28.6. `contextlib` — Utilities for `with`-statement contexts¶

New in version 2.5.

This module provides utilities for common tasks involving the `with` statement. For more information see also *Context Manager Types* and *With Statement Context Managers*.

Functions provided:

`contextlib.contextmanager`(*func*)¶

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

A simple example (this is not recommended as a real way of generating HTML!):

```
from contextlib import contextmanager

@contextmanager
def tag(name):
   print "<%s>" % name
   yield
   print "</%s>" % name

>>> with tag("h1"):
...    print "foo"
...
<h1>
foo
</h1>
```

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

At the point where the generator yields, the block nested in the `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 `try`...`except`...`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 `with` statement that the exception has been handled, and execution will resume with the statement immediately following the `with` statement.

`contextlib.nested`(*mgr1*[, *mgr2*[, *...*]])¶

Combine multiple context managers into a single nested context manager.

Code like this:

```
from contextlib import nested

with nested(A(), B(), C()) as (X, Y, Z):
   do_something()
```

is equivalent to this:

```
m1, m2, m3 = A(), B(), C()
with m1 as X:
   with m2 as Y:
       with m3 as Z:
           do_something()
```

Note that if the `__exit__()` method of one of the nested context managers indicates an exception should be suppressed, no exception information will be passed to any remaining outer context managers. Similarly, if the `__exit__()` method of one of the nested managers raises an exception, any previous exception state will be lost; the new exception will be passed to the `__exit__()` methods of any remaining outer context managers. In general, `__exit__()` methods should avoid raising exceptions, and in particular they should not re-raise a passed-in exception.

`contextlib.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
import urllib

with closing(urllib.urlopen('http://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 `with` block is exited.

See also

PEP 0343 - The “with” statement
The specification, background, and examples for the Python `with` statement.

Previous topic

28.5. `warnings` — Warning control

Next topic

28.7. `abc` — Abstract Base Classes

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 28. Python Runtime Services »