zope.copy Documentation

Release 4.0

Zope Foundation Contributors

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Using `zope.copy`

Copying persistent objects

This package provides a pluggable way to copy persistent objects. It was once extracted from the zc.copy package to contain much less dependencies. In fact, we only depend on `zope.interface` to provide pluggability.

The package provides a `clone()` function that does the object cloning and the `copy()` wrapper that sets `__parent__` and `__name__` attributes of object's copy to None. This is useful when working with Zope's located objects (see `zope.location` package). The `copy()` function actually calls the `clone()` function, so we'll use the first one in the examples below. We'll also look a bit at their differences in the end of this document.

The `clone()` function (and thus the `copy()` function that wraps it) uses pickling to copy the object and all its subobjects recursively. As each object and subobject is pickled, the function tries to adapt it to `zope.copy.interfaces.ICopyHook`. If a copy hook is found, the recursive copy is halted. The hook is called with two values: the main, top-level object that is being copied; and a callable that supports registering functions to be called after the copy is made. The copy hook should return the exact object or subobject that should be used at this point in the copy, or raise `zope.copy.interfaces.ResumeCopy` exception to resume copying the object or subobject recursively after all.

Note that we use zope's component architecture provided by the `zope.component` package in this document, but the `zope.copy` package itself doesn't use or depend on it, so you can provide another adaptation mechanism as described in `zope.interface`'s adapter documentation.

Simple hooks

First let's examine a simple use. A hook is to support the use case of resetting the state of data that should be changed in a copy – for instance, a log, or freezing or versioning data. The canonical way to do this is by storing the changable data on a special sub-object of the object that is to be copied. We'll look at a simple case of a subobject that should be converted to None when it is copied – the way that the zc.freeze copier hook works. Also see the zc.objectlog copier module for a similar example.

So, here is a simple object that stores a boolean on a special object.

```
# zope.copy.examples.Demo
class Demo(object):

    _frozen = None

    def isFrozen(self):
        return self._frozen is not None

    def freeze(self):
        self._frozen = Data()
```

```
# zope.copy.examples.Data
class Data(object):
    pass
```

Here's what happens if we copy one of these objects without a copy hook.

```
>>> from zope.copy.examples import Demo, Data
>>> original = Demo()
>>> original.isFrozen()
False
>>> original.freeze()
>>> original.isFrozen()
True
>>> import zope.copy
>>> copy = zope.copy.copy(original)
>>> copy is original
False
>>> copy.isFrozen()
True
```

Now let's make a super-simple copy hook that always returns None, no matter what the top-level object being copied is. We'll register it and make another copy.

```
>>> import zope.component
>>> import zope.interface
>>> import zope.copy.interfaces
>>> def _factory(obj, register):
...     return None
>>> @zope.component.adapter(Data)
... @zope.interface.implementer(zope.copy.interfaces.ICopyHook)
... def data_copyfactory(obj):
...     return _factory
...

>>> zope.component.provideAdapter(data_copyfactory)
>>> copy2 = zope.copy.copy(original)
>>> copy2 is original
False
>>> copy2.isFrozen()
False
```

Much better.

Post-copy functions

Now, let's look at the registration function that the hook can use. It is useful for resetting objects within the new copy – for instance, back references such as __parent__ pointers. This is used concretely in the zc.objectlog.copier module; we will come up with a similar but artificial example here.

Imagine an object with a subobject that is "located" (i.e., zope.location) on the parent and should be replaced whenever the main object is copied.

```
# zope.copy.examples.Subobject
class Subobject(zope.location.location.Location):

    def __init__(self):
        self.counter = 0

    def __call__(self):
        res = self.counter
        self.counter += 1
        return res
```

```
>>> import zope.location.location
>>> from zope.copy.examples import Subobject
>>> o = zope.location.location.Location()
>>> s = Subobject()
>>> o.subobject = s
>>> zope.location.location.locate(s, o, 'subobject')
>>> s.__parent__ is o
True
>>> o.subobject()
0
>>> o.subobject()
1
>>> o.subobject()
2
```

Without an ICopyHook, this will simply duplicate the subobject, with correct new pointers.

```
>>> c = zope.copy.copy(o)
>>> c.subobject.__parent__ is c
True
```

Note that the subobject has also copied state.

```
>>> c.subobject()
3
>>> o.subobject()
3
```

Our goal will be to make the counters restart when they are copied. We'll do that with a copy hook.

This copy hook is different: it provides an object to replace the old object, but then it needs to set it up further after the copy is made. This is accomplished by registering a callable, `reparent()` here, that sets up the `__parent__`. The callable is passed a function that can translate something from the original object into the equivalent on the new object. We use this to find the new parent, so we can set it.

```
>>> import zope.component
>>> import zope.interface
```

```
>>> import zope.copy.interfaces
>>> @zope.component.adapter(Subobject)
... @zope.interface.implementer(zope.copy.interfaces.ICopyHook)
... def subobject_copyfactory(original):
...     def factory(obj, register):
...         obj = Subobject()
...         def reparent(translate):
...             obj.__parent__ = translate(original.__parent__)
...         register(reparent)
...         return obj
...     return factory
...
>>> zope.component.provideAdapter(subobject_copyfactory)
```

Now when we copy, the new subobject will have the correct, revised __parent__, but will be otherwise reset (here, just the counter)

```
>>> c = zope.copy.copy(o)
>>> c.subobject.__parent__ is c
True
>>> c.subobject()
0
>>> o.subobject()
4
```

Resuming recursive copy

One thing we didn't examine yet is the use of ResumeCopy exception in the copy hooks. For example, when copying located objects we don't want to copy referenced subobjects that are not located in the object that is being copied. Imagine, we have a content object that has an image object, referenced by the `cover` attribute, but located in an independent place.

```
>>> root = zope.location.location.Location()

>>> content = zope.location.location.Location()
>>> zope.location.location.locate(content, root, 'content')

>>> image = zope.location.location.Location()
>>> zope.location.location.locate(image, root, 'image.jpg')

>>> content.cover = image
```

Without any hooks, the image object will be cloned as well:

```
>>> new = zope.copy.copy(content)
>>> new.cover is image
False
```

That's not what we'd expect though, so, let's provide a copy hook to deal with that. The copy hook for this case is provided by zope.location package, but we'll create one from scratch as we want to check out the usage of the ResumeCopy.

```
>>> @zope.component.adapter(zope.location.interfaces.ILocation)
... @zope.interface.implementer(zope.copy.interfaces.ICopyHook)
... def location_copyfactory(obj):
```

```
...     def factory(location, register):
...         if not zope.location.location.inside(obj, location):
...             return obj
...         raise zope.copy.interfaces.ResumeCopy
...     return factory
...
>>> zope.component.provideAdapter(location_copyfactory)
```

This hook returns objects as they are if they are not located inside object that's being copied, or raises ResumeCopy to signal that the recursive copy should be continued and used for the object.

```
>>> new = zope.copy.copy(content)
>>> new.cover is image
True
```

Much better :-)

clone() vs copy()

As we stated before, there's two functions that is used for copying objects. The `clone()` - that does the job, and its wrapper, `copy()` that calls `clone()` and then clears copy's `__parent__` and `__name__` attribute values.

Let's create a location object with __name__ and __parent__ set.

```
>>> root = zope.location.location.Location()
>>> folder = zope.location.location.Location()
>>> folder.__name__ = 'files'
>>> folder.__parent__ = root
```

The `clone()` function will leave those attributes as is. Note that the referenced __parent__ won't be cloned, as we registered a hook for locations in the previous section.

```
>>> folder_clone = zope.copy.clone(folder)
>>> folder_clone.__parent__ is root
True
>>> folder_clone.__name__ == 'files'
True
```

However, the `copy()` function will reset those attributes to None, as we will probably want to place our object into another container with another name.

```
>>> folder_clone = zope.copy.copy(folder)
>>> folder_clone.__parent__ is None
True
>>> folder_clone.__name__ is None
True
```

Notice, that if your object doesn't have __parent__ and __name__ attributes at all, or these attributes could'nt be got or set because of some protections (as with zope.security's proxies, for example), you still can use the `copy()` function, because it works for objects that don't have those attributes.

It won't set them if original object doesn't have them:

```
# zope.copy.examples.Something
class Something(object):
    pass
```

```
>>> from zope.copy.examples import Something
>>> s = Something()
>>> s_copy = zope.copy.copy(s)
>>> s_copy.__parent__
Traceback (most recent call last):
...
AttributeError: ...
>>> s_copy.__name__
Traceback (most recent call last):
...
AttributeError: ...
```

And it won't fail if original object has them but doesn't allow to set them.

```
# zope.copy.examples.Other
class Other(object):
    root = object()  # immutable
    __name__ = property(lambda _self: 'something')
    __parent__ = property(lambda self: self.__class__.root)
```

```
>>> from zope.copy.examples import Other
>>> s = Other()
>>> s_copy = zope.copy.copy(s)
>>> s_copy.__parent__ is Other.root
True
>>> s_copy.__name__ == 'something'
True
```

LocationCopyHook

The location copy hook is defined in `zope.location` but only activated if this package is installed.

It's job is to allow copying referenced objects that are not located inside object that's being copied.

To see the problem, imagine we want to copy an `ILocation` object that contains an attribute-based reference to another ILocation object and the referenced object is not contained inside object being copied.

Without this hook, the referenced object will be cloned:

```
>>> from zope.component.globalregistry import base
>>> base.__init__('base') # blow away previous registrations
>>> from zope.location.location import Location, locate
>>> root = Location()
>>> page = Location()
>>> locate(page, root, 'page')
>>> image = Location()
>>> locate(page, root, 'image')
>>> page.thumbnail = image

>>> from zope.copy import copy
>>> page_copy = copy(page)
>>> page_copy.thumbnail is image
False
```

But if we will provide a hook, the attribute will point to the original object as we might want.

```
>>> from zope.component import provideAdapter
>>> from zope.location.pickling import LocationCopyHook
>>> from zope.location.interfaces import ILocation
>>> provideAdapter(LocationCopyHook, (ILocation,))

>>> from zope.copy import copy
>>> page_copy = copy(page)
>>> page_copy.thumbnail is image
True
```

CHAPTER 2

zope.copy API Reference

zope.copy.interfaces

CHAPTER 3

Hacking on `zope.copy`

Getting the Code

The main repository for `zope.copy` is in the Zope Foundation Github repository:

https://github.com/zopefoundation/zope.copy

You can get a read-only checkout from there:

```
$ git clone https://github.com/zopefoundation/zope.copy.git
```

or fork it and get a writeable checkout of your fork:

```
$ git clone git@github.com/jrandom/zope.copy.git
```

The project also mirrors the trunk from the Github repository as a Bazaar branch on Launchpad:

https://code.launchpad.net/zope.copy

You can branch the trunk from there using Bazaar:

```
$ bzr branch lp:zope.copy
```

Working in a `virtualenv`

Installing

If you use the `virtualenv` package to create lightweight Python development environments, you can run the tests using nothing more than the `python` binary in a virtualenv. First, create a scratch environment:

```
$ /path/to/virtualenv --no-site-packages /tmp/hack-zope.copy
```

Next, get this package registered as a "development egg" in the environment:

```
$ /tmp/hack-zope.copy/bin/python setup.py develop
```

Running the tests

Then, you canrun the tests using the build-in `setuptools` testrunner:

```
$ /tmp/hack-zope.copy/bin/python setup.py test -q
..........
----------------------------------------------------------------------
Ran 11 tests in 0.000s

OK
```

If you have the `nose` package installed in the virtualenv, you can use its testrunner too:

```
$ /tmp/hack-zope.copy/bin/nosetests
............
----------------------------------------------------------------------
Ran 12 tests in 0.011s

OK
```

If you have the `coverage` pacakge installed in the virtualenv, you can see how well the tests cover the code:

```
$ .tox/coverage/bin/nosetests --with-coverage
............
Name                    Stmts   Miss  Cover   Missing
-----------------------------------------------------
zope.copy                  59      0   100%
zope.copy._compat           6      0   100%
zope.copy.examples          4      0   100%
zope.copy.interfaces        5      0   100%
-----------------------------------------------------
TOTAL                      74      0   100%
----------------------------------------------------------------------
Ran 12 tests in 0.062s

OK
```

Building the documentation

`zope.copy` uses the nifty `Sphinx` documentation system for building its docs. Using the same virtualenv you set up to run the tests, you can build the docs:

```
$ /tmp/hack-zope.copy/bin/easy_install Sphinx
...
$ cd docs
$ PATH=/tmp/hack-zope.copy/bin:$PATH make html
sphinx-build -b html -d _build/doctrees   . _build/html
...
build succeeded.

Build finished. The HTML pages are in _build/html.
```

You can also test the code snippets in the documentation:

```
$ PATH=/tmp/hack-zope.copy/bin:$PATH make doctest
sphinx-build -b doctest -d _build/doctrees   . _build/doctest
...
running tests...

Document: narr
--------------
1 items passed all tests:
  93 tests in default
93 tests in 1 items.
93 passed and 0 failed.
Test passed.

Doctest summary
===============
   93 tests
    0 failures in tests
    0 failures in setup code
    0 failures in cleanup code
build succeeded.
Testing of doctests in the sources finished, look at the  \
    results in _build/doctest/output.txt.
```

Using `zc.buildout`

Setting up the buildout

`zope.copy` ships with its own `buildout.cfg` file and `bootstrap.py` for setting up a development buildout:

```
$ /path/to/python2.6 bootstrap.py
...
Generated script '.../bin/buildout'
$ bin/buildout
Develop: '/home/jrandom/projects/Zope/zope.event/.'
...
```

Running the tests

You can now run the tests:

```
$ bin/test --all
Running zope.testing.testrunner.layer.UnitTests tests:
  Set up zope.testing.testrunner.layer.UnitTests in 0.000 seconds.
  Ran 2 tests with 0 failures and 0 errors in 0.000 seconds.
Tearing down left over layers:
  Tear down zope.testing.testrunner.layer.UnitTests in 0.000 seconds.
```

Using tox

Running Tests on Multiple Python Versions

tox is a Python-based test automation tool designed to run tests against multiple Python versions. It creates a virtualenv for each configured version, installs the current package and configured dependencies into each virtualenv, and then runs the configured commands.

zope.copy configures the following tox environments via its tox.ini file:

- The py26, py27, py32, py33, py34, and pypy environments build a virtualenv with the appropriate interpreter, installs zope.copy and dependencies, and runs the tests via python setup.py test -q.
- The coverage environment builds a virtualenv with python2.6, installs zope.copy, installs nose and coverage, and runs nosetests with statement coverage.
- The docs environment builds a virtualenv with python2.6, installs zope.copy, installs Sphinx and dependencies, and then builds the docs and exercises the doctest snippets.

This example requires that you have a working python2.6 on your path, as well as installing tox:

```
$ tox -e py26
GLOB sdist-make: .../zope.copy/setup.py
py26 sdist-reinst: .../zope.copy/.tox/dist/zope.copy-4.0.2dev.zip
py26 runtests: commands[0]
..........
----------------------------------------------------------------------
Ran 11 tests in 0.000s

OK
_____________________________________ summary ______________________________________
py26: commands succeeded
congratulations :)
```

Running tox with no arguments runs all the configured environments, including building the docs and testing their snippets:

```
$ tox
GLOB sdist-make: .../zope.copy/setup.py
py26 sdist-reinst: .../zope.copy/.tox/dist/zope.copy-4.0.2dev.zip
py26 runtests: commands[0]
...
Doctest summary
===============
 93 tests
   0 failures in tests
   0 failures in setup code
   0 failures in cleanup code
build succeeded.
_____________________________________ summary ______________________________________
py26: commands succeeded
py27: commands succeeded
py32: commands succeeded
py33: commands succeeded
py34: commands succeeded
pypy: commands succeeded
pypy3: commands succeeded
coverage: commands succeeded
```

```
docs: commands succeeded
congratulations :)
```

Contributing to zope.copy

Submitting a Bug Report

zope.copy tracks its bugs on Github:

https://github.com/zopefoundation/zope.copy/issues

Please submit bug reports and feature requests there.

Sharing Your Changes

Note: Please ensure that all tests are passing before you submit your code. If possible, your submission should include new tests for new features or bug fixes, although it is possible that you may have tested your new code by updating existing tests.

If have made a change you would like to share, the best route is to fork the Githb repository, check out your fork, make your changes on a branch in your fork, and push it. You can then submit a pull request from your branch:

https://github.com/zopefoundation/zope.copy/pulls

If you branched the code from Launchpad using Bazaar, you have another option: you can "push" your branch to Launchpad:

```
$ bzr push lp:~jrandom/zope.copy/cool_feature
```

After pushing your branch, you can link it to a bug report on Github, or request that the maintainers merge your branch using the Launchpad "merge request" feature.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search