Safely using destructors in Python

June 12, 2009 at 08:40 Tags Articles , Python

This post applies to Python 2.5 and 2.6 - if you see any difference for Python 3, please let me know.

Destructors are a very important concept in C++, where they're an essential ingredient of RAII - virtually the only real safe way to write code that involves allocation and deallocation of resources in an exception-throwing program.

In Python, destructors are needed much less, because Python has a garbage collector that handles memory management. However, while memory is the most common resource allocated, it is not the only one. There are also sockets and database connections to be closed, files, buffers and caches flushed and a few more resources that need to be released when an object is done with them.

So Python has the destructor concept - the `__del__` method. For some reason, many in the Python community believe that `__del__` is evil and shouldn't be used. However, a simple `grep` of the standard library shows dozens of uses of `__del__` in classes we all use and love, so where's the catch? In this article I'll try to make it clear (first and foremost for myself), when `__del__` should be used, and how.

Simple code samples

First a basic example:

```
class FooType(object):
    def __init__(self, id):
        self.id = id
        print self.id, 'born'

    def __del__(self):
        print self.id, 'died'


ft = FooType(1)
```

This prints:

```
1 born
1 died
```

Now, recall that due to the usage of a reference-counting garbage collector, Python won't clean up an object when it goes out of scope. It will clean it up when the last reference to it has gone out of scope. Here's a demonstration:

```
class FooType(object):
    def __init__(self, id):
        self.id = id
        print self.id, 'born'

    def __del__(self):
        print self.id, 'died'

def make_foo():
    print 'Making...'
    ft = FooType(1)
    print 'Returning...'
    return ft

print 'Calling...'
ft = make_foo()
print 'End...'
```

This prints:

```
Calling...
Making...
1 born
Returning...
End...
1 died
```

The destructor was called after the program ended, not when `ft` went out of scope inside `make_foo`.

Alternatives to the destructor

Before I proceed, a proper disclosure: Python provides a better method for managing resources than destructors - contexts. I won't turn this into a tutorial of contexts, but you should really get yourself familiar with the `with` statement and objects that can be used inside. For example, the best way to handle writing to a file is:

```
with open('out.txt', 'w') as of:
    of.write('222')
```

This makes sure the file is properly closed when the block inside `with` exits, even if exceptions are thrown. Note that this demonstrates a standard context manager. Another is `threading.lock`, which returns a context manager very suitable to be used in a `with` statement. You should read PEP 343 for more details.

While recommended, `with` isn't always applicable. For example, assume you have an object that encapsulates

some sort of a database that has to be committed and closed when the object ends its existence. Now suppose the object should be a member variable of some large and complex class (say, a GUI dialog, or a MVC model class). The parent interacts with the DB object from time to time in different methods, so using `with` isn't practical. What's needed is a functioning destructor.

Where destructors go astray

To solve the use case I presented in the last paragraph, you can employ the `__del__` destructor. However, it's important to know that this doesn't always work well. The nemesis of a reference-counting garbage collector is circular references. Here's an example:

```
class FooType(object):
    def __init__(self, id, parent):
        self.id = id
        self.parent = parent
        print 'Foo', self.id, 'born'

    def __del__(self):
        print 'Foo', self.id, 'died'


class BarType(object):
    def __init__(self, id):
        self.id = id
        self.foo = FooType(id, self)
        print 'Bar', self.id, 'born'

    def __del__(self):
        print 'Bar', self.id, 'died'


b = BarType(12)
```

Output:

```
Foo 12 born
Bar 12 born
```

Ouch... what has happened? Where are the destructors? Here's what the Python documentation has to say on the matter:

> 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.

Python doesn't know the order in which it's safe to destroy objects that hold circular references to each other, so as a design decision, it just doesn't call the destructors for such methods!

So, now what?

Shouldn't we use destructors because of this deficiency? I'm very surprised to see that many Pythonistas think so, and recommend to use explicit `close` methods. But I disagree - explicit `close` methods are less safe, since they are easy to forget to call. Moreover, when exceptions can happen (and in Python they happen all the time), managing explicit closing becomes very difficult and burdensome.

I actually think that destructors can and should be used safely in Python. With a couple of precautions, it's definitely possible.

First and foremost, note that justified cyclic references are a rare occurrence. I say *justified* on purpose - a lot of uses in which cyclic references arise are an example of bad design and leaky abstractions.

As a general rule of thumb, resources should be held by the lowest-level objects possible. Don't hold a DB resource directly in your GUI dialog. Use an object to encapsulate the DB connection and close it safely in the destructor. The DB object has no reason whatsoever to hold references to other objects in your code. If it does - it violates several good-design practices.

Sometimes Dependency Injection can help prevent cyclic references in complex code, but even in those rare few cases when you find yourself needing a true cyclic reference, there's a solution. Python provides the weakref module for this purpose. The documentation quickly reveals that this is exactly what we need here:

> A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, garbage collection is free to destroy the referent and reuse its memory for something else. A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

Here's the previous example rewritten with `weakref`:

```
import weakref

class FooType(object):
    def __init__(self, id, parent):
        self.id = id
        self.parent = weakref.ref(parent)
        print 'Foo', self.id, 'born'

    def __del__(self):
        print 'Foo', self.id, 'died'
```

```
class BarType(object):
    def __init__(self, id):
        self.id = id
        self.foo = FooType(id, self)
        print 'Bar', self.id, 'born'

    def __del__(self):
        print 'Bar', self.id, 'died'

b = BarType(12)
```

Now we get the result we want:

```
Foo 12 born
Bar 12 born
Bar 12 died
Foo 12 died
```

The tiny change in this example is that I use `weakref.ref` to assign the `parent` reference in the constructor `FooType`. This is a weak reference, so it doesn't really create a cycle. Since the GC sees no cycle, it destroys both objects.

Conclusion

Python has perfectly usable object destruction via the `__del__` method. It works fine for the vast majority of use-cases, but chokes on cyclic references. Cyclic references, however, are often a sign of bad design, and few of them are justified. For the teeny tiny amount of uses cases where justified cyclic references have to be used, the cycles can be easily broken with weak references, which Python provides in the `weakref` module.

References

Some links that were useful in the preparation of this article: