

🧐 Objetos mutables e inmutables

Tipos de datos Mutables objetos que pueden modificar su valor		Tipos de datos Inmutables objetos que no pueden modificar su valor	
Lista	<pre>>>> lista = [1,2,3] >>> type(lista) <class 'list'></pre>	Cadena	<pre>>>> nombre = "Pepe" >>> type(nombre) <class 'str'></pre>
Diccionario	<pre>>>> diccionario = {"nombre": "Pepe", "edad": 35 } >>> type(diccionario) <class 'dict'></pre>	Entero	<pre>>>> numero = 20 >>> type(numero) <class 'int'></pre>
Conjunto	<pre>>>> conjunto = {"mesa", "silla", "armario"} >>> type(conjunto) <class 'set'></pre>	Flotante	<pre>>>> decimal = 30.5 >>> type(decimal) <class 'float'></pre>
Bytearray	<pre>>>> vehiculo = bytearray("camion", "cp1252") >>> type(vehiculo) <class 'bytearray'></pre>	Imaginario	<pre>>>> imaginario = 4 + 4j >>> type(imaginario) <class 'complex'></pre>
MemoryView	<pre>>>> memv = memoryview(vehiculo) >>> type(memv) <class 'memoryview'></pre>	Tupla	<pre>>>> tupla = ('a', 'b', 'c') >>> type(tupla) <class 'tuple'></pre>
		Rango	<pre>>>> rango = range(5) >>> type(rango) <class 'range'></pre>
		Frozenset	<pre>>>> frozen = frozenset({"mesa", "silla", "armario"}) >>> type(frozen) <class 'frozenset'></pre>
		Booleano	<pre>>>> booleano = True >>> type(booleano) <class 'bool'></pre>
		Byte	<pre>>>> serie = b'abc' >>> type(serie) <class 'bytes'></pre>
		None	<pre>>>> a = None >>> type(a) <class 'NoneType'></pre>

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()`: devuelve la lista de atributos y métodos del objeto que le pasemos.

```
In [1]: nombre = "Python"
dir(nombre)
```

```
Out[1]: ['__add__',
         '__class__',
         '__contains__',
         '__delattr__',
         '__dir__',
         '__doc__',
         '__eq__',
         '__format__',
         '__ge__',
         '__getattr__',
         '__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',
         'removeprefix',
         'removesuffix',
         'replace',
         'rfind',
         'rindex',
         'rjust',
         'rpartition',
         'rsplit',
         'rstrip',
         'split',
         'splitlines',
         'startswith',
         'strip',
         'swapcase',
         'title',
         'translate',
         'upper',
         'zfill']
```

help(): devuelve documentación de lo que le pasemos.

```
In [2]: 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.

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 - inmutabilidad de cadenas

```
In [3]: mi_texto = "hoy es un gran día"
mi_texto
```

```
Out[3]: 'hoy es un gran día'
```

```
In [4]: mi_texto[0]='H'
```

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

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

```
In [5]: id(mi_texto)
```

```
Out[5]: 1988730436944
```

```
In [6]: type(mi_texto)
```

```
Out[6]: str
```

```
>>> 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 - inmutabilidad de cadenas

```
In [7]: mi_texto = "Comienzo de mi texto"
mi_texto
```

```
Out[7]: 'Comienzo de mi texto'
```

```
In [8]: id(mi_texto)
```

```
Out[8]: 1988759403776
```

```
In [9]: mi_texto = mi_texto + " fin de mi texto"
mi_texto
```

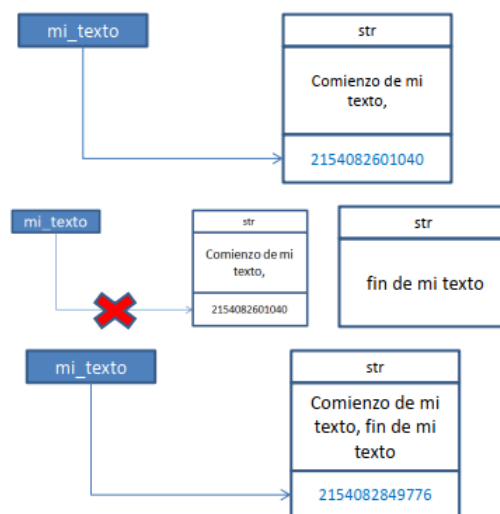
```
Out[9]: 'Comienzo de mi texto fin de mi texto'
```

```
In [10]: id(mi_texto)
```

```
Out[10]: 1988759322224
```

```
>>> 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 **immutable** 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 (immutable) aunque nosotros creamos que se modifica (falsa creencia de mutabilidad)

Ejemplo 3 - inmutabilidad de cadenas

```
In [11]: un_texto = "Me gusta Python"
id(un_texto)
```

```
Out[11]: 1988759023792
```

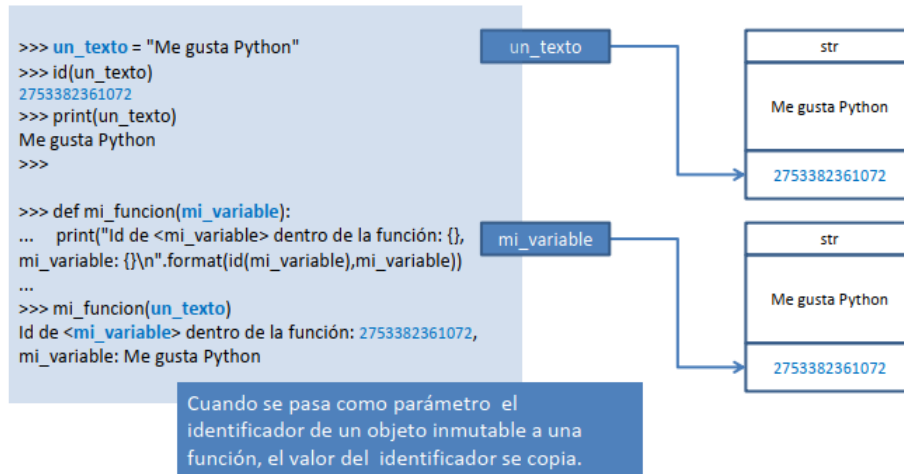
In [12]: `un_texto`

Out[12]: 'Me gusta Python'

In [13]: `def mi_funcion(mi_variable):
 print("Id de <mi_variable> dentro de la función: {}, mi_variable: {}".format(id(mi_variable),mi_variable))

mi_funcion(un_texto)`

Id de <mi_variable> dentro de la función: 1988759023792, mi_variable: Me gusta Python



Ejemplo 4 - inmutabilidad de cadenas - Conclusiones

In [14]: `mi_variable_inmutable = "Me gusta Python"
print("mi_variable_inmutable (id: {}): {}".format(id(mi_variable_inmutable), mi_variable_inmutable))`

mi_variable_inmutable (id: 1988759039920): Me gusta Python

In [15]: `def mi_funcion(var_de_func):
 print("var_de_func antes de retornar (id: {}): {}".format(id(var_de_func), var_de_func))
 var_de_func += ", y a vos?"
 print("var_de_func (id: {}): {}".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: 1988759039920): Me gusta Python

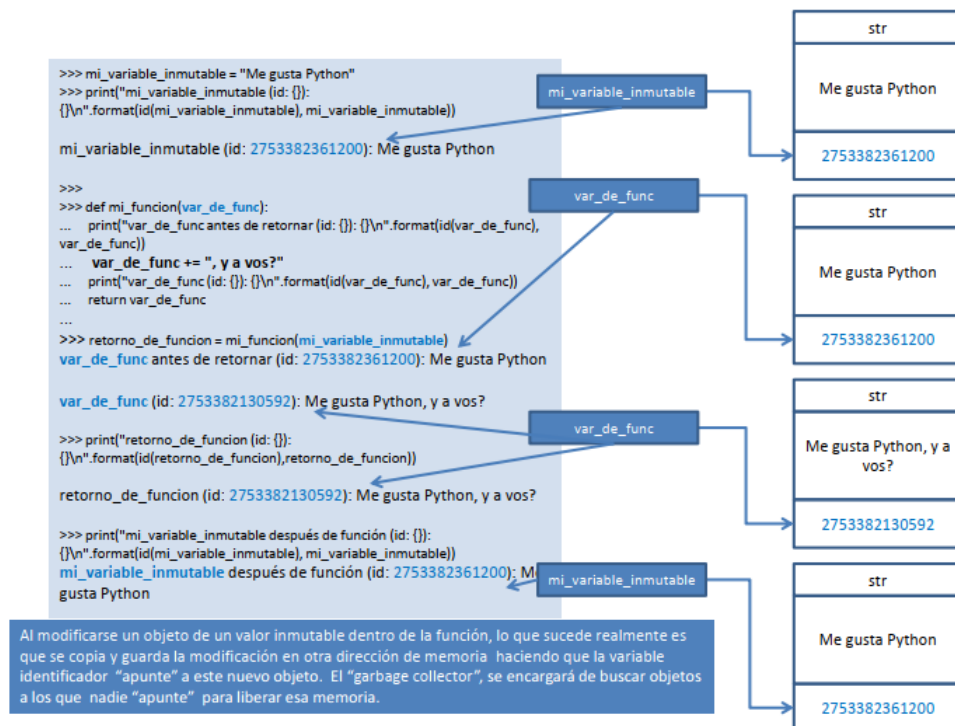
var_de_func (id: 1988759485056): Me gusta Python, y a vos?

In [16]: `print("retorno_de_funcion (id: {}): {}".format(id(retorno_de_funcion), retorno_de_funcion))`

retorno_de_funcion (id: 1988759485056): Me gusta Python, y a vos?

In [17]: `print("mi_variable_inmutable después de función \n
 (id: {}): {}".format(id(mi_variable_inmutable), mi_variable_inmutable))`

mi_variable_inmutable después de función (id: 1988759039920): Me gusta Python



Números enteros

In [18]: `x = 1`
`id(x)`

Out[18]: 1988642761008

In [19]: `y = 1`
`id(y)`

Out[19]: 1988642761008

In [20]: `x is y`

Out[20]: True

In [21]: `x == y`

Out[21]: True

In [22]: `x = 1000`
`id(x)`

Out[22]: 1988758295856

In [23]: `y = 1000`
`id(y)`

Out[23]: 1988758297456

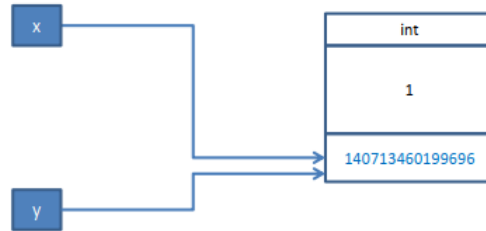
In [24]: `x is y`

Out[24]: False

In [25]: `x == y`

Out[25]: True

```
>>> x=1
>>> id(x)
140713460199696
>>> y=1
>>> id(y)
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
```



```
In [26]: a = 1
         id(a)

Out[26]: 1988642761008
```

```
In [27]: a = 1000
         id(a)

Out[27]: 1988758295280
```

```
In [28]: b = 1
         id(b)

Out[28]: 1988642761008
```

```
In [29]: id(1)

Out[29]: 1988642761008
```



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

```
In [30]: x = 10
         id(x), type(x)

Out[30]: (1988642761296, int)
```

```
In [31]: y = 10
         id(y), type(y)

Out[31]: (1988642761296, int)
```

```
In [32]: x is y
```

```
Out[32]: True
```

```
In [33]: x == y
```

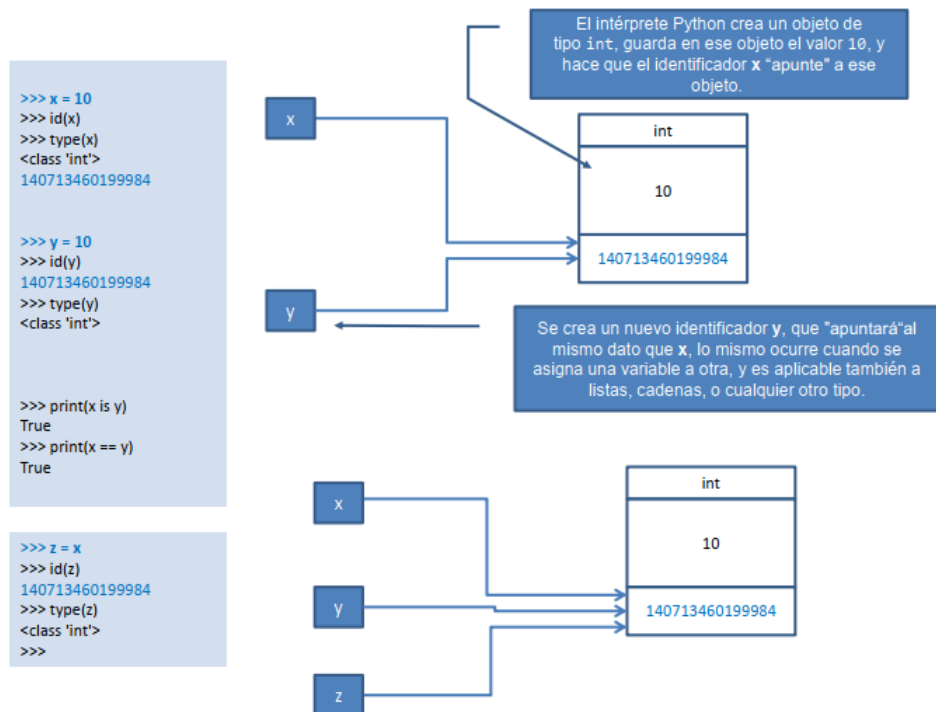
```
Out[33]: True
```

```
In [34]: z = x  
         id(z)
```

```
Out[34]: 1988642761296
```

```
In [35]: type(z)
```

```
Out[35]: int
```



Ejemplo 1 - Números enteros , inmutabilidad y asignaciones

```
In [36]: x = 1000  
         id(x), type(x)
```

```
Out[36]: (1988759576784, int)
```

```
In [37]: y = x  
         id(y), type(y)
```

```
Out[37]: (1988759576784, int)
```

```
In [38]: x == y
```

```
Out[38]: True
```

```
In [39]: x is y
```

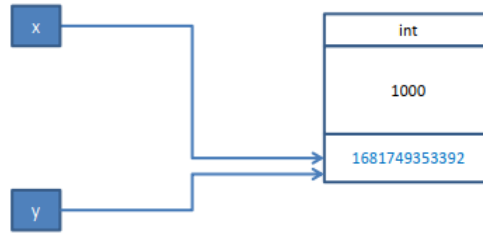
```
Out[39]: True
```



```
>>> 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
>>>
```

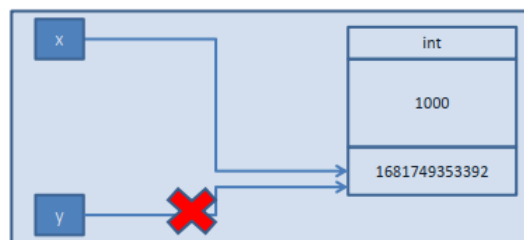


```
In [40]: ❏ y = y + 1
         id(y)
```

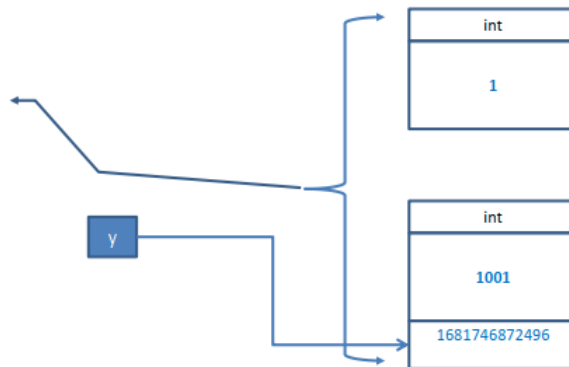
```
Out[40]: 1988759577264
```

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 **reassignará** 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

```
In [41]: ❏ x is y
```

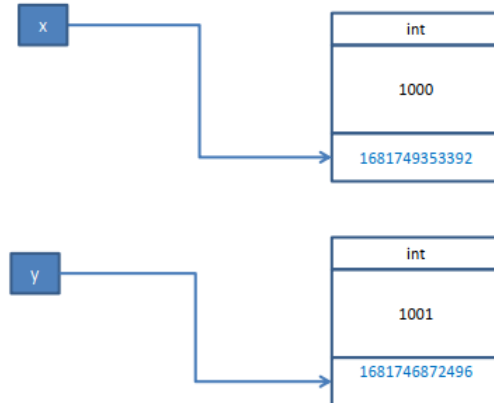
```
Out[41]: False
```

```
In [42]: ❏ x == y
```

```
Out[42]: False
```

Un entero es *immutable*, 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

```
In [43]: una_lista = ["una cosa", "dos cosas", "tres cosas"]
una_lista
```

```
Out[43]: ['una cosa', 'dos cosas', 'tres cosas']
```

```
In [44]: id(una_lista)
```

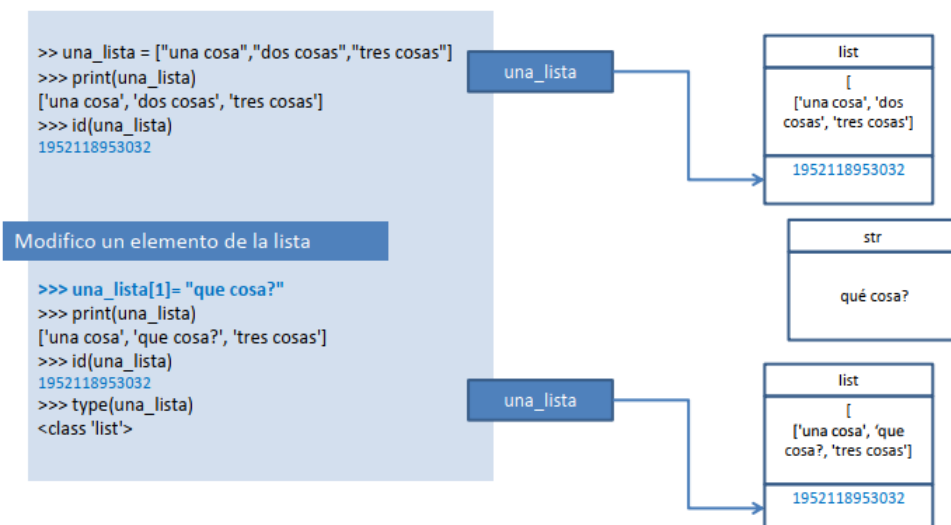
```
Out[44]: 1988759113856
```

```
In [45]: una_lista[1] = "que cosa?"
una_lista
```

```
Out[45]: ['una cosa', 'que cosa?', 'tres cosas']
```

```
In [46]: id(una_lista), type(una_lista)
```

```
Out[46]: (1988759113856, list)
```



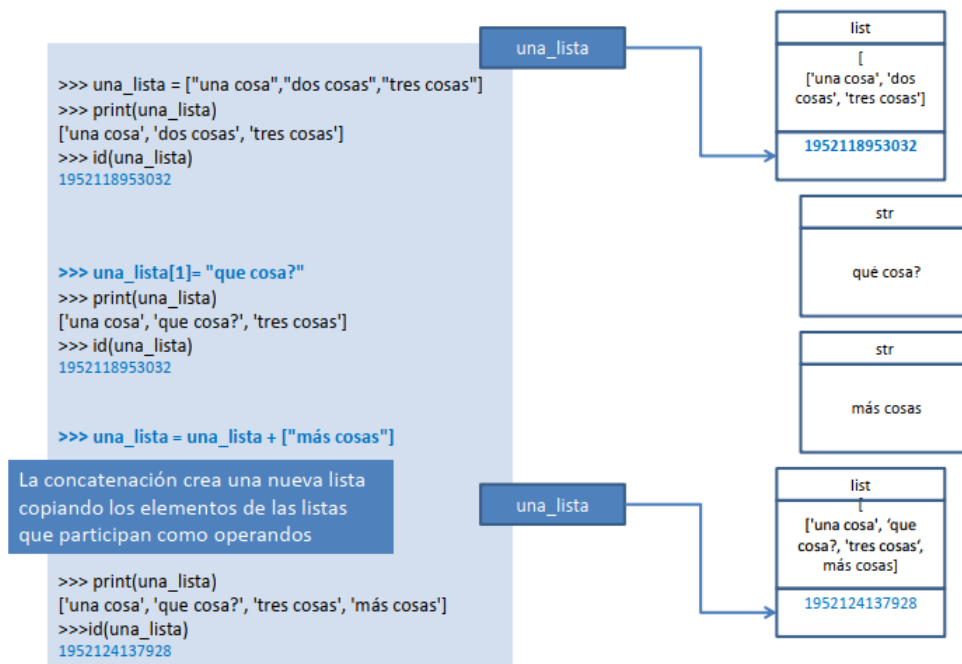
Ejemplo 2 – Listas y mutabilidad

```
In [47]: una_lista = una_lista + ["más cosas"]
una_lista
```

```
Out[47]: ['una cosa', 'que cosa?', 'tres cosas', 'más cosas']
```

```
In [48]: id(un_a_lista)
```

```
Out[48]: 1988759063616
```



Ejemplo 3 – Listas y mutabilidad

```
In [49]: otra_lista = una_lista
         otra_lista
```

```
Out[49]: ['una cosa', 'que cosa?', 'tres cosas', 'más cosas']
```

```
In [50]: id(otra_lista)
```

```
Out[50]: 1988759063616
```

```
In [51]: otra_lista = otra_lista + ["una más"]
         otra_lista
```

```
Out[51]: ['una cosa', 'que cosa?', 'tres cosas', 'más cosas', 'una más']
```

```
In [52]: id(otra_lista)
```

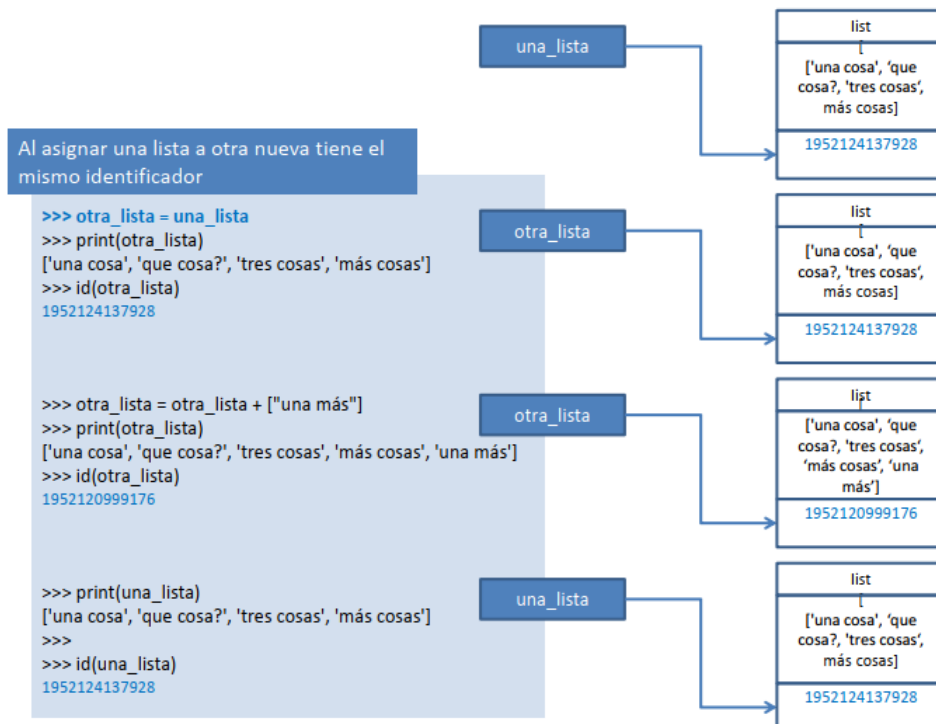
```
Out[52]: 1988759135360
```

```
In [53]: una_lista
```

```
Out[53]: ['una cosa', 'que cosa?', 'tres cosas', 'más cosas']
```

```
In [54]: id(una_lista)
```

```
Out[54]: 1988759063616
```



Ejemplo 4 – Listas y mutabilidad

```

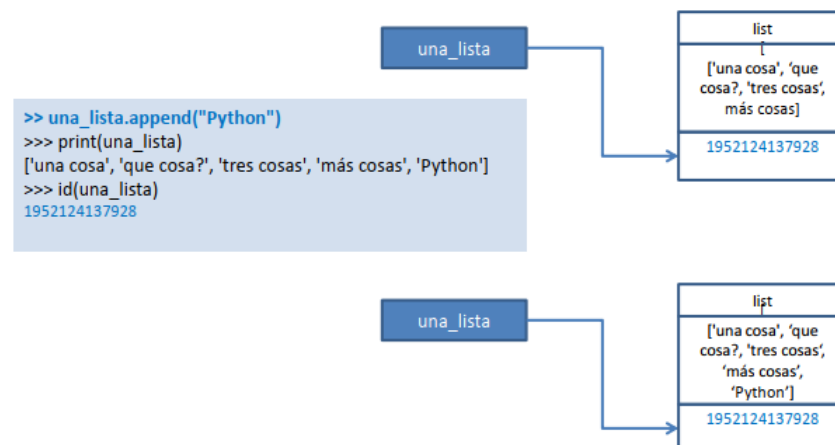
In [55]: una_lista.append("Python")
una_lista

Out[55]: ['una cosa', 'que cosa?', 'tres cosas', 'más cosas', 'Python']

In [56]: id(una_lista)

Out[56]: 1988759063616

```



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

```

In [57]: una_lista = ["una cosa", "dos cosas"]
una_lista

Out[57]: ['una cosa', 'dos cosas']

In [58]: id(una_lista)

Out[58]: 1988759063296

```

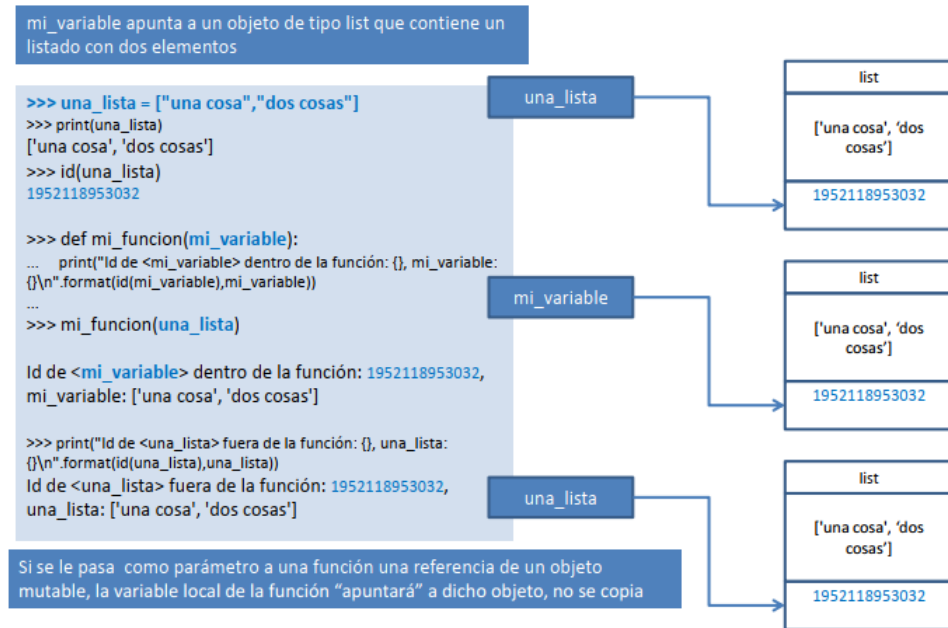
```
In [59]: def mi_funcion(mi_variable):
          print("Id de <mi_variable> dentro de la función: {}, mi_variable: {}".format(id(mi_variable),mi_variable))

          mi_funcion(una_lista)
```

Id de <mi_variable> dentro de la función: 1988759063296, mi_variable: ['una cosa', 'dos cosas']

```
In [60]: print("Id de <una_lista> fuera de la función: {}, una_lista: {}".format(id(una_lista),una_lista))
```

Id de <una_lista> fuera de la función: 1988759063296, una_lista: ['una cosa', 'dos cosas']



Ejemplo 6 – Listas y mutabilidad - Conclusiones

```
In [61]: def mi_funcion(mi_variable):
          print("Id de <mi_variable> dentro de la función: {}, mi_variable: \
                {}".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: \
                {}".format(id(mi_variable),mi_variable))

          mi_funcion(una_lista)
```

Id de <mi_variable> dentro de la función: 1988759063296, mi_variable: ['una cosa', 'dos cosas']

Id de <mi_variable> dentro de la función
después de la modificación: 1988759063296, mi_variable: ['una cosa', 'dos cosas', 'tres cosas']

```
In [62]: print("Id de <una_lista> fuera de la función: {}, una_lista: {}".format(id(una_lista),una_lista))
```

Id de <una_lista> fuera de la función: 1988759063296, una_lista: ['una cosa', 'dos cosas', 'tres cosas']

```
In [63]: una_lista
```

Out[63]: ['una cosa', 'dos cosas', 'tres cosas']

```

>>> 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: {}".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: {}".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: {}".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.

una_lista

mi_variable

mi_variable

una_lista

list
[['una cosa', 'dos cosas']
1881381426248

list
[['una cosa', 'dos cosas']
1881381426248

list
[['una cosa', 'dos cosas', 'tres cosas']
1881381426248

list
[['una cosa', 'dos cosas', 'tres cosas']
1881381426248