contextlib2 Documentation

Release 0.5.0

Nick Coghlan

January 12, 2016

Contents

1 Additions Relative to the Standard Library **3**

2 API Reference **5**

3 Examples and Recipes **11**
3.1 Cleaning up in an `__enter__` implementation 11
3.2 Replacing any use of `try-finally` and flag variables 12
3.3 Using a context manager as a function decorator 13

4 Context Management Concepts **15**
4.1 Single use, reusable and reentrant context managers 15

5 Obtaining the Module **17**
5.1 Development and Support 17
5.2 Release History 17

6 Indices and tables **19**

Python Module Index **21**

This module provides backports of features in the latest version of the standard library's `contextlib` module to earlier Python versions. It also serves as a real world proving ground for potential future enhancements to that module.

Like `contextlib`, this module provides utilities for common tasks involving the `with` statement.

CHAPTER 1

Additions Relative to the Standard Library

This module is primarily a backport of the Python 3.5 version of `contextlib` to earlier releases. However, it is also a proving ground for new features not yet part of the standard library.

There are currently no such features in the module.

Refer to the `contextlib` documentation for details of which versions of Python 3 introduce the various APIs provided in this module.

CHAPTER 2

API Reference

Functions and classes provided:

@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).

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.

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 0.5: Part of the standard library in Python 3.4 and later

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 a `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 0.5: Part of the standard library in Python 3.4 and later

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 0.5: Part of the standard library in Python 3.5 and later

class 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`:

```
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.

---

class `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 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
```

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 0.4: Part of the standard library in Python 3.3 and later

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 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.
```

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.

CHAPTER 3

Examples and Recipes

This section describes some examples and recipes for making effective use of the tools provided by *contextlib2*. Some of them may also work with `contextlib` in sufficiently recent versions of Python. When this is the case, it is noted at the end of the example.

3.1 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:

```
from contextlib2 import ExitStack

class ResourceManager(object):

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

    def __enter__(self):
        resource = self.acquire_resource()
        if self.check_resource_ok is not None:
            with ExitStack() as stack:
                stack.push(self)
                if not self.check_resource_ok(resource):
                    msg = "Failed validation for {!r}"
                    raise RuntimeError(msg.format(resource))
                # 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()
        return resource

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

This example will also work with `contextlib` in Python 3.3 or later.

3.2 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.

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:

```
from contextlib2 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 you find yourself using this pattern a lot, it can be simplified even further by means of a small helper class:

```
from contextlib2 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:

```
from contextlib2 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.

This example will also work with `contextlib` in Python 3.3 or later.

3.3 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. The `ContextDecorator.refresh_cm()` method even makes it possible to use otherwise single use context managers (such as those created by *`contextmanager()`*) that way.

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, *`contextmanager()`* provides both capabilities in a single definition:

```
from contextlib2 import contextmanager
import logging

logging.basicConfig(level=logging.INFO)

@contextmanager
def track_entry_and_exit(name):
    logging.info('Entering: {}'.format(name))
    yield
    logging.info('Exiting: {}'.format(name))
```

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

This example will also work with `contextlib` in Python 3.2.1 or later.

CHAPTER 4

Context Management Concepts

4.1 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
```

4.1.1 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 is *`suppress()`*. Here's a toy example of reentrant use (real world examples of reentrancy are more likely to occur with objects like recursive locks and are likely to be far more complicated than this example):

```
>>> from contextlib import suppress
>>> ignore_raised_exception = suppress(ZeroDivisionError)
>>> with ignore_raised_exception:
...     with ignore_raised_exception:
...         1/0
...     print("This line runs")
...     1/0
...     print("This is skipped")
...
This line runs
>>> # The second exception is also suppressed
```

4.1.2 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.

An example of a reusable context manager is *`redirect_stdout()`*:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> f = StringIO()
>>> collect_output = redirect_stdout(f)
>>> with collect_output:
...     print("Collected")
...
>>> print("Not collected")
Not collected
>>> with collect_output:
...     print("Also collected")
...
>>> print(f.getvalue())
Collected
Also collected
```

However, this context manager is not reentrant, so attempting to reuse it within a containing with statement fails:

```
>>> with collect_output:
...     # Nested reuse is not permitted
...     with collect_output:
...         pass
...
Traceback (most recent call last):
  ...
RuntimeError: Cannot reenter <...>
```

CHAPTER 5

Obtaining the Module

This module can be installed directly from the Python Package Index with pip:

```
pip install contextlib2
```

Alternatively, you can download and unpack it manually from the contextlib2 PyPI page.

There are no operating system or distribution specific versions of this module - it is a pure Python module that should work on all platforms.

Supported Python versions are currently 2.7 and 3.2+.

5.1 Development and Support

contextlib2 is developed and maintained on BitBucket. Problems and suggested improvements can be posted to the issue tracker.

5.2 Release History

- Updated to include all features from the Python 3.4 and 3.5 releases of contextlib (also includes some `ExitStack` enhancements made following the integration into the standard library for Python 3.3)
- The legacy `ContextStack` and `ContextDecorator.refresh_cm` APIs are no longer documented and emit `DeprecationWarning` when used
- Python 2.6, 3.2 and 3.3 have been dropped from compatibility testing
- Issue #8: Replace ContextStack with ExitStack (old ContextStack API retained for backwards compatibility)
- Fall back to unittest2 if unittest is missing required functionality
- Issue #7: Add MANIFEST.in so PyPI package contains all relevant files
- Issue #5: ContextStack.register no longer pointlessly returns the wrapped function
- Issue #2: Add examples and recipes section to docs
- Issue #3: ContextStack.register_exit() now accepts objects with __exit__ attributes in addition to accepting exit callbacks directly
- Issue #1: Add ContextStack.preserve() to move all registered callbacks to a new ContextStack object
- Wrapped callbacks now expose __wrapped__ (for direct callbacks) or __self__ (for context manager methods) attributes to aid in introspection

- Moved version number to a VERSION.txt file (read by both docs and setup.py)
- Added NEWS.rst (and incorporated into documentation)
- Renamed CleanupManager to ContextStack (hopefully before anyone started using the module for anything, since I didn't alias the old name at all)
- Initial release as a backport module
- Added CleanupManager (based on a Python feature request)
- Added ContextDecorator.refresh_cm() (based on a Python tracker issue)

CHAPTER 6

Indices and tables

- genindex
- search

C

contextlib2, 3

C

callback() (ExitStack method), 9
close() (ExitStack method), 9
closing() (in module contextlib2), 5
ContextDecorator (class in contextlib2), 7
contextlib2 (module), 1
contextmanager() (in module contextlib2), 5

E

enter_context() (ExitStack method), 9
ExitStack (class in contextlib2), 8

P

pop_all() (ExitStack method), 9
push() (ExitStack method), 9

R

redirect_stderr() (in module contextlib2), 7
redirect_stdout() (in module contextlib2), 6

S

suppress() (in module contextlib2), 6