Context Managers (“with” Statement) Python 2.x 2.5–2.7 , Python 3.x 3.0–3.6

While Python's context managers are widely used, few understand the purpose behind their use. These statements, commonly used with reading and writing files, assist the application in conserving system memory and improve resource management by ensuring specific resources are only in use for certain processes. This topic explains and demonstrates the use of Python's context managers.

Examples

Introduction to context managers and the with statement

A context manager is an object that is notified when a context (a block of code) *starts* and *ends* . You commonly use one with the `with` statement. It takes care of the notifying.

For example, file objects are context managers. When a context ends, the file object is closed automatically:

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

The above example is usually simplified by using the `as` keyword:

```
with open(filename) as open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

Anything that ends execution of the block causes the context manager's exit method to be called. This includes exceptions, and can be useful when an error causes you to prematurely exit from an open file or connection. Exiting a script without properly closing files/connections is a bad idea, that may cause data loss or other problems. By using a context manager you can ensure that precautions are always taken to prevent damage or loss in this way. This feature was added in Python 2.5.

Writing your own context manager

A context manager is any object that implements two magic methods `__enter__()` and `__exit__()` (although it can implement other methods as well):

```
class AContextManager():

    def __enter__(self):
        print("Entered")
        # optionally return an object
        return "A-instance"

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exited" + (" (with an exception)" if exc_type else ""))
        # return True if you want to suppress the exception
```

If the context exits with an exception, the information about that exception will be passed as a triple `exc_type` , `exc_value` , `traceback` (these are the same variables as returned by the `sys.exc_info()` function). If the context exits normally, all three of these arguments will be `None` .

If an exception occurs and is passed to the `__exit__` method, the method can return `True` in order to suppress the exception, or the exception will be re-raised at the end of the `__exit__` function.

```
with AContextManager() as a:
    print("a is %r" % a)
# Entered
# a is 'A-instance'
# Exited

with AContextManager() as a:
    print("a is %d" % a)
# Entered
# Exited (with an exception)
# Traceback (most recent call last):
#   File "<stdin>", line 2, in <module>
# TypeError: %d format: a number is required, not str
```

Note that in the second example even though an exception occurs in the middle of the body of the with-statement, the `__exit__` handler still gets executed, before the exception propagates to the outer scope.

If you only need an `__exit__` method, you can return the instance of the context manager:

```
class MyContextManager:
    def __enter__(self):
        return self

    def __exit__(self):
        print('something')
```

Writing your own contextmanager using generator syntax

It is also possible to write a context manager using generator syntax thanks to the `contextlib.contextmanager` decorator:

```
import contextlib

@contextlib.contextmanager
def context_manager(num):
    print('Enter')
    yield num + 1
    print('Exit')

with context_manager(2) as cm:
    # the following instructions are run when the 'yield' point of the context
    # manager is reached.
    # 'cm' will have the value that was yielded
    print('Right in the middle with cm = {}'.format(cm))
```

produces:

```
Enter
Right in the middle with cm = 3
Exit
```

The decorator simplifies the task of writing a context manager by converting a generator into one. Everything before the yield expression becomes the `__enter__` method, the value yielded becomes the value returned by the generator (which can be bound to a variable in the with statement), and everything after the yield expression becomes the `__exit__` method.

If an exception needs to be handled by the context manager, a `try..except..finally` -block can be written in the generator and any exception raised in the `with` -block will be handled by this exception block.

```
@contextlib.contextmanager
def error_handling_context_manager(num):
    print("Enter")
    try:
        yield num + 1
    except ZeroDivisionError:
        print("Caught error")
    finally:
        print("Cleaning up")
    print("Exit")

with error_handling_context_manager(-1) as cm:
    print("Dividing by cm = {}".format(cm))
    print(2 / cm)
```

This produces:

```
Enter
Dividing by cm = 0
Caught error
Cleaning up
Exit
```

Multiple context managers

You can open several content managers at the same time:

```
with open(input_path) as input_file, open(output_path, 'w') as output_file:

    # do something with both files.

    # e.g. copy the contents of input_file into output_file
    for line in input_file:
        output_file.write(line + '\n')
```

It has the same effect as nesting context managers:

```
with open(input_path) as input_file:
    with open(output_path, 'w') as output_file:
        for line in input_file:
            output_file.write(line + '\n')
```

Assigning to a target

Many context managers return an object when entered. You can assign that object to a new name in the `with` statement.

For example, using a database connection in a `with` statement could give you a cursor object:

```
with database_connection as cursor:
    cursor.execute(sql_query)
```

File objects return themselves, this makes it possible to both open the file object and use it as a context manager in one expression:

```
with open(filename) as open_file:
    file_contents = open_file.read()
```

Manage Resources

```
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()
```

`__init__()` method sets up the object, in this case setting up the file name and mode to open file. `__enter__()` opens and returns the file and `__exit__()` just closes it.

Using these magic methods (`__enter__` , `__exit__`) allows you to implement objects which can be used easily `with` the with statement.

Use File class:

```
for _ in range(10000):
    with File('foo.txt', 'w') as f:
        f.write('foo')
```

Syntax

```
with "context_manager"( as "alias")(, "context_manager"( as "alias")?)*:
```

Parameters

Remarks

Context managers are defined in PEP 343 . They are intended to be used as more succinct mechanism for resource management than `try ... finally` constructs. The formal definition is as follows.

> In this PEP, context managers provide `__enter__()` and `__exit__()` methods that are invoked on entry to and exit from the body of the with statement.

It then goes on to define the `with` statement as follows.

```
with EXPR as VAR:
    BLOCK
```

The translation of the above statement is:

```
   mgr = (EXPR)
   exit = type(mgr).__exit__  # Not calling it yet
   value = type(mgr).__enter__(mgr)
   exc = True
   try:
       try:
           VAR = value  # Only if "as VAR" is present
           BLOCK
       except:
           # The exceptional case is handled here
           exc = False
           if not exit(mgr, *sys.exc_info()):
               raise
           # The exception is swallowed if exit() returns true
   finally:
       # The normal and non-local-goto cases are handled here
       if exc:
           exit(mgr, None, None, None)
```