

Understanding Python's "with" statement

Fredrik Lundh | October 2006 | Originally posted to online.effbot.org

Judging from comp.lang.python and other forums, Python 2.5's new [with statement](#) (dead link) seems to be a bit confusing even for experienced Python programmers.

As most other things in Python, the **with** statement is actually very simple, once you understand the problem it's trying to solve. Consider this piece of code:

```
set things up
try:
    do something
finally:
    tear things down
```

Here, "set things up" could be opening a file, or acquiring some sort of external resource, and "tear things down" would then be closing the file, or releasing or removing the resource. The **try-finally** construct guarantees that the "tear things down" part is always executed, even if the code that does the work doesn't finish.

If you do this a lot, it would be quite convenient if you could put the "set things up" and "tear things down" code in a library function, to make it easy to reuse. You can of course do something like

```
def controlled_execution(callback):
    set things up
    try:
        callback(thing)
    finally:
        tear things down

def my_function(thing):
    do something

controlled_execution(my_function)
```

But that's a bit verbose, especially if you need to modify local variables. Another approach is to use a one-shot generator, and use the **for-in** statement to "wrap" the code:

```
def controlled_execution():
    set things up
    try:
        yield thing
    finally:
        tear things down

for thing in controlled_execution():
    do something with thing
```

But **yield** isn't even allowed inside a **try-finally** in 2.4 and earlier. And while that could be fixed (and it has been fixed in 2.5), it's still a bit weird to use a loop construct when you know that you only want to execute something once.

So after contemplating a number of alternatives, GvR and the python-dev team finally came up with a generalization of the latter, using an object instead of a generator to control the behaviour of an external piece of code:

```
class controlled_execution:
    def __enter__(self):
        set things up
        return thing
    def __exit__(self, type, value, traceback):
        tear things down

with controlled_execution() as thing:
    some code
```

Now, when the "with" statement is executed, Python evaluates the expression, calls the **__enter__** method on the resulting value (which is called a "context guard"), and assigns whatever **__enter__** returns to the variable given by **as**. Python will then execute the code body, and *no matter what happens in that code*, call the guard object's **__exit__** method.

As an extra bonus, the **__exit__** method can look at the exception, if any, and suppress it or act on it as necessary. To suppress the exception, just return a true value. For example, the following **__exit__** method swallows any **TypeError**, but lets all other exceptions through:

```
def __exit__(self, type, value, traceback):  
    return isinstance(value, TypeError)
```


In Python 2.5, the file object has been equipped with `__enter__` and `__exit__` methods; the former simply returns the file object itself, and the latter closes the file:

```
>>> f = open("x.txt")  
>>> f  
<open file 'x.txt', mode 'r' at 0x00AE82F0>  
>>> f.__enter__()  
<open file 'x.txt', mode 'r' at 0x00AE82F0>  
>>> f.read(1)  
'X'  
>>> f.__exit__(None, None, None)  
>>> f.read(1)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: I/O operation on closed file
```

so to open a file, process its contents, and make sure to close it, you can simply do:

```
with open("x.txt") as f:  
    data = f.read()  
    do something with data
```

This wasn't very difficult, was it?

 rendered by a [django](#) application. hosted by [webfaction](#).