

Re: Semi-newbie, rolling my own __deepcopy__

Re: Semi-newbie, rolling my own __deepcopy__

Source: <http://coding.derkeiler.com/Archive/Python/comp.lang.python/2005-04/msg01085.html>

- *From:* "ladasky@xxxxxxxxxxxx" <ladasky@xxxxxxxxxxxx>
 - *Date:* 6 Apr 2005 04:34:25 -0700
-

Michael Spencer wrote:

> ladasky@xxxxxxxxxxxx wrote:

[snip]

>> Anyway, my present problem is that I want to make copies of instances

>> of my own custom classes. I'm having a little trouble understanding

>> the process. Not that I think that it matters -- but in case it does,

>> I'll tell you that I'm running Python 2.3.4 on a Win32 machine.

[snip]

> If you google for:

> python __deepcopy__ cookbook

> you will find a couple of examples of this method in use, among them:

> <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/259179>

>

> 1 class deque(object):

> 2

> 3 def __init__(self, iterable=()):

> 4 if not hasattr(self, 'data'):

> 5 self.left = self.right = 0

> 6 self.data = {}

> 7 self.extend(iterable)

>

> [...snip methods...]

>

> 8 def __deepcopy__(self, memo={ }):

> 9 from copy import deepcopy

> 10 result = self.__class__()

> 11 memo[id(self)] = result

> 12 result.__init__(deepcopy(tuple(self), memo))

> 13 return result

>

> HTH

> Michael

Hi, Michael,

Re: Semi-newbie, rolling my own `__deepcopy__`

I was indeed finding code like this in my web searches, though not this particular example. I'm glad that you cut out code that is irrelevant to the `deepcopy` operation. Still, I want to understand what is going on here, and I don't. I've numbered the lines in your example.

So, entering `deepcopy`, I encounter the first new concept (for me) on line 10. We obtain the class/type of `self`. On line 11 we create a dictionary item in `memo`, `[id(self):type(self)]`. So now I'm confused as to the purpose of `memo`. Why should it contain the ID of the **original** object?

Things get even stranger for me on line 12. Working from the inside of the parentheses outward, an attempt is made to convert `self` to a tuple. Shouldn't this generate a `TypeError` when given a complex, non-iterable item like the `deque` class? I just tried running one of my programs, which assigns the name "x" to one of my custom objects, and when I execute `tuple(x)`, a `TypeError` is what I get.

Anyway, I like what I think I see when you finally call `__init__` for the copied object, on lines 4–6. If `__deepcopy__` calls `__init__`, then the new `deque` object should already have a "data" attribute. So "data" is only initialized if it doesn't already exist.

I do not understand whether the "iterable" variable, which appears on lines 3 and 7, is relevant to the copying operation. Line 12 calls `__init__` with a single argument, not two, so `iterable` should be declared as an empty tuple. Again I see what I think should generate a `TypeError` on line 7, when an attempt is made to extend the non-sequence object "self" with the iterable tuple.

Is there another section of the Python docs that will clarify all this for me? I got hung up on all the "static", "public", etc. declarations in Java early on. Python has taken me an awful lot farther, but perhaps I'm hitting the conceptual wall again.

—
Rainforest laid low.
"Wake up and smell the ozone,"
Says man with chainsaw.
John J. Ladasky Jr., Ph.D.

• *Follow-Ups:*

- ◆ *Re: Semi-newbie, rolling my own `__deepcopy__`*
 ◇ *From: Michael Spencer*
- ◆ *Re: Semi-newbie, rolling my own `__deepcopy__`*
 ◇ *From: Steven Bethard*

Re: Semi-newbie, rolling my own __deepcopy__

- **References:**

- ◆ **Semi-newbie, rolling my own deepcopy**
◇ From: ladasky@xxxxxxxxxxxx
- ◆ **Re: Semi-newbie, rolling my own deepcopy**
◇ From: Michael Spencer

- Prev by Date: **Re: boring the reader to death (was Re: Lambda: the Ultimate Design Flaw**
- Next by Date: **ignoring keywords on func. call**
- Previous by thread: **Re: Semi-newbie, rolling my own deepcopy**
- Next by thread: **Re: Semi-newbie, rolling my own deepcopy**
- Index(es):
 - ◆ **Date**
 - ◆ **Thread**