`!traceback` --- Print or retrieve a stack traceback

- Module-Level Functions
- !TracebackException Objects
- !StackSummary Objects
- !FrameSummary Objects
- Examples of Using the Module-Level Functions
- Examples of Using TracebackException

traceback

Source code: `Lib/traceback.py`

This module provides a standard interface to extract, format and print stack traces of Python programs. It is more flexible than the interpreter's default traceback display, and therefore makes it possible to configure certain aspects of the output. Finally, it contains a utility for capturing enough information about an exception to print it later, without the need to save a reference to the actual exception. Since exceptions can be the roots of large objects graph, this utility can significantly improve memory management.

pair: object; traceback

The module uses `traceback objects <traceback-objects>` --- these are objects of type `types.TracebackType`, which are assigned to the `~BaseException.__traceback__` field of `BaseException` instances.

Module `faulthandler`
Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

Module `pdb`
Interactive source code debugger for Python programs.

The module's API can be divided into two parts:

- Module-level functions offering basic functionality, which are useful for interactive inspection of exceptions and tracebacks.
- `TracebackException` class and its helper classes `StackSummary` and `FrameSummary`. These offer both more flexibility in the output generated and the ability to store the information necessary for later formatting without holding references to actual exception and traceback objects.

Module-Level Functions

print_tb(tb, limit=None, file=None)

Print up to *limit* stack trace entries from `traceback object <traceback-objects>` *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 <file object>` or `file-like object` to receive the output.

> [!NOTE] The meaning of the *limit* parameter is different than the meaning of `sys.tracebacklimit`. A negative *limit* value corresponds to a positive value of `!sys.tracebacklimit`, whereas the behaviour of a positive *limit* value cannot be achieved with `!sys.tracebacklimit`.

3.5 Added negative *limit* support.

print_exception(exc, /[, value, tb], limit=None, file=None, chain=True)

Print exception information and stack trace entries from `traceback object <traceback-objects>` *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 type and *value* after the stack trace

single: ^ (caret); marker

- 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.

Since Python 3.10, instead of passing *value* and *tb*, an exception object can be passed as the first argument. If *value* and *tb* are provided, the first argument is ignored in order to provide backwards compatibility.

The optional *limit* argument has the same meaning as for `print_tb`. If *chain* is true (the default), then chained exceptions (the `~BaseException.__cause__` or `~BaseException.__context__` attributes of

the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

3.5 The *etype* argument is ignored and inferred from the type of *value*.

3.10 The *etype* parameter has been renamed to *exc* and is now positional-only.

print_exc(limit=None, file=None, chain=True)

This is a shorthand for `print_exception(sys.exception(), limit=limit, file=file, chain=chain)`.

print_last(limit=None, file=None, chain=True)

This is a shorthand for `print_exception(sys.last_exc, limit=limit, file=file, chain=chain)`. In general it will work only after an exception has reached an interactive prompt (see `sys.last_exc`).

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 <frame-objects>` to start. The optional *file* argument has the same meaning as for `print_tb`.

3.5 Added negative *limit* support.

extract_tb(tb, limit=None)

Return a `StackSummary` object representing a list of "pre-processed" stack trace entries extracted from the `traceback object <traceback-objects>` *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 `FrameSummary` object with attributes representing the information that is usually printed for a stack trace.

extract_stack(f=None, limit=None)

Extract the raw traceback from the current `stack frame <frame-objects>`. 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`.

print_list(extracted_list, file=None)

Print the list of tuples as returned by `extract_tb` or `extract_stack` as a formatted stack trace to the given file. If *file* is `None`, the output is written to `sys.stderr`.

format_list(extracted_list)

Given a list of tuples or `FrameSummary` objects 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`.

format_exception_only(exc, /[, value], *, show_group=False)

Format the exception part of a traceback using an exception value such as given by `sys.last_exc`. The return value is a list of strings, each ending in a newline. The list contains the exception's message, which is normally a single string; however, for `SyntaxError` exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. Following the message, the list contains the exception's `notes <BaseException.__notes__>`.

Since Python 3.10, instead of passing *value*, an exception object can be passed as the first argument. If *value* is provided, the first argument is ignored in order to provide backwards compatibility.

When *show_group* is `True`, and the exception is an instance of `BaseExceptionGroup`, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

3.10 The *etype* parameter has been renamed to *exc* and is now positional-only.

3.11 The returned list now includes any `notes <BaseException.__notes__>` attached to the exception.

3.13 *show_group* parameter was added.

format_exception(exc, /[, 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`.

3.5 The *etype* argument is ignored and inferred from the type of *value*.

3.10 This function's behavior and signature were modified to match `print_exception`.

format_exc(limit=None, chain=True)

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

format_tb(tb, limit=None)

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

format_stack(f=None, limit=None)

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

clear_frames(tb)

Clears the local variables of all the stack frames in a `traceback <traceback-objects>` *tb* by calling the `~frame.clear` method of each `frame object <frame-objects>`.

3.4

walk_stack(f)

Walk a stack following `f.f_back <frame.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`.

3.5

3.14 This function previously returned a generator that would walk the stack when first iterated over. The generator returned now is the state of the stack when `walk_stack` is called.

walk_tb(tb)

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

3.5

`!TracebackException` Objects

3.5

`!TracebackException` objects are created from actual exceptions to capture data for later printing. They offer a more lightweight method of storing this information by avoiding holding references to `traceback<traceback-objects>` and `frame<frame-objects>` objects. In addition, they expose more options to configure the output compared to the module-level functions described above.

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

If *compact* is true, only data that is required by `!TracebackException`'s `format` method is saved in the class attributes. In particular, the `__context__` field is calculated only if `__cause__` is `None` and `__suppress_context__` is false.

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

max_group_width and *max_group_depth* control the formatting of exception groups (see `BaseExceptionGroup`). The depth refers to the nesting level of the group, and the width refers to the size of a single exception group's exceptions array. The formatted output is truncated when either limit is exceeded.

3.10 Added the *compact* parameter.

3.11 Added the *max_group_width* and *max_group_depth* parameters.

__cause__

A `!TracebackException` of the original `~BaseException.__cause__`.

__context__

A `!TracebackException` of the original `~BaseException.__context__`.

exceptions

If `self` represents an `ExceptionGroup`, this field holds a list of `!TracebackException` instances representing the nested exceptions. Otherwise it is `None`.

3.11

__suppress_context__

The `~BaseException.__suppress_context__` value from the original exception.

__notes__

The `~BaseException.__notes__` value from the original exception, or `None` if the exception does not have any notes. If it is not `None` is it formatted in the traceback after the exception string.

3.11

stack

A `StackSummary` representing the traceback.

exc_type

The class of the original exception.

3.13

exc_type_str

String display of the class of the original exception.

3.13

filename

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

lineno

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

end_lineno

For syntax errors - the end line number where the error occurred. Can be `None` if not present.

3.10

text

For syntax errors - the text where the error occurred.

offset

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

end_offset

For syntax errors - the end offset into the text where the error occurred. Can be `None` if not present.

3.10

msg

For syntax errors - the compiler error message.

from_exception(exc, *, limit=None, lookup_lines=True, capture_locals=False, compact=False, max_group_width=15, max_group_depth=10)

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.

print(*, file=None, chain=True)

Print to *file* (default `sys.stderr`) the exception information returned by `format`.

3.11

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. `~traceback.print_exception` is a wrapper around this method which just prints the lines to a file.

format_exception_only(*, show_group=False)

Format the exception part of the traceback.

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

When *show_group* is `False`, the generator emits the exception's message followed by its notes (if it has any). The exception message is normally a single string; however, for `SyntaxError` exceptions, it consists of several lines that (when printed) display detailed information about where the syntax error occurred.

When *show_group* is `True`, and the exception is an instance of `BaseExceptionGroup`, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

3.11 The exception's `notes <BaseException.__notes__>` are now included in the output.

3.13 Added the *show_group* parameter.

`!StackSummary` Objects

3.5

`!StackSummary` objects represent a call stack ready for formatting.

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

Construct a `!StackSummary` object from a frame generator (such as is returned by `~traceback.walk_stack` or `~traceback.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.

3.12 Exceptions raised from `repr` on a local variable (when *capture_locals* is `True`) are no longer propagated to the caller.

from_list(a_list)

Construct a `!StackSummary` object from a supplied list of `FrameSummary` objects or 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 <frame-objects>` 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.

3.6 Long sequences of repeated frames are now abbreviated.

format_frame_summary(frame_summary)

Returns a string for printing one of the `frames <frame-objects>` involved in the stack. This method is called for each `FrameSummary` object to be printed by `StackSummary.format`. If it returns `None`, the frame is omitted from the output.

3.11

`!FrameSummary` Objects

3.5

A `!FrameSummary` object represents a single `frame <frame-objects>` in a `traceback <traceback-objects>`.

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

`!FrameSummary` instances have the following attributes:

FrameSummary.filename

The filename of the source code for this frame. Equivalent to accessing `f.f_code.co_filename <codeobject.co_filename>` on a `frame object <frame-objects>` *f*.

FrameSummary.lineno

The line number of the source code for this frame.

FrameSummary.name

Equivalent to accessing `f.f_code.co_name <codeobject.co_name>` on a `frame object <frame-objects>` *f*.

FrameSummary.line

A string representing the source code for this frame, with leading and trailing whitespace stripped. If the source is not available, it is `None`.

FrameSummary.end_lineno

The last line number of the source code for this frame. By default, it is set to `lineno` and indexation starts from 1.

3.13 The default value changed from `None` to `lineno`.

FrameSummary.colno

The column number of the source code for this frame. By default, it is `None` and indexation starts from 0.

FrameSummary.end_colno

The last column number of the source code for this frame. By default, it is `None` and indexation starts from 0.

Examples of Using the Module-Level Functions

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_life()

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

try:
lumberjack()

except IndexError as exc:
print("*** print_tb:") traceback.print_tb(exc.__traceback__, limit=1, file=sys.stdout) print("*** print_exception:") traceback.print_exception(exc, 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:") print(repr(traceback.format_exception(exc))) 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_life() ~~~~~~~~~~~~~~~~~~~~~^^ 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_life() ~~~~~~~~~~~~~~~~~~~~~^^ 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 default[0]>", line 10, in <module>n lumberjack()n ~~~~~~~~~~^^n', ' File "<doctest default[0]>", line 4, in lumberjackn bright_side_of_life()n ~~~~~~~~~~~~~~~~~~~~~^^n', ' File "<doctest default[0]>", line 7, in bright_side_of_lifen return tuple()[0]n ~~~~~~~^^^n', 'IndexError: tuple index out of rangen']*** 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_life>] * **format_tb: [' File "<doctest default[0]>", line 10, in <module>n lumberjack()n ~~~~~~~~~~^^n', ' File "<doctest default[0]>", line 4, in lumberjackn bright_side_of_life()n ~~~~~~~~~~~~~~~~~~~~~^^n', ' File "<doctest default[0]>", line 7, in bright_side_of_lifen return tuple()[0]n ~~~~~~~^^^n']*** 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.format_stack()))\n']
```

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 eggsn return "bacon"n'] >>> an_error = IndexError('tuple index out of range') >>> traceback.format_exception_only(an_error) ['IndexError: tuple index out of rangen']

Examples of Using `TracebackException`

With the helper class, we have more options:

```
>>> import sys
>>> from traceback import TracebackException
>>>
>>> def lumberjack():
...     bright_side_of_life()
...
>>> def bright_side_of_life():
...     t = "bright", "side", "of", "life"
...     return t[5]
...
>>> try:
```

```
...     lumberjack()
... except IndexError as e:
...     exc = e
...
>>> try:
...     try:
...         lumberjack()
...     except:
...         1/0
... except Exception as e:
...     chained_exc = e
...
>>> # limit works as with the module-level functions
>>> TracebackException.from_exception(exc, limit=-2).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

>>> # capture_locals adds local variables in frames
>>> TracebackException.from_exception(exc, limit=-2, capture_locals=True).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
    t = ("bright", "side", "of", "life")
IndexError: tuple index out of range

>>> # The *chain* kwarg to print() controls whether chained
>>> # exceptions are displayed
>>> TracebackException.from_exception(chained_exc).print()
Traceback (most recent call last):
  File "<python-input-19>", line 4, in <module>
    lumberjack()
```

```
    ~~~~~~~~~~^^
  File "<python-input-8>", line 7, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
  File "<python-input-8>", line 11, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
ZeroDivisionError: division by zero

>>> TracebackException.from_exception(chained_exc).print(chain=False)
Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
ZeroDivisionError: division by zero
```

□□□ #2
□ □□□ □□□ 2026-10-31 17:22:28 UTC
□ □□□ □□□ 2026-10-31 17:22:30 UTC