29.6. contextlib — Utilities for with-statement contexts

Source code: Lib/contextlib.py

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

29.6.1. Utilities

Functions and classes provided:

@ contextlib. **contextmanager**

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.

contextmanager() uses ContextDecorator so the context managers it creates can be used as decorators as well as in 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 contextmanager() to meet the requirement that context managers support multiple invocations in order to be used as decorators).

Changed in version 3.2: Use of ContextDecorator.

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
from urllib.request import urlopen

with closing(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.

contextlib. **suppress**(*exceptions*)

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

New in version 3.4.

contextlib.**redirect_stdout**(*new_target*)

Context manager for temporarily redirecting 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 help() normally is sent to *sys.stdout*. You can capture that output in a string by redirecting the output to an io.StringIO object:

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

To send the output of 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 help() to *sys.stderr*:

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

Note that the global side effect on 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 reentrant.

New in version 3.4.

contextlib. **redirect_stderr**(*new_target*)

Similar to redirect_stdout() but redirecting sys.stderr to another file or file-like object.

This context manager is reentrant.

New in version 3.5.

class contextlib. **ContextDecorator**

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 contextmanager(), so you get this functionality automatically.

Example of ContextDecorator:

```python
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 with statements. If this is not the case, then the original construct with the explicit with statement inside the function should be used.

New in version 3.2.

class contextlib.**ExitStack**

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 filenar
    # 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
```

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 with statement). Note that callbacks are *not* invoked implicitly when the context stack instance is garbage collected.

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 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.

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.

New in version 3.3.

enter_context(*cm*)

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

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

push(*exit*)

Adds a context manager's __exit__() method to the callback stack.

As __enter__ is *not* invoked, this method can be used to cover part of an __enter__() implementation with a context manager's own __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 __exit__() method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager __exit__() methods can.

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

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.

pop_all()

Transfers the callback stack to a fresh 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 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 file
    # 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.
```

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.

29.6.2. Examples and Recipes

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

29.6.2.1. Supporting a variable number of context managers

The primary use case for ExitStack is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single 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:

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

29.6.2.2. Simplifying support for single optional context managers

In the specific case of a single optional context manager, ExitStack instances can be used as a "do nothing" context manager, allowing a context manager to easily be omitted without affecting the overall structure of the source code:

```python
def debug_trace(details):
    if __debug__:
        return TraceContext(details)
    # Don't do anything special with the context in release mode
    return ExitStack()

with debug_trace():
    # Suite is traced in debug mode, but runs normally otherwise
```

29.6.2.3. Catching exceptions from __enter__ methods

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

```python
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 try/except/finally statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then ExitStack can make it easier to handle various situations that can't be handled directly in a with statement.

29.6.2.4. Cleaning up in an __enter__ implementation

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

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:

```python
from contextlib import contextmanager, ExitStack

class ResourceManager:

    def __init__(self, acquire_resource, release_resource, check_res
        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()
```

29.6.2.5. 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:

```python
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.

`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:

```python
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:

```python
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, *args, **kwds):
        super(Callback, self).__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 ExitStack.callback() to declare the resource cleanup in advance:

```python
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.

29.6.2.6. Using a context manager as a function decorator

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

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 ContextDecorator provides both capabilities in a single definition:

```python
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: {}'.format(self.name))

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: {}'.format(self.name))
```

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

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

And also as a function decorator:

```python
@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 __enter__(). If that value is needed, then it is still necessary to use an explicit with statement.

29.6.3. 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 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.

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

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

Context managers created using 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:

```
>>> 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
```

29.6.3.1. Reentrant context managers

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

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

```
>>> 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. redirect_stdout(), for example, is definitely not thread safe, as it makes a global modification to the system state by binding sys.stdout to a different stream.

29.6.3.2. 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.

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

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

```
>>> 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 ExitStack instances instead of reusing a single instance avoids that problem:

```
>>> 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
```