

---

**tblib**

***Release 1.3.2***

**Oct 23, 2019**



---

## Contents

---

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	Installation . . . . .	2
1.2	Documentation . . . . .	2
1.3	Credits . . . . .	11
<b>2</b>	<b>Installation</b>	<b>13</b>
<b>3</b>	<b>Usage</b>	<b>15</b>
<b>4</b>	<b>Reference</b>	<b>17</b>
4.1	tblib . . . . .	17
<b>5</b>	<b>Contributing</b>	<b>19</b>
5.1	Bug reports . . . . .	19
5.2	Documentation improvements . . . . .	19
5.3	Feature requests and feedback . . . . .	19
5.4	Development . . . . .	20
<b>6</b>	<b>Authors</b>	<b>21</b>
<b>7</b>	<b>Changelog</b>	<b>23</b>
7.1	1.3.2 (2017-04-09) . . . . .	23
7.2	1.3.1 (2017-03-27) . . . . .	23
7.3	1.3.0 (2016-03-08) . . . . .	23
7.4	1.2.0 (2015-12-18) . . . . .	23
7.5	1.1.0 (2015-07-27) . . . . .	23
7.6	1.0.0 (2015-03-30) . . . . .	24
<b>8</b>	<b>Indices and tables</b>	<b>25</b>
	<b>Python Module Index</b>	<b>27</b>
	<b>Index</b>	<b>29</b>



# CHAPTER 1

---

## Overview

---

docs	
tests	
package	

Traceback serialization library.

- 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.

**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

- *Pickling tracebacks*
- *Unpickling*
- *Raising*
  - *What if we have a local stack, does it show correctly ?*
  - *It also supports more contrived scenarios*
- *Reference*
  - *tblib.Traceback*
    - \* *tblib.Traceback.to\_dict*
    - \* *tblib.Traceback.from\_dict*
    - \* *tblib.Traceback.from\_string*
  - *tblib.decorators.return\_error*
    - \* *What if we have a local call stack ?*
    - \* *Other weird stuff*

### 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
```

(continues on next page)

(continued from previous page)

```

True
>>> try:
...     inner_2()
... except:
...     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

```

>>> 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):
...

```

(continues on next page)

(continued from previous page)

```

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(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

```

### 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

```

(continues on next page)



(continued from previous page)

```

    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())
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()

```

(continues on next page)

(continued from previous page)

```

...     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()
...
RuntimeError: maximum recursion depth exceeded

```

## 1.2.4 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:

```

(continues on next page)



(continued from previous page)

```

...   File "tests/examples.py", line 10, in func_c
...       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()

```

(continues on next page)

(continued from previous page)

```
>>> 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:

```
>>> import traceback
>>> from multiprocessing import Pool
>>> from examples import func_a
>>> if sys.version_info[:2] >= (3, 4):
...     import multiprocessing.pool
...     # Undo the fix for http://bugs.python.org/issue13831 so that we can see the_
...     ↪effects of our change.
...     # because Python 3.4 will show the remote traceback (but as a string sadly)
...     multiprocessing.pool.ExceptionWithTraceback = lambda e, t: e
>>> pool = Pool()
>>> try:
...     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 !

>>> pool.terminate()
```

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

```
>>> from tblib.decorators import apply_with_return_error, Error
>>> from itertools import repeat
>>> pool = Pool()
>>> try:
...     for i in pool.map(apply_with_return_error, zip(repeat(func_a), range(5))):
...         if isinstance(i, Error):
...             i.reraise()
```

(continues on next page)

(continued from previous page)

```

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

```

(continues on next page)

(continued from previous page)

```

File "<doctest README.rst[...]>", line 5, 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 2

---

### Installation

---

At the command line:

```
pip install tblib
```



## CHAPTER 3

---

### Usage

---

To use tblib in a project:

```
import tblib
```



## CHAPTER 4

---

### Reference

---

#### 4.1 tblib



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

### 5.1 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.

### 5.2 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.

### 5.3 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 :)

## 5.4 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:your_name_here/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, doc builder and spell checker with `tox` 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.

### 5.4.1 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`)<sup>1</sup>.
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`.

### 5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

---

<sup>1</sup> If you don’t have all the necessary python versions available locally you can rely on Travis - it will [run the tests](#) for each change you add in the pull request.  
It will be slower though ...



## CHAPTER 6

---

### 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>



### 7.1 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](#).

### 7.2 1.3.1 (2017-03-27)

- Fixed handling for tracebacks due to exceeding the recursion limit. Fixes [#15](#).

### 7.3 1.3.0 (2016-03-08)

- Added `Traceback.from_string`.

### 7.4 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](#).

### 7.5 1.1.0 (2015-07-27)

- Added support for Python 2.6. Contributed by Arcadiy Ivanov in [#8](#).

## 7.6 1.0.0 (2015-03-30)

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

## CHAPTER 8

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



**t**

tblib, [17](#)





## T

`tblib` (*module*), 17