Introduction to Context Managers in Python

John Jacobsen

This is the third of a series of posts loosely associated with integration testing, mostly focused on Python.

Context managers are a way of allocating and releasing some sort of resource exactly where you need it. The simplest example is file access:

```
with file(“/tmp/foo", “w”) as foo:
    print >> foo, “Hello!”
```

This is essentially equivalent to:

```
foo = file(“/tmp/foo", “w”)
try:
    print >> foo, “Hello!”
finally:
    foo.close()
```

Locks are another example. Given:

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

then

```
with lock:
    my_list.append(item)
```

replaces the more verbose:

```
lock.acquire()
try:
    my_list.append(item)
finally:
    lock.release()
```

In each case a bit of boilerplate is eliminated, and the "context" of the file or the lock is acquired, used, and released cleanly.Context managers are a common idiom in Lisp, where they are usually defined using macros (examples are `with-open` and `with-out-str` in Clojure and `with-open-file` and `with-output-to-string` in Common Lisp).

Python, not having macros, must include context managers as part of the language. Since 2.5, it does so, providing an easy mechanism for rolling your own. Though the default, "low level" way to make a context manager is to make a class which follows the context management protocol, by implementing `__enter__` and `__exit__` methods, the simplest way is using the `contextmanager` decorator from the contextlib library, and invoking `yield` in your context manager function in between the setup and teardown steps.

Here is a context manager which you could use to time our threads-vs-processes testing, discussed previously:

```
import contextlib
import time

@contextlib.contextmanager
def time_print(task_name):
    t = time.time()
    try:
        yield
    finally:
        print task_name, “took", time.time() - t, “seconds.”

with time_print(“processes”):
    [doproc() for _ in range(500)]

# processes took 15.236166954 seconds.

with time_print(“threads”):
    [dothread() for _ in range(500)]

# threads took 0.11357998848 seconds.
```

Composition

Context managers can be composed very nicely. While you can certainly do the following,

```
with a(x, y) as A:
    with b(z) as B:
        # Do stuff
```

in Python 2.7 or above, the following also works:

```
with a(x, y) as A, b(z) as B:
    # Do stuff
```

with Python 2.6, using `contextlib.nested` does almost the same thing:

```
with contextlib.nested(a(x, y), b(z)) as (A, B):
    # Do the same stuff
```

the difference being that with the 2.7+ syntax, you can use the value yielded from the first context manager as the argument to the second (e.g., `with a(x, y) as A, b(A) as C:...`).

If multiple contexts occur together repeatedly, you can also roll them together into a new context manager:

```
import contextlib

@contextlib.contextmanager
def c(x, y, z):
    with a(x, y) as A, b(z) as B:
        yield (A, B)

with c(x, y, z) as C:  # C == (A, B)
    # Do that same old stuff
```

What does all this have to do with testing? I have found that for complex integration tests where there is a lot of setup and teardown, context managers provide a helpful pattern for making compact, simple code, by putting the "context" (state) needed for any given test close to where it is actually needed (and not everywhere else). Careful isolation of state is an important aspect of functional programming.

More on the use of context managers in actual integration tests in the next post.