Preshing on Programming

https://preshing.com/20110920/the-python-with-statement-by-example/

这个是英文版的online链接，这个文档我没有仔细的看，就是只是看了前面一部分，其实和中文文档的内容都是差不多的。如果有需要，也是可以后来反过来再看一遍的。

Sep 20, 2011

The Python "with" Statement by Example

Python's `with` statement was first introduced five years ago, in Python 2.5. It's handy when you have two related operations which you'd like to execute as a pair, with a block of code in between. The classic example is opening a file, manipulating the file, then closing it:

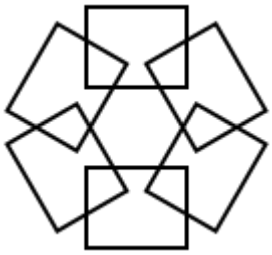
```
with open('output.txt', 'w') as f:
    f.write('Hi there!')
```

The above `with` statement will automatically close the file after the nested block of code. (Continue reading to see exactly how the close occurs.) The advantage of using a `with` statement is that it is guaranteed to close the file no matter *how* the nested block exits. If an exception occurs before the end of the block, it will close the file before the exception is caught by an outer exception handler. If the nested block were to contain a `return` statement, or a `continue` or `break` statement, the `with` statement would automatically close the file in those cases, too.

Here's another example. The pycairo drawing library contains a `Context` class which exposes a `save` method, to push the current drawing state on an internal stack, and a `restore` method, to restore the drawing state from the stack. These two functions are always called in a pair, with some code in between.

This code sample uses a `Context` object ("cairo context") to draw six rectangles, each with a different rotation. Each call to `rotate` is actually combined with the current transformation, so we use a pair of calls to `save` and `restore` to preserve the drawing state on each iteration of the loop. This prevents the rotations from combining with each other:

```
cr.translate(68, 68)
for i in xrange(6):
    cr.save()
    cr.rotate(2 * math.pi * i / 6)
    cr.rectangle(-25, -60, 50, 40)
    cr.stroke()
    cr.restore()
```



That's a fairly simple example, but for larger scripts, it can become cumbersome to keep track of which `save` goes with which `restore`, and to keep them correctly matched. The `with` statement can help tidy things up a bit.

By themselves, pycairo's `save` and `restore` methods do not support the `with` statement, so we'll have to add the support on our own. There are two ways to support the `with` statement: by implementing a context manager class, or by writing a generator function. I'll demonstrate both approaches.

Implementing the Context Manager as a Class

Here's the first approach. To implement a context manager, we define a class containing an `__enter__` and `__exit__` method. The class below accepts a cairo context, `cr`, in its constructor:

```
class Saved():
    def __init__(self, cr):
        self.cr = cr
    def __enter__(self):
        self.cr.save()
        return self.cr
    def __exit__(self, type, value, traceback):
        self.cr.restore()
```

Thanks to those two methods, it's valid to instantiate a `Saved` object and use it in a `with` statement. The `Saved` object is considered to be the context manager.

```
cr.translate(68, 68)
for i in xrange(6):
    with Saved(cr):
        cr.rotate(2 * math.pi * i / 6)
        cr.rectangle(-25, -60, 50, 40)
        cr.stroke()
```

Here are the exact steps taken by the Python interpreter when it reaches the `with` statement:

1. The `with` statement stores the `Saved` object in a temporary, hidden variable, since it'll be needed later. (Actually, it only stores the bound `__exit__` method, but that's a detail.)
2. The `with` statement calls `__enter__` on the `Saved` object, giving the context manager a chance to do its job.
3. The `__enter__` method calls `save` on the cairo context.
4. The `__enter__` method returns the cairo context, but as you can see, we have not specified the optional `"as" target` part of the `with` statement. Therefore, the return value is not saved anywhere. We don't need it; we know it's the same cairo context that we passed in.
5. The nested block of code is executed. It sets up the rotation and draws a rectangle.
6. At the end of the nested block, the `with` statement calls the `Saved` object's `__exit__` method, passing the arguments `(None, None, None)` to indicate that no exception occured.
7. The `__exit__` method calls `restore` on the cairo context.

Once we understand what the Python interpreter is doing, we can make better sense of the example at the beginning of this blog post, where we opened a file in the `with` statement: File objects expose their own `__enter__` and `__exit__` methods, and can therefore act as their own context managers. Specifically, the `__exit__` method closes the file.

Exception Handling

Returning to the drawing example, what happens if an exception occurs within the nested code block? For example, suppose we mistakenly passed the wrong number of arguments to the `rectangle` call. In that case, the steps taken by the Python interpreter would be:

1. The `rectangle` method raises a `TypeError` exception: "Context.rectangle() takes exactly 4 arguments."
2. The `with` statement catches this exception.
3. The `with` statement calls `__exit__` on the `Saved` object. It passes information about the exception in three arguments: (*type*, *value*, *traceback*) – the same values you'd get by calling `sys.exc_info`. This tells the `__exit__` method everything it could possibly need to know about the exception that occurred.
4. In this case, our `__exit__` method doesn't particularly care. It calls `restore` on the cairo context anyway, and returns `None`. (In Python, when no `return` statement is specified, the function actually returns `None`.)
5. The `with` statement checks to see whether this return value is true. Since it isn't, the `with` statement re-raises the `TypeError` exception to be handled by someone else.

In this manner, we can guarantee that `restore` will always be called on the cairo context, whether an exception occurs or not.

Implementing the Context Manager as a Generator

That brings us to the second approach for supporting the `with` statement. Instead of implementing a class for the context manager, we can write a generator function. Here's a simplified example of such a generator function. Let me point out right away that this example is incomplete, since it does not handle exceptions very well. Read on for more details:

```
from contextlib import contextmanager

@contextmanager
def saved(cr):
    cr.save()
    yield cr
    cr.restore()
```

There is a certain charm to writing a generator like this one. At first glance, it appears simpler than the previous approach: A single function takes the place of an entire class definition. But don't be fooled! This approach involves many more steps, and a lot more complexity than the previous approach. It took me several reads of PEP 343 – which is more of a historical document than a reference – before I could claim to understand it completely. It requires familiarity with Python decorators, generators, iterators and functions-returning-functions, in addition to the object-oriented programming and exception handling we've already seen.

To make this generator work, two entities from `contextlib`, a standard Python module, are required: the `contextmanager` function, and an internal class named `GeneratorContextManager`. The source code, `contextlib.py`, is a bit hairy, but at least it's short. I'll simply describe what happens, and you are free to refer to the source code, and any other supplementary materials, as needed.

Let's start with the generator itself. Here's what happens when the above code snippet runs:

1. The Python interpreter recognizes the `yield` statement in the middle of the function definition. As a result, the `def` statement does not create a normal function; it creates a generator function.
2. Because of the presence of the `@contextmanager` decorator, `contextmanager` is called with the generator function as its argument.
3. The `contextmanager` function returns a "factory" function, which creates `GeneratorContextManager` objects wrapped around the provided generator. (line 83 of `contextlib.py`)
4. Finally, the factory function is assigned to `saved`. From this point on, when we call `saved`, we'll actually be calling the factory function.

Equipped with all that good stuff, we can now write:

```
for i in xrange(6):
    with saved(cr):
        cr.rotate(2 * math.pi * i / 6)
        cr.rectangle(-25, -60, 50, 40)
        cr.stroke()
```

Here are all the steps taken by the Python interpreter when it reaches the `with` statement.

1. The `with` statement calls `saved`, which of course, calls the factory function, passing `cr`, a cairo context, as its only argument.
2. The factory function passes the cairo context to our generator function, creating a generator.
3. The generator is passed to the constructor of `GeneratorContextManager`, an internal class which will act as our context manager.
4. The `with` statement saves the `GeneratorContextManager` object in a temporary hidden variable. (Actually, it only stores the bound `__exit__` method, but that's a detail.)
5. The `with` statement calls `__enter__` on the `GeneratorContextManager` object.
6. `__enter__` calls `next` on the generator.
7. Our generator function – the block of code we defined under `def saved(cr)` – runs up until the `yield` statement. This calls `save` on the cairo context.
8. The `yield` statement yields the cairo context, which becomes the return value for the call to `next` on the iterator.
9. The `__enter__` method returns the cairo context, but as you can see, we have not specified the optional `"as" target` part of the `with` statement. Therefore, the return value is not saved anywhere. We don't need it; we know it's the same cairo context that we passed in.
10. The nested code block is executed. It sets up the rotation and draws a rectangle.
11. At the end of the nested block, the `with` statement calls the `__exit__` method on the `GeneratorContextManager` object, passing the arguments `(None, None, None)` to indicate that no exception occured.
12. The `__exit__` method calls `next` on the iterator (expecting a `StopIteration` exception).
13. Our generator resumes execution after the `yield` statement. This calls `restore` on the cairo context.
14. The generator returns, raising a `StopIteration` exception (as expected).
15. The `__exit__` method catches the `StopIteration` exception, and returns normally.

And that's it! We've successfully used this generator function as a `with` statement context manager. In this example, it helped that no exceptions occured. To correctly deal with exceptions, we'll have to improve the generator function a little bit.

Exception Handling

Now, what happens if an exception occurs within the nested block while using this approach? Again, let's suppose we've mistakenly passed the wrong number of arguments to the `rectangle` call. Here's what would happen:

1. The `rectangle` method raises a `TypeError` exception: "Context.rectangle() takes exactly 4 arguments."
2. The `with` statement catches this exception.
3. The `with` statement calls `__exit__` on the `GeneratorContextManager` object. It passes information about the exception in three arguments: (*type*, *value*, *traceback*).
4. `__exit__` calls `throw` on the iterator, passing the same three arguments.
5. The `TypeError` exception is raised in the context of our generator function, on the line containing the `yield` statement.

Uh oh! At this point, our current generator function has a problem: `restore` will *not* be called on the cairo context. An exception has been raised on the line containing the `yield` statement, so the rest of the generator function will not be executed. We need to make the generator more robust, by inserting a `try/finally` block around the `yield`:

```
@contextmanager
def saved(cr):
    cr.save()
    try:
        yield cr
    finally:
        cr.restore()
```

Continuing where we left off:

1. Inside our generator, the `finally` block executes. This calls `restore` on the cairo context.
2. The `TypeError` exception went unhandled by the generator, so it is re-raised in the `__exit__` method, on the line containing the call to `throw` on the iterator. (line 35 of `contextlib.py`)
3. The `TypeError` exception is caught by `__exit__`.
4. `__exit__` sees that the exception caught is the same exception that was passed in, and as a result, returns `None`.
5. The `with` statement checks to see whether this return value is true. Since it isn't, the `with` statement re-raises the `TypeError` exception, to be handled by someone else.

Thus concludes our journey through the Python `with` statement. If, like me, you've had a hard time understanding this statement completely – especially if you were attracted to the generator form of writing context managers – don't feel bad. It's complicated! It cleverly ties together several of Python's language features, many of which were themselves introduced fairly recently in Python's history. If any Pythonistas out there spot an error or oversight in the above explanation, please let me know in the comments.

Drawing a Fractal Tree

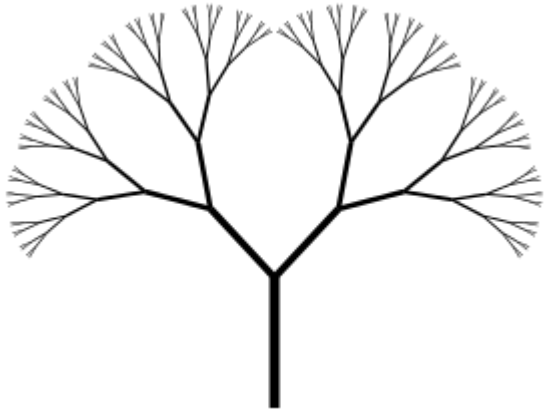
For those of you who have endured the entire blog post up to this point, here's a small bonus script. It uses our newly minted cairo context manager to recursively draw a fractal tree.

```
import cairo
from contextlib import contextmanager

@contextmanager
def saved(cr):
    cr.save()
    try:
        yield cr
    finally:
        cr.restore()

def Tree(angle):
    cr.move_to(0, 0)
    cr.translate(0, -65)
    cr.line_to(0, 0)
    cr.stroke()
    cr.scale(0.72, 0.72)
    if angle > 0.12:
        for a in [-angle, angle]:
            with saved(cr):
                cr.rotate(a)
                Tree(angle * 0.75)

surf = cairo.ImageSurface(cairo.FORMAT_ARGB32, 280, 204)
cr = cairo.Context(surf)
cr.translate(140, 203)
cr.set_line_width(5)
Tree(0.75)
surf.write_to_png('fractal-tree.png')
```



For yet another example of `with` statement usage in Python, see [Timing Your Code Using Python's "with" Statement](#).

« Penrose Tiling Explained Timing Your Code Using Python's "with" Statement »

Comments (21)

Commenting Disabled
Further commenting on this page has been disabled by the blog admin.

stump · *501 weeks ago*

Those are awesome demos of how the with statement can simplify code, but the examples violate PEP 343's suggested naming convention for context managers, which the context managers in the standard library obey.

Just the name "saving" implies that the context manager will do nothing in __enter__ and will do something that can be described as "saving" in __exit__. (See, for example, contextlib.closing().)

It would be better if it were called "saved", indicating that __enter__ places the object in a state that can be described as "saved", and that this is undone upon __exit__.

(Relatedly, perhaps this naming convention should be mentioned in the article.)

Just my two cents.

Reply **2 replies** · *active 333 weeks ago*

Jeff Preshing · *501 weeks ago*

Thanks for pointing this out. I've gone ahead and changed the name based on your feedback.

Reply

Stephen Stack · *333 weeks ago*

Good feedback, naming is extremely in all languages, especially in such an expressive syntax like Python.

Reply

Pierre · *471 weeks ago*

Thanks a lot for those examples!!! It was very usefull!

Reply

Hassan · *458 weeks ago*

Thanks for this helpful description.

Reply

 Bah Mait · *451 weeks ago*

This is really clear - thank you

Reply

 Rod · *437 weeks ago*

Finally! I now understand what with is used for. I couldn't grok it until I knew how it worked under the hood. Thanks a lot!

Reply

 Nic Watson · *427 weeks ago*

Awesome article!

Reply

 Le23 · *426 weeks ago*

Thank you for blogging it clearly here.

Reply

 san · *420 weeks ago*

useful...

Reply

 Murphy Randle · *411 weeks ago*

Simplest explanation I've seen yet of the "with" statement.
Thank you!

Reply

 Luis romero · *389 weeks ago*

Just Awesome (O.O)

Reply

 vindolin · *344 weeks ago*

I was working on a program with lots of cairo calls and thought to myself: I should replace all those save/restore's with a context manager, googled for "python context with" and landed on this page.. destiny.. :)

Reply

 Liang Ma · *316 weeks ago*

very useful

Reply

Richard · *263 weeks ago*

Many thanks for the great explanation!
Just one question if self.cr.save() or cr.save() were to cause an error on entry into the context manager (let's say it returned False for failure), is our only choice to raise an Exception and not yield? What if we would like to just not run the 'with' block at all on error?

Reply

steve · *248 weeks ago*

A very good explanation, thanks
Posted some sample code to further demo this

```
class PR():
def pr_open(self):
print("PR, open")

def pr_whatsit(self,i):
print("PR, whatsit ",i)

def pr_close(self):
print("PR, close (clean up resources)")

class MyContext():
def __init__(self, pr):
self.pr = pr
print("MyContext, init")

def __enter__(self):
self.pr.pr_open()
print("MyContext, enter, (I was called automatically at start of with block)")
return self.pr
def __exit__(self, type, value, traceback):
print("MyContext, exit (I was called automatically at end of with block, even it exception")
self.pr.pr_close()

#We see that for with, that the exit method is called automatically, even if there
#is an exception, which allows for clean up of resources

pr=PR()
with MyContext(pr):
pr.pr_whatsit(1)
pr.pr_whatsit(2)
raise "Error in block"
pr.pr_whatsit(3)
```

run the program, shows what happens

Reply

OlyDLG · *234 weeks ago*

First, thanks: I agree, generally, very clearly explained. Now, a sincere (i.e., non-rhetorical) question: given how much more complicated it seems, in this context, why would anyone ever choose the generator approach over the class approach: are there (classes of) use-cases where the generator approach is actually either simpler or performs better, or is it always simply a matter of taste? (Was the generator approach added later to appease Functionalists?)

Reply

Patrick Duc · *219 weeks ago*

Thanks a lot, very clear explanation.

Reply

 ErnestoZ · *199 weeks ago*

Great explanation of Python context managers. Kudos!!!

Reply

 Rusty · *198 weeks ago*

Excellent article, it was very useful for me to see the possibilities of using the 'with' clause!

Thank You for writing it!

Reply

 Ofer · *198 weeks ago*

Thanks a lot for this great article. It is well written and most enjoyable!

Reply
