tblib

Release 3.0.0

Ionel Cristian Mărieș

Oct 22, 2023

CONTENTS

CHAPTER

ONE

OVERVIEW

docs
tests
package

Serialization library for Exceptions and Tracebacks.

- Free software: BSD license

It allows you to:

- Pickle tracebacks and raise exceptions with pickled tracebacks in different processes. This allows better error handling when running code over multiple processes (imagine multiprocessing, billiard, futures, celery etc).
- Create traceback objects from strings (the `from_string` method). *No pickling is used.*
- Serialize tracebacks to/from plain dicts (the `from_dict` and `to_dict` methods). *No pickling is used.*
- Raise the tracebacks created from the aforementioned sources.
- Pickle an Exception together with its traceback and exception chain (`raise ... from ...`) *(Python 3 only)*

Again, note that using the pickle support is completely optional. You are solely responsible for security problems should you decide to use the pickle support.

1.1 Installation

```
pip install tblib
```

1.2 Documentation

1.2.1 Pickling tracebacks

Note: The traceback objects that come out are stripped of some attributes (like variables). But you'll be able to raise exceptions with those tracebacks or print them - that should cover 99% of the usecases.

```
>>> from tblib import pickling_support
>>> pickling_support.install()
>>> import pickle, sys
>>> def inner_0():
...     raise Exception('fail')
...
>>> def inner_1():
...     inner_0()
...
>>> def inner_2():
...     inner_1()
...
>>> try:
...     inner_2()
... except:
...     s1 = pickle.dumps(sys.exc_info())
...
>>> len(s1) > 1
True
>>> try:
...     inner_2()
... except:
```

(continues on next page)

(continued from previous page)

```
...     s2 = pickle.dumps(sys.exc_info(), protocol=pickle.HIGHEST_PROTOCOL)
...
>>> len(s2) > 1
True

>>> try:
...     import cPickle
... except ImportError:
...     import pickle as cPickle
>>> try:
...     inner_2()
... except:
...     s3 = cPickle.dumps(sys.exc_info(), protocol=pickle.HIGHEST_PROTOCOL)
...
>>> len(s3) > 1
True
```

1.2.2 Unpickling tracebacks

```
>>> pickle.loads(s1)
(<...Exception'>, Exception('fail'...), <traceback object at ...>)

>>> pickle.loads(s2)
(<...Exception'>, Exception('fail'...), <traceback object at ...>)

>>> pickle.loads(s3)
(<...Exception'>, Exception('fail'...), <traceback object at ...>)
```

1.2.3 Raising

```
>>> from six import reraise
>>> reraise(*pickle.loads(s1))
Traceback (most recent call last):
  ...
  File "<doctest README.rst[14]>", line 1, in <module>
    reraise(*pickle.loads(s2))
  File "<doctest README.rst[8]>", line 2, in <module>
    inner_2()
  File "<doctest README.rst[5]>", line 2, in inner_2
    inner_1()
  File "<doctest README.rst[4]>", line 2, in inner_1
    inner_0()
  File "<doctest README.rst[3]>", line 2, in inner_0
    raise Exception('fail')
Exception: fail
>>> reraise(*pickle.loads(s2))
Traceback (most recent call last):
  ...
  File "<doctest README.rst[14]>", line 1, in <module>
```

(continues on next page)

(continued from previous page)

```
    reraise(*pickle.loads(s2))
  File "<doctest README.rst[8]>", line 2, in <module>
    inner_2()
  File "<doctest README.rst[5]>", line 2, in inner_2
    inner_1()
  File "<doctest README.rst[4]>", line 2, in inner_1
    inner_0()
  File "<doctest README.rst[3]>", line 2, in inner_0
    raise Exception('fail')
Exception: fail
>>> reraise(*pickle.loads(s3))
Traceback (most recent call last):
  ...
  File "<doctest README.rst[14]>", line 1, in <module>
    reraise(*pickle.loads(s2))
  File "<doctest README.rst[8]>", line 2, in <module>
    inner_2()
  File "<doctest README.rst[5]>", line 2, in inner_2
    inner_1()
  File "<doctest README.rst[4]>", line 2, in inner_1
    inner_0()
  File "<doctest README.rst[3]>", line 2, in inner_0
    raise Exception('fail')
Exception: fail
```

1.2.4 Pickling Exceptions together with their traceback and chain (Python 3 only)

```
>>> try:
...     try:
...         1 / 0
...     except Exception as e:
...         raise Exception("foo") from e
... except Exception as e:
...     s = pickle.dumps(e)
>>> raise pickle.loads(s)
Traceback (most recent call last):
  File "<doctest README.rst[16]>", line 3, in <module>
    1 / 0
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<doctest README.rst[17]>", line 1, in <module>
    raise pickle.loads(s)
  File "<doctest README.rst[16]>", line 5, in <module>
    raise Exception("foo") from e
Exception: foo
```

BaseException subclasses defined after calling `pickling_support.install()` will **not** retain their traceback and exception chain pickling. To cover custom Exceptions, there are three options:

1. Use `@pickling_support.install` as a decorator for each custom Exception

```
>>> from tblib import pickling_support
>>> # Declare all imports of your package's dependencies
>>> import numpy

>>> pickling_support.install()  # install for all modules imported so far

>>> @pickling_support.install
... class CustomError(Exception):
...     pass
```

   Eventual subclasses of `CustomError` will need to be decorated again.

2. Invoke `pickling_support.install()` after all modules have been imported and all Exception subclasses have been declared

```
>>> # Declare all imports of your package's dependencies
>>> import numpy
>>> from tblib import pickling_support

>>> # Declare your own custom Exceptions
>>> class CustomError(Exception):
...     pass

>>> # Finally, install tblib
>>> pickling_support.install()
```

3. Selectively install tblib for Exception instances just before they are pickled

```
pickling_support.install(<Exception instance>, [Exception instance], ...)
```

   The above will install tblib pickling for all listed exceptions as well as any other exceptions in their exception chains.

   For example, one could write a wrapper to be used with ProcessPoolExecutor, Dask.distributed, or similar libraries:

```
>>> from tblib import pickling_support
>>> def wrapper(func, *args, **kwargs):
...     try:
...         return func(*args, **kwargs)
...     except Exception as e:
...         pickling_support.install(e)
...         raise
```

What if we have a local stack, does it show correctly ?

Yes it does:

```
>>> exc_info = pickle.loads(s3)
>>> def local_0():
...     reraise(*exc_info)
...
>>> def local_1():
...     local_0()
...
>>> def local_2():
...     local_1()
...
>>> local_2()
Traceback (most recent call last):
  File "...doctest.py", line ..., in __run
    compileflags, 1) in test.globs
  File "<doctest README.rst[24]>", line 1, in <module>
    local_2()
  File "<doctest README.rst[23]>", line 2, in local_2
    local_1()
  File "<doctest README.rst[22]>", line 2, in local_1
    local_0()
  File "<doctest README.rst[21]>", line 2, in local_0
    reraise(*exc_info)
  File "<doctest README.rst[11]>", line 2, in <module>
    inner_2()
  File "<doctest README.rst[5]>", line 2, in inner_2
    inner_1()
  File "<doctest README.rst[4]>", line 2, in inner_1
    inner_0()
  File "<doctest README.rst[3]>", line 2, in inner_0
    raise Exception('fail')
Exception: fail
```

It also supports more contrived scenarios

Like tracebacks with syntax errors:

```
>>> from tblib import Traceback
>>> from examples import bad_syntax
>>> try:
...     bad_syntax()
... except:
...     et, ev, tb = sys.exc_info()
...     tb = Traceback(tb)
...
>>> reraise(et, ev, tb.as_traceback())
Traceback (most recent call last):
  ...
  File "<doctest README.rst[58]>", line 1, in <module>
```

(continues on next page)

(continued from previous page)

```
    reraise(et, ev, tb.as_traceback())
  File "<doctest README.rst[57]>", line 2, in <module>
    bad_syntax()
  File "...tests...examples.py", line 18, in bad_syntax
    import badsyntax
  File "...tests...badsyntax.py", line 5
    is very bad
     ^
SyntaxError: invalid syntax
```

Or other import failures:

```
>>> from examples import bad_module
>>> try:
...     bad_module()
... except:
...     et, ev, tb = sys.exc_info()
...     tb = Traceback(tb)
...
>>> reraise(et, ev, tb.as_traceback())
Traceback (most recent call last):
  ...
  File "<doctest README.rst[61]>", line 1, in <module>
    reraise(et, ev, tb.as_traceback())
  File "<doctest README.rst[60]>", line 2, in <module>
    bad_module()
  File "...tests...examples.py", line 23, in bad_module
    import badmodule
  File "...tests...badmodule.py", line 3, in <module>
    raise Exception("boom!")
Exception: boom!
```

Or a traceback that's caused by exceeding the recursion limit (here we're forcing the type and value to have consistency across platforms):

```
>>> def f(): f()
>>> try:
...    f()
... except RuntimeError:
...    et, ev, tb = sys.exc_info()
...    tb = Traceback(tb)
...
>>> reraise(RuntimeError, RuntimeError("maximum recursion depth exceeded"), tb.as_
↪traceback())
Traceback (most recent call last):
  ...
  File "<doctest README.rst[32]>", line 1, in f
    def f(): f()
  File "<doctest README.rst[32]>", line 1, in f
    def f(): f()
  File "<doctest README.rst[32]>", line 1, in f
    def f(): f()
```

(continues on next page)

(continued from previous page)

```
  ...
RuntimeError: maximum recursion depth exceeded
```

1.2.5 Reference

tblib.Traceback

It is used by the `pickling_support`. You can use it too if you want more flexibility:

```
>>> from tblib import Traceback
>>> try:
...     inner_2()
... except:
...     et, ev, tb = sys.exc_info()
...     tb = Traceback(tb)
...
>>> reraise(et, ev, tb.as_traceback())
Traceback (most recent call last):
  ...
  File "<doctest README.rst[21]>", line 6, in <module>
    reraise(et, ev, tb.as_traceback())
  File "<doctest README.rst[21]>", line 2, in <module>
    inner_2()
  File "<doctest README.rst[5]>", line 2, in inner_2
    inner_1()
  File "<doctest README.rst[4]>", line 2, in inner_1
    inner_0()
  File "<doctest README.rst[3]>", line 2, in inner_0
    raise Exception('fail')
Exception: fail
```

tblib.Traceback.to_dict

You can use the `to_dict` method and the `from_dict` classmethod to convert a Traceback into and from a dictionary serializable by the stdlib json.JSONDecoder:

```
>>> import json
>>> from pprint import pprint
>>> try:
...     inner_2()
... except:
...     et, ev, tb = sys.exc_info()
...     tb = Traceback(tb)
...     tb_dict = tb.to_dict()
...     pprint(tb_dict)
{'tb_frame': {'f_code': {'co_filename': '<doctest README.rst[...]>',
                         'co_name': '<module>'},
              'f_globals': {'__name__': '__main__'},
              'f_lineno': 5},
```

(continues on next page)

(continued from previous page)

```
 'tb_lineno': 2,
 'tb_next': {'tb_frame': {'f_code': {'co_filename': ...,
                                     'co_name': 'inner_2'},
                          'f_globals': {'__name__': '__main__'},
                          'f_lineno': 2},
             'tb_lineno': 2,
             'tb_next': {'tb_frame': {'f_code': {'co_filename': ...,
                                                 'co_name': 'inner_1'},
                                      'f_globals': {'__name__': '__main__'},
                                      'f_lineno': 2},
                         'tb_lineno': 2,
                         'tb_next': {'tb_frame': {'f_code': {'co_filename': ...,
                                                             'co_name': 'inner_0'},
                                                  'f_globals': {'__name__': '__main__'},
                                                  'f_lineno': 2},
                                     'tb_lineno': 2,
                                     'tb_next': None}}}}
```

tblib.Traceback.from_dict

Building on the previous example:

```
>>> tb_json = json.dumps(tb_dict)
>>> tb = Traceback.from_dict(json.loads(tb_json))
>>> reraise(et, ev, tb.as_traceback())
Traceback (most recent call last):
  ...
  File "<doctest README.rst[21]>", line 6, in <module>
    reraise(et, ev, tb.as_traceback())
  File "<doctest README.rst[21]>", line 2, in <module>
    inner_2()
  File "<doctest README.rst[5]>", line 2, in inner_2
    inner_1()
  File "<doctest README.rst[4]>", line 2, in inner_1
    inner_0()
  File "<doctest README.rst[3]>", line 2, in inner_0
    raise Exception('fail')
Exception: fail
```

tblib.Traceback.from_string

```
>>> tb = Traceback.from_string("""
... File "skipped.py", line 123, in func_123
... Traceback (most recent call last):
...   File "tests/examples.py", line 2, in func_a
...     func_b()
...   File "tests/examples.py", line 6, in func_b
...     func_c()
...   File "tests/examples.py", line 10, in func_c
```

(continues on next page)

(continued from previous page)

```
...     func_d()
...   File "tests/examples.py", line 14, in func_d
... Doesn't: matter
... """)
>>> reraise(et, ev, tb.as_traceback())
Traceback (most recent call last):
  ...
  File "<doctest README.rst[42]>", line 6, in <module>
    reraise(et, ev, tb.as_traceback())
  File "...examples.py", line 2, in func_a
    func_b()
  File "...examples.py", line 6, in func_b
    func_c()
  File "...examples.py", line 10, in func_c
    func_d()
  File "...examples.py", line 14, in func_d
    raise Exception("Guessing time !")
Exception: fail
```

If you use the `strict=False` option then parsing is a bit more lax:

```
>>> tb = Traceback.from_string("""
... File "bogus.py", line 123, in bogus
... Traceback (most recent call last):
...  File "tests/examples.py", line 2, in func_a
...   func_b()
...    File "tests/examples.py", line 6, in func_b
...     func_c()
...    File "tests/examples.py", line 10, in func_c
...   func_d()
...  File "tests/examples.py", line 14, in func_d
... Doesn't: matter
... """, strict=False)
>>> reraise(et, ev, tb.as_traceback())
Traceback (most recent call last):
  ...
  File "<doctest README.rst[42]>", line 6, in <module>
    reraise(et, ev, tb.as_traceback())
  File "bogus.py", line 123, in bogus
  File "...examples.py", line 2, in func_a
    func_b()
  File "...examples.py", line 6, in func_b
    func_c()
  File "...examples.py", line 10, in func_c
    func_d()
  File "...examples.py", line 14, in func_d
    raise Exception("Guessing time !")
Exception: fail
```

tblib.decorators.return_error

```
>>> from tblib.decorators import return_error
>>> inner_2r = return_error(inner_2)
>>> e = inner_2r()
>>> e
<tblib.decorators.Error object at ...>
>>> e.reraise()
Traceback (most recent call last):
  ...
  File "<doctest README.rst[26]>", line 1, in <module>
    e.reraise()
  File "...tblib...decorators.py", line 19, in reraise
    reraise(self.exc_type, self.exc_value, self.traceback)
  File "...tblib...decorators.py", line 25, in return_exceptions_wrapper
    return func(*args, **kwargs)
  File "<doctest README.rst[5]>", line 2, in inner_2
    inner_1()
  File "<doctest README.rst[4]>", line 2, in inner_1
    inner_0()
  File "<doctest README.rst[3]>", line 2, in inner_0
    raise Exception('fail')
Exception: fail
```

How's this useful? Imagine you're using multiprocessing like this:

```
# Note that Python 3.4 and later will show the remote traceback (but as a string sadly)␣
↪so we skip testing this.
>>> import traceback
>>> from multiprocessing import Pool
>>> from examples import func_a
>>> pool = Pool()  # doctest: +SKIP
>>> try:  # doctest: +SKIP
...     for i in pool.map(func_a, range(5)):
...         print(i)
... except:
...     print(traceback.format_exc())
...
Traceback (most recent call last):
  File "<doctest README.rst[...]>", line 2, in <module>
    for i in pool.map(func_a, range(5)):
  File "...multiprocessing...pool.py", line ..., in map
    ...
  File "...multiprocessing...pool.py", line ..., in get
    ...
Exception: Guessing time !
<BLANKLINE>
>>> pool.terminate()  # doctest: +SKIP
```

Not very useful is it? Let's sort this out:

```
>>> from tblib.decorators import apply_with_return_error, Error
>>> from itertools import repeat
```

(continues on next page)

(continued from previous page)

```
>>> pool = Pool()
>>> try:
...     for i in pool.map(apply_with_return_error, zip(repeat(func_a), range(5))):
...         if isinstance(i, Error):
...             i.reraise()
...         else:
...             print(i)
... except:
...     print(traceback.format_exc())
...
Traceback (most recent call last):
  File "<doctest README.rst[...]>", line 4, in <module>
    i.reraise()
  File "...tblib...decorators.py", line ..., in reraise
    reraise(self.exc_type, self.exc_value, self.traceback)
  File "...tblib...decorators.py", line ..., in return_exceptions_wrapper
    return func(*args, **kwargs)
  File "...tblib...decorators.py", line ..., in apply_with_return_error
    return args[0](*args[1:])
  File "...examples.py", line 2, in func_a
    func_b()
  File "...examples.py", line 6, in func_b
    func_c()
  File "...examples.py", line 10, in func_c
    func_d()
  File "...examples.py", line 14, in func_d
    raise Exception("Guessing time !")
Exception: Guessing time !

>>> pool.terminate()
```

Much better !

What if we have a local call stack ?

```
>>> def local_0():
...     pool = Pool()
...     try:
...         for i in pool.map(apply_with_return_error, zip(repeat(func_a), range(5))):
...             if isinstance(i, Error):
...                 i.reraise()
...             else:
...                 print(i)
...     finally:
...         pool.close()
...
>>> def local_1():
...     local_0()
...
>>> def local_2():
...     local_1()
```

(continues on next page)

(continued from previous page)

```
...
>>> try:
...     local_2()
... except:
...     print(traceback.format_exc())
Traceback (most recent call last):
  File "<doctest README.rst[...]>", line 2, in <module>
    local_2()
  File "<doctest README.rst[...]>", line 2, in local_2
    local_1()
  File "<doctest README.rst[...]>", line 2, in local_1
    local_0()
  File "<doctest README.rst[...]>", line 6, in local_0
    i.reraise()
  File "...tblib...decorators.py", line 20, in reraise
    reraise(self.exc_type, self.exc_value, self.traceback)
  File "...tblib...decorators.py", line 27, in return_exceptions_wrapper
    return func(*args, **kwargs)
  File "...tblib...decorators.py", line 47, in apply_with_return_error
    return args[0](*args[1:])
  File "...tests...examples.py", line 2, in func_a
    func_b()
  File "...tests...examples.py", line 6, in func_b
    func_c()
  File "...tests...examples.py", line 10, in func_c
    func_d()
  File "...tests...examples.py", line 14, in func_d
    raise Exception("Guessing time !")
Exception: Guessing time !
```

Other weird stuff

Clearing traceback works (Python 3.4 and up):

```
>>> tb = Traceback.from_string("""
... File "skipped.py", line 123, in func_123
... Traceback (most recent call last):
...   File "tests/examples.py", line 2, in func_a
...     func_b()
...   File "tests/examples.py", line 6, in func_b
...     func_c()
...   File "tests/examples.py", line 10, in func_c
...     func_d()
...   File "tests/examples.py", line 14, in func_d
... Doesn't: matter
... """)
>>> import traceback, sys
>>> if sys.version_info > (3, 4):
...     traceback.clear_frames(tb)
```

1.3 Credits

- mitsuhiko/jinja2 for figuring a way to create traceback objects.

CHAPTER

TWO

INSTALLATION

At the command line:

```
pip install tblib
```

CHAPTER

THREE

USAGE

To use tblib in a project:

```
import tblib
```

CHAPTER

FOUR

CONTRIBUTING

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

4.1 Bug reports

When 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.

4.2 Documentation improvements

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

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/ionelmc/python-tblib/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 code contributions are welcome :)

4.4 Development

To set up *python-tblib* for local development:

1. Fork python-tblib (look for the "Fork" button).
2. Clone your fork locally:

   ```
   git clone git@github.com:YOURGITHUBNAME/python-tblib.git
   ```

3. Create a branch for local development:

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

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with one command:

   ```
   tox
   ```

5. 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
   ```

6. Submit a pull request through the GitHub website.

4.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`).
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

4.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

CHAPTER

FIVE

API REFERENCE

This page contains auto-generated API reference documentation[1].

5.1 tblib

5.1.1 Submodules

tblib.decorators

Module Contents

Classes

Error	

Functions

return_error(func[, exc_type])	
apply_with_return_error(args)	args is a tuple where the first argument is a callable.

class tblib.decorators.**Error**(*exc_type*, *exc_value*, *traceback*)

property traceback

reraise()

tblib.decorators.**return_error**(*func*, *exc_type=Exception*)

tblib.decorators.**apply_with_return_error**(*args*)

args is a tuple where the first argument is a callable.

eg:

[1] Created with sphinx-autoapi

```
apply_with_return_error((func, 1, 2, 3)) - this will call func(1, 2, 3)
```

tblib.pickling_support

Module Contents

Functions

unpickle_traceback(tb_frame, tb_lineno, tb_next)	
pickle_traceback(tb, *[, get_locals])	
unpickle_exception(func, args, cause, tb[, context, ...])	
pickle_exception(obj)	
_get_subclasses(cls)	
install(*exc_classes_or_instances[, get_locals])	**Parameters** **get_locals** (*callable*) -- A function that take a frame argument and returns a dict. See *tblib.Traceback* class for example.
_install_for_instance(exc, seen)	

tblib.pickling_support.**unpickle_traceback**(*tb_frame*, *tb_lineno*, *tb_next*)

tblib.pickling_support.**pickle_traceback**(*tb*, *, *get_locals=None*)

tblib.pickling_support.**unpickle_exception**(*func*, *args*, *cause*, *tb*, *context=None*, *suppress_context=False*, *notes=None*)

tblib.pickling_support.**pickle_exception**(*obj*)

tblib.pickling_support.**_get_subclasses**(*cls*)

tblib.pickling_support.**install**(**exc_classes_or_instances*, *get_locals=None*)

Parameters
get_locals (*callable*) – A function that take a frame argument and returns a dict. See *tblib.Traceback* class for example.

tblib.pickling_support.**_install_for_instance**(*exc*, *seen*)

5.1.2 Package Contents

Classes

Code	Class that replicates just enough of the builtin Code object to enable serialization and traceback rendering.
Frame	Class that replicates just enough of the builtin Frame object to enable serialization and traceback rendering.
Traceback	Class that wraps builtin Traceback objects.

exception tblib.**TracebackParseError**

Bases: Exception

Common base class for all non-exit exceptions.

class tblib.**Code**(*code*)

Class that replicates just enough of the builtin Code object to enable serialization and traceback rendering.

co_code

class tblib.**Frame**(*frame*, *, *get_locals=None*)

Class that replicates just enough of the builtin Frame object to enable serialization and traceback rendering.

Parameters

get_locals (*callable*) – A function that take a frame argument and returns a dict.

See *Traceback* class for example.

clear()

For compatibility with PyPy 3.5; clear() was added to frame in Python 3.4 and is called by traceback.clear_frames(), which in turn is called by unittest.TestCase.assertRaises

class tblib.**Traceback**(*tb*, *, *get_locals=None*)

Class that wraps builtin Traceback objects.

Parameters

get_locals (*callable*) – A function that take a frame argument and returns a dict.

Ideally you will only return exactly what you need, and only with simple types that can be json serializable.

Example:

```
def get_locals(frame):
    if frame.f_locals.get("__tracebackhide__"):
        return {"__tracebackhide__": True}
    else:
        return {}
```

tb_next

to_traceback

to_dict

as_traceback()

Convert to a builtin Traceback object that is usable for raising or rendering a stacktrace.

as_dict()

Converts to a dictionary representation. You can serialize the result to JSON as it only has builtin objects like dicts, lists, ints or strings.

classmethod from_dict(*dct*)

Creates an instance from a dictionary with the same structure as `.as_dict()` returns.

classmethod from_string(*string*, *strict=True*)

Creates an instance by parsing a stacktrace. Strict means that parsing stops when lines are not indented by at least two spaces anymore.

CHAPTER

SIX

AUTHORS

- Ionel Cristian Mărieș - https://blog.ionelmc.ro
- Arcadiy Ivanov - https://github.com/arcivanov
- Beckjake - https://github.com/beckjake
- DRayX - https://github.com/DRayX
- Jason Madden - https://github.com/jamadden
- Jon Dufresne - https://github.com/jdufresne
- Elliott Sales de Andrade - https://github.com/QuLogic
- Victor Stinner - https://github.com/vstinner
- Guido Imperiale - https://github.com/crusaderky
- Alisa Sireneva - https://github.com/purplesyringa
- Michał Górny - https://github.com/mgorny
- Tim Maxwell - https://github.com/tmaxwell-anthropic

CHAPTER

SEVEN

CHANGELOG

7.1 3.0.0 (2023-10-22)

- Added support for `__context__`, `__suppress_context__` and `__notes__`. Contributed by Tim Maxwell in #72.
- Added the `get_locals` argument to `tblib.pickling_support.install()`, `tblib.Traceback` and `tblib.Frame`. Fixes #41.
- Dropped support for now-EOL Python 3.7 and added 3.12 in the test grid.

7.2 2.0.0 (2023-06-22)

- Removed support for legacy Pythons (2.7 and 3.6) and added Python 3.11 in the test grid.
- Some cleanups and refactors (mostly from ruff).

7.3 1.7.0 (2020-07-24)

- Add more attributes to `Frame` and `Code` objects for pytest compatibility. Contributed by Ivanq in #58.

7.4 1.6.0 (2019-12-07)

- When pickling an Exception, also pickle its traceback and the Exception chain (`raise ... from ...`). Contributed by Guido Imperiale in #53.

7.5 1.5.0 (2019-10-23)

- Added support for Python 3.8. Contributed by Victor Stinner in #42.
- Removed support for end of life Python 3.4.
- Few CI improvements and fixes.

7.6 1.4.0 (2019-05-02)

- Removed support for end of life Python 3.3.
- Fixed tests for Python 3.7. Contributed by Elliott Sales de Andrade in #36.
- Fixed compatibility issue with Twised (`twisted.python.failure.Failure` expected a `co_code` attribute).

7.7 1.3.2 (2017-04-09)

- Add support for PyPy3.5-5.7.1-beta. Previously `AttributeError: 'Frame' object has no attribute 'clear'` could be raised. See PyPy issue #2532.

7.8 1.3.1 (2017-03-27)

- Fixed handling for tracebacks due to exceeding the recursion limit. Fixes #15.

7.9 1.3.0 (2016-03-08)

- Added `Traceback.from_string`.

7.10 1.2.0 (2015-12-18)

- Fixed handling for tracebacks from generators and other internal improvements and optimizations. Contributed by DRayX in #10 and #11.

7.11 1.1.0 (2015-07-27)

- Added support for Python 2.6. Contributed by Arcadiy Ivanov in #8.

7.12 1.0.0 (2015-03-30)

- Added `to_dict` method and `from_dict` classmethod on Tracebacks. Contributed by beckjake in #5.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

A

C

E

F

I

M

P

R

T

U