mxProxy

Object Access Control for Python

Version 3.2

Printed in Germany.

Contents

1. Introduction

You've probably ran into that problem too while writing an application with multi-user access: Python has no apparent way to effectively hide attributes from unauthorized access other than running in an restricted environment (which isn't supported anymore in Python 2.1 and up) or using the standard lib's Bastion object wrapper.

Since using the restricted mode is not an option anymore or can't be used due to being too restrictive (e.g. your code does assignments to __class__ and __dict__ in a few vital places), we've created a new type that hides Python object attributes away in the C implementation. These attributes are not accessible from within Python. Of course, a debugger will give you access and some special extension module that knows about the internal structure of the types could too. But for most applications, this level security suffices.

Instead of creating a special Bastion-like type we decided to write a more generic and extendable version of a *Proxy* object. The object itself doesn't provide too much functionality, but it effectively hides the wrapped object and provides controlled access to its attributes.

1.1 Object Reference: Weak or Strong

Another nice feature that a proxy object allows you to implement, is that of *weak references*, i.e. references to objects that don't prevent them to be garbage collected.

Since Python uses a reference count garbage collection scheme, circular references cause memory leakage as soon as the reference to the cycle is dropped (the involved objects are not garbage collected because their reference count never drops to zero).

Even though Python does have weak reference support for a number of basic types nowadays[1], this support only works if the referencing objects know how to handle weak references. mxProxy object by contrast can be used to create weak references to any Python object and store them in any Python object container or attribute.

---

[1] It didn't support weak references at the time mxProxy was written. The weak reference implementation in Python was in part inspired by mxProxy.

There are two flavors of Proxy object references: strong references which act just like any other reference you use to the object in Python and weak references. The `WeakProxy()` constructor must be used for the latter. Details are described in section 1.5 Weak References below.

1.2 Access Protocol

To control access to the wrapped objects attributes and methods, the Proxy provides two features:

- Access is only granted if the attributes name appears in a special *interface list* of accessible names.
- Public access can be further controlled if the object provides the methods `.__public_getattr__()` and/or `.__public_setattr__()`.

The access scheme first checks the interface list. If given, only attributes with names appearing in it can be set or fetched. All other accesses are cancelled by raising an `AccessError` exception (which is a subclass of the standard `AttributeError`).

The presence of the interface list also indicates whether methods that are to be returned to the requesting object should be Proxy-wrapped on-the-fly or not. Wrapped methods only have the callable slot enabled, thus inhibiting access to the enclosed object reference.

In case an attribute it to be fetched, the Proxy checks the availability of a `.__public_getattr__()` method. If found, this method is called with the name as parameter and whatever this method returns is returned to the requesting object. Otherwise, the Proxy uses the standard `getattr()`-functionality to fetch the attribute.

Setting attributes is done in a similar way: the Proxy checks the availability of a `.__public_setattr__()` method. If found, this method is called with parameters name and value. Otherwise, the standard `setattr()` functionality is used.

1.3 Cleanup Protocol

While Python provides support for garbage collection nowadays, it is still possible to create systems of cyclically referenced objects that the built-in mechanism cannot collect. Since these reference loops cannot automatically be broken by the system, the user has to provide means of breaking the circles at the proper places. One such place is the object destructor of objects that explicitly contain such references.

The Proxy object can be used to provide an indirect pointer into such a circle of objects that reference each other. When the system deletes the Proxy object, its destructor tries to find a special method `.__cleanup__()` in the wrapped object and calls it before continuing the destruction. This allows the object to break the reference circles, e.g. by performing a `self.__dict__.clear()` or something similar.

Errors raised in `.__cleanup__()` calls are ignored. Warnings are printed to `stderr` in case Python is run in verbose mode (`python -v`) and tracebacks printed if run in debug mode (`python -d`).

1.4 Implicit Access

On many occasions object attributes are not explicitly accessed by e.g. using `object.attribute`, but indirect through built-in functions or the interpreter itself. This poses a problem to the Proxy, since there are different ways to reach an objects implementation. For Python instances all important hooks are reachable via `getattr()`, e.g. `.__len__()` and `.__add__()`. This is different for extension and built-in types: they use a system of slots for providing access to their data.

Proxy objects also allow managing the type slot interface for most applications. This means that you can transparently use the Proxy to wrap built-in types such as lists or tuples and use the Proxy just like you would use the referenced built-in object itself.

Since Proxies are about access management, you can also restrict access to these slots. For simplicity, the names you have to use for different slots are exactly those you would define for Python classes, e.g. the length slot is named '`__len__`', the sequence and mapping get item slot are grouped under the name '`__getitem__`'. These names have to be explicitly stated in the interface list you pass to the Proxy constructor when creating the proxy in case you do define an interface list. If you don't specify such a list, no direct interface restriction is applied.

These type slot names are defined:

Generic:

`__call__, __hash__, __str__, __cmp__`

Mapping:

`__getitem__, __setitem__, __len__`

Sequence:

`__add__, __repeat__, __getitem__, __getslice__ __setitem__, __getitem__`

Number:

`__add__, __sub__, __mul__, __div__ __mod__, __divmod__, __pow__, __neg__, __pos__, __abs__, __true__ __invert__, __lshift__, __rshift__, __and__, __str__, __or__, __int__ __long__, __float__`

> Note: number coercion does not yet work, so most of these are currently useless !

Omissions currently are: `__coerce__`, `__hex__` and `__oct__`. `__repr__` is handled by the Proxy object itself since it would otherwise possibly expose address information about the underlying object.

Proxies also work transparently for Python instances. To achieve this, another proxy-like object which is a real Python instance has to be put in front of the Proxy object. Access to this instance then gets translated into `getattr()`-calls by the interpreter, these calls are filtered through the Proxy and the wrapped instance object's attribute then gets accessed in the usual way. The result is passed back to the requesting object.

1.5 Weak References

Proxy objects work in two modes: keeping a strong or a weak reference to the object. The `Proxy()` constructor returns a Proxy object using a strong reference, the `WeakProxy()` constructor one using a weak reference.

Weak references are called weak because they don't keep the object alive by incrementing the reference count on the referenced object. Since objects get garbage collected when this reference count falls down to 0, a weak reference can become invalid at any time. The mxProxy implementation raises a `LostReferenceError` in case a weak reference to such an object is used.

This may sound like a pretty flaky feature at first, but the main pro argument for these weak references is that you can build up circular references without having to fear about them not being properly garbage collected. Using strong references (which do increment the reference count and thus keep the object alive as long as the reference is around) would produce cycles in the referencing scheme which the Python garbage collection (GC) mechanism cannot automatically break causing the cycle to become unreachable from the Python namespaces: a severe memory leak.

Weak references in mxProxy work by using a global dictionary of all objects handled through weak reference Proxies. This dictionary is checked prior to every action on a weak Proxy and after its deletion. You can also force a check by calling the `checkweakrefs()` anytime you like, e.g. at regular intervals.

The dictionary holds a strong reference to the object keeping it alive until the next check is done. During the check all handled objects are inspected to see if their reference count has gone down to 1 (*phantom objects*: only the dictionary references them). If this is the case, all weak proxies are marked defunct and the object is removed from the dictionary causing it to be garbage collected by the Python GC mechanism. All subsequent actions on the weak references to this object will then cause a `LostReferenceError` exception to be raised.

2. mx.Proxy.Proxy Object

The Proxy object implements all of the above and provides the following interface.

2.1 Proxy Object Constructors

These constructors are available in the package:

`Proxy(object,interface=None,passobj=None)`

Returns a new Proxy instance that wraps object.

`interface` can be given as sequence of strings and/or objects with `__name__` attribute or as dictionary with string keys (only the keys are currently used) and tells the Proxy to only allow access to these names. If not given or None, no filtering is done by the Proxy (see above on how access is managed).

`passobj` can be provided to retrieve the wrapped object from the Proxy at a later point using the `.proxy_object()` method.

`InstanceProxy(object,interface=None,passobj=None)`

Same as above, except that a Python instance wraps the Proxy object.

This makes the Proxy transparent for access to wrapped Python instances, meaning that the Proxy will act as if it were the wrapped object itself (with the added features mentioned above).

`CachingInstanceProxy(object,interface=None,passobj=None)`

Same as `InstanceProxy()`, except that a read cache is used by the Proxy which caches all queried attributes in the Proxy instance's dictionary.

Note that this may introduce circular references if not used properly. Cached attributes are not looked up in the wrapped instance after the first lookup -- if their value changes, this won't be noticed by objects that access the object through this wrapper.

`SelectiveCachingInstanceProxy(object,interface=None,passobj=None)`

Same as `InstanceProxy()`, except that a read cache is used by the Proxy which caches certain queried attributes in the Proxy instance's dictionary depending on their type.

The cached types are defined by the `.proxy_cacheable_types` attribute. It defaults to only cache Python methods.

`MethodCachingProxy(object,interface=None,passobj=None)`

Alias for `SelectiveCachingInstanceProxy()`.

`ReadonlyInstanceProxy(object,interface=None,passobj=None)`

Same as `InstanceProxy()`, except that write access will result in an `AccessError` being raised

`ProxyFactory(Class,interface=None)`

Returns a factory object for producing Class instances that are automatically wrapped using Proxy-instances.

interface is passed to the Proxy constructor, pass-objects are not supported.

`InstanceProxyFactory(Class,interface=None)`

Returns a factory object for producing Class instances that are automatically wrapped using `InstanceProxy`-instances.

`WeakProxy(object,interface=None,passobj=None)`

Returns a new weak referencing Proxy instance that points to object.

interface can be given as sequence of strings and/or objects with `__name__` attribute or as dictionary with string keys (only the keys are currently used) and tells the Proxy to only allow access to these names. If not given or None, no filtering is done by the Proxy (see above on how access is managed).

`passobj` can be provided to retrieve the wrapped object from the Proxy at a later point using the `.proxy_object()` method.

For details on weak references and how they work, see section 1.5 Weak References.

2.2 Proxy Object Instance Methods

A `Proxy` instance `proxy` defines these methods in addition to the ones available through restricted access to the wrapped object:

`.proxy_defunct()`

Return 1 iff the referenced object has already been garbage collected.

`.proxy_getattr(name)`

Same as `getattr(proxy,name)`.

`.proxy_object(passobj)`

Returns the wrapped object provided the given `passobj` is identical to the one used for creating the Proxy.

Simple equality is not enough -- it has to be the same object.

`.proxy_setattr(name,value)`

Same as `setattr(proxy,name,value)`.

> Note that all attribute names starting with '`proxy_`' are interpreted as being Proxy attributes and are not passed to the wrapped object. Access to these attributes is *not* subject to the restrictions explained above.

2.3 Proxy Object Instance Variables

`Proxy` instances do not provide any instance variables themselves. They do provide restricted access to the variables of the wrapped object though.

> Note that all attribute names starting with '`proxy_`' are interpreted as being Proxy attributes and are not passed to the wrapped object. Also, access to these attributes is *not* subject to the restrictions explained above.

3. mx.Proxy Functions

These functions are available:

`checkweakrefs()`

Calling this function causes the global dictionary used for managing weak references to be checked for phantom objects. If such objects are found, they are garbage collected during the call and weak referencing Proxies pointing to them are defunct.

Weak references cause the objects to stay alive until either a proxy is used on them (causing an exception), a proxy referencing them is deleted or this function is called. To ensure the timely garbage collection of the objects, call this function at regular intervals.

`finalizeweakrefs()`

For internal use only:
Forces finalization of the weak reference implementation. Subsequent usage of weak references will cause errors to be raised.

Calling this function after finalization is not an error.

`initweakrefs()`

For internal use only:
Initializes or reinitializes the weak reference implementation. This first forces a finalization of the previous state if the implementation has already been used and then starts again with a clean weak reference dictionary.

4. mx.Proxy Constants

These constants are available:

`AccessError`

Exception object used for access specific errors that the module raises. It is a subclass of `AttributeError`.

5. Examples of Use

Here is a very simple one:

```
from mx import Proxy

class DataRecord:
    a = 2
    b = 3

    # Make read-only:
    def __public_setattr__(self,what,to):
     raise Proxy.AccessError('read-only')

    # Cleanup protocol
    def __cleanup__(self):
     print 'cleaning up',self

o = DataRecord()

# Wrap the instance:
p = Proxy.InstanceProxy(o,('a',))
# Remove o from the accessible system:
del o

print 'Read p.a through Proxy:',p.a

# This will cause an exception, because the object is read-only
p.a = 3

# This will cause an exception, because no access is given to .b
print p.b

# Clear all traces of the provious exceptions (they might contain
# references to p) by issuing another one. Note that not doing
# so will cause the following 'del p' to not destroy the final
# reference to p... (don't ask why).
1/0

# Deleting the Proxy will also delete the wrapped object, if there
# is no other reference to it in the system. It will invoke
# the __cleanup__ method in that case.
del p

# If you want to have the wrapping done automatically, you can use
# the InstanceProxyFactory:
DataRecord = Proxy.InstanceProxyFactory(DataRecord,('a',))

# This gives the same behaviour...
p = DataRecord()
print p.a
p.a = 3
print p.b
```

More examples will eventually appear in the Examples subdirectory of the package.

6. Package Structure

```
[Proxy]
       Doc/
       Examples/
       [mxProxy]
       Proxy.py
```

Entries enclosed in brackets are packages (i.e. they are directories that include a `__init__.py` file). Ones without brackets are just simple subdirectories that are not accessible via `import`.

The package imports all symbols from the Proxy sub module which in turn imports the extension module, so you only need to '`from mx import Proxy`' to start working.

7. Support

eGenix.com is providing commercial support for this package. If you are interested in receiving information about this service please see the *eGenix.com Support Conditions*.

8. Copyright & License

This software is covered by the ***eGenix.com Public License Agreement***, which is included in the following section. The text of the license is also included as file "LICENSE" in the package's main directory.

By downloading, copying, installing or otherwise using the software, you agree to be bound by the terms and conditions of the following *eGenix.com Public License Agreement*.

EGENIX.COM PUBLIC LICENSE AGREEMENT

Version 1.1.0

This license agreement is based on the Python CNRI License Agreement, a widely accepted open-source license.

1. Introduction

This "License Agreement" is between eGenix.com Software, Skills and Services GmbH ("eGenix.com"), having an office at Pastor-Loeh-Str. 48, D-40764 Langenfeld, Germany, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. License

Subject to the terms and conditions of this eGenix.com Public License Agreement, eGenix.com hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the eGenix.com Public License Agreement is retained in the Software, or in any derivative version of the Software prepared by Licensee.

3. NO WARRANTY

eGenix.com is making the Software available to Licensee on an "AS IS" basis. SUBJECT TO ANY STATUTORY WARRANTIES WHICH CAN NOT BE EXCLUDED, EGENIX.COM MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, EGENIX.COM MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. LIMITATION OF LIABILITY

EGENIX.COM SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) AS A RESULT OF USING, MODIFYING OR

DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE EXCLUSION OR LIMITATION MAY NOT APPLY TO LICENSEE.

5. Termination

This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. Third Party Rights

Any software or documentation in source or binary form provided along with the Software that is associated with a separate license agreement is licensed to Licensee under the terms of that license agreement. This License Agreement does not apply to those portions of the Software. Copies of the third party licenses are included in the Software Distribution.

7. General

Nothing in this License Agreement affects any statutory rights of consumers that cannot be waived or limited by contract.

Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between eGenix.com and Licensee.

If any provision of this License Agreement shall be unlawful, void, or for any reason unenforceable, such provision shall be modified to the extent necessary to render it enforceable without losing its intent, or, if no such modification is possible, be severed from this License Agreement and shall not affect the validity and enforceability of the remaining provisions of this License Agreement.

This License Agreement shall be governed by and interpreted in all respects by the law of Germany, excluding conflict of law provisions. It shall not be governed by the United Nations Convention on Contracts for International Sale of Goods.

This License Agreement does not grant permission to use eGenix.com trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party.

The controlling language of this License Agreement is English. If Licensee has received a translation into another language, it has been provided for Licensee's convenience only.

8. Agreement

By downloading, copying, installing or otherwise using the Software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

For question regarding this License Agreement, please write to:

eGenix.com Software, Skills and Services GmbH

Pastor-Loeh-Str. 48

D-40764 Langenfeld

Germany