:mod:`!contextlib` --- Utilities for :keyword:`!with`-statement contexts

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 1); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 1); *backlink*

Unknown interpreted text role "keyword".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 4)

Unknown directive type "module".

```
.. module:: contextlib
   :synopsis: Utilities for with-statement contexts.
```

Source code: :source:`Lib/contextlib.py`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 7); *backlink*

Unknown interpreted text role "source".

This module provides utilities for common tasks involving the :keyword:`with` statement. For more information see also :ref:`typecontextmanager` and :ref:`context-managers`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 11); *backlink*

Unknown interpreted text role "keyword".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 11); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 11); *backlink*

Unknown interpreted text role "ref".

Utilities

Functions and classes provided:

An :term:`abstract base class` for classes that implement :meth:`object.__enter__` and :meth:`object.__exit__`. A default implementation for :meth:`object.__enter__` is provided which returns `self` while :meth:`object.__exit__` is an abstract method which by default returns `None`. See also the definition of :ref:`typecontextmanager`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 23); *backlink*

Unknown interpreted text role "term".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 23); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 23); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 23); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 23); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 23); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 29)

Unknown directive type "versionadded".

```
.. versionadded:: 3.6
```

An :term:`abstract base class` for classes that implement :meth:`object.__aenter__` and :meth:`object.__aexit__`. A default implementation for :meth:`object.__aenter__` is provided which returns `self` while :meth:`object.__aexit__` is an abstract method which by default returns `None`. See also the definition of :ref:`async-context-managers`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 34); *backlink*

Unknown interpreted text role "term".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 34); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 34); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 34); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 34); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 34); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 41)

Unknown directive type "versionadded".

```
.. versionadded:: 3.7
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 44)

Unknown directive type "decorator".

```
.. decorator:: contextmanager

   This function is a :term:`decorator` that can be used to define a factory
   function for :keyword:`with` statement context managers, without needing to
   create a class or separate :meth:`__enter__` and :meth:`__exit__` methods.

   While many objects natively support use in with statements, sometimes a
   resource needs to be managed that isn't a context manager in its own right,
   and doesn't implement a ``close()`` method for use with ``contextlib.closing``

   An abstract example would be the following to ensure correct resource
   management::

      from contextlib import contextmanager

      @contextmanager
      def managed_resource(*args, **kwds):
          # Code to acquire resource, e.g.:
          resource = acquire_resource(*args, **kwds)
          try:
              yield resource
          finally:
              # Code to release resource, e.g.:
              release_resource(resource)

      >>> with managed_resource(timeout=3600) as resource:
      ...     # Resource is released at the end of this block,
      ...     # even if code in the block raises an exception

   The function being decorated must return a :term:`generator`-iterator when
   called. This iterator must yield exactly one value, which will be bound to
   the targets in the :keyword:`with` statement's :keyword:`!as` clause, if any.

   At the point where the generator yields, the block nested in the :keyword:`with`
   statement is executed.  The generator is then resumed after the block is exited.
   If an unhandled exception occurs in the block, it is reraised inside the
   generator at the point where the yield occurred.  Thus, you can use a
   :keyword:`try`...\ :keyword:`except`...\ :keyword:`finally` statement to trap
   the error (if any), or ensure that some cleanup takes place. If an exception is
   trapped merely in order to log it or to perform some action (rather than to
   suppress it entirely), the generator must reraise that exception. Otherwise the
   generator context manager will indicate to the :keyword:`!with` statement that
   the exception has been handled, and execution will resume with the statement
   immediately following the :keyword:`!with` statement.

   :func:`contextmanager` uses :class:`ContextDecorator` so the context managers
   it creates can be used as decorators as well as in :keyword:`with` statements.
   When used as a decorator, a new generator instance is implicitly created on
   each function call (this allows the otherwise "one-shot" context managers
   created by :func:`contextmanager` to meet the requirement that context
   managers support multiple invocations in order to be used as decorators).

   .. versionchanged:: 3.2
      Use of :class:`ContextDecorator`.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 100)

Unknown directive type "decorator".

```
.. decorator:: asynccontextmanager

   Similar to :func:`~contextlib.contextmanager`, but creates an
   :ref:`asynchronous context manager <async-context-managers>`.
```

```
   This function is a :term:`decorator` that can be used to define a factory
   function for :keyword:`async with` statement asynchronous context managers,
   without needing to create a class or separate :meth:`__aenter__` and
   :meth:`__aexit__` methods. It must be applied to an :term:`asynchronous
   generator` function.

   A simple example::

      from contextlib import asynccontextmanager

      @asynccontextmanager
      async def get_connection():
          conn = await acquire_db_connection()
          try:
              yield conn
          finally:
              await release_db_connection(conn)

      async def get_all_users():
          async with get_connection() as conn:
              return conn.query('SELECT ...')

   .. versionadded:: 3.7

   Context managers defined with :func:`asynccontextmanager` can be used
   either as decorators or with :keyword:`async with` statements::

     import time
     from contextlib import asynccontextmanager

     @asynccontextmanager
     async def timeit():
         now = time.monotonic()
         try:
             yield
         finally:
             print(f'it took {time.monotonic() - now}s to run')

      @timeit()
      async def main():
          # ... async code ...

   When used as a decorator, a new generator instance is implicitly created on
   each function call. This allows the otherwise "one-shot" context managers
   created by :func:`asynccontextmanager` to meet the requirement that context
   managers support multiple invocations in order to be used as decorators.

  .. versionchanged:: 3.10
     Async context managers created with :func:`asynccontextmanager` can
     be used as decorators.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 157)

Unknown directive type "function".

```
.. function:: closing(thing)

   Return a context manager that closes *thing* upon completion of the block.  This
   is basically equivalent to::

      from contextlib import contextmanager

      @contextmanager
      def closing(thing):
          try:
              yield thing
          finally:
              thing.close()

   And lets you write code like this::

      from contextlib import closing
      from urllib.request import urlopen

      with closing(urlopen('https://www.python.org')) as page:
          for line in page:
```

```
                print(line)

    without needing to explicitly close ``page``.  Even if an error occurs,
    ``page.close()`` will be called when the :keyword:`with` block is exited.
```

Return an async context manager that calls the `aclose()` method of *thing* upon completion of the block. This is basically equivalent to:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def aclosing(thing):
    try:
        yield thing
    finally:
        await thing.aclose()
```

Significantly, `aclosing()` supports deterministic cleanup of async generators when they happen to exit early by :keyword:`break` or an exception. For example:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 198); *backlink*

Unknown interpreted text role "keyword".

```
from contextlib import aclosing

async with aclosing(my_generator()) as values:
    async for value in values:
        if value == 42:
            break
```

This pattern ensures that the generator's async exit code is executed in the same context as its iterations (so that exceptions and context variables work as expected, and the exit code isn't run after the lifetime of some task it depends on).

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 214)

Unknown directive type "versionadded".

```
.. versionadded:: 3.10
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 219)

Unknown directive type "function".

```
.. function:: nullcontext(enter_result=None)

   Return a context manager that returns *enter_result* from ``__enter__``, but
   otherwise does nothing. It is intended to be used as a stand-in for an
   optional context manager, for example::

      def myfunction(arg, ignore_exceptions=False):
          if ignore_exceptions:
              # Use suppress to ignore all exceptions.
              cm = contextlib.suppress(Exception)
          else:
              # Do not ignore any exceptions, cm has no effect.
              cm = contextlib.nullcontext()
          with cm:
              # Do something

   An example using *enter_result*::

      def process_file(file_or_path):
          if isinstance(file_or_path, str):
              # If string, open file
              cm = open(file_or_path)
          else:
              # Caller is responsible for closing file
              cm = nullcontext(file_or_path)

          with cm as file:
```

```
            # Perform processing on the file

    It can also be used as a stand-in for
    :ref:`asynchronous context managers <async-context-managers>`::

        async def send_http(session=None):
            if not session:
                # If no http session, create it with aiohttp
                cm = aiohttp.ClientSession()
            else:
                # Caller is responsible for closing the session
                cm = nullcontext(session)

            async with cm as session:
                # Send http requests with session

    .. versionadded:: 3.7

    .. versionchanged:: 3.10
       :term:`asynchronous context manager` support was added.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 269)

Unknown directive type "function".

```
.. function:: suppress(*exceptions)

   Return a context manager that suppresses any of the specified exceptions
   if they occur in the body of a :keyword:`!with` statement and then
   resumes execution with the first statement following the end of the
   :keyword:`!with` statement.

   As with any other mechanism that completely suppresses exceptions, this
   context manager should be used only to cover very specific errors where
   silently continuing with program execution is known to be the right
   thing to do.

   For example::

       from contextlib import suppress

       with suppress(FileNotFoundError):
           os.remove('somefile.tmp')

       with suppress(FileNotFoundError):
           os.remove('someotherfile.tmp')

   This code is equivalent to::

       try:
           os.remove('somefile.tmp')
       except FileNotFoundError:
           pass

       try:
           os.remove('someotherfile.tmp')
       except FileNotFoundError:
           pass

   This context manager is :ref:`reentrant <reentrant-cms>`.

   .. versionadded:: 3.4
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 308)

Unknown directive type "function".

```
.. function:: redirect_stdout(new_target)

   Context manager for temporarily redirecting :data:`sys.stdout` to
   another file or file-like object.
```

```
This tool adds flexibility to existing functions or classes whose output
is hardwired to stdout.

For example, the output of :func:`help` normally is sent to *sys.stdout*.
You can capture that output in a string by redirecting the output to an
:class:`io.StringIO` object. The replacement stream is returned from the
``__enter__`` method and so is available as the target of the
:keyword:`with` statement::

     with redirect_stdout(io.StringIO()) as f:
         help(pow)
     s = f.getvalue()

To send the output of :func:`help` to a file on disk, redirect the output
to a regular file::

     with open('help.txt', 'w') as f:
         with redirect_stdout(f):
             help(pow)

To send the output of :func:`help` to *sys.stderr*::

     with redirect_stdout(sys.stderr):
         help(pow)

Note that the global side effect on :data:`sys.stdout` means that this
context manager is not suitable for use in library code and most threaded
applications. It also has no effect on the output of subprocesses.
However, it is still a useful approach for many utility scripts.

This context manager is :ref:`reentrant <reentrant-cms>`.

.. versionadded:: 3.4
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 348)

Unknown directive type "function".

```
.. function:: redirect_stderr(new_target)

   Similar to :func:`~contextlib.redirect_stdout` but redirecting
   :data:`sys.stderr` to another file or file-like object.

   This context manager is :ref:`reentrant <reentrant-cms>`.

   .. versionadded:: 3.5
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 358)

Unknown directive type "function".

```
.. function:: chdir(path)

   Non parallel-safe context manager to change the current working directory.
   As this changes a global state, the working directory, it is not suitable
   for use in most threaded or async contexts. It is also not suitable for most
   non-linear code execution, like generators, where the program execution is
   temporarily relinquished -- unless explicitely desired, you should not yield
   when this context manager is active.

   This is a simple wrapper around :func:`~os.chdir`, it changes the current
   working directory upon entering and restores the old one on exit.

   This context manager is :ref:`reentrant <reentrant-cms>`.

   .. versionadded:: 3.11
```

A base class that enables a context manager to also be used as a decorator.

Context managers inheriting from `ContextDecorator` have to implement `__enter__` and `__exit__` as normal. `__exit__` retains its optional exception handling even when used as a decorator.

`ContextDecorator` is used by :func:`contextmanager`, so you get this functionality automatically.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 383); *backlink***
>
> Unknown interpreted text role "func".

Example of `ContextDecorator`:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False

>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

This change is just syntactic sugar for any construct of the following form:

```
def f():
    with cm():
        # Do stuff
```

`ContextDecorator` lets you instead write:

```
@cm()
def f():
    # Do stuff
```

It makes it clear that the `cm` applies to the whole function, rather than just a piece of it (and saving an indentation level is nice, too).

Existing context managers that already have a base class can be extended by using `ContextDecorator` as a mixin class:

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

> **Note**
>
> As the decorated function must be able to be called multiple times, the underlying context manager must support use in multiple :keyword:`with` statements. If this is not the case, then the original construct with the explicit :keyword:`!with` statement inside the function should be used.
>
> > **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 443); *backlink***
> >
> > Unknown interpreted text role "keyword".
>
> > **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 443); *backlink***

Unknown interpreted text role "keyword".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 448)

Unknown directive type "versionadded".

```
.. versionadded:: 3.2
```

Similar to :class:`ContextDecorator` but only for asynchronous functions.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 453); *backlink*

Unknown interpreted text role "class".

Example of `AsyncContextDecorator`:

```
from asyncio import run
from contextlib import AsyncContextDecorator

class mycontext(AsyncContextDecorator):
    async def __aenter__(self):
        print('Starting')
        return self

    async def __aexit__(self, *exc):
        print('Finishing')
        return False

>>> @mycontext()
... async def function():
...     print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing

>>> async def function():
...    async with mycontext():
...         print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 487)

Unknown directive type "versionadded".

```
.. versionadded:: 3.10
```

A context manager that is designed to make it easy to programmatically combine other context managers and cleanup functions, especially those that are optional or otherwise driven by input data.

For example, a set of files may easily be handled in a single with statement as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

The :meth:`__enter__` method returns the :class:`ExitStack` instance, and performs no additional operations.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 505); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 505); *backlink*

Unknown interpreted text role "class".

Each instance maintains a stack of registered callbacks that are called in reverse order when the instance is closed (either explicitly or implicitly at the end of a :keyword:`with` statement). Note that callbacks are *not* invoked implicitly when the context stack instance is garbage collected.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 508); *backlink*

Unknown interpreted text role "keyword".

This stack model is used so that context managers that acquire their resources in their `__init__` method (such as file objects) can be handled correctly.

Since registered callbacks are invoked in the reverse order of registration, this ends up behaving as if multiple nested :keyword:`with` statements had been used with the registered set of callbacks. This even extends to exception handling - if an inner callback suppresses or replaces an exception, then outer callbacks will be passed arguments based on that updated state.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 517); *backlink*

Unknown interpreted text role "keyword".

This is a relatively low level API that takes care of the details of correctly unwinding the stack of exit callbacks. It provides a suitable foundation for higher level context managers that manipulate the exit stack in application specific ways.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 529)

Unknown directive type "versionadded".

```
.. versionadded:: 3.3
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 531)

Unknown directive type "method".

```
.. method:: enter_context(cm)

   Enters a new context manager and adds its :meth:`__exit__` method to
   the callback stack. The return value is the result of the context
   manager's own :meth:`__enter__` method.

   These context managers may suppress exceptions just as they normally
   would if used directly as part of a :keyword:`with` statement.

   .. versionchanged:: 3.11
      Raises :exc:`TypeError` instead of :exc:`AttributeError` if *cm*
      is not a context manager.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 544)

Unknown directive type "method".

```
.. method:: push(exit)

   Adds a context manager's :meth:`__exit__` method to the callback stack.

   As ``__enter__`` is *not* invoked, this method can be used to cover
   part of an :meth:`__enter__` implementation with a context manager's own
   :meth:`__exit__` method.

   If passed an object that is not a context manager, this method assumes
   it is a callback with the same signature as a context manager's
   :meth:`__exit__` method and adds it directly to the callback stack.
```

```
By returning true values, these callbacks can suppress exceptions the
same way context manager :meth:`__exit__` methods can.

The passed in object is returned from the function, allowing this
method to be used as a function decorator.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 562)

Unknown directive type "method".

```
.. method:: callback(callback, /, *args, **kwds)

   Accepts an arbitrary callback function and arguments and adds it to
   the callback stack.

   Unlike the other methods, callbacks added this way cannot suppress
   exceptions (as they are never passed the exception details).

   The passed in callback is returned from the function, allowing this
   method to be used as a function decorator.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 573)

Unknown directive type "method".

```
.. method:: pop_all()

   Transfers the callback stack to a fresh :class:`ExitStack` instance
   and returns it. No callbacks are invoked by this operation - instead,
   they will now be invoked when the new stack is closed (either
   explicitly or implicitly at the end of a :keyword:`with` statement).

   For example, a group of files can be opened as an "all or nothing"
   operation as follows::

      with ExitStack() as stack:
          files = [stack.enter_context(open(fname)) for fname in filenames]
          # Hold onto the close method, but don't call it yet.
          close_files = stack.pop_all().close
          # If opening any file fails, all previously opened files will be
          # closed automatically. If all files are opened successfully,
          # they will remain open even after the with statement ends.
          # close_files() can then be invoked explicitly to close them all.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 592)

Unknown directive type "method".

```
.. method:: close()

   Immediately unwinds the callback stack, invoking callbacks in the
   reverse order of registration. For any context managers and exit
   callbacks registered, the arguments passed in will indicate that no
   exception occurred.
```

An :ref:`asynchronous context manager <async-context-managers>`, similar to :class:`ExitStack`, that supports combining both synchronous and asynchronous context managers, as well as having coroutines for cleanup logic.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 601); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 601); *backlink*

Unknown interpreted text role "class".

The :meth:`close` method is not implemented, :meth:`aclose` must be used instead.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 606); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 606); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 609)

Unknown directive type "coroutinemethod".

```
.. coroutinemethod:: enter_async_context(cm)

   Similar to :meth:`enter_context` but expects an asynchronous context
   manager.

   .. versionchanged:: 3.11
      Raises :exc:`TypeError` instead of :exc:`AttributeError` if *cm*
      is not an asynchronous context manager.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 618)

Unknown directive type "method".

```
.. method:: push_async_exit(exit)

   Similar to :meth:`push` but expects either an asynchronous context manager
   or a coroutine function.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 623)

Unknown directive type "method".

```
.. method:: push_async_callback(callback, /, *args, **kwds)

   Similar to :meth:`callback` but expects a coroutine function.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 627)

Unknown directive type "coroutinemethod".

```
.. coroutinemethod:: aclose()

   Similar to :meth:`close` but properly handles awaitables.
```

Continuing the example for :func:`asynccontextmanager`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 631); *backlink*

Unknown interpreted text role "func".

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
        for i in range(5)]
    # All opened connections will automatically be released at the end of
    # the async with statement, even if attempts to open a connection
    # later in the list raise an exception.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 640)

Unknown directive type "versionadded".

```
.. versionadded:: 3.7
```

Examples and Recipes

This section describes some examples and recipes for making effective use of the tools provided by :mod:`contextlib`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 645); *backlink*

Unknown interpreted text role "mod".

Supporting a variable number of context managers

The primary use case for :class:`ExitStack` is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single :keyword:`with` statement. The variability may come from the number of context managers needed being driven by user input (such as opening a user specified collection of files), or from some of the context managers being optional:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 652); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 652); *backlink*

Unknown interpreted text role "keyword".

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

As shown, :class:`ExitStack` also makes it quite easy to use :keyword:`with` statements to manage arbitrary resources that don't natively support the context management protocol.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 667); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 667); *backlink*

Unknown interpreted text role "keyword".

Catching exceptions from ``__enter__`` methods

It is occasionally desirable to catch exceptions from an ``__enter__`` method implementation, *without* inadvertently catching exceptions from the :keyword:`with` statement body or the context manager's ``__exit__`` method. By using :class:`ExitStack` the steps in the context management protocol can be separated slightly in order to allow this:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 675); *backlink*

Unknown interpreted text role "keyword".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 675); backlink

Unknown interpreted text role "class".

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

Actually needing to do this is likely to indicate that the underlying API should be providing a direct resource management interface for use with :keyword:`try`/:keyword:`except`/:keyword:`finally` statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then :class:`ExitStack` can make it easier to handle various situations that can't be handled directly in a :keyword:`with` statement.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 690); backlink

Unknown interpreted text role "keyword".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 690); backlink

Unknown interpreted text role "keyword".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 690); backlink

Unknown interpreted text role "keyword".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 690); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 690); backlink

Unknown interpreted text role "keyword".

Cleaning up in an `__enter__` implementation

As noted in the documentation of :meth:`ExitStack.push`, this method can be useful in cleaning up an already allocated resource if later steps in the :meth:`__enter__` implementation fail.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 702); backlink

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 702); backlink

Unknown interpreted text role "meth".

Here's an example of doing this for a context manager that accepts resource acquisition and release functions, along with an optional validation function, and maps them to the context management protocol:

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
```

```
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

Replacing any use of `try-finally` and flag variables

A pattern you will sometimes see is a `try-finally` statement with a flag variable to indicate whether or not the body of the `finally` clause should be executed. In its simplest form (that can't already be handled just by using an `except` clause instead), it looks something like this:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

As with any `try` statement based code, this can cause problems for development and review, because the setup code and the cleanup code can end up being separated by arbitrarily long sections of code.

:class:`ExitStack` makes it possible to instead register a callback for execution at the end of a `with` statement, and then later decide to skip executing that callback:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 766); *backlink***
>
> Unknown interpreted text role "class".

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

This allows the intended cleanup up behaviour to be made explicit up front, rather than requiring a separate flag variable.

If a particular application uses this pattern a lot, it can be simplified even further by means of a small helper class:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, /, *args, **kwds):
        super().__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

If the resource cleanup isn't already neatly bundled into a standalone function, then it is still possible to use the decorator form of :meth:`ExitStack.callback` to declare the resource cleanup in advance:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 799); *backlink***
>
> Unknown interpreted text role "meth".

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

Due to the way the decorator protocol works, a callback function declared this way cannot take any parameters. Instead, any resources to be released must be accessed as closure variables.

Using a context manager as a function decorator

:class:`ContextDecorator` makes it possible to use a context manager in both an ordinary `with` statement and also as a function decorator.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 822); *backlink***
>
> Unknown interpreted text role "class".

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, inheriting from :class:`ContextDecorator` provides both capabilities in a single definition:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 825); *backlink***
>
> Unknown interpreted text role "class".

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Instances of this class can be used as both a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

And also as a function decorator:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Note that there is one additional limitation when using context managers as function decorators: there's no way to access the return value of :meth:`__enter__`. If that value is needed, then it is still necessary to use an explicit `with` statement.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 859); *backlink***

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 864)

Unknown directive type "seealso".

```
.. seealso::

   :pep:`343` - The "with" statement
      The specification, background, and examples for the Python :keyword:`with`
      statement.
```

Single use, reusable and reentrant context managers

Most context managers are written in a way that means they can only be used effectively in a :keyword:`with` statement once. These single use context managers must be created afresh each time they're used - attempting to use them a second time will trigger an exception or otherwise not work correctly.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 875); *backlink*

Unknown interpreted text role "keyword".

This common limitation means that it is generally advisable to create context managers directly in the header of the :keyword:`with` statement where they are used (as shown in all of the usage examples above).

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 881); *backlink*

Unknown interpreted text role "keyword".

Files are an example of effectively single use context managers, since the first :keyword:`with` statement will close the file, preventing any further IO operations using that file object.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 885); *backlink*

Unknown interpreted text role "keyword".

Context managers created using :func:`contextmanager` are also single use context managers, and will complain about the underlying generator failing to yield if an attempt is made to use them a second time:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 889); *backlink*

Unknown interpreted text role "func".

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

Reentrant context managers

More sophisticated context managers may be "reentrant". These context managers can not only be used in multiple :keyword:`with` statements, but may also be used *inside* a :keyword:`!with` statement that is already using the same context manager.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 919); *backlink*

Unknown interpreted text role "keyword".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 919); *backlink*

Unknown interpreted text role "keyword".

:class:`threading.RLock` is an example of a reentrant context manager, as are :func:`suppress`, :func:`redirect_stdout`, and :func:`chdir`. Here's a very simple example of reentrant use:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 924); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 924); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 924); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 924); *backlink*

Unknown interpreted text role "func".

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

Real world examples of reentrancy are more likely to involve multiple functions calling each other and hence be far more complicated than this example.

Note also that being reentrant is *not* the same thing as being thread safe. :func:`redirect_stdout`, for example, is definitely not thread safe, as it makes a global modification to the system state by binding :data:`sys.stdout` to a different stream.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 947); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 947); *backlink*

Unknown interpreted text role "data".

Reusable context managers

Distinct from both single use and reentrant context managers are "reusable" context managers (or, to be completely explicit, "reusable, but not reentrant" context managers, since reentrant context managers are also reusable). These context managers support being used multiple times, but will fail (or otherwise not work correctly) if the specific context manager instance has already been used in a containing with statement.

:class:`threading.Lock` is an example of a reusable, but not reentrant, context manager (for a reentrant lock, it is necessary to use :class:`threading.RLock` instead).

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 965); *backlink***
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 965); *backlink***
>
> Unknown interpreted text role "class".

Another example of a reusable, but not reentrant, context manager is :class:`ExitStack`, as it invokes *all* currently registered callbacks when leaving any with statement, regardless of where those callbacks were added:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 969); *backlink***
>
> Unknown interpreted text role "class".

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

As the output from the example shows, reusing a single stack object across multiple with statements works correctly, but attempting to nest them will cause the stack to be cleared at the end of the innermost with statement, which is unlikely to be desirable behaviour.

Using separate :class:`ExitStack` instances instead of reusing a single instance avoids that problem:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)contextlib.rst, line 1005); *backlink***
>
> Unknown interpreted text role "class".

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```