Padme Documentation

Release 1.0

Zygmunt Krynicki

February 16, 2015

Contents

Contents:

CHAPTER 1

Padme - a mostly transparent proxy class for Python

1.1 Features

- Free software: LGPLv3 license
- 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

3.1 `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.

Let's consider a simple example:

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

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):
...     @unproxied
...     def __repr__(self):
...         return super(censor_cat, self).__repr__().replace('cat', '***')
```

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).

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

The `type()` function:

```
>>> type(pets_proxy)
<class 'padme...boundproxy'>
```

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 `unproxied()` decorator. See below for examples.

3.2 Reference

class `padme.`**`proxy`**

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 = ['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] = '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 `@unproxied` decorator on them.

```
>>> import codecs
>>> class crypto(proxy):
...
...     @unproxied
...     def __repr__(self):
...         return codecs.encode(super().__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 = ['ala ma kota', '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']
```

__class__
alias of proxy_meta

__del__()
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.

__init__
Initialize self. See help(type(self)) for accurate signature.

static __new__(*proxy_cls*, *proxiee*)
Create a new instance of proxy() wrapping proxiee

Parameters **proxiee** – The object to proxy

Returns An instance of new subclass of proxy, called boundproxy that uses a new meta-class that lexically bounds the proxiee argument. The new sub-class has a different implementation of __new__ and can be instantiated without additional arguments.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__sizeof__() → int
size of object in memory, in bytes

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

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

padme.**unproxied**(*fn*)
Mark an object (attribute) as not-to-be-proxied.

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

3.3 Internals

class padme.**proxy_meta**
Meta-class for all proxy types

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

`padme.`**`make_boundproxy_meta`**(*proxiee*)

Make a new bound proxy meta-class for the specified object

Parameters **proxiee** – The object that will be proxied

Returns A new meta-class that lexically wraps `proxiee` 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__()

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:

4.1 Types of Contributions

4.1.1 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.

4.1.2 Fix Bugs

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

4.1.3 Implement Features

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

4.1.4 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.

4.1.5 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 :)

4.2 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.

4.3 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.

4.4 Tips

To run a subset of tests:

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

CHAPTER 5

Credits

5.1 Development Lead

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

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 1.0 (2014-02-11)

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

6.2 2015

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

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*

p

Symbols

M

P

U