Python: With That
Five Easy Context Managers

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009) and *Python Cookbook* (3rd Edition, O'Reilly Media, 2013). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

At the last PyCon conference, Raymond Hettinger gave a keynote talk in which he noted that context managers might be one of Python's most powerful yet underappreciated features. In case you're new to the concept of a context manager, we're talking about the `with` statement that was added to Python 2.6. You'll most often see it used in the context of file I/O. For instance, this is the "modern" style of reading a file line-by-line:

```
with open('data.csv') as f:
    for line in f:
        # Do something with line
        ...

# f automatically closed here
```

In this example, the variable `f` holds an open file instance that is automatically closed when control leaves the block of statements under the `with` statement. Thus, you don't have to invoke `f.close()` explicitly when you use the `with` statement as shown. If you're not quite convinced, you can also try an interactive example:

```
>>> with open('/etc/passwd') as f:
...     print(f)
...
<open file '/etc/passwd', mode 'r' at 0x2b4180>
>>> print(f)
<closed file '/etc/passwd', mode 'r' at 0x2b4180>
>>>
```

With that in mind, seeing how something so minor could be one of the language's most powerful features as claimed might be a bit of a stretch. So, in this article, we'll simply take a look at some examples involving context managers and see that so much more is possible.

Make a Sandwich

What is a context manager anyways? To steal an analogy from Raymond Hettinger, a context manager is kind of like the slices of bread that make up a sandwich. That is, you have a top and a bottom piece, in-between which you put some kind of filling. The choice of filling is immaterial—the bread doesn't pass judgment on your dubious choice to make a sandwich filled with peanut-butter, jelly, and tuna.

In terms of programming, a context manager allows you to write code that wraps around the execution of a block of statements. To make it work, objects must implement a specific protocol, as shown here:

```
class Manager(object):
   def __enter__(self):
      print('Entering')
      return "SomeValue"    # Can return anything
   def __exit__(self, e_ty, e_val, e_tb):
      if e_ty is not None:
         print('exception %s occurred' % e_ty)
      print('Exiting')
```

Before proceeding, try the code yourself:

```
>>> m = Manager()
>>> with m as val:
...     print('Hello World')
...     print(val)
...
Entering
Hello World
SomeValue
Exiting
>>>
```

Notice how the "Entering" and "Exiting" messages get wrapped around the statements under the `with`. Also observe how the value returned by the `__enter__()` method is placed into the variable name given with the optional as specifier. Now, try an example with an error:

```
>>> with m:
...     print('About to die')
...     x = int('not a number')
...
Entering
About to die
exception <class 'ValueError'> occurred
Exiting
Traceback (most recent call last):
  File "<stdin>", line 3, in
ValueError: invalid literal for int() with base 10: 'not a number'
>>>
```

Here, carefully observe that the `__exit__()` method was invoked and presented with the type, value, and traceback of the pending exception. This occurred prior to the traceback being generated.

You can make any object work as a context manager by implementing the `__enter__()` and `__exit__()` methods as shown; however, the contextlib library provides a decorator that can also be used to write context managers in the form of a simple generator function. For example:

```
from contextlib import contextmanager

@contextmanager
def manager():
   # Everything before yield is part of __enter__
   print("Entering")
   try:
      yield "SomeValue"
   # Everything beyond the yield is part of __exit__
   except Exception as e:
      print("An error occurred: %s" % e)
      raise
   else:
      print("No errors occurred")
```

If you try the above function, you'll see that it works in the same way.

```
>>> with manager() as val:
...     print("Hello World")
...     print(val)
...
Entering
Hello World
SomeValue
No errors occurred
>>>
```

Sandwiches Everywhere!

Once you've seen your first sandwich, you'll quickly realize that they are everywhere! Consider some of the following common programming patterns:

```
# File I/O
f = open('somefile')
...
f.close()

# Temporary files/directories
name = mktemp()
...
remove(name)

# Timing
start_time = time()
...
end_time = time()

# Locks (threads)
lock.acquire()
...
lock.release()
```

```
# Publish-subscribe
channel.subscribe(recipient)
...
channel.unsubscribe(recipient)

# Database transactions
cur = db.cursor()
...
db.commit()
```

Indeed, the same pattern repeats itself over and over again in all sorts of real-world code. In fact, any time you find yourself working with code that follows this general pattern, consider the use of a context manager instead. Indeed, many of Python's built-in objects already support it. For example:

```
# File I/O
with open('somefile') as f:
    ...

# Temporary files
from tempfile import NamedTemporaryFile
with NamedTemporaryFile() as f:
    ...

# Locks
lock = threading.Lock()
with lock:
    ...
```

The main benefit of using the context-manager version is that it more precisely defines your usage of some resource and is less error prone should you forget to perform the final step (e.g., closing a file, releasing a lock, etc.).

Making Your Own Managers

Although it's probably most common to use the `with` statement with existing objects in the library, you shouldn't shy away from making your own context managers. In fact, it's pretty easy to write custom context manager code.

The remainder of this article simply presents some different examples of custom context managers in action. It turns out that they can be used for so much more than simple resource management if you use your imagination. The examples are presented with little in the way of discussion, so you'll need to enter the code and play around with them yourself.

Temporary Directories with Automatic Deletion

Sometimes you need to create a temporary directory to perform a bunch of file operations. Here's a context manager that does just that, but it takes care of destroying the directory contents when done:

```
import tempfile
import shutil
from contextlib import contextmanager

@contextmanager
def tempdir():
    name = tempfile.mkdtemp()
    try:
        yield name
    finally:
        shutil.rmtree(name)
```

To use it, you would write code like this:

```
with tempdir() as dirname:
    # Create files and perform operations
    filename = os.path.join(dirname, 'example.txt')
    with open(filename, 'w') as f:
        f.write('Hello World\n')
    ...

# dirname (and all contents) automatically deleted here
```

Ignoring Exceptions

Sometimes you just want to ignore an exception. Traditionally, you might write code like this:

```
try:
    ...
except SomeError:
    pass
```

However, here's a context manager that allows you to reduce it all to one line:

```
@contextmanager
def ignore(exc):
    try:
        yield
    except exc:
        pass

# Example use. Parse data and ignore bad conversions
records = []
for row in lines:
    with ignore(ValueError):
        record = (int(row[0]), int(row[1]), float(row[2]))
        records.append(record)
```

With a few minor modifications, you could adapt this code to perform other kinds of exception handling actions: for example, routing exceptions to a log file, or simply packaging up a complex exception handling block into a simple function that can be easily reused as needed.

Making a Stopwatch

Here's an object that implements a simple stopwatch:

```
import time

class Timer(object):
    def __init__(self):
        self.elapsed = 0.0
        self._start = None

    def __enter__(self):
        assert self._start is None, "Timer already started"
        self._start = time.time()

    def __exit__(self, e_ty, e_val, e_tb):
        assert self._start is not None, "Timer not started"
        end = time.time()
        self.elapsed += end - self._start
        self._start = None

    def reset(self):
        self.__init__()
```

To use the timer, you simply use the `with` statement to indicate the operations you want timed. For example:

```
# Example use
my_timer = Timer()

...
with my_timer:
    statement
    statement
    ...
...
print("Total time: %s" % my_timer.elapsed)
```

Deadlock Avoidance

A common problem in threaded programs is deadlock arising from the use of too many locks at once. Here is a context manager that implements a simple deadlock avoidance scheme that can be used to acquire multiple locks at once. It works by simply forcing multiple locks always to be acquired in ascending order of their object IDs.

```
from contextlib import contextmanager

@contextmanager
def acquire(*locks):
    sorted_locks = sorted(locks, key=id)
    for lock in sorted_locks:
        lock.acquire()
    try:
        yield
    finally:
        for lock in reversed(sorted_locks):
            lock.release()
```

This one might take a bit of pondering, but if you throw it at the classic "Dining Philosopher's" problem from operating systems, you'll find that it works.

```
import threading

def philosopher(n, left_stick, right_stick):
    while True:
        with acquire(left_stick, right_stick):
            print("%d eating" % n)

def dining_philosophers():
    sticks = [ threading.Lock() for n in range(5) ]
    for n in range(5):
        left_stick = sticks[n]
        right_stick = sticks[(n + 1) % 5]
        t = threading.Thread(target=philosopher,
                             args=(n, left_stick, right_stick))
        t.daemon = True
        t.start()

if __name__ == '__main__':
    import time
    dining_philosophers()
    time.sleep(10)
```

If you run the above code, you should see all of the philosophers running deadlock free for about 10 seconds. After that, the program simply terminates.

Making a Temporary Patch to Module

Here's a context manager that allows you to make a temporary patch to a variable defined in an already loaded module:

```
from contextlib import contextmanager
import sys

@contextmanager
def patch(qualname, newvalue):
    parts = qualname.split('.')
    assert len(parts) > 1, "Must use fully qualified name"
    obj = sys.modules[parts[0]]
    for part in parts[1:-1]:
        obj = getattr(obj, part)

    name = parts[-1]
    oldvalue = getattr(obj, name)
    try:
```

```
        setattr(obj, name, newvalue)
        yield newvalue
    finally:
        setattr(obj, name, oldvalue)
```

Here's an example of using this manager:

```
>>> import io
>>> with patch('sys.stdout', io.StringIO()) as out:
...     for i in range(10):
...         print(i)
...
>>> out.getvalue()
'0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
>>>
```

In this example, the value of `sys.stdout` is temporarily replaced by a `StringIO` object that allows you to capture output directed toward standard output. This might be useful in the context of certain tasks such as tests. In fact, the popular mock tool (https://pypi.python.org/pypi/mock) has a similar, but much more powerful variant of this decorator.

More Information

This article is really only scratching the surface of what's possible with context managers; however, the key takeaway is that context managers can be used to address a wide variety of problems that come up in real-world programming. Not only that, they are relatively easy to define, so you're definitely not limited to using them only with Python's built-in objects such as files. For more ideas and inspiration, a good starting point might be documentation for the `contextlib` module as well as PEP 343 (http://www.python.org/dev/peps/pep-0343/).