Python destructor and garbage collection notes

Monday, July 7th, 2008

I hardly ever use destructors in Python objects. I guess Python's dynamic nature often negates the need for destructors. Today though, I needed to write some data to disk when an object was destroyed, or more accurately, when the program exited. So I defined a destructor in the main controller object using the `__del__` magic method. To my surprise, the destructor was *never* called. Not only was it never called upon program exit, but also not when I deleted it manually (using `del`). The code I needed this in was written a while ago, so I wasn't intimately familiar with it anymore, leading me to think it was some strange bug in my program. I eventually traced the problem to some code which basically did this:

```
class Foo:
        def __init__(self, x):
                print "Foo: Hi"
                self.x = x
        def __del__(self):
                print "Foo: bye"

class Bar:
        def __init__(self):
                print "Bar: Hi"
                self.foo = Foo(self) # x = this instance

        def __del__(self):
                print "Bar: Bye"

bar = Bar()
# del bar # This doesn't work either.
```

What the above code does is that the Foo instance keeps a reference to its creator class, which is an instance of Bar. The output is:

```
Bar: Hi
Foo: Hi
```

As you can see, the destructors are never called, not even when we add a `del bar` at the end of the program. Removing the `self.x = x` solves (well, makes it disappear) the problem.

Garbage collection
The reason that `__del__` is never called suddenly becomes obvious when looking at the above code. It's a 'problem' with certain garbage collectors, namely: circular referencing. Python uses a reference counting

garbage collecting algorithm. Such an garbage collection algorithm increases a counter on each data instance for each reference that exists to that data instance and decreases the counter when a reference to the data instance is removed. When the counter reaches zero, the data instance is garbage collected because nothing points to it anymore. Reference counting has a problem with circular links. In the above code `foo.x` points to `bar`, and `bar.foo` points to `foo`. This means that the reference counter never goes down, and the objects never get garbage collected. The destructors never gets called because of this.

The reason `del foo` doesn't work is also simple to explain. I initially confused myself by thinking that `del foo` would call the destructor, but it only decreases the reference counter on the object (and removes the reference from the local scope). Since the count in the code above for Foo and Bar is 2 (one in the main program, one in the other instances), the count will only go down to 1 for the object.

Further info
After figuring this out, somebody (thanks Cris) pointed me to the documentation on `__del__`. Had I bothered to read it earlier I would have noticed the note there:

> "del x" doesn't directly call x.__del__() — the former decrements the reference count for x by one, and the latter is only called when x's reference count reaches zero. Some common situations that may prevent the reference count of an object from going to zero include: circular references between objects

Something else I noticed in the documentation for `__del__`:

> Circular references which are garbage are detected when the option cycle detector is enabled (it's on by default), but can only be cleaned up if there are no Python-level __del__() methods involved.

Furher reading reveals:

> A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). By default, this list contains only objects with __del__() methods.26.1Objects that have __del__() methods and are part of a reference cycle cause the entire reference cycle to be uncollectable, including objects not necessarily in the cycle but reachable only from it. Python doesn't collect such cycles automatically because, in general, it isn't possible for Python to guess a safe order in which to run the __del__() methods. […] It's generally better to avoid the issue by not creating cycles containing objects with __del__() methods

This means that objects with cyclic references and `__del__` methods will generate memory leaks in your Python program, unless the cyclic references are manually broken before the object is going to be deleted. Something to keep under consideration.

Program exit
You may wonder why Python doesn't simply set all reference counts to 0 when exiting the program? As outlined in this post by the BDFL on __del__:

> One final thing to ponder: if we have a __del__ method, should the interpreter guarantee that it is called when the program exits? (Like C++, which guarantees that destructors of global variables are called.) The only way to guarantee this is to go running around all modules and delete all their variables. But this means

> that __del__ method cannot trust that any global variables it might want to use still exist, since there is no way to know in what order variables are to be deleted.

Like the manual mentions: *It is not guaranteed that __del__() methods are called for objects that still exist when the interpreter exits.*

Exceptions inside destructors
On a side-note, as mentioned in the post by Van Rossom, exceptions raised in destructors are ignored:

```
        def __del__(self):
                raise Exception("Oopsy")
                print "Bar: Bye"
```

```
Exception exceptions.Exception: Exception('Oopsy',) in > ignored
Bar: Hi
Foo: Hi
Foo: bye
```

(The warning is generated during compile-time, not run-time)

More side-notes
On another side-note: This is why it pays to learn software engineering students basic stuff like C programming and how garbage collectors and various algorithms work, even though some educators seem to think such information is not required anymore with today's high-level languages.

Also, don't rely on your IDE's ability to display code-completions with a short description of the method and its parameters. Even though I probably wouldn't have read the full documentation on `__del__` and `del` anyway, I've often found that important notes and limitations are missed when not reading the full documentation of methods (deprecation notices, security concerns and nasty side effects in particular). If you want to have some fun with that, try to find a way to securely generate temporary files on a Unix system using Python or C using the manuals.

Update: A good way to prevent circular references seems to be the weakref module: weakref — Weak references. A quick introduction: Mindtrove: Python Weak References. (Thanks to zzzeek @ reddit for pointing out weakrefs)