<http://stackoverflow.com/questions/13091221/reinitialize-an-object-with-self-init>

**問題描述:**

Could anybody explain whether it is safe to reinitialize an object by calling "self.init(". as shown in the following simplified example?

The reason i'm asking is that i couldn't find this method neither in several python books nor in internet. There are some who suggest to list all attributes and set them to initial value one by one. Basically i want to set my object to initial state after it has finished some tasks.

**原始碼:**

class Book(object):

def \_\_init\_\_(self,name,author):

self.name = name

self.author = author

self.copies = 5

def reset(self):

self.\_\_init\_\_(self.name,self.author)

def incrementCopy(self):

self.copies += 1

Kite = Book('kite runner','khaled hosseini')

print 'initial number of copies:', Kite.copies

Kite.incrementCopy()

Kite.incrementCopy()

Kite.incrementCopy()

print '3 copies are added:', Kite.copies

Kite.reset()

print 'number of copies are reinitialized', Kite.copies

initial number of copies: 5

3 copies are added: 8

number of copies are reinitialized 5

A

Why can't you do that in reset function itself, after you will have to pass the initial values again? self.name and self.author accesses the current value, not the initial. – Rohit Jain Oct 26 '12 at 17:02

self.name and self.author are not going to be changed, they can be ignored in this example. My question is about self.\_\_init\_\_ part. I didn't quite understand your question. What do you mean by "that"? – user1638532 Oct 26 '12 at 17:09

B

The only thing special about \_\_init\_\_ is that it is called automatically when an instance is created. Other than that it is a normal method, and it is safe to use it to set your object back to its initial state.

That being said, just because it is safe doesn't mean it is a good idea. Other people looking at your code might be confused by it, and it isn't difficult to do everything in a reset method (that \_\_init\_\_ can even call) to be more explicit about how the method is being used.

C

I would consider it a very bad practice - you should not \_\_init\_\_ manually (unless calling \_\_init\_\_ of the parent class). Also, passing object's data back to \_\_init\_\_ is somewhat strange.

Why not something like this

class Book(object):

def \_\_init\_\_(self,name,author):

self.name = name

self.author = author

self.reset()

def reset(self):

self.copies = 5

D

<http://blog.csdn.net/yatusiter/article/details/16953905>

# [Python class 初始化参数为 list 等 可变类型时遇到的问题](http://blog.csdn.net/yatusiter/article/details/16953905)

写了一个类，结果在初始化的时候，发现无论class怎么重新初始化，里面的list类型总是会带上上一个对象的内容，也就是无法清空

使用id()函数查看 class对应的attribute，结果发现id居然是一样的。。attribute变成了静态变量，以往使用强类型语言的经验直接傻了。。

查看了后面的文章发现：

    Python 默认初始化变量，只在def的时候做，也就是初始化以后，无论再如何操作，都不会执行默认变量，也就是import之后，就不会再初始化了

    如果要对可变变量进行初始化，则需要先使用占位类型（一般使用None）

**[python]** [view plaincopy](http://blog.csdn.net/yatusiter/article/details/16953905)[在CODE上查看代码片](https://code.csdn.net/snippets/81830)

1. **class** test(arg=None):
2. **if** (test **is** None):
3. test = []
4. esle:
5. ....

参考：<http://effbot.org/zone/default-values.htm>

# 

# Default Parameter Values in Python

Fredrik Lundh | July 17, 2008 | based on a comp.lang.python post

(It happened to me in one of the first Python programs I ever wrote, and it took several years before we spotted the (non-critical) bug, when someone looked a bit more carefully at the contents of a property file, and wondered what all those things were doing there…)

Python’s handling of default parameter values is one of a few things that tends to trip up most new Python programmers (but usually only once).

What causes the confusion is the behaviour you get when you use a “mutable” object as a default value; that is, a value that can be modified in place, like a list or a dictionary.

An example:

>>> def function(data=[]):

... data.append(1)

... return data

...

>>> function()

[1]

>>> function()

[1, 1]

>>> function()

[1, 1, 1]

As you can see, the list keeps getting longer and longer. If you look at the list identity, you’ll see that the function keeps returning the same object:

**[python]** [view plaincopy](http://blog.csdn.net/yatusiter/article/details/16953905)

1. <code style="margin:0px; padding:0px">>>> id(function())
2. 12516768
3. >>> id(function())
4. 12516768
5. >>> id(function())
6. 12516768
7. </code>

The reason is simple: the function keeps using the same object, in each call. The modifications we make are “sticky”.

### Why does this happen?

Default parameter values are always evaluated when, and only when, the “def” statement they belong to is executed; see:

<http://docs.python.org/ref/function.html>

for the relevant section in the Language Reference.

Also note that “def” is an executable statement in Python, and that default arguments are evaluated in the “def” statement’s environment. If you execute “def” multiple times, it’ll create a new function object (with freshly calculated default values) each time. We’ll see examples of this below.

### What to do instead?

The workaround is, as others have mentioned, to use a placeholder value instead of modifying the default value.**None** is a common value:

def myfunc(value=None):

if value is None:

value = []

# modify value here

If you need to handle arbitrary objects (including None), you can use a sentinel object:

sentinel = object()

def myfunc(value=sentinel):

if value is sentinel:

value = expression

# use/modify value here

In older code, written before “object” was introduced, you sometimes see things like

**[python]** [view plaincopy](http://blog.csdn.net/yatusiter/article/details/16953905)

1. <code style="margin:0px; padding:0px">sentinel = ['placeholder']
2. </code>

used to create a non-false object with a unique identity; [] creates a new list every time it is evaluated.

### Valid uses for mutable defaults

Finally, it should be noted that more advanced Python code often uses this mechanism to its advantage; for example, if you create a bunch of UI buttons in a loop, you might try something like:

**[python]** [view plaincopy](http://blog.csdn.net/yatusiter/article/details/16953905)

1. <code style="margin:0px; padding:0px">**for** i **in** range(10):
2. **def** callback():
3. **print** "clicked button", i
4. UI.Button("button %s" % i, callback)
5. </code>

only to find that all callbacks print the same value (most likely 9, in this case). The reason for this is that Python’s nested scopes bind to variables, not object values, so all callback instances will see the current (=last) value of the “i” variable. To fix this, use explicit binding:

**[python]** [view plaincopy](http://blog.csdn.net/yatusiter/article/details/16953905)

1. <code style="margin:0px; padding:0px">**for** i **in** range(10):
2. **def** callback(i=i):
3. **print** "clicked button", i
4. UI.Button("button %s" % i, callback)
5. </code>

The “i=i” part binds the parameter “i” (a local variable) to the current value of the outer variable “i”.

Two other uses are local caches/memoization; e.g.

**[python]** [view plaincopy](http://blog.csdn.net/yatusiter/article/details/16953905)

1. <code style="margin:0px; padding:0px">**def** calculate(a, b, c, memo={}):
2. **try**:
3. value = memo[a, b, c] # return already calculated value
4. **except** KeyError:
5. value = heavy\_calculation(a, b, c)
6. memo[a, b, c] = value # update the memo dictionary
7. **return** value
8. </code>

(this is especially nice for certain kinds of recursive algorithms)

and, for highly optimized code, local rebinding of global names:

**[python]** [view plaincopy](http://blog.csdn.net/yatusiter/article/details/16953905)

1. <code style="margin:0px; padding:0px">**import** math
3. **def** this\_one\_must\_be\_fast(x, sin=math.sin, cos=math.cos):
4. ...
5. </code>

### How does this work, in detail?

When Python executes a “def” statement, it takes some ready-made pieces (including the compiled code for the function body and the current namespace), and creates a new function object. When it does this, it also evaluates the default values.

The various components are available as attributes on the function object; using the function we used above:

**[python]** [view plaincopy](http://blog.csdn.net/yatusiter/article/details/16953905)

1. <code style="margin:0px; padding:0px">>>> function.func\_name
2. 'function'
3. >>> function.func\_code
4. <code object function at 00BEC770, file "<stdin>", line 1>
5. >>> function.func\_defaults
6. ([1, 1, 1],)
7. >>> function.func\_globals
8. {'function': <function function at 0x00BF1C30>,
9. '\_\_builtins\_\_': <module '\_\_builtin\_\_' (built-**in**)>,
10. '\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None}
11. </code>

Since you can access the defaults, you can also modify them:

**[python]** [view plaincopy](http://blog.csdn.net/yatusiter/article/details/16953905)

1. <code style="margin:0px; padding:0px">>>> function.func\_defaults[0][:] = []
2. >>> function()
3. [1]
4. >>> function.func\_defaults
5. ([1],)
6. </code>

However, this is not exactly something I’d recommend for regular use…

Another way to reset the defaults is to simply re-execute the same “def” statement. Python will then create a new binding to the code object, evaluate the defaults, and assign the function object to the same variable as before. But again, only do that if you know exactly what you’re doing.

And yes, if you happen to have the pieces but not the function, you can use the **function** class in the **new** module to create your own function object.