

29.9. `traceback` — Print or retrieve a stack traceback

Source code: [Lib/traceback.py](#)

This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, such as in a “wrapper” around the interpreter.

The module uses traceback objects — this is the object type that is stored in the `sys.last_traceback` variable and returned as the third item from `sys.exc_info()`.

The module defines the following functions:

`traceback.print_tb(tb, limit=None, file=None)`

Print up to *limit* stack trace entries from traceback object *tb* (starting from the caller’s frame) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. If *file* is omitted or `None`, the output goes to `sys.stderr`; otherwise it should be an open file or file-like object to receive the output.

Changed in version 3.5: Added negative *limit* support.

`traceback.print_exception(etype, value, tb, limit=None, file=None, chain=True)`

Print exception information and stack trace entries from traceback object *tb* to *file*. This differs from `print_tb()` in the following ways:

- if *tb* is not `None`, it prints a header `Traceback (most recent call last):`
- it prints the exception *etype* and *value* after the stack trace
- if `type(value)` is `SyntaxError` and *value* has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

The optional *limit* argument has the same meaning as for `print_tb()`. If *chain* is true (the default), then chained exceptions (the `__cause__` or `__context__` attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

Changed in version 3.5: The *etype* argument is ignored and inferred from the type of *value*.

`traceback.print_exc(limit=None, file=None, chain=True)`

This is a shorthand for `print_exception(*sys.exc_info(), limit, file, chain)`.

`traceback.print_last(limit=None, file=None, chain=True)`

This is a shorthand for `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file, chain)`. In general it will work only after an exception has reached an interactive prompt (see [sys.last_type](#)).

`traceback.print_stack(f=None, limit=None, file=None)`

Print up to *limit* stack trace entries (starting from the invocation point) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *file* argument has the same meaning as for [print_tb\(\)](#).

Changed in version 3.5: Added negative *limit* support.

`traceback.extract_tb(tb, limit=None)`

Return a list of “pre-processed” stack trace entries extracted from the traceback object *tb*. It is useful for alternate formatting of stack traces. The optional *limit* argument has the same meaning as for [print_tb\(\)](#). A “pre-processed” stack trace entry is a 4-tuple (*filename*, *line number*, *function name*, *text*) representing the information that is usually printed for a stack trace. The *text* is a string with leading and trailing whitespace stripped; if the source is not available it is `None`.

`traceback.extract_stack(f=None, limit=None)`

Extract the raw traceback from the current stack frame. The return value has the same format as for [extract_tb\(\)](#). The optional *f* and *limit* arguments have the same meaning as for [print_stack\(\)](#).

`traceback.format_list(extracted_list)`

Given a list of tuples as returned by [extract_tb\(\)](#) or [extract_stack\(\)](#), return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not `None`.

`traceback.format_exception_only(etype, value)`

Format the exception part of a traceback. The arguments are the exception type and value such as given by `sys.last_type` and `sys.last_value`. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for [SyntaxError](#) exceptions, it contains several lines that (when printed) display detailed information about where the syntax error oc-

curred. The message indicating which exception occurred is the always last string in the list.

`traceback.format_exception(etype, value, tb, limit=None, chain=True)`

Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to `print_exception()`. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does `print_exception()`.

Changed in version 3.5: The `etype` argument is ignored and inferred from the type of `value`.

`traceback.format_exc(limit=None, chain=True)`

This is like `print_exc(limit)` but returns a string instead of printing to a file.

`traceback.format_tb(tb, limit=None)`

A shorthand for `format_list(extract_tb(tb, limit))`.

`traceback.format_stack(f=None, limit=None)`

A shorthand for `format_list(extract_stack(f, limit))`.

`traceback.clear_frames(tb)`

Clears the local variables of all the stack frames in a traceback `tb` by calling the `clear()` method of each frame object.

New in version 3.4.

`traceback.walk_stack(f)`

Walk a stack following `f.f_back` from the given frame, yielding the frame and line number for each frame. If `f` is `None`, the current stack is used. This helper is used with `StackSummary.extract()`.

New in version 3.5.

`traceback.walk_tb(tb)`

Walk a traceback following `tb_next` yielding the frame and line number for each frame. This helper is used with `StackSummary.extract()`.

New in version 3.5.

The module also defines the following classes:

29.9.1. `TracebackException` Objects

New in version 3.5.

`TracebackException` objects are created from actual exceptions to capture data for later printing in a lightweight fashion.

`class traceback.TracebackException(exc_type, exc_value, exc_traceback, *, limit=None, lookup_lines=True, capture_locals=False)`

Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the `StackSummary` class.

Note that when locals are captured, they are also shown in the traceback.

`__cause__`

A `TracebackException` of the original `__cause__`.

`__context__`

A `TracebackException` of the original `__context__`.

`__suppress_context__`

The `__suppress_context__` value from the original exception.

`stack`

A `StackSummary` representing the traceback.

`exc_type`

The class of the original traceback.

`filename`

For syntax errors - the file name where the error occurred.

`lineno`

For syntax errors - the line number where the error occurred.

`text`

For syntax errors - the text where the error occurred.

`offset`

For syntax errors - the offset into the text where the error occurred.

`msg`

For syntax errors - the compiler error message.

classmethod **from_exception**(exc, *, limit=None, lookup_lines=True, capture_locals=False)

Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the [StackSummary](#) class.

Note that when locals are captured, they are also shown in the traceback.

format(*, chain=True)

Format the exception.

If *chain* is not True, `__cause__` and `__context__` will not be formatted.

The return value is a generator of strings, each ending in a newline and some containing internal newlines. [print_exception\(\)](#) is a wrapper around this method which just prints the lines to a file.

The message indicating which exception occurred is always the last string in the output.

format_exception_only()

Format the exception part of the traceback.

The return value is a generator of strings, each ending in a newline.

Normally, the generator emits a single string; however, for [SyntaxError](#) exceptions, it emits several lines that (when printed) display detailed information about where the syntax error occurred.

The message indicating which exception occurred is always the last string in the output.

29.9.2. [StackSummary](#) Objects

New in version 3.5.

[StackSummary](#) objects represent a call stack ready for formatting.

class traceback. **StackSummary**

classmethod **extract**(frame_gen, *, limit=None, lookup_lines=True, capture_locals=False)

Construct a [StackSummary](#) object from a frame generator (such as is returned by [walk_stack\(\)](#) or [walk_tb\(\)](#)).

If *limit* is supplied, only this many frames are taken from *frame_gen*. If *lookup_lines* is `False`, the returned `FrameSummary` objects will not have read their lines in yet, making the cost of creating the `StackSummary` cheaper (which may be valuable if it may not actually get formatted). If *capture_locals* is `True` the local variables in each `FrameSummary` are captured as object representations.

classmethod **from_list**(*a_list*)

Construct a `StackSummary` object from a supplied old-style list of tuples. Each tuple should be a 4-tuple with filename, lineno, name, line as the elements.

format()

Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.

For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

Changed in version 3.6: Long sequences of repeated frames are now abbreviated.

29.9.3. `FrameSummary` Objects

New in version 3.5.

`FrameSummary` objects represent a single frame in a traceback.

class `traceback.FrameSummary(filename, lineno, name, lookup_line=True, locals=None, line=None)`

Represent a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frames locals included in it. If *lookup_line* is `False`, the source code is not looked up until the `FrameSummary` has the `line` attribute accessed (which also happens when casting it to a tuple). `line` may be directly provided, and will prevent line lookups happening at all. *locals* is an optional local variable dictionary, and if supplied the variable representations are stored in the summary for later display.

29.9.4. Traceback Examples

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the [code](#) module.

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("*** print_tb:")
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print("*** print_exception:")
    # exc_type below is ignored on 3.5 and later
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                              limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
```

```

print("*** format_exception:")
# exc_type below is ignored on 3.5 and later
print(repr(traceback.format_exception(exc_type, exc_value,
                                     exc_traceback)))

print("*** extract_tb:")
print(repr(traceback.extract_tb(exc_traceback)))
print("*** format_tb:")
print(repr(traceback.format_tb(exc_traceback)))
print("*** tb_lineno:", exc_traceback.tb_lineno)

```

The output for the example would look similar to this:

```

*** print_tb:
File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_dea
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return t
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_death>]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_dea
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return t
*** tb_lineno: 10

```

The following example shows the different ways to print and format the stack:


```

>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
File "<doctest>", line 10, in <module>
    another_function()
File "<doctest>", line 3, in another_function
    lumberstack()
File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback

```

This last example demonstrates the final few formatting functions:

```

>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range\n']

```