contextlib — Context Manager Utilities

Purpose: Utilities for creating and working with context managers.

The `contextlib` module contains utilities for working with context managers and the `with` statement.

Context Manager API

A *context manager* is responsible for a resource within a code block, possibly creating it when the block is entered and then cleaning it up after the block is exited. For example, files support the context manager API to make it easy to ensure they are closed after all reading or writing is done.

```
# contextlib_file.py

with open('/tmp/pymotw.txt', 'wt') as f:
    f.write('contents go here')
# file is automatically closed
```

A context manager is enabled by the `with` statement, and the API involves two methods. The `__enter__()` method is run when execution flow enters the code block inside the `with`. It returns an object to be used within the context. When execution flow leaves the `with` block, the `__exit__()` method of the context manager is called to clean up any resources being used.

```
# contextlib_api.py

class Context:

    def __init__(self):
        print('__init__()')

    def __enter__(self):
        print('__enter__()')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__()')


with Context():
    print('Doing work in the context')
```

Combining a context manager and the `with` statement is a more compact way of writing a `try:finally` block, since the context manager’s `__exit__()` method is always called, even if an exception is raised.

```
$ python3 contextlib_api.py

__init__()
__enter__()
Doing work in the context
__exit__()
```

The `__enter__()` method can return any object to be associated with a name specified in the `as` clause of the `with` statement. In this example, the `Context` returns an object that uses the open context.

```
# contextlib_api_other_object.py

class WithinContext:

    def __init__(self, context):
        print('WithinContext.__init__({})'.format(context))

    def do_something(self):
        print('WithinContext.do_something()')
        print('WithinContext.do_something()')

    def __del__(self):
        print('WithinContext.__del__')


class Context:

    def __init__(self):
        print('Context.__init__()')

    def __enter__(self):
        print('Context.__enter__()')
        return WithinContext(self)

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('Context.__exit__()')


with Context() as c:
    c.do_something()
```

The value associated with the variable c is the object returned by `__enter__()`, which is not necessarily the `Context` instance created in the with statement.

```
$ python3 contextlib_api_other_object.py

Context.__init__()
Context.__enter__()
WithinContext.__init__(<__main__.Context object at 0x101f046d8>)
WithinContext.do_something()
Context.__exit__()
WithinContext.__del__
```

The `__exit__()` method receives arguments containing details of any exception raised in the `with` block.

```
# contextlib_api_error.py

class Context:

    def __init__(self, handle_error):
        print('__init__({})'.format(handle_error))
        self.handle_error = handle_error

    def __enter__(self):
        print('__enter__()')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__()')
        print('  exc_type =', exc_type)
        print('  exc_val  =', exc_val)
        print('  exc_tb   =', exc_tb)
        return self.handle_error


with Context(True):
    raise RuntimeError('error message handled')

print()

with Context(False):
    raise RuntimeError('error message propagated')
```

If the context manager can handle the exception, `__exit__()` should return a true value to indicate that the exception does not need to be propagated. Returning false causes the exception to be re-raised after `__exit__()` returns.

```
$ python3 contextlib_api_error.py

__init__(True)
__enter__()
__enter__()
__exit__()
  exc_type = <class 'RuntimeError'>
  exc_val  = error message handled
  exc_tb   = <traceback object at 0x101c94948>

__init__(False)
__enter__()
__exit__()
  exc_type = <class 'RuntimeError'>
  exc_val  = error message propagated
  exc_tb   = <traceback object at 0x101c94948>
Traceback (most recent call last):
  File "contextlib_api_error.py", line 34, in <module>
    raise RuntimeError('error message propagated')
RuntimeError: error message propagated
```

Context Managers as Function Decorators

The class `ContextDecorator` adds support to regular context manager classes to let them be used as function decorators as well as context managers.

```python
# contextlib_decorator.py

import contextlib


class Context(contextlib.ContextDecorator):

    def __init__(self, how_used):
        self.how_used = how_used
        print('__init__({})'.format(how_used))

    def __enter__(self):
        print('__enter__({})'.format(self.how_used))
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__({})'.format(self.how_used))


@Context('as decorator')
def func(message):
    print(message)


print()
with Context('as context manager'):
    print('Doing work in the context')

print()
func('Doing work in the wrapped function')
```

One difference with using the context manager as a decorator is that the value returned by `__enter__()` is not available inside the function being decorated, unlike when using `with` and `as`. Arguments passed to the decorated function are available in the usual way.

```
$ python3 contextlib_decorator.py

__init__(as decorator)

__init__(as context manager)
__enter__(as context manager)
Doing work in the context
__exit__(as context manager)

__enter__(as decorator)
Doing work in the wrapped function
__exit__(as decorator)
```

From Generator to Context Manager

Creating context managers the traditional way, by writing a class with `__enter__()` and `__exit__()` methods, is not difficult. But sometimes writing everything out fully is extra overhead for a trivial bit of context. In those sorts of situations, use the `contextmanager()` decorator to convert a generator function into a context manager.

```
# contextlib_contextmanager.py

import contextlib


@contextlib.contextmanager
def make_context():
    print('  entering')
    try:
        yield {}
    except RuntimeError as err:
        print('  ERROR:', err)
    finally:
        print('  exiting')


print('Normal:')
with make_context() as value:
    print('  inside with statement:', value)

print('\nHandled error:')
with make_context() as value:
    raise RuntimeError('showing example of handling an error')

print('\nUnhandled error:')
with make_context() as value:
    raise ValueError('this exception is not handled')
```

The generator should initialize the context, yield exactly one time, then clean up the context. The value yielded, if any, is bound to the variable in the as clause of the `with` statement. Exceptions from within the `with` block are re-raised inside the generator, so they can be handled there.

```
$ python3 contextlib_contextmanager.py

Normal:
  entering
  inside with statement: {}
  exiting

Handled error:
  entering
  ERROR: showing example of handling an error
  exiting

Unhandled error:
  entering
  exiting
Traceback (most recent call last):
  File "contextlib_contextmanager.py", line 33, in <module>
    raise ValueError('this exception is not handled')
ValueError: this exception is not handled
```

The context manager returned by `contextmanager()` is derived from `ContextDecorator`, so it also works as a function decorator.

```
# contextlib_contextmanager_decorator.py

import contextlib


@contextlib.contextmanager
def make_context():
    print('  entering')
    try:
        # Yield control, but not a value, because any value
        # yielded is not available when the context manager
        # is used as a decorator.
        yield
    except RuntimeError as err:
        print('  ERROR:', err)
    finally:
        print('  exiting')


@make_context()
def normal():
    print('  inside with statement')


@make_context()
def throw_error(err):
    raise err


print('Normal:')
normal()

print('\nHandled error:')
throw_error(RuntimeError('showing example of handling an error'))

print('\nUnhandled error:')
throw_error(ValueError('this exception is not handled'))
```

As in the `ContextDecorator` example above, when the context manager is used as a decorator the value yielded by the generator is not available inside the function being decorated. Arguments passed to the decorated function are still available, as demonstrated by `throw_error()` in this example.

```
$ python3 contextlib_contextmanager_decorator.py

Normal:
  entering
  inside with statement
  exiting

Handled error:
  entering
  ERROR: showing example of handling an error
  exiting

Unhandled error:
  entering
  exiting
Traceback (most recent call last):
  File "contextlib_contextmanager_decorator.py", line 43, in
<module>
    throw_error(ValueError('this exception is not handled'))
  File ".../lib/python3.7/contextlib.py", line 74, in inner
    return func(*args, **kwds)
  File "contextlib_contextmanager_decorator.py", line 33, in
throw_error
    raise err
ValueError: this exception is not handled
```

Closing Open Handles

The `file` class supports the context manager API directly, but some other objects that represent open handles do not. The example given in the standard library documentation for `contextlib` is the object returned from `urllib.urlopen()`. There are other legacy classes that use a `close()` method but do not support the context manager API. To ensure that a handle is closed, use `closing()` to create a context manager for it.

```
# contextlib_closing.py
```

```
import contextlib


class Door:

    def __init__(self):
        print('  __init__()')
        self.status = 'open'

    def close(self):
        print('  close()')
        self.status = 'closed'


print('Normal Example:')
with contextlib.closing(Door()) as door:
    print('  inside with statement: {}'.format(door.status))
print('  outside with statement: {}'.format(door.status))

print('\nError handling example:')
try:
    with contextlib.closing(Door()) as door:
        print('  raising from inside with statement')
        raise RuntimeError('error message')
except Exception as err:
    print('  Had an error:', err)
```

The handle is closed whether there is an error in the with block or not.

```
$ python3 contextlib_closing.py

Normal Example:
  __init__()
  inside with statement: open
  close()
  outside with statement: closed

Error handling example:
  __init__()
  raising from inside with statement
  close()
  Had an error: error message
```

Ignoring Exceptions

It is frequently useful to ignore exceptions raised by libraries, because the error indicates that the desired state has already been achieved, or it can otherwise be ignored. The most common way to ignore exceptions is with a `try:except` statement with only a pass statement in the except block.

```
# contextlib_ignore_error.py

import contextlib


class NonFatalError(Exception):
    pass


def non_idempotent_operation():
    raise NonFatalError(
        'The operation failed because of existing state'
    )


try:
    print('trying non-idempotent operation')
    non_idempotent_operation()
    print('succeeded!')
except NonFatalError:
    pass
    pass

print('done')
```

In this case, the operation fails and the error is ignored.

```
$ python3 contextlib_ignore_error.py

trying non-idempotent operation
done
```

The `try:except` form can be replaced with `contextlib.suppress()` to more explicitly suppress a class of exceptions happening anywhere in the `with` block.

```python
# contextlib_suppress.py

import contextlib


class NonFatalError(Exception):
    pass


def non_idempotent_operation():
    raise NonFatalError(
        'The operation failed because of existing state'
    )


with contextlib.suppress(NonFatalError):
    print('trying non-idempotent operation')
    non_idempotent_operation()
    print('succeeded!')

print('done')
```

In this updated version, the exception is discarded entirely.

```
$ python3 contextlib_suppress.py

trying non-idempotent operation
done
```

Redirecting Output Streams

Poorly designed library code may write directly to `sys.stdout` or `sys.stderr`, without providing arguments to configure different output destinations. The `redirect_stdout()` and `redirect_stderr()` context managers can be used to capture output from functions like this, for which the source cannot be changed to accept a new output argument.

```python
# contextlib_redirect.py

from contextlib import redirect_stdout, redirect_stderr
import io
import sys


def misbehaving_function(a):
    sys.stdout.write('(stdout) A: {!r}\n'.format(a))
    sys.stderr.write('(stderr) A: {!r}\n'.format(a))


capture = io.StringIO()
with redirect_stdout(capture), redirect_stderr(capture):
    misbehaving_function(5)

print(capture.getvalue())
```

In this example, `misbehaving_function()` writes to both `stdout` and `stderr`, but the two context managers send that output to the same `io.StringIO` instance where it is saved to be used later.

to the same io.StringIO instance where it is saved to be used later.

```
$ python3 contextlib_redirect.py

(stdout) A: 5
(stderr) A: 5
```

Note

Both redirect_stdout() and redirect_stderr() modify global state by replacing objects in the sys module, and should be used with care. The functions are not thread-safe, and may interfere with other operations that expect the standard output streams to be attached to terminal devices.

Dynamic Context Manager Stacks

Most context managers operate on one object at a time, such as a single file or database handle. In these cases, the object is known in advance and the code using the context manager can be built around that one object. In other cases, a program may need to create an unknown number of objects in a context, while wanting all of them to be cleaned up when control flow exits the context. ExitStack was created to handle these more dynamic cases.

An ExitStack instance maintains a stack data structure of cleanup callbacks. The callbacks are populated explicitly within the context, and any registered callbacks are called in the reverse order when control flow exits the context. The result is like having multple nested with statements, except they are established dynamically.

Stacking Context Managers

There are several ways to populate the ExitStack. This example uses enter_context() to add a new context manager to the stack.

```python
# contextlib_exitstack_enter_context.py

import contextlib


@contextlib.contextmanager
def make_context(i):
    print('{} entering'.format(i))
    yield {}
    print('{} exiting'.format(i))


def variable_stack(n, msg):
    with contextlib.ExitStack() as stack:
        for i in range(n):
            stack.enter_context(make_context(i))
        print(msg)


variable_stack(2, 'inside context')
```

enter_context() first calls __enter__() on the context manager, and then registers its __exit__() method as a callback to be invoked as the stack is undone.

```
$ python3 contextlib_exitstack_enter_context.py

0 entering
1 entering
inside context
1 exiting
0 exiting
```

The context managers given to ExitStack are treated as though they are in a series of nested with statements. Errors that happen anywhere within the context propagate through the normal error handling of the context managers. These context manager classes illustrate the way errors propagate.

```python
# contextlib_context_managers.py

import contextlib
import contextlib


class Tracker:
    "Base class for noisy context managers."

    def __init__(self, i):
        self.i = i

    def msg(self, s):
        print('  {}({}): {}'.format(
            self.__class__.__name__, self.i, s))

    def __enter__(self):
        self.msg('entering')


class HandleError(Tracker):
    "If an exception is received, treat it as handled."

    def __exit__(self, *exc_details):
        received_exc = exc_details[1] is not None
        if received_exc:
            self.msg('handling exception {!r}'.format(
                exc_details[1]))
        self.msg('exiting {}'.format(received_exc))
        # Return Boolean value indicating whether the exception
        # was handled.
        return received_exc


class PassError(Tracker):
    "If an exception is received, propagate it."

    def __exit__(self, *exc_details):
        received_exc = exc_details[1] is not None
        if received_exc:
            self.msg('passing exception {!r}'.format(
                exc_details[1]))
        self.msg('exiting')
        # Return False, indicating any exception was not handled.
        return False


class ErrorOnExit(Tracker):
    "Cause an exception."

    def __exit__(self, *exc_details):
        self.msg('throwing error')
        raise RuntimeError('from {}'.format(self.i))


class ErrorOnEnter(Tracker):
    "Cause an exception."

    def __enter__(self):
        self.msg('throwing error on enter')
        raise RuntimeError('from {}'.format(self.i))

    def __exit__(self, *exc_info):
        self.msg('exiting')
```

The examples using these classes are based around `variable_stack()`, which uses the context managers passed to construct an ExitStack, building up the overall context one by one. The examples below pass different context managers to explore the error handling behavior. First, the normal case of no exceptions.

```
print('No errors:')
variable_stack([
    HandleError(1),
    PassError(2),
])
```

Then, an example of handling exceptions within the context managers at the end of the stack, in which all of the open contexts are closed as the stack is unwound.

```
print('\nError at the end of the context stack:')
variable_stack([
    HandleError(1),
    HandleError(2),
    ErrorOnExit(3),
])
```

Next, an example of handling exceptions within the context managers in the middle of the stack, in which the error does not occur until some contexts are already closed, so those contexts do not see the error.

```
print('\nError in the middle of the context stack:')
variable_stack([
    HandleError(1),
    PassError(2),
    ErrorOnExit(3),
    HandleError(4),
])
```

Finally, an example of the exception remaining unhandled and propagating up to the calling code.

```
try:
    print('\nError ignored:')
    variable_stack([
        PassError(1),
        ErrorOnExit(2),
    ])
except RuntimeError:
    print('error handled outside of context')
```

If any context manager in the stack receives an exception and returns a True value, it prevents that exception from propagating up to any other context managers.

```
$ python3 contextlib_exitstack_enter_context_errors.py

No errors:
  HandleError(1): entering
  PassError(2): entering
  PassError(2): exiting
  HandleError(1): exiting False
  outside of stack, any errors were handled

Error at the end of the context stack:
  HandleError(1): entering
  HandleError(2): entering
  ErrorOnExit(3): entering
  ErrorOnExit(3): throwing error
  HandleError(2): handling exception RuntimeError('from 3')
  HandleError(2): exiting True
  HandleError(1): exiting False
  outside of stack, any errors were handled

Error in the middle of the context stack:
  HandleError(1): entering
  PassError(2): entering
  ErrorOnExit(3): entering
  HandleError(4): entering
  HandleError(4): exiting False
  ErrorOnExit(3): throwing error
  PassError(2): passing exception RuntimeError('from 3')
  PassError(2): exiting
  HandleError(1): handling exception RuntimeError('from 3')
  HandleError(1): exiting True
  outside of stack, any errors were handled

Error ignored:
  PassError(1): entering
  ErrorOnExit(2): entering
  ErrorOnExit(2): throwing error
  PassError(1): passing exception RuntimeError('from 2')
  PassError(1): exiting
error handled outside of context
```

Arbitrary Context Callbacks

ExitStack also supports arbitrary callbacks for closing a context, making it easy to clean up resources that are not controlled via a context manager.

```
# contextlib_exitstack_callbacks.py

import contextlib


def callback(*args, **kwds):
    print('closing callback({}, {})'.format(args, kwds))


with contextlib.ExitStack() as stack:
    stack.callback(callback, 'arg1', 'arg2')
    stack.callback(callback, arg3='val3')
```

Just as with the `__exit__()` methods of full context managers, the callbacks are invoked in the reverse order that they are registered.

```
$ python3 contextlib_exitstack_callbacks.py

closing callback((), {'arg3': 'val3'})
closing callback(('arg1', 'arg2'), {})
```

The callbacks are invoked regardless of whether an error occurred, and they are not given any information about whether an error occurred. Their return value is ignored.

```
# contextlib_exitstack_callbacks_error.py

import contextlib


def callback(*args, **kwds):
    print('closing callback({}, {})'.format(args, kwds))


try:
    with contextlib.ExitStack() as stack:
        stack.callback(callback, 'arg1', 'arg2')
        stack.callback(callback, arg3='val3')
        raise RuntimeError('thrown error')
except RuntimeError as err:
    print('ERROR: {}'.format(err))
```

Because they do not have access to the error, callbacks are unable to suppress exceptions from propagating through the rest of the stack of context managers.

```
$ python3 contextlib_exitstack_callbacks_error.py

closing callback((), {'arg3': 'val3'})
closing callback(('arg1', 'arg2'), {})
ERROR: thrown error
```

Callbacks make a convenient way to clearly define cleanup logic without the overhead of creating a new context manager class. To improve code readability, that logic can be encapsulated in an inline function, and `callback()` can be used as a decorator.

```
# contextlib_exitstack_callbacks_decorator.py

import contextlib
```

```


with contextlib.ExitStack() as stack:

    @stack.callback
    def inline_cleanup():
        print('inline_cleanup()')
        print('local_resource = {!r}'.format(local_resource))

    local_resource = 'resource created in context'
    print('within the context')
```

There is no way to specify the arguments for functions registered using the decorator form of callback(). However, if the cleanup callback is defined inline, scope rules give it access to variables defined in the calling code.

```
$ python3 contextlib_exitstack_callbacks_decorator.py

within the context
inline_cleanup()
local_resource = 'resource created in context'
```

Partial Stacks

Sometimes when building complex contexts it is useful to be able to abort an operation if the context cannot be completely constructed, but to delay the cleanup of all resources until a later time if they can all be set up properly. For example, if an operation needs several long-lived network connections, it may be best to not start the operation if one connection fails. However, if all of the connections can be opened they need to stay open longer than the duration of a single context manager. The pop_all() method of ExitStack can be used in this scenario.

pop_all() clears all of the context managers and callbacks from the stack on which it is called, and returns a new stack pre-populated with those same context managers and callbacks. The close() method of the new stack can be invoked later, after the original stack is gone, to clean up the resources.

```
# contextlib_exitstack_pop_all.py

import contextlib

from contextlib_context_managers import *


def variable_stack(contexts):
    with contextlib.ExitStack() as stack:
        for c in contexts:
            stack.enter_context(c)
        # Return the close() method of a new stack as a clean-up
        # function.
        return stack.pop_all().close
    # Explicitly return None, indicating that the ExitStack could
    # not be initialized cleanly but that cleanup has already
    # occurred.
    return None


print('No errors:')
cleaner = variable_stack([
    HandleError(1),
    HandleError(2),
])
cleaner()

print('\nHandled error building context manager stack:')
try:
    cleaner = variable_stack([
        HandleError(1),
        ErrorOnEnter(2),
    ])
except RuntimeError as err:
    print('caught error {}'.format(err))
else:
    if cleaner is not None:
        cleaner()
    else:
        print('no cleaner returned')

print('\nUnhandled error building context manager stack:')
try:
    cleaner = variable_stack([
        PassError(1),
        ErrorOnEnter(2),
    ])
except RuntimeError as err:
    print('caught error {}'.format(err))
else:
    if cleaner is not None:
        cleaner()
    else:
        print('no cleaner returned')
```

This example uses the same context manager classes defined earlier, with the difference that `ErrorOnEnter` produces an error on `__enter__()` instead of `__exit__()`. Inside `variable_stack()`, if all of the contexts are entered without error then the `close()` method of a new `ExitStack` is returned. If a handled error occurs, `variable_stack()` returns None to indicate that the cleanup work is already done. And if an unhandled error occurs, the partial stack is cleaned up and the error is propagated.

```
$ python3 contextlib_exitstack_pop_all.py

No errors:
  HandleError(1): entering
  HandleError(2): entering
  HandleError(2): exiting False
  HandleError(1): exiting False

Handled error building context manager stack:
  HandleError(1): entering
  ErrorOnEnter(2): throwing error on enter
  HandleError(1): handling exception RuntimeError('from 2')
  HandleError(1): exiting True
no cleaner returned

Unhandled error building context manager stack:
  PassError(1): entering
  ErrorOnEnter(2): throwing error on enter
  PassError(1): passing exception RuntimeError('from 2')
  PassError(1): exiting
caught error from 2
```

See also

- Standard library documentation for contextlib
- **PEP 343** – The `with` statement.
- Context Manager Types – Description of the context manager API from the standard library documentation.
- With Statement Context Managers – Description of the context manager API from the Python Reference Guide.
- Resource management in Python 3.3, or contextlib.ExitStack FTW! – Description of using `ExitStack` to deploy safe code from Barry Warsaw.

operator — Functional Interface to Built-in Operators

Dates and Times

This page was last updated 2018-12-09.

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?