The Curious Case of Python's Context Manager

Discovering the quirks of Python's context manager

Mar 26, 2020 • 9 min read

Python

Python's context managers are great for resource management and stopping the propagation of leaked abstractions. You've probably used it while opening a file or a database connection. Usually it starts with a `with` statement like this:

```python
with open("file.txt", "wt") as f:
    f.write("contents go here")
```

In the above case, `file.txt` gets automatically closed when the execution flow goes out of the scope. This is equivalent to writing:

```python
try:
    f = open("file.txt", "wt")
    text = f.write("contents go here")
```

```
finally:
    f.close()
```

Writing Custom Context Managers

To write a custom context manager, you need to create a class that includes the `__enter__` and `__exit__` methods. Let's recreate a custom context manager that will execute the same workflow as above.

```
class CustomFileOpen:
    """Custom context manager for opening files."""

    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.f = open(self.filename, self.mode)
        return self.f

    def __exit__(self, *args):
        self.f.close()
```

You can use the above class just like a regular context manager.

```
with CustomFileOpen("file.txt", "wt") as f:
    f.write("contents go here")
```

From Generators to Context Managers

Creating context managers by writing a class with `__enter__` and `__exit__` methods, is not difficult. However, you can achieve better brevity by defining them using `contextlib.contextmanager` decorator. This decorator converts a generator function into a context manager. The blueprint for creating context manager decorators goes something like this:

```
@contextmanager
def some_generator(<arguments>):
    <setup>
    try:
        yield <value>
    finally:
        <cleanup>
```

When you use the context manager with the `with` statement:

```
with some_generator(<arguments>) as <variable>:
    <body>
```

It roughly translates to:

```
<setup>
try:
    <variable> = <value>
    <body>
finally:
    <cleanup>
```

The setup code goes before the `try..finally` block. Notice the point where the generator yields. This is where the code block nested in the `with` statement gets executed. After the completion of the code block, the generator is then resumed. If an unhandled exception occurs in the block, it's re-raised inside the generator at the point where the `yield` occurred and then the `finally` block is executed. If no unhandled exception occurs, the code gracefully proceeds to the `finally` block where you run your cleanup code.

Let's implement the same `CustomFileOpen` context manager with `contextmanager` decorator.

```
from contextlib import contextmanager


@contextmanager
def CustomFileOpen(filename, method):
    """Custom context manager for opening a file."""

    f = open(filename, method)
    try:
        yield f
    finally:
        f.close()
```

Now use it just like before:

```
with CustomFileOpen("file.txt", "wt") as f:
    f.write("contents go here")
```

Writing Context Managers as Decorators

You can use context managers as decorators also. To do so, while defining the class, you have to inherit from `contextlib.ContextDecorator` class. Let's make a `RunTime` decorator that will be applied on a file-opening function. The decorator will:

- Print a user provided description of the function
- Print the time it takes to run the function

```
from contextlib import ContextDecorator
from time import time


class RunTime(ContextDecorator):
    """Timing decorator."""

    def __init__(self, description):
        self.description = description

    def __enter__(self):
        print(self.description)
        self.start_time = time()

    def __exit__(self, *args):
        self.end_time = time()
        run_time = self.end_time - self.start_time
        print(f"The function took {run_time} seconds to run.")
```

You can use the decorator like this:

```
@RunTime("This function opens a file")
def custom_file_write(filename, mode, content):
    with open(filename, mode) as f:
        f.write(content)
```

Using the function like this should return:

```
print(custom_file_write("file.txt", "wt", "jello"))
```

```
This function opens a file
The function took 0.0005390644073486328 seconds to run.
None
```

You can also create the same decorator via `contextlib.contextmanager` decorator.

```
from contextlib import contextmanager


@contextmanager
def runtime(description):

    print(description)
    start_time = time()
    try:
        yield
    finally:
        end_time = time()
        run_time = end_time - start_time
        print(f"The function took {run_time} seconds to run.")
```

Nesting Contexts

You can nest multiple context managers to manage resources simultaneously. Consider the following dummy manager:

```
from contextlib import contextmanager


@contextmanager
def get_state(name):
    print("entering:", name)
    yield name
    print("exiting :", name)


# multiple get_state can be nested like this
with get_state("A") as A, get_state("B") as B, get_state("C") as C:
    print("inside with statement:", A, B, C)
```

```
entering: A
entering: B
entering: C
inside with statement: A B C
exiting : C
exiting : B
exiting : A
```

Notice the order they're closed. Context managers are treated as a stack, and should be exited in reverse order in which they're entered. If an exception occurs, this order matters, as any context manager could suppress the exception, at which point the remaining managers will not even get notified of this. The `__exit__` method is also permitted to raise a different exception, and other context managers then should be able to handle that new exception.

Combining Multiple Context Managers

You can combine multiple context managers too. Let's consider these two managers.

```
from contextlib import contextmanager


@contextmanager
def a(name):
    print("entering a:", name)
    yield name
    print("exiting a:", name)


@contextmanager
def b(name):
    print("entering b:", name)
    yield name
    print("exiting b:", name)
```

Now combine these two using the decorator syntax. The following function takes the above define managers `a` and `b` and returns a combined context manager `ab`.

```
@contextmanager
def ab(a, b):
    with a("A") as A, b("B") as B:
        yield (A, B)
```

This can be used as:

```
with ab(a, b) as AB:
    print("Inside the composite context manager:", AB)
```

```
entering a: A
entering b: B
Inside the composite context manager: ('A', 'B')
exiting b: B
exiting a: A
```

If you have variable numbers of context managers and you want to combine them gracefully, `contextlib.ExitStack` is here to help. Let's rewrite context manager `ab` using `ExitStack`. This function takes the individual context managers and their arguments as tuples and returns the combined manager.

```
from contextlib import contextmanager, ExitStack


@contextmanager
def ab(cms, args):
    with ExitStack() as stack:
        yield [stack.enter_context(cm(arg)) for cm, arg in zip(cms, args)]
```

```
with ab((a, b), ("A", "B")) as AB:
    print("Inside the composite context manager:", AB)
```

```
entering a: A
entering b: B
Inside the composite context manager: ['A', 'B']
exiting b: B
exiting a: A
```

`ExitStack` can be also used in cases where you want to manage multiple resources gracefully. For example, suppose, you need to create a list from the contents of multiple files in a directory. Let's see, how you can do so while avoiding accidental memory leakage with robust resource management.

```
from contextlib import ExitStack
from pathlib import Path

# ExitStack ensures all files are properly closed after o/p
with ExitStack() as stack:
    streams = (
        stack.enter_context(open(fname, "r")) for fname in Path("src").rglob("*.py")
    )
    contents = [f.read() for f in streams]
```

Using Context Managers to Create SQLAlchemy Session

If you are familiar with SQLALchemy, Python's SQL toolkit and Object Relational Mapper, then you probably know the usage of `Session` to run a query. A `Session` basically turns any query into a transaction and make it atomic. Context managers can help you write a transaction session in a very elegant way. A basic querying workflow in SQLAlchemy may look like this:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from contextlib import contextmanager

# an Engine, which the Session will use for connection resources
some_engine = create_engine("sqlite://")

# create a configured "Session" class
Session = sessionmaker(bind=some_engine)
```

```python
@contextmanager
def session_scope():
    """Provide a transactional scope around a series of operations."""
    session = Session()
    try:
        yield session
        session.commit()
    except:
        session.rollback()
        raise
    finally:
        session.close()
```

The excerpt above creates an in memory `SQLite` connection and a `session_scope` function with context manager. The session_scope function takes care of committing and rolling back in case of exception automatically. The `session_scope` function can be used to run queries in the following way:

```python
with session_scope() as session:
    myobject = MyObject("foo", "bar")
    session.add(myobject)
```

Abstract Away Exception Handling Monstrosity with Context Managers

This is my absolute favorite use case of context managers. Suppose you want to write a function but want the exception handling logic out of the way. Exception handling logics with sophisticated logging can often obfuscate the core logic of your function. You can write a decorator type context manager that will handle the exceptions for you and decouple these additional code from your main logic. Let's write a decorator that will handle `ZeroDivisionError` and `TypeError` simultaneously.

```python
from contextlib import contextmanager


@contextmanager
```

```python
def errhandler():
    try:
        yield
    except ZeroDivisionError:
        print("This is a custom ZeroDivisionError message.")
        raise
    except TypeError:
        print("This is a custom TypeError message.")
        raise
```

Now use this in a function where these exceptions occur.

```python
@errhandler()
def div(a, b):
    return a // b
```

```python
div("b", 0)
```

```
This is a custom TypeError message.
---------------------------------------------------------------------------


TypeError                                 Traceback (most recent call last)

<ipython-input-43-65497ed57253> in <module>
----> 1 div('b',0)

/usr/lib/python3.8/contextlib.py in inner(*args, **kwds)
        73         def inner(*args, **kwds):
        74             with self._recreate_cm():
---> 75                 return func(*args, **kwds)
        76         return inner
        77



<ipython-input-42-b7041bcaa9e6> in div(a, b)
        1 @errhandler()
        2 def div(a, b):
----> 3     return a // b



TypeError: unsupported operand type(s) for //: 'str' and 'int'
```

You can see that the `errhandler` decorator is doing the heavylifting for you. Pretty neat, huh?

The following one is a more sophisticated example of using context manager to decouple your error handling monstrosity from the main logic. It also hides the elaborate logging logics from the main method.

```
import logging
from contextlib import contextmanager
import traceback
import sys

logging.getLogger(__name__)

logging.basicConfig(
    level=logging.INFO,
    format="\n(asctime)s [%(levelname)s] %(message)s",
    handlers=[logging.FileHandler("./debug.log"), logging.StreamHandler()],
)


class Calculation:
    """Dummy class for demonstrating exception decoupling with contextmanager."""

    def __init__(self, a, b):
        self.a = a
        self.b = b

    @contextmanager
    def errorhandler(self):
        try:
            yield
        except ZeroDivisionError:
            print(
                f"Custom handling of Zero Division Error! Printing "
                "only 2 levels of traceback.."
            )
            logging.exception("ZeroDivisionError")

    def main_func(self):
        """Function that we want to save from nasty error handling logic."""
        with self.errorhandler():
            return self.a / self.b


obj = Calculation(2, 0)
print(obj.main_func())
```

This will return

```
(asctime)s [ERROR] ZeroDivisionError
Traceback (most recent call last):
    File "<ipython-input-44-ff609edb5d6e>", line 25, in errorhandler
    yield
    File "<ipython-input-44-ff609edb5d6e>", line 37, in main_func
    return self.a / self.b
ZeroDivisionError: division by zero


Custom handling of Zero Division Error! Printing only 2 levels of traceback..
None
```

Persistent Parameters Across Http Requests with Context Managers

Another great use case for context managers is making parameters persistent across multiple http requests. Python's `requests` library has a `Session` object that will let you easily achieve this. So, if you're making several requests to the same host, the underlying TCP connection will be reused, which can result in a significant performance increase. The following example is taken directly from requests' official docs. Let's persist some cookies across requests.

```
with requests.Session() as session:
    session.get("http://httpbin.org/cookies/set/sessioncookie/123456789")
    response = session.get("http://httpbin.org/cookies")
    print(response.text)
```

This should show:

```
{
  "cookies": {
    "sessioncookie": "123456789"
  }
}
```

Remarks

All the code snippets are updated for python `3.8`. To avoid redundencies, I have purposefully excluded examples of nested with statements and now deprecated `contextlib.nested` function to create nested context managers.

Resources

1. Python Contextlib Documentation
2. Python with Context Manager - Jeff Knupp
3. SQLALchemy Session Creation
4. Scipy Lectures: Context Managers
5. Merging Context Managers

4 Comments - *powered by utteranc.es*

rednafi commented on May 20, 2020 Owner

The comment section is now working!

Mlevydaniel commented on Jun 14, 2020

Great article! I learned a lot from him. Thanks for taking the time to write it.

❤️ 1

arantesdv commented on Apr 27, 2021

Great article

johnklee commented on Apr 28, 2021

Great article!

Write Preview

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub

Subscribe

Sporadic cogitations on software, tech & personal beliefs