Context managers

Reuven M. Lerner, PhD
reuven@lerner.co.il

Context managers

- Idea: A class that defines the behavior that should happen before and after a block of code is executed
- Sort of like “advice” in Lisp
- The class defines __enter__ and __exit__

with

- Execute code within a particular context manager using “with”:

```
with CM():

    # __enter__ executes here

    print ‘hello’

    # __exit__ executes here
```

with .. as

- You can add an “as” to the call:

```
with CM() as thing:

    # __enter__ executes here

    print ‘hello’

    # __exit__ executes here
```

with .. as

- When you specify “as,” the named object is assigned to whatever __enter__ returns

Files

```
with open(filename, 'r') as myfile:

  for line in myfile:

    print line
```

CM example

```
class Squealer(object):
    def __init__(self):
        print "Now in init!"
    def __enter__(self):
        print "Now in enter!"
        return self
    def __exit__(self, type, value, traceback):
        print "Now in exit!"
```

__exit__ params

- self is the normal self
- exc_type, exc_value, and traceback describe the exception that was raised (if one was)
- No exception? They're all set to None

What do you return?

- Often, a context manager won't return self (i.e., the context manager object)
- Rather, you set things up, and then return one of the created objects

Using Squealer

```
from squealer import Squealer
with Squealer():
  print "Hello inside"

Now in init!
Now in enter!
Hello inside
Now in exit!
```

If there's an exception

```
from squealer import Squealer
with Squealer():
  raise IOError

Now in init!
Now in enter!
Hello inside
Now in exit!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Trap exception

```
class Squealer():
    def __init__(self):
        print("Now in init!")
    def __enter__(self):
        print("Now in enter!")
    def __exit__(self, type, value, traceback):
       print("Now in exit!")
        return True
```

Trap the exception!

```
from squealer import Squealer
with Squealer():
  raise IOError

Now in init!
Now in enter!
Hello inside
Now in exit!
```

Another example

```
import hashlib
class SecureLog( object ):
    def __init__( self, someFile, marker="-----HASH-----" ):
        self.theFile= someFile
        self.marker= marker
        self.hash= hashlib.md5()
    def write( self, aLine ):
        self.theFile.write( aLine )
        self.hash.update(aLine)
    def finalize( self ):
        self.theFile.write("{}\n{}\n".format(self.marker,
                                             self.hash.hexdigest()))
    def close(self):
        theFile.close()
```

Using SecureLog

```
>>> log = open('/tmp/logfile', 'w')
>>> sl = securelog.SecureLog(log)
>>> sl.write('hi\n')
>>> sl.write('out\n')
>>> sl.write('there\n')
>>> sl.finalize()
>>> sl.close()
```

Manage SecureLog

```
class SecureLogManager( object ):
    def __init__( self, someFile ):
        self.theFile= someFile
    def __enter__( self ):
        self.transLog= SecureLog( self.theFile )
        return self.transLog
    def __exit__( self, type, value, tb ):
        if type is not None:
            pass # Exception occurred
        self.transLog.finalize()
        self.transLog.close()
```

Using the CM

```
result = open( '/tmp/cmlog.txt', 'w' )
with SecureLogManager( result ) as log:
    log.write( "Some Configuration\n" )
    log.write('foo\n')
    log.write('bar\n')
```

Uses for context managers

- Common setup/teardown scenarios
- Closing connections
- Logging
- Trapping and logging errors
- Checking permissions

Using an old Python?

- You have to say

```
from __future__ import with
```

Redirecting I/O

```
import sys
from StringIO import StringIO
class redirect_stdout:
    def __init__(self, target):
        self.stdout = sys.stdout
        self.target = target
    def __enter__(self):
        sys.stdout = self.target
    def __exit__(self, type, value, tb):
        sys.stdout = self.stdout
```

Redirecting

```
out = StringIO()

with redirect_stdout(out):

    print 'Test'
```

contextlib

- You don’t have to create __enter__ and __exit_ methods!
- Instead, you can use contextlib to automate things a bit

Using contextlib

```
from contextlib import contextmanager
@contextmanager
def tag(name):
    print "<{}>".format(name)
    yield
    print "</{}>".format(name)
```

Better yet

```
from contextlib import contextmanager
@contextmanager
def tag(name):
    try:
        print "<{}>".format(name)
        yield
    finally:
        print "</{}>".format(name)
```

Using contextlib

```
#!/usr/bin/env python

from contextlib import contextmanager
@contextmanager
def tag(name):
    print "<{}>".format(name)
    yield
    print "</{}>".format(name)
with tag('b') as t:
    print 'hello'
```