[Diferencia entre atributos de instancia y atributos de clase](https://es.stackoverflow.com/questions/132561/diferencia-entre-atributos-de-instancia-y-atributos-de-clase)

[Formular una pregunta](https://es.stackoverflow.com/questions/ask)

**¿Cuál es la diferencia entre un atributo/propiedad de instancia y un atributo/propiedad de clase? ¿Cuándo uso uno y cuándo otro en Python?**

Es decir, si tenemos:

class Foo:

a = 5

def print\_a(self):

print(self.a)

y:

class Foo:

def \_\_init\_\_(self):

self.a = 5

def print\_a(self):

print(self.a)

¿Qué diferencias hay con respecto al atributo a?

**Respuesta:**

↓↓↓

La diferencia fundamental es que **los atributos de clase son compartidos por todas las instancias de esa clase, mientras que los atributos de instancia son particulares para cada objeto creado con esa clase.** Por tanto, las variables de instancia son para datos únicos y propios para cada objeto y las variables de clase son para atributos que deban ser compartidos por todas las instancias de esa clase.

* **Los atributos de instancia** son creados usando la sintaxis instance.attr o self.attr (self es realmente una convención, simplemente es un identificador que hace referencia a la propia instancia). Como se ha dicho son locales para esa instancia y por tanto solo accesibles desde una de ellas.

Por norma general son creados dentro de un método. Se pueden crear atributos de instancia en cualquier método de instancia, pero generalmente es buena práctica hacerlo en el inicializador de la clase (\_\_init\_\_) o el constructor (\_\_new\_\_) o como mucho en un método que sea directamente llamado desde uno de ellos. Si no se hace así (aunque solo sea inicializarlos a un valor nulo) el código pierde legibilidad y lo que es más importante, el atributo no existe hasta que el método que lo crea sea llamado, lo que crea incertidumbre en cuanto a si una instancia tiene o no cierto atributo en un momento dado.

En cuanto a su utilidad, genéricamente podemos decir que permiten definir un determinado "estado" a ese objeto particular y que lo diferencia de otros objetos de su clase. Como puede ser el atributo "matrícula" en una clase "Automóvil".

Son almacenados en un diccionario (diccionario de instancia u objeto) al cual puedes acceder mediante NombreInstancia.\_\_dict\_\_.

* **Los atributos de clase** se definen fuera de cualquier método de la clase, normalmente justo debajo de la cadena de documentación. Se pueden referenciar directamente desde la clase y también se puede hacer referencia a ellas desde cualquier instancia de la clase. Usos comunes son crear una constante enumerada, como variable de control en un Singleton, para establecer valores predeterminados para variables de instancia, definir constantes, etc.

Son almacenados en un diccionario distinto (diccionario de clase) e independiente al usado para los atributos de instancia, el cual puedes consultar mediante NombreClase.\_\_dict\_\_ o desde una instancia con NombreInstancia.\_\_class\_\_.\_\_dict\_\_. Las *properties* y los descriptores en general también están a nivel de clase, no de instancia.

**Nota:** En Python 3 el diccionario de clase es de tipo mappingproxy, sin extenderse demasiado una consecuencia directa es hacer que el diccionario sea de solo lectura y en principio solo \_\_setattr\_\_ podrá modificarlo en su caso. Esto se hace por eficiencia y estabilidad, entre otras cosas se asegura que las claves solo puedan ser cadenas.

Es importante conocer el orden en el que se resuelve el acceso a un atributo, Python sigue el siguiente orden al buscar un atributo desde un objeto:

1. Atributos especiales (\_\_dict\_\_, \_\_slots\_\_, \_\_weakref\_\_, etc.).
2. Descriptores de datos (implementan \_\_set\_\_) en diccionario de clase y posibles clases padre.
3. Atributos de instancia.
4. Descriptores de no-datos (implementan \_\_get\_\_ pero no \_\_set\_\_) en diccionario de clase y padres.
5. Finalmente intenta delegar en el método \_\_getattr\_\_.

si obviamos los descriptores de datos (y por tanto properties con *setter*) Python busca primero el atributo en los atributos de instancia, y de no encontrarlo busca en los de clase. Esto implica que si creas un atributo de instancia con el mismo nombre que un atributo de clase ya presente, el atributo de clase queda "oculto" si se intenta acceder a el vía self.atributo o instancia.atributo.

En el caso de asignación el orden es:

1. Descriptores de datos en la clase y padres.
2. Método \_\_setattr\_\_
3. Atributos de instancia, **si no existe el atributo y lo anterior falla se agrega el nuevo atributo como atributo de instancia automáticamente**.

Lo anterior contesta básicamente a la pregunta, lo siguiente que es algo más extenso (principalmente por los ejemplos) e intenta explicar un error común causado por la diferencia que hay entre asignar a un atributo de clase desde su clase o desde una instancia y directamente relacionado con el orden de resolución explicado antes.

Si alguien no quiere leer más ツ, la moraleja es:

Si se va a asignar un nuevo valor a un atributo de clase (que no sea un descriptor de datos) hacerlo siempre referenciando desde la clase, no desde la instancia. Es decir, vía NombreClase.atributo o usando un método de clase (@classmethod) no mediante self.atributo o nombre\_instancia.atributo, ya que esto no modifica el atributo de clase sino que crea un nuevo atributo de instancia con el mismo nombre.

Cómo se comenta arriba podemos acceder a los atributos de clase de ambas formas:

>>> class Foo:

n = 7

>>> inst = Foo()

>>> inst.n # Acceso mediante nombre de una instancia

7

>>> Foo.n # Acceso mediante nombre de la clase

7

>>> inst.\_\_class\_\_.n # Acceso mediante el atributo \_\_class\_\_ de la instancia

7

Cuando en vez de solo acceder al valor queremos realizar una asignación, las cosas cambian:

* Si asignamos a través de la clase el comportamiento es el esperado, el valor cambia para todas las instancias de la clase:
* >>> class Foo:
* cls\_n = 13
* @classmethod
* def set\_cls\_n(cls, v):
* cls.cls\_n = v
* >>> inst1 = Foo()
* >>> inst2 = Foo()
* >>> inst1.cls\_n
* 13
* >>> inst2.cls\_n
* 13
* >>> Foo.cls\_n = 7
* >>> inst1.cls\_n
* 7
* >>> inst2.cls\_n
* 7
* >>> inst1.set\_cls\_n(23)
* >>> inst1.cls\_n
* 23
* >>> inst2.cls\_n
* 23

Se puede observar como modificando el valor de n tanto usando Foo.n como mediante un método de clase (dónde cls hace referencia a Foo igual que \_\_class\_\_) el cambio se refleja en todas las instancias de la clase.

* Si se asigna un nuevo valor a la variable de clase usando una instancia para hacer referencia a ella nos llevamos una sorpresa:
* >>> class Foo:
* cls\_n = 13
* >>> inst1 = Foo()
* >>> inst2 = Foo()
* >>> inst1.cls\_n
* 13
* >>> inst2.cls\_n
* 13
* >>> inst1.cls\_n = 7
* >>> inst1.cls\_n
* 7
* >>> inst2.cls\_n
* 13 # ????????????????????

**Al intentar asignar un nuevo objeto al atributo de clase usando instancia.atributo o desde un método de instancia vía self.atributo (siempre que no estemos tratando con un descriptor de datos que como vimos tiene preferencia y se comportaría como esperamos) no se modifica el atributo de clase, se crea un atributo de instancia con el mismo nombre y se le asigna el nuevo valor**.

Recordemos que los atributos (descriptores de datos aparte de nuevo) primero se buscan en \_\_dict\_\_ y después en \_\_class\_\_.\_\_dict\_\_ si no se encuentran, esto hace que el atributo de clase quede oculto detrás del de instancia, podemos verlo mediante el contenido de \_\_dict\_\_ y \_\_class\_\_.\_\_dict\_\_:

>>> inst1 = Foo()

>>> inst1.\_\_dict\_\_

{}

>>> inst1.\_\_class\_\_.\_\_dict\_\_

mappingproxy({'\_\_module\_\_': '\_\_main\_\_',

'cls\_n': 13,

'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Foo' objects>,

'\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Foo' objects>,

'\_\_doc\_\_': None

})

>>> inst1.cls\_n = 7

>>> inst1.\_\_dict\_\_

{'cls\_n': 7} # <<<< Nuestro nuevo atributo de instancia

>>> inst1.\_\_class\_\_.\_\_dict\_\_

mappingproxy({'\_\_module\_\_': '\_\_main\_\_',

'cls\_n': 13,

'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Foo' objects>,

'\_\_weakref\_\_': <attribute

'\_\_weakref\_\_' of 'Foo' objects>,

'\_\_doc\_\_': None

})

Por último, recordar que en Python hay objetos mutables como (list, dict, set) e inmutables (int, float, bool, tuple, frozenset, str). Los objetos inmutables no pueden ser modificados una vez creados, cuando concatenamos una cadena se crea un nuevo objeto, cosa que no pasa con una lista:

>>> a = "Hola"

>>> b = "StackOverflow"

>>> id(a)

2105306070744

>>> id(b)

2105306118320

>>> a += b

>>> a

'HolaStackOverflow'

>>> id(a)

2105306110544 # a ahora hace referencia a otro objeto

>>> a = ["Hola"]

>>> b = ["StackOverflow"]

>>> id(a)

2105306091848

>>> id(b)

2105305993800

>>> a += b

>>> a

['Hola', 'StackOverflow']

>>> id(a)

2105306091848 # a sigue haciendo referencia al mismo objeto

Python maneja la asignación por referencia, una variable o atributo no es más que un identificador, un nombre, que hace referencia a un objeto en memoria. Los atributos de clase se comparten entre todas las instancias porque hacen referencia al mismo objeto en memoria, por lo tanto cuando modificamos atributos de clase mutables a través de instancia.atributo o self.atributo el cambio si que se refleja en todas las instancias:

class Foo:

a = [100] # Atributo de clase

def \_\_init\_\_(self, n):

self.b = [n] # Atributo de instancia

>>> inst1 = Foo(1)

>>> inst2 = Foo(2)

>>> inst1.a

[100]

>>> inst2.a

[100]

>>> inst1.a += [5]

>>> inst1.a

[100, 5]

>>> inst2.a

[100, 5]

Esto no ocurre con el atributo de instancia ya que es propio de cada objeto:

>>> inst1.b

[1]

>>> inst2.b

[2]

>>> inst1.b += [5]

>>> inst1.b

[1, 5]

>>> inst2.b

[2]

Si el atributo de clase hace referencia a un objeto inmutable aunque intentemos "modificarlo" en una instancia esto no implica hacerlo en el resto, ya que "modificar" un objeto inmutable implica siempre una asignación de por medio y como se ha visto antes, una asignación desde la instancia ocasiona la creación de un nuevo atributo de instancia, no en la modificación del atributo de clase (a no ser que sea un descriptor de datos):

>>> class Foo:

s = "Hola"

>>> a = Foo()

>>> b = Foo()

>>> a.s += " StackOverflow"

>>> a.s

'Hola StackOverflow'

>>> b.s

'Hola'

A diferencia de lo que pasaba con la lista la concatenación de cadenas (o las operaciones con enteros, floats, etc) implica la creación de un nuevo objeto y un intento de asignación al atributo, lo cual implica la creación de un atributo de instancia como ya vimos antes y no la alteración del atributo de clase.

*Aunque los descriptores están íntimamente ligados a todo esto y se mencionan repetidamente no se entra en ellos porque explicar el protocolo descriptor implicaría extender demasiado la respuesta cuando ya es posiblemente demasiado larga, podemos ver en la propia documentación (en inglés) una guía general acerca de esto:*[*Descriptor HowTo Guide*](https://docs.python.org/3/howto/descriptor.html#descriptor-protocol)