traceback — Exceptions and Stack Traces

Purpose: Extract, format, and print exceptions and stack traces.

The `traceback` module works with the call stack to produce error messages. A *traceback* is a stack trace from the point of an exception handler down the call chain to the point where the exception was raised. Tracebacks also can be accessed from the current call stack up from the point of a call (and without the context of an error), which is useful for finding out the paths being followed into a function.

The high-level API in `traceback` uses `StackSummary` and `FrameSummary` instances to hold the representation of the stack. These classes can be constructed from a traceback or the current execution stack, and then processed in the same ways.

The low-level functions in `traceback` fall into several common categories. There are functions for extracting raw tracebacks from the current runtime environment (either an exception handler for a traceback, or the regular stack). The extracted stack trace is a sequence of tuples containing the filename, line number, function name, and text of the source line.

Once extracted, the stack trace can be formatted using functions like `format_exception()`, `format_stack()`, etc. The format functions return a list of strings with messages formatted to be printed. There are shorthand functions for printing the formatted values, as well.

Although the functions in `traceback` mimic the behavior of the interactive interpreter by default, they also are useful for handling exceptions in situations where dumping the full stack trace to the console is not desirable. For example, a web application may need to format the traceback so it looks good in HTML and an IDE may convert the elements of the stack trace into a clickable list that lets the user browse the source.

Supporting Functions

The examples in this section use the module `traceback_example.py`.

```
# traceback_example.py

import traceback
import sys


def produce_exception(recursion_level=2):
    sys.stdout.flush()
    if recursion_level:
        produce_exception(recursion_level - 1)
    else:
        raise RuntimeError()


def call_function(f, recursion_level=2):
    if recursion_level:
        return call_function(f, recursion_level - 1)
    else:
        return f()
```

Examining the Stack

To examine the current stack, construct a `StackSummary` from `walk_stack()`.

```
# traceback_stacksummary.py

import traceback
import sys

from traceback_example import call_function


def f():
    summary = traceback.StackSummary.extract(
```

```
        traceback.walk_stack(None)
    )
    print(''.join(summary.format()))


print('Calling f() directly:')
f()

print()
print('Calling f() from 3 levels deep:')
call_function(f)
```

The `format()` method produces a sequence of formatted strings ready to be printed.

```
$ python3 traceback_stacksummary.py

Calling f() directly:
  File "traceback_stacksummary.py", line 18, in f
    traceback.walk_stack(None)
  File "traceback_stacksummary.py", line 24, in <module>
    f()


Calling f() from 3 levels deep:
  File "traceback_stacksummary.py", line 18, in f
    traceback.walk_stack(None)
  File ".../traceback_example.py", line 26, in call_function
    return f()
  File ".../traceback_example.py", line 24, in call_function
    return call_function(f, recursion_level - 1)
  File ".../traceback_example.py", line 24, in call_function
    return call_function(f, recursion_level - 1)
  File "traceback_stacksummary.py", line 28, in <module>
    call_function(f)
```

The StackSummary is an iterable container holding FrameSummary instances.

```
# traceback_framesummary.py

import traceback
import sys

from traceback_example import call_function

template = (
    '{fs.filename:<26}:{fs.lineno}:{fs.name}:\n'
    '    {fs.line}'
)


def f():
    summary = traceback.StackSummary.extract(
        traceback.walk_stack(None)
    )
    for fs in summary:
        print(template.format(fs=fs))


print('Calling f() directly:')
f()

print()
print('Calling f() from 3 levels deep:')
call_function(f)
```

Each FrameSummary describes a frame of the stack, including information about where in the program source files the execution context is.

```
$ python3 traceback_framesummary.py
```

```
Calling f() directly:
traceback_framesummary.py :23:f:
    traceback.walk_stack(None)
traceback_framesummary.py :30:<module>:
    f()

Calling f() from 3 levels deep:
traceback_framesummary.py :23:f:
    traceback.walk_stack(None)
.../traceback_example.py:26:call_function:
    return f()
.../traceback_example.py:24:call_function:
    return call_function(f, recursion_level - 1)
.../traceback_example.py:24:call_function:
    return call_function(f, recursion_level - 1)
traceback_framesummary.py :34:<module>:
    call_function(f)
```

TracebackException

The TracebackException class is a high-level interface for building a StackSummary while processing a traceback.

```
# traceback_tracebackexception.py

import traceback
import sys

from traceback_example import produce_exception

print('with no exception:')
exc_type, exc_value, exc_tb = sys.exc_info()
tbe = traceback.TracebackException(exc_type, exc_value, exc_tb)
print(''.join(tbe.format()))

print('\nwith exception:')
try:
    produce_exception()
except Exception as err:
    exc_type, exc_value, exc_tb = sys.exc_info()
    tbe = traceback.TracebackException(
        exc_type, exc_value, exc_tb,
    )
    print(''.join(tbe.format()))

    print('\nexception only:')
    print(''.join(tbe.format_exception_only()))
```

The `format()` method produces a formatted version of the full traceback, while `format_exception_only()` shows only the exception message.

```
$ python3 traceback_tracebackexception.py

with no exception:
None: None


with exception:
Traceback (most recent call last):
  File "traceback_tracebackexception.py", line 22, in <module>
    produce_exception()
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 19, in produce_exception
    raise RuntimeError()
RuntimeError
```

```
exception only:
RuntimeError
```

Low-level Exception APIs

Another way to handle exception reporting is with `print_exc()`. It uses `sys.exc_info()` to obtain the exception information for the current thread, formats the results, and prints the text to a file handle (`sys.stderr`, by default).

```
# traceback_print_exc.py

import traceback
import sys

from traceback_example import produce_exception

print('print_exc() with no exception:')
traceback.print_exc(file=sys.stdout)
print()

try:
    produce_exception()
except Exception as err:
    print('print_exc():')
    traceback.print_exc(file=sys.stdout)
    print()
    print('print_exc(1):')
    traceback.print_exc(limit=1, file=sys.stdout)
```

In this example, the file handle for `sys.stdout` is substituted so the informational and traceback messages are mingled correctly:

```
$ python3 traceback_print_exc.py

print_exc() with no exception:
NoneType: None

print_exc():
Traceback (most recent call last):
  File "traceback_print_exc.py", line 20, in <module>
    produce_exception()
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 19, in produce_exception
    raise RuntimeError()
RuntimeError

print_exc(1):
Traceback (most recent call last):
  File "traceback_print_exc.py", line 20, in <module>
    produce_exception()
RuntimeError
```

`print_exc()` is just a shortcut for `print_exception()`, which requires explicit arguments.

```
# traceback_print_exception.py

import traceback
import sys

from traceback_example import produce_exception

try:
    produce_exception()
except Exception as err:
    print('print_exception():')
    exc_type, exc_value, exc_tb = sys.exc_info()
    traceback.print_exception(exc_type, exc_value, exc_tb)
```

The arguments to `print_exception()` are produced by `sys.exc_info()`.

```
$ python3 traceback_print_exception.py

Traceback (most recent call last):
  File "traceback_print_exception.py", line 16, in <module>
    produce_exception()
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 19, in produce_exception
    raise RuntimeError()
RuntimeError
print_exception():
```

`print_exception()` uses `format_exception()` to prepare the text.

```
# traceback_format_exception.py

import traceback
import sys
from pprint import pprint

from traceback_example import produce_exception

try:
    produce_exception()
except Exception as err:
    print('format_exception():')
    exc_type, exc_value, exc_tb = sys.exc_info()
    pprint(
        traceback.format_exception(exc_type, exc_value, exc_tb),
        width=65,
    )
```

The same three arguments, exception type, exception value, and traceback, are used with `format_exception()`.

```
$ python3 traceback_format_exception.py

format_exception():
['Traceback (most recent call last):\n',
 '  File "traceback_format_exception.py", line 17, in
<module>\n'
 '    produce_exception()\n',
 '  File '
 '".../traceback_example.py", '
 'line 17, in produce_exception\n'
 '    produce_exception(recursion_level - 1)\n',
 '  File '
 '".../traceback_example.py", '
 'line 17, in produce_exception\n'
 '    produce_exception(recursion_level - 1)\n',
 '  File '
 '".../traceback_example.py", '
 'line 19, in produce_exception\n'
 '    raise RuntimeError()\n',
 'RuntimeError\n']
```

To process the traceback in some other way, such as formatting it differently, use `extract_tb()` to get the data in a usable form.

```
# traceback_extract_tb.py

import traceback
import sys
import os
from traceback_example import produce_exception
```

```
template = '{filename:<23}:{linenum}:{funcname}:\n    {source}'

try:
    produce_exception()
except Exception as err:
    print('format_exception():')
    exc_type, exc_value, exc_tb = sys.exc_info()
    for tb_info in traceback.extract_tb(exc_tb):
        filename, linenum, funcname, source = tb_info
        if funcname != '<module>':
            funcname = funcname + '()'
        print(template.format(
            filename=os.path.basename(filename),
            linenum=linenum,
            source=source,
            funcname=funcname)
        )
```

The return value is a list of entries from each level of the stack represented by the traceback. Each entry is a tuple with four parts: the name of the source file, the line number in that file, the name of the function, and the source text from that line with whitespace stripped (if the source is available).

```
$ python3 traceback_extract_tb.py

format_exception():
traceback_extract_tb.py:18:<module>:
    produce_exception()
traceback_example.py   :17:produce_exception():
    produce_exception(recursion_level - 1)
traceback_example.py   :17:produce_exception():
    produce_exception(recursion_level - 1)
traceback_example.py   :19:produce_exception():
    raise RuntimeError()
```

Low-level Stack APIs

There are a similar set of functions for performing the same operations with the current call stack instead of a traceback. `print_stack()` prints the current stack, without generating an exception.

```
# traceback_print_stack.py

import traceback
import sys

from traceback_example import call_function


def f():
    traceback.print_stack(file=sys.stdout)


print('Calling f() directly:')
f()

print()
print('Calling f() from 3 levels deep:')
call_function(f)
```

The output look like a traceback without an error message.

```
$ python3 traceback_print_stack.py

Calling f() directly:
  File "traceback_print_stack.py", line 21, in <module>
    f()
  File "traceback_print_stack.py", line 17, in f
    traceback.print_stack(file=sys.stdout)
```

```
Calling f() from 3 levels deep:
  File "traceback_print_stack.py", line 25, in <module>
    call_function(f)
  File ".../traceback_example.py", line 24, in call_function
    return call_function(f, recursion_level - 1)
  File ".../traceback_example.py", line 24, in call_function
    return call_function(f, recursion_level - 1)
  File ".../traceback_example.py", line 26, in call_function
    return f()
  File "traceback_print_stack.py", line 17, in f
    traceback.print_stack(file=sys.stdout)
```

`format_stack()` prepares the stack trace in the same way that `format_exception()` prepares the traceback.

```
# traceback_format_stack.py

import traceback
import sys
from pprint import pprint

from traceback_example import call_function


def f():
    return traceback.format_stack()


formatted_stack = call_function(f)
pprint(formatted_stack)
```

It returns a list of strings, each of which makes up one line of the output.

```
$ python3 traceback_format_stack.py

['  File "traceback_format_stack.py", line 21, in <module>\n'
 '    formatted_stack = call_function(f)\n',
 '  File '
 '".../traceback_example.py", '
 'line 24, in call_function\n'
 '    return call_function(f, recursion_level - 1)\n',
 '  File '
 '".../traceback_example.py", '
 'line 24, in call_function\n'
 '    return call_function(f, recursion_level - 1)\n',
 '  File '
 '".../traceback_example.py", '
 'line 26, in call_function\n'
 '    return f()\n',
 '  File "traceback_format_stack.py", line 18, in f\n'
 '    return traceback.format_stack()\n']
```

The `extract_stack()` function works like `extract_tb()`.

```
# traceback_extract_stack.py

import traceback
import sys
import os

from traceback_example import call_function

template = '{filename:<26}:{linenum}:{funcname}:\n    {source}'


def f():
    return traceback.extract_stack()


stack = call_function(f)
```

```
for filename, linenum, funcname, source in stack:
    if funcname != '<module>':
        funcname = funcname + '()'
    print(template.format(
        filename=os.path.basename(filename),
        linenum=linenum,
        source=source,
        funcname=funcname)
    )
```

It also accepts arguments, not shown here, to start from an alternate place in the stack frame or to limit the depth of traversal.

```
$ python3 traceback_extract_stack.py

traceback_extract_stack.py:23:<module>:
    stack = call_function(f)
traceback_example.py      :24:call_function():
    return call_function(f, recursion_level - 1)
traceback_example.py      :24:call_function():
    return call_function(f, recursion_level - 1)
traceback_example.py      :26:call_function():
    return f()
traceback_extract_stack.py:20:f():
    return traceback.extract_stack()
```

See also

- Standard library documentation for traceback
- sys – The sys module includes singletons that hold the current exception.
- inspect – The inspect module includes other functions for probing the frames on the stack.
- cgitb – Another module for formatting tracebacks nicely.

trace — Follow Program Flow

cgitb — Detailed Traceback Reports

Quick Links

This page was last updated 2018-03-18.

Navigation

trace — Follow Program Flow
cgitb — Detailed Traceback Reports

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?