# zope.exceptions Documentation

*Release 4.0*

**Zope Foundation contributors.**

**Sep 27, 2017**

# Contents

Contents:

CHAPTER 1

# Using `zope.exceptions`

This module extends the standard library's `traceback` module, allowing application code to add additional information to the formatted tracebacks using specially-named variables in the scope of a given frame.

We will use examples of a rendering function that's meant to produce some output given a template and a set of options. But this rendering function is quite broken and produces a useless exception, so when we call it we'd like to be able to provide some more contextual information.

```
>>> import sys
>>> def render(source, options):
...     raise Exception("Failed to render")
```

## Annotating Application Code

### `__traceback_info__`

This variable can only be defined at local scope. It will be converted to a string when added to the formatted traceback.

```
>>> def render_w_info(template_file, options):
...     with open(template_file) as f:
...         source = f.read()
...     __traceback_info__ = '%s\n\n%s' % (template_file, source)
...     render(source, options)
```

This is convenient for quickly adding context information in an unstructured way, especially if you already have a string, or an object with a custom `__str__` or `__repr__` that provides the information you need (tuples of multiple such items also work well). However, if you need to format a string to produce readable information, as in the example above, this may have an undesirable runtime cost because it is calculated even when no traceback is formatted. For such cases, `__traceback_supplement__` may be helpful.

### `__traceback_supplement__`

This variable can be defined either at either local or global (module) scope. Unlike `__traceback__info__` this is structured data. It must consist of a sequence containing a function and the arguments to pass to that function. At runtime, only if a traceback needs to be formatted will the function be called, with the arguments, to produce a *supplement object*. Because the construction of the object is delayed until needed, this can be a less expensive way to produce lots of useful information with minimal runtime overhead.

The formatting functions treat the resulting supplement object as if it supports the `ITracebackSupplement` interface. The various attributes (all optional) of that interface will be used to add structured information to the formatted traceback.

For example, assuming your code renders a template:

```
>>> import os
>>> class Supplement(object):
...     def __init__(self, template_file, options):
...         self.source_url = 'file://%s' % os.path.abspath(template_file)
...         self.options = options
...         self.expression = 'an expression'
...     def getInfo(self):
...         return "Options: " + str(self.options)
>>> def render_w_supplement(template_file, options):
...     with open(template_file) as f:
...         source = f.read()
...     __traceback_supplement__ = Supplement, template_file, options
...     render(source, options)
```

Here, the filename and options of the template will be rendered as part of the traceback.

---

**Note:** If there is an exception calling the constructor function, no supplement will be formatted, and (by default) the exception will be printed on `sys.stderr`.

---

## API Functions

Three API functions support these features when formatting Python exceptions and their associated tracebacks:

### `format_exception()`

Use this API to format an exception and traceback as a list of strings, using the special annotations. E.g.:

```
>>> from zope.exceptions import format_exception
>>> try:
...     render_w_info('docs/narr.rst', {})
... except:
...     t, v, tb = sys.exc_info()
...     report = format_exception(t, v, tb)
...     del tb # avoid a leak
...     # Now do something with report, e.g., send e-mail.
>>> print('\n'.join(report))
Traceback (most recent call last):

  Module <doctest default[1]>, line 2, in <module>
```

```
    render_w_info('docs/narr.rst', {})

  Module <doctest default[0]>, line 5, in render_w_info
   - __traceback_info__: docs/narr.rst
...
```

### print_exception()

Use this API to write the formated exception and traceback to a file-like object, using the special annotations. E.g.:

```
>>> from zope.exceptions import print_exception
>>> try:
...     render_w_supplement('docs/narr.rst', {})
... except:
...     t, v, tb = sys.exc_info()
...     print_exception(t, v, tb, file=sys.stdout)
...     del tb # avoid a leak
Traceback (most recent call last):
  File "<doctest default[1]>", line 2, in <module>
    render_w_supplement('docs/narr.rst', {})
  File "<doctest default[2]>", line 5, in render_w_supplement
   - file:///...
   - Expression: an expression
Options: {}
  File "<doctest default[1]>", line 2, in render
    render_w_supplement('docs/narr.rst', {})
Exception: Failed to render
```

### extract_stack()

Use this API to format just the traceback as a list of string,s using the special annotations. E.g.:

```
>>> import sys
>>> from zope.exceptions import extract_stack
>>> try:
...     raise ValueError('demo')
... except:
...     for line in extract_stack(sys.exc_info()[2].tb_frame):
...         pass # do something with each line
```

CHAPTER 2

# zope.exceptions API documentation

## zope.exceptions.interfaces

### ITracebackSupplement

## zope.exceptions.exceptionformatter

### format_exception()

### print_exception()

### extract_stack()

CHAPTER 3

# Hacking on `zope.exceptions`

## Getting the Code

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

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

You can get a read-only checkout from there:

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

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

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

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

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

You can branch the trunk from there using Bazaar:

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

## 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.exceptions
```

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

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

## Running the tests

Run the tests using the build-in `setuptools` testrunner:

```
$ /tmp/hack-zope.exceptions/bin/python setup.py test -q
running test
...------------------------------------------------------------------------
----------------------------------------------------------------------
Ran 72 tests in 0.017s

OK
```

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

```
$ /tmp/hack-zope.exceptions/bin/easy_install nose
...
$ /tmp/hack-zope.exceptions/bin/python setup.py nosetests
running nosetests
.....................................................................
----------------------------------------------------------------------
Ran 73 tests in 0.010s

OK
```

or:

```
$ /tmp/hack-zope.exceptions/bin/nosetests
.....................................................................
----------------------------------------------------------------------
Ran 73 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:

```
$ /tmp/hack-zope.exceptions/bin/easy_install nose coverage
...
$ /tmp/hack-zope.exceptions/bin/python setup.py nosetests \
    --with coverage --cover-package=zope.exceptions
running nosetests
...
.....................................................................
Name                                 Stmts   Miss  Cover   Missing
------------------------------------------------------------------
zope.exceptions                         10      0   100%
zope.exceptions.exceptionformatter     171      0   100%
zope.exceptions.interfaces              18      0   100%
zope.exceptions.log                     13      0   100%
------------------------------------------------------------------
TOTAL                                  212      0   100%
----------------------------------------------------------------------

OK
```

### Building the documentation

`zope.exceptions` 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.exceptions/bin/easy_install Sphinx
...
$ cd docs
$ PATH=/tmp/hack-zope.exceptions/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:

```
$ bin/sphinx-build -b docs/doctest -d docs/_build/doctrees docs docs/_build/doctest
...
12 tests in 1 items.
12 passed and 0 failed.
Test passed.

Doctest summary
===============
   12 tests
    0 failures in tests
    0 failures in setup 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.exceptions` 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/BTK/exceptions/.'
...
Generated script '.../bin/sphinx-quickstart'.
Generated script '.../bin/sphinx-build'.
```

### Running the tests

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.exceptions` configures the following `tox` environments via its `tox.ini` file:

- The `py26`, `py27`, `py33`, `py34`, and `pypy` environments builds a `virtualenv` with `pypy`, installs `zope.exceptions` and dependencies, and runs the tests via `python setup.py test -q`.
- The `coverage` environment builds a `virtualenv` with `python2.6`, installs `zope.exceptions`, installs `nose` and `coverage`, and runs `nosetests` with statement coverage.
- The `docs` environment builds a virtualenv with `python2.6`, installs `zope.exceptions`, 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.interface/setup.py
py26 sdist-reinst: .../zope.interface/.tox/dist/zope.interface-4.0.2dev.zip
py26 runtests: commands[0]
...----------------------------------------------------------------------
----------------------------------------------------------------------
Ran 72 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.interface/setup.py
py26 sdist-reinst: .../zope.interface/.tox/dist/zope.interface-4.0.2dev.zip
py26 runtests: commands[0]
...
Doctest summary
===============
 12 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
pypy: commands succeeded
coverage: commands succeeded
docs: commands succeeded
congratulations :)
```

# Contributing to `zope.exceptions`

## Submitting a Bug Report

`zope.exceptions` tracks its bugs on Github:

https://github.com/zopefoundation/zope.exceptions/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.exceptions/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.exceptions/cool_feature
```

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

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search