Jeff Knupp

PYTHON PROGRAMMER

Everything I know about Python...

Python with Context Managers

Of all of the most commonly used Python constructs, `context managers` are neck-and-neck with `decorators` in a "Things I use but don't really understand how they work" contest. As every schoolchild will tell you, the canonical way to open and read from a file is:

```
with open('what_are_context_managers.txt', 'r') as inf
ile:
    for line in infile:
        print('> {}'.format(line))
```

But how many of those who correctly handle file IO know *why* it's correct, or even that there's an *incorrect* way to do it? Hopefully a lot, or else this post won't get read much...

Managing Resources

Perhaps the most common (and important) use of context managers is to properly manage resources. In fact, that's the reason we use a context manager when reading from a file. The act of opening a file consumes a resource (called a file descriptor), and this resource is limited by your OS. That is to say, there are a maximum number of files a process can have open at one time. To

prove it, try running this code:

```
files = []
for x in range(100000):
    files.append(open('foo.txt', 'w'))
```

If you're on Mac OS X or Linux, you probably got an error message like the following:

```
> python test.py
Traceback (most recent call last):
  File "test.py", line 3, in <module>
OSError: [Errno 24] Too many open files: 'foo.txt'
```

If you're on Windows, your computer probably crashed and your motherboard is now on fire. Let this be a lesson: **don't leak file descriptors!**

Joking aside, what is a "file descriptor" and what does it mean to "leak" one? Well, when you open a file, the operating system assigns an integer to the open file, allowing it to essentially give you a *handle* to the open file rather than direct access to the underlying file itself. This is beneficial for a variety of reasons, including being able to pass references to files between processes and to maintain a certain level of security enforced by the kernel.

So how does one "leak" a file descriptor. Simply: by **not closing opened files**. When working with files, it's easy to forget that any file that is `open()`-ed must also be `close()`-ed. Failure to do so will lead you to discover that there is (usually) a limit to the number of file descriptors a process can be assigned. On UNIX-like systems, `$ ulimit -n` should give you the value of that upper limit (it's `7168` on my system). If you want to prove it to yourself, re-run the example code and replace `100000` with whatever number you got (minus about `5`, to account for files the Python interpreter opens on startup). You should now see the program run to completion.

Of course, there's a simpler (and better) way to get the program to complete: **close each file!**. Here's a contrived example of how to fix the issue:

```
files = []
for x in range(10000):
    f = open('foo.txt', 'w')
    f.close()
    files.append(f)
```

A Better Way To Manage Resources

In real systems, it's difficult to make sure that `close()` is called on every file opened, especially if the file is in a function that may raise an

exception or has multiple return paths. In a complicated function that opens a file, how can you possibly be expected to remember to add `close()` to every place that function could return from? And that's not counting exceptions, either (which may happen from anywhere). The short answer is: you can't be.

In other languages, developers are forced to use `try...except...finally` every time they work with a file (or any other type of resource that needs to be closed, like sockets or database connections). Luckily, Python loves us and gives us a simple way to make sure all resources we use are properly cleaned up, regardless of if the code returns or an exception is thrown: *context managers*.

By now, the premise should be obvious. We need a convenient method for indicating a particular variable has some cleanup associated with it, and to guarantee that cleanup happens, no matter what. Given that requirement, the syntax for using context managers makes a lot of sense:

```
with something_that_returns_a_context_manager() as my_
resource:
    do_something(my_resource)
    ...
    print('done using my_resource')
```

That's it! Using `with`, we can call anything that returns a context manager (like the built-in `open()` function). We assign it to a variable using `... as <variable_name>`. Crucially, **the variable only exists within the indented block below the `with` statement**. Think of `with` as creating a mini-function: we can use the variable freely in the indented portion, but once that block ends, the variable goes out of scope. When the variable goes out of scope, it automatically calls a special method that contains the code to clean up the resource.

But where is the code that is actually being called when the variable goes out of scope? The short answer is, "wherever the context manager is defined." You see, there are a number of ways to create a context manager. The simplest is to define a class that contains two special methods: `__enter__()` and `__exit__()`. `__enter__()` returns the resource to be managed (like a file object in the case of `open()`). `__exit__()` does any cleanup work and returns nothing.

To make things a bit more clear, let's create a

totally redundant context manager for working with files:

```
class File():

    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.open_file = open(self.filename, self.mode
)
        return self.open_file

    def __exit__(self, *args):
        self.open_file.close()

files = []
for _ in range(10000):
    with File('foo.txt', 'w') as infile:
        infile.write('foo')
        files.append(infile)
```

Let's go over what we have. Like any class, there's an `__init__()` method that sets up the object (in our case, setting the file name to open and the mode to open it in). `__enter__()` opens and returns the file (also creating an attribute `open_file` so that we can refer to it in `__exit__()`). `__exit__()` just closes the file. Running the code above works because the file is being closed when it leaves the `with File('foo.txt', 'w') as infile:` block. Even if code in that block raised an exception, the file would still be closed.

Other Useful Context Managers

Given that context managers are so helpful, they were added to the Standard Library in a number of places. `Lock` objects in `threading` are context managers, as are `zipfile.ZipFile`s. `subprocess.Popen`, `tarfile.TarFile`, `telnetlib.Telnet`, `pathlib.Path`... the list goes on and on. Essentially, any object that needs to have `close` called on it after use is (or should be) a context manager.

The `Lock` usage is particularly interesting. In this case, the resource in question is a mutex (e.g. a "Lock"). Using context managers prevents a common source of deadlocks in multi-threaded programs which occur when a thread "acquires" a mutex and never "releases" it. Consider the following:

```
from threading import Lock
lock = Lock()

def do_something_dangerous():
    lock.acquire()
    raise Exception('oops I forgot this code could rai
se exceptions')
    lock.release()

try:
    do_something_dangerous()
except:
    print('Got an exception')
lock.acquire()
print('Got here')
```

Clearly `lock.release()` will never be called, causing all other threads calling `do_something_dangerous()` to become deadlocked. In our program, this is represented by never hitting the `print('Got here')` line. This, however, is easily fixed by taking advantage of the fact that `Lock` is a context manager:

```
from threading import Lock
lock = Lock()

def do_something_dangerous():
    with lock:
        raise Exception('oops I forgot this code could
 raise exceptions')

try:
    do_something_dangerous()
except:
    print('Got an exception')
lock.acquire()
print('Got here')
```

Indeed, there is no reasonable way to acquire `lock` using a context manager and not release it. And that's exactly how it should be.

Fun With `contextlib`

Context managers are so useful, they have a whole Standard Library module devoted to them! `contextlib` contains tools for creating and working with context managers. One nice shortcut to creating a context manager from a class is to use the `@contextmanager` decorator. To use it, decorate a generator function that calls `yield` exactly *once*. Everything *before* the call to `yield` is considered the code for `__enter__()`. Everything after is the code for `__exit__()`. Let's rewrite our `File` context manager using the decorator approach:

```
from contextlib import contextmanager

@contextmanager
def open_file(path, mode):
    the_file = open(path, mode)
    yield the_file
    the_file.close()

files = []

for x in range(100000):
    with open_file('foo.txt', 'w') as infile:
        files.append(infile)

for f in files:
    if not f.closed:
        print('not closed')
```

As you can see, the implementation is considerably shorter. In fact, it's only five lines long! We open the file, `yield` it, then close it. The code that follows is just proof that all of the files are, indeed, closed. The fact that the program didn't crash is extra insurance it worked.

The official Python docs have a particularly fun/stupid example:

```
from contextlib import contextmanager

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

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

My favorite piece of context manager-lunacy, however, has to be `contextlib.ContextDecorator`. It lets you define a context manager using the class-based approach, but inheriting from `contextlib.ContextDecorator`. By doing so, you can use your context manager with the `with` statement as normal *or as a function decorator*. We could do something similar to the HTML example above using this pattern (which is truly insane and shouldn't be done):

```
from contextlib import ContextDecorator

class makeparagraph(ContextDecorator):
    def __enter__(self):
        print('<p>')
        return self

    def __exit__(self, *exc):
        print('</p>')
        return False

@makeparagraph()
def emit_html():
    print('Here is some non-HTML')

emit_html()
```

The output will be:

```
<p>
Here is some non-HTML
</p>
```

Truly useless and horrifying...

Wrapping Up

Hopefully by now you understand what a context manager is, how it works, and why it's useful. As we just saw, there are *a lot* of useful (and not-so-useful) things you can do with context managers. Their goal is a noble one: to make working with resources and creating managed contexts easier. Now it's up to you to not only use them, but create new ones as well, thus making *other* people's lives easier. Unless you use them to generate HTML. Then you're just an awful person and you shouldn't read this blog (but I'm glad you did).

Posted on Mar 07, 2016 by Jeff Knupp

Tweet

« Improve Your Python: the with Statement and Context Managers

Subscribe