Padme Documentation

Release 1.1.1

Zygmunt Krynicki

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Padme - a mostly transparent proxy class for Python

Features

- Free software: LGPLv3 license
- Support for Python 2.7 and Python 3.2+
- Documentation: https://padme.readthedocs.org.
- Create proxy classes for any object with *padme.proxy*.
- Redirect particular methods in subclasses using `padme.unproxied`.

CHAPTER 2

Installation

At the command line:

```
$ easy_install padme
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv padme
$ pip install padme
```

CHAPTER 3

Usage

padme – a mostly transparent proxy class for Python.

Padme, named after the Star Wars (tm) character, is a library for creating proxy objects out of any other python object.

The resulting object is as close to mimicking the original as possible. Some things are impossible to fake in CPython so those are highlighted below. All other operations are silently forwarded to the original.

Terminology

proxy: An intermediate object that is used in place of some original object.

proxiee: The original object hidden behind one or more proxies.

Basic features

Let's consider a simple example:

```
>>> pets = [str('cat'), str('dog'), str('fish')]
>>> pets_proxy = proxy(pets)
>>> pets_proxy
['cat', 'dog', 'fish']
>>> isinstance(pets_proxy, list)
True
>>> pets_proxy.append(str('rooster'))
>>> pets
['cat', 'dog', 'fish', 'rooster']
```

By default, a proxy object is not that interesting. What is more interesting is the ability to create subclasses that change a subset of the behavior. For implementation simplicity such methods need to be decorated with `@proxy.direct`.

Let's consider a crazy proxy that overrides the `__repr__()` method to censor the word 'cat'. This is how it can be implemented:

```
>>> class censor_cat(proxy):
...     @proxy.direct
...     def __repr__(self):
...         return repr(proxy.original(self)).replace(
...             str('cat'), str('***'))
```

Now let's create a proxy for our pets collection and see how it looks like:

```
>>> pets_proxy = censor_cat(pets)
>>> pets_proxy
['***', 'dog', 'fish', 'rooster']
```

As before, all other aspects of the proxy behave the same way. All of the methods work and are forwarded to the original object. The type of the proxy object is correct, event the meta-class of the object is correct (this matters for `issubclass()`, for instance).

Accessing the original object

At any time one can access the original object hidden behind any proxy by using the *`proxy.original()`* function. For example:

```
>>> obj = 'hello world'
>>> proxy.original(proxy(obj)) is obj
True
```

Accessing proxy state

At any time the state of any proxy object can be accessed using the *`proxy.state()`* function. The state object behaves as a regular object with attributes. It can be used to add custom state to an object that cannot hold it, for example:

```
>>> obj = 42
>>> obj.foo = 42
Traceback (most recent call last):
    ...
AttributeError: 'int' object has no attribute 'foo'
>>> obj = proxy(obj)
>>> obj.foo = 42
Traceback (most recent call last):
    ...
AttributeError: 'int' object has no attribute 'foo'
>>> proxy.state(obj).foo = 42
>>> proxy.state(obj).foo
42
```

Using the @proxy.direct decorator

The `@proxy.direct` decorator can be used to disable the automatic pass-through behavior that is exhibited by any proxy object. In practice we can use it to either intercept and substitute an existing functionality or to add a new functionality that doesn't exist in the original object.

First, let's write a custom proxy class for the `bool` class (which cannot be used as a base class anymore) and change the core functionality.

```
>>> class nay(proxy):
...
...     @proxy.direct
...     def __nonzero__(self):
...         return not bool(proxiee(self))
...
...     @proxy.direct
...     def __bool__(self):
...         return not bool(proxiee(self))
```

```
>>> bool(nay(True))
False
>>> bool(nay(False))
True
>>> if nay([]):
...     print("It works!")
It works!
```

Now, let's write a different proxy class that will add some new functionality

Here, the self_aware_proxy class gives any object a new property, `is_proxy` which always returns `True`.

```
>>> class self_aware_proxy(proxy):
...     @proxy.direct
...     def is_proxy(self):
...         return True
>>> self_aware_proxy('hello').is_proxy()
True
```

Limitations

There are only two things that that give our proxy away.

The `type()` function:

```
>>> type(pets_proxy)
<class '...censor_cat[list]'>
```

And the `id` function (and anything that checks object identity):

```
>>> pets_proxy is pets
False
>>> id(pets) == id(pets_proxy)
False
```

That's it, enjoy. You can read the unit tests for additional interesting details of how the proxy class works. Those are not covered in this short introduction.

Note: There are a number of classes and meta-classes but the only public interface is the *proxy* class and the *proxy.direct()* decorator. See below for examples.

Deprecated 1.0 APIs

If you've used Padme before you may have seen `@unproxied()` and `proxiee()`. They are still here but `@unproxied` is now spelled `@proxy.direct` and `proxiee()` is now `proxy.original()`. This was done to allow all of Padme to be used from the one *proxy* class.

Reference

class padme.**proxy**(*proxy_obj*, *proxiee*)

A mostly transparent proxy type.

The proxy class can be used in two different ways. First, as a callable `proxy(obj)`. This simply returns a proxy for a single object.

```
>>> truth = [str('trust no one')]
>>> lie = proxy(truth)
```

This will return an instance of a new `proxy` sub-class which for all intents and purposes, to the extent possible in CPython, forwards all requests to the original object.

One can still examine the proxy with some ways:

```
>>> lie is truth
False
>>> type(lie) is type(truth)
False
```

Having said that, the vast majority of stuff will make the proxy behave identically to the original object.

```
>>> lie[0]
'trust no one'
>>> lie[0] = str('trust the government')
>>> truth[0]
'trust the government'
```

The second way of using the `proxy` class is as a base class. In this way, one can actually override certain methods. To ensure that all the dunder methods work correctly please use the `@proxy.direct` decorator on them.

```
>>> import codecs
>>> class crypto(proxy):
...
...     @proxy.direct
...     def __repr__(self):
...         return codecs.encode(
...             super(crypto, self).__repr__(), "rot_13")
```

With this weird class, we can change the repr() of any object we want to be ROT-13 encoded. Let's see:

```
>>> orig = [str('ala ma kota'), str('a kot ma ale')]
>>> prox = crypto(orig)
```

We can sill access all of the data through the proxy:

```
>>> prox[0]
'ala ma kota'
```

But the whole repr() is now a bit different than usual:

```
>>> prox
['nyn zn xbgn', 'n xbg zn nyr']
```

direct (*fn*)
Mark a method as not-to-be-proxied.

This decorator can be used inside *proxy* sub-classes. Please consult the documentation of `proxy` for details.

In practical terms there are two reasons one can use `proxy.direct`.

- First, as a way to change the behaviour of a proxy. In this mode a method that already exists on the proxied object is intercepted and custom code is executed. The custom code can still call the original, if desired, by using the *proxy.original()* function to access the original object
- Second, as a way to introduce new functionality to an object. In that sense the resulting proxy will be less transparent as all `proxy.direct` methods are explicitly visible and available to access but this may be exactly what is desired in some situations.

For additional details on how to use this decorator, see the documentation of the *padme* module.

original (*proxy_obj*)
Return the proxiee hidden behind the given proxy.

Parameters **proxy** – An instance of *proxy* or its subclass.

Returns The original object that the proxy is hiding.

This function can be used to access the object hidden behind a proxy. This is useful when access to original object is necessary, for example, to implement an method decorated with `@proxy.direct`.

In the following example, we cannot use `super()` to get access to the append method because the proxy does not really subclass the list object. To override the `append` method in a way that allows us to still call the original we must use the *proxy.original()* function:

```
>>> class verbose_list(proxy):
...     @proxy.direct
...     def append(self, item):
...         print("Appending:", item)
...         proxy.original(self).append(item)
```

Now that we have a `verbose_list` class, we can use it to see that it works as expected:

```
>>> l = verbose_list([])
>>> l.append(42)
Appending: 42
>>> l
[42]
```

state (*proxy_obj*)
Support function for accessing the state of a proxy object.

The main reason for this function to exist is to facilitate creating stateful proxy objects. This allows you to put state on objects that cannot otherwise hold it (typically built-in classes or classes using `__slots__`) and to keep the state invisible to the original object so that it cannot interfere with any future APIs.

To use it, just call it on any proxy object and use the return value as a normal object you can get/set attributes on. For example:

```
>>> life = proxy(42)
```

We cannot set attributes on integer instances:

```
>>> life.foo = True
Traceback (most recent call last):
    ...
AttributeError: 'int' object has no attribute 'foo'
```

But we can do that with a proxy around the integer object.

```
>>> proxy.state(life).foo = True
>>> proxy.state(life).foo
True
```

Internals

class padme.**proxy_meta**

Meta-class for all proxy types.

This meta-class is responsible for gathering the __unproxied__ attributes on each created class. The attribute is a frozenset of names that will not be forwarded to the `proxiee` but instead will be looked up on the proxy itself.

padme.**make_typed_proxy_meta**(*proxiee_cls*)

Make a new proxy meta-class for the specified class of proxiee objects.

Note: Had python had an easier way of doing this, it would have been spelled as `proxy_meta[cls]` but I didn't want to drag pretty things into something nobody would ever see.

Parameters **proxiee_cls** – The type of the that will be proxied

Returns A new meta-class that lexically wraps `proxiee` and `proxiee_cls` and subclasses *proxy_meta*.

class padme.**proxy_base**

Base class for all proxies.

This class implements the bulk of the proxy work by having a lot of dunder methods that delegate their work to a `proxiee` object. The `proxiee` object must be available as the `__proxiee__` attribute on a class deriving from `base_proxy`. Apart from `__proxiee__`, the ``__unproxied__` attribute, which should be a frozenset, must also be present in all derived classes.

In practice, the two special attributes are injected via `boundproxy_meta` created by `make_boundproxy_meta()`. This class is also used as a base class for the tricky *proxy* below.

NOTE: Look at `pydoc3 SPECIALMETHODS` section titled `Special method lookup` for a rationale of why we have all those dunder methods while still having __getattribute__()

__abs__()

__add__(*other*)

__and__(*other*)

__bool__()

__bytes__()

__call__(**args*, ***kwargs*)

__complex__()

__contains__(*item*)

__del__()
No-op object delete method.

Note: This method is handled specially since it must be called after an object becomes unreachable. As long as the proxy object itself exits, it holds a strong reference to the original object.

__delattr__(*name*)

__delete__(*instance*)

__delitem__(*item*)

__dict__ = mappingproxy({'__rfloordiv__': <function proxy_base.__rfloordiv__>, '__round__': <function proxy_base.

__dir__()

__divmod__(*other*)

__enter__()

__eq__(*other*)

__exit__(*exc_type*, *exc_value*, *traceback*)

__float__()

__floordiv__(*other*)

__format__(*format_spec*)

__ge__(*other*)

__get__(*instance*, *owner*)

__getattr__(*name*)

__getattribute__(*name*)

__getitem__(*item*)

__gt__(*other*)

__hash__()

__iadd__(*other*)

__iand__(*other*)

__ifloordiv__(*other*)

__ilshift__(*other*)

__imatmul__(*other*)

__imod__(*other*)

__imul__(*other*)

__index__()

__int__()

__invert__()

__ior__(*other*)

__ipow__(*other*, *modulo=None*)

__irshift__(*other*)

__isub__(*other*)

__iter__()

__itruediv__(*other*)

__ixor__(*other*)

__le__(*other*)

__len__()

__length_hint__()

__lshift__(*other*)

__lt__(*other*)

__matmul__(*other*)

__mod__(*other*)

__module__ = 'padme'

__mul__(*other*)

__ne__(*other*)

__neg__()

__or__(*other*)

__pos__()

__pow__(*other*, *modulo=None*)

__radd__(*other*)

__rand__(*other*)

__rdivmod__(*other*)

__repr__()

__reversed__()

__rfloordiv__(*other*)

__rlshift__(*other*)

__rmatmul__(*other*)

__rmod__(*other*)

__rmul__(*other*)

__ror__(*other*)

__round__(*n*)

__rpow__(*other*)

__rrshift__(*other*)

__rshift__(*other*)

__rsub__(*other*)

__rtruediv__(*other*)

__rxor__(*other*)

__set__(*instance*, *value*)

__setattr__(*name*, *value*)

__setitem__(*item*, *value*)

__str__()

__sub__(*other*)

__truediv__(*other*)

__weakref__
list of weak references to the object (if defined)

__xor__(*other*)

class padme.**proxy_state**(*proxy_obj*)
Support class for working with proxy state.

This class implements simple attribute-based access methods. It is normally instantiated internally for each proxy object. You don't want to fuss with it manually, instead just use *proxy.state()* function to access it.

__dict__ = mappingproxy({'__repr__': <function proxy_state.__repr__>, '__init__': <function proxy_state.__init__>, '

__init__(*proxy_obj*)

__module__ = 'padme'

__repr__()

__weakref__
list of weak references to the object (if defined)

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/zyga/padme/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

Write Documentation

padme could always use more documentation, whether as part of the official padme docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/zyga/padme/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *padme* for local development.

1. Fork the *padme* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/padme.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv padme
$ cd padme/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 padme
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.2, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/zyga/padme/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ python -m unittest padme.tests
```

CHAPTER 5

Credits

Development Lead

- Zygmunt Krynicki <zygmunt.krynicki@canonical.com>

Contributors

None yet. Why not be the first?

CHAPTER 6

History

1.1.1 (2015-03-04)

- Add general support for **Python 2.7**.
- All numeric methods are now supported with some methods exclusive to Python 2.x (`__div__()`, `__coerce__()`, `__oct__()`, `__hex__()`).
- Add support for the new matrix multiplication operator `@`.
- Make `__nonzero__()` and `__unicode__()` exlusive to Python 2.x.
- Make `__bool__()` and `__bytes__()` exclusive to Python 3.x.
- Make `__length_hint()` exclusive to Python 3.4.
- Add support for the `__cmp__()` method, exclusive to Python 2.x.
- Add support for accessing the proxied object with the new *`original()`* function.
- Add support for accessing proxy state with the new *`state()`* function.
- De-couple proxy classes from proxied objects, much more lightweight proxy design is possible this way (less objects, lower cost to create each new proxy).

1.0 (2014-02-11)

- First release on PyPI.
- Add a short introduction.
- Enable travis-ci.org integration.
- Remove numbering of generated meta-classes

2015

- Released on PyPI as a part of plainbox as `plainbox.impl.proxy`

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

D

M

O

P

S