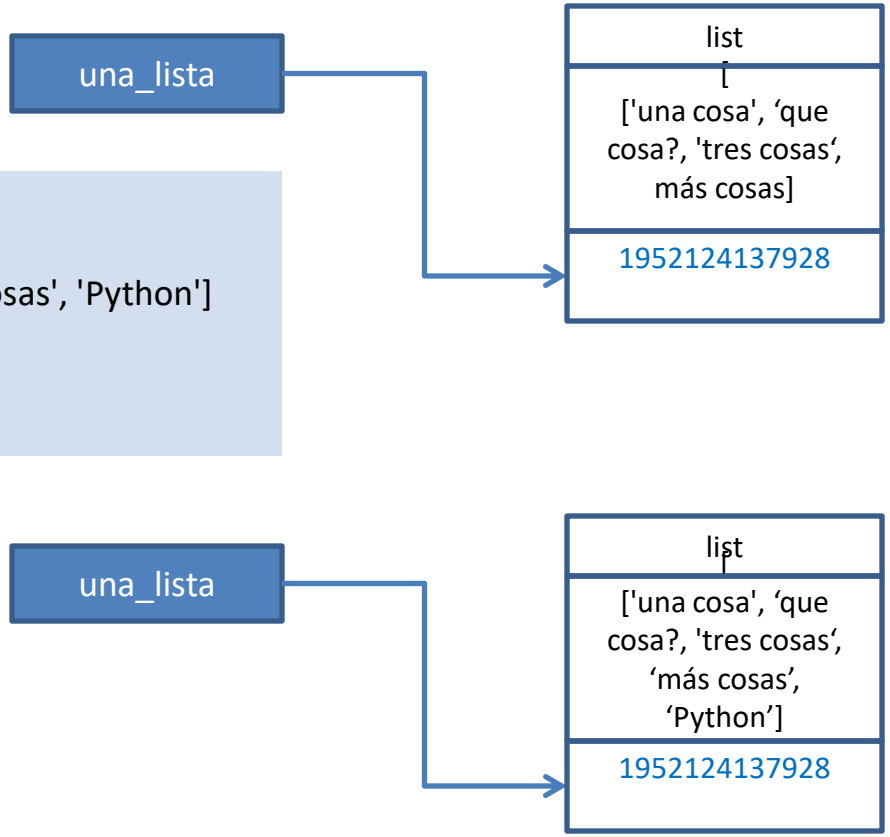
```
>>> otra_lista = otra_lista + ["una más"]
>>> print(otra_lista)
['una cosa', 'que cosa?', 'tres cosas', 'más cosas', 'una más']
>>> id(otra_lista)
1952120999176
```

```
>>> print(una_lista)
['una cosa', 'que cosa?', 'tres cosas', 'más cosas']
>>>
>>> id(una_lista)
1952124137928
```



Ejemplo 4 – Listas y mutabilidad

```
>> una_lista.append("Python")
>>> print(una_lista)
['una cosa', 'que cosa?', 'tres cosas', 'más cosas', 'Python']
>>> id(una_lista)
1952124137928
```

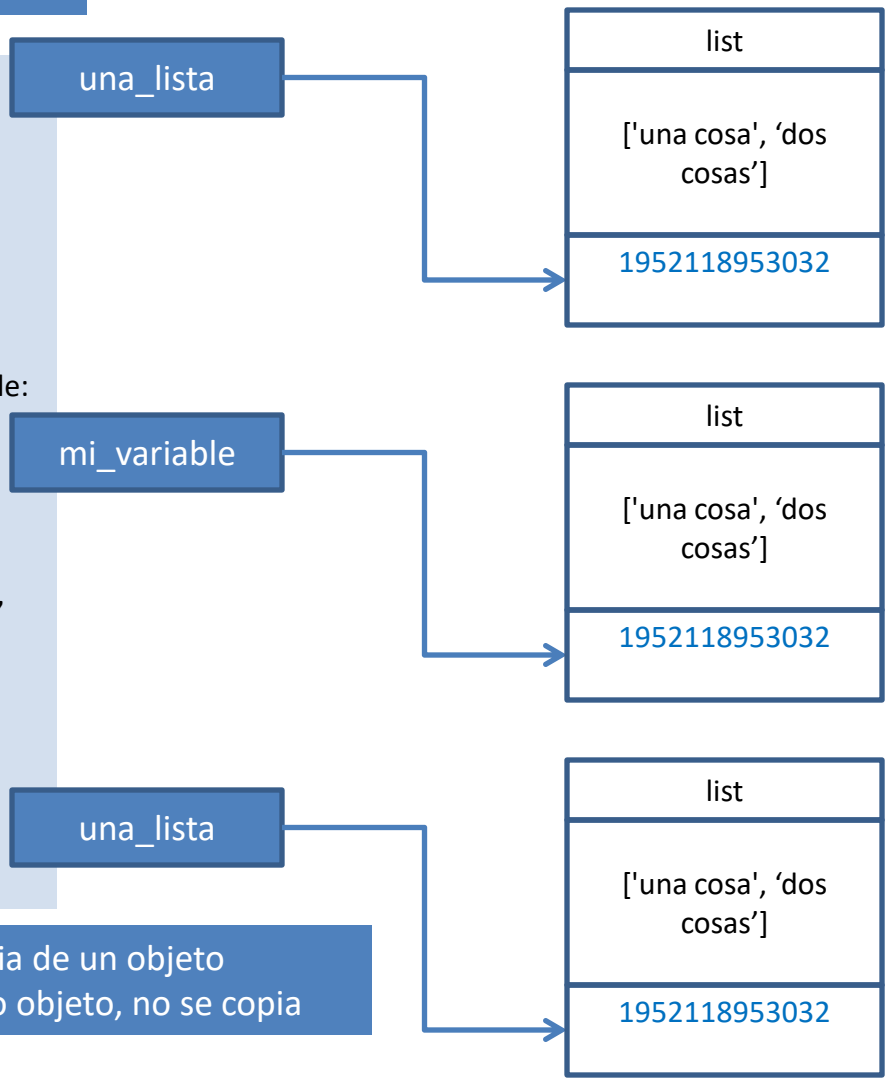


Hay una diferencia fundamental entre usar el operador de concatenación + y usar append: la concatenación crea una nueva lista copiando los elementos de las listas que participan como operandos y append modifica la lista original.

Ejemplo 5 – Listas y mutabilidad

mi_variable apunta a un objeto de tipo list que contiene un listado con dos elementos

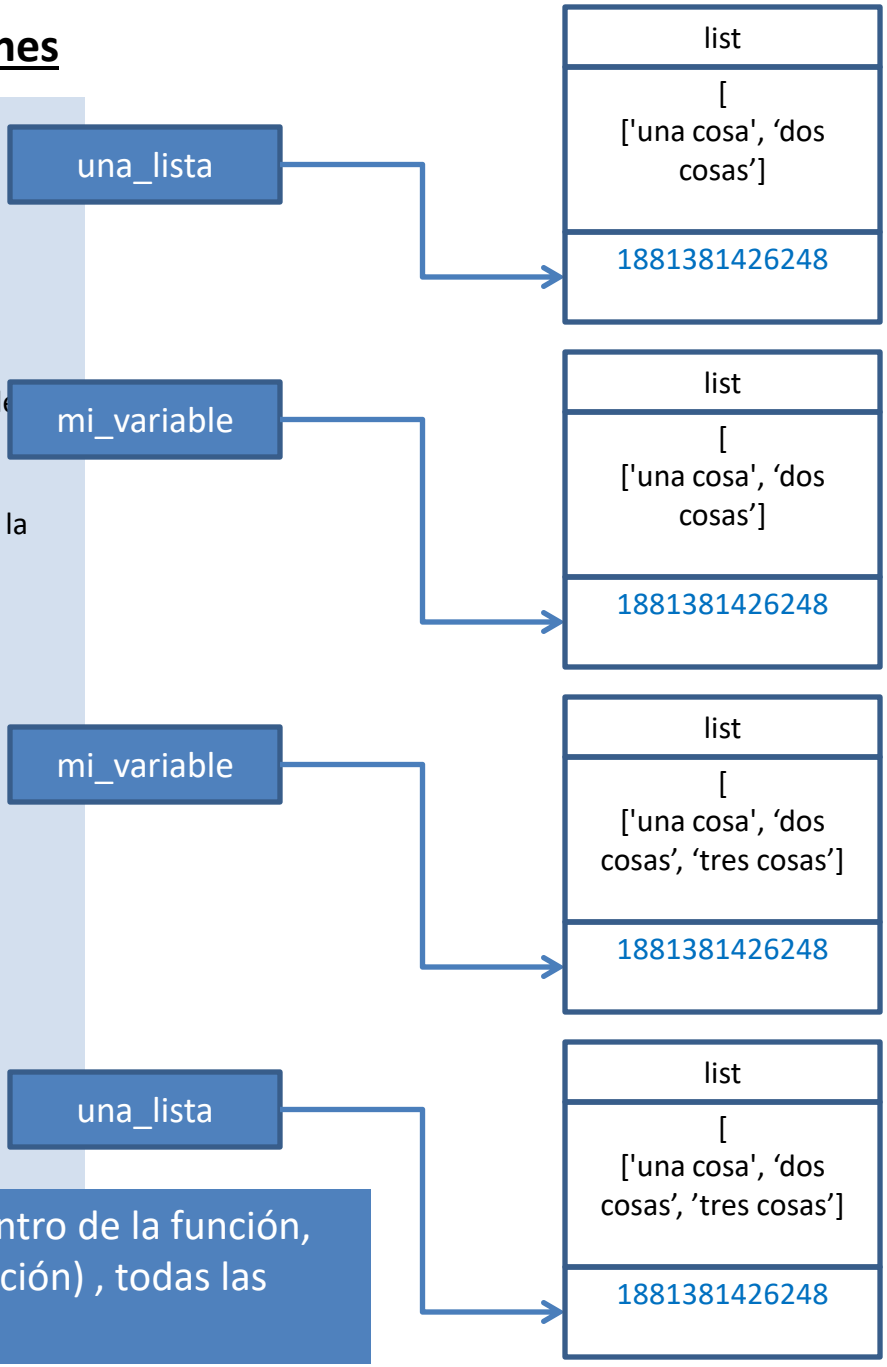
```
>>> una_lista = ["una cosa","dos cosas"]
>>> print(una_lista)
['una cosa', 'dos cosas']
>>> id(una_lista)
1952118953032

>>> def mi_funcion(mi_variable):
...     print("Id de <mi_variable> dentro de la función: {}, mi_variable: {}\n".format(id(mi_variable),mi_variable))
...
>>> mi_funcion(una_lista)

Id de <mi_variable> dentro de la función: 1952118953032, mi_variable: ['una cosa', 'dos cosas']

>>> print("Id de <una_lista> fuera de la función: {}, una_lista: {}\n".format(id(una_lista),una_lista))
Id de <una_lista> fuera de la función: 1952118953032, una_lista: ['una cosa', 'dos cosas']
```



Si se le pasa como parámetro a una función una referencia de un objeto mutable, la variable local de la función "apuntará" a dicho objeto, no se copia

Ejemplo 6 – Listas y mutabilidad - Conclusiones

```
>>> una_lista = ["una cosa","dos cosas"]
>>> print(una_lista)
['una cosa', 'dos cosas']
>>> id(una_lista)
1881381426248
>>>
>>> def mi_funcion(mi_variable):
...     print("Id de <mi_variable> dentro de la función: {}, mi_variable:
{}\n".format(id(mi_variable),mi_variable))
...     mi_variable += ["tres cosas"]
...     print("Id de <mi_variable> dentro de la función \ndespués de la
modificación: {}, mi_variable:
{}\n".format(id(mi_variable),mi_variable))
...
>>> mi_funcion(una_lista)
Id de <mi_variable> dentro de la función: 1881381426248,
mi_variable: ['una cosa', 'dos cosas']

Id de <mi_variable> dentro de la función
después de la modificación: 1881381426248, mi_variable:
['una cosa', 'dos cosas', 'tres cosas']

>>> print("Id de <una_lista> fuera de la función: {}, una_lista:
{}\n".format(id(una_lista),una_lista))
Id de <una_lista> fuera de la función: 1881381426248,
una_lista: ['una cosa', 'dos cosas', 'tres cosas']
```



Cuando se modifique el objeto mutable desde dentro de la función, se modificará para todos (dentro y fuera de la función) , todas las modificaciones se realizan sobre el mismo objeto.

FIN