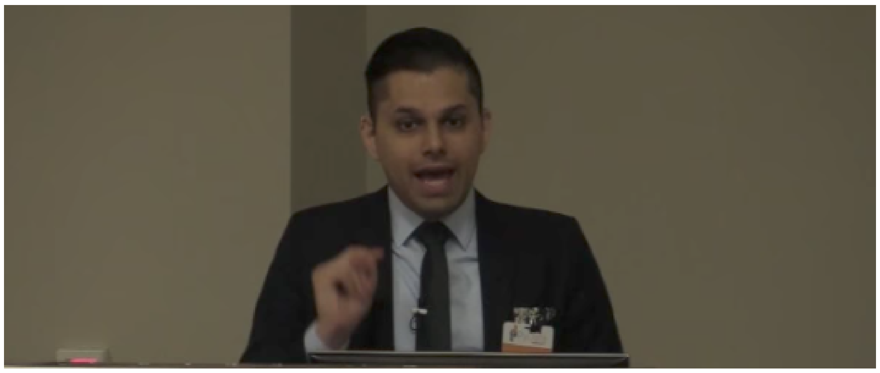
Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

data to do so

Aug 7, 2017 · 5 min read

PyData17—"What Does It Take To Be A Python Expert?"



I came across an interesting lecture from PyData17 from James Powell, that works for Microsoft. In his lecture, he focuses on 4 aspects that make an expert Python programmer.

watch the video

First aspect—data model classes, underscore functions, protocol view in python

The behavior of classes for primal operations of the language should be overlooked and is not used enough. Use the 'underscore' functions when writing Classes to compact your code and add modularity.

- __init__- object constructor
- __add__- addition of objects
- __repr__- what to show when string representation of the object is required
- __len__- what to return when calling len(self)
- __call__—*()->?

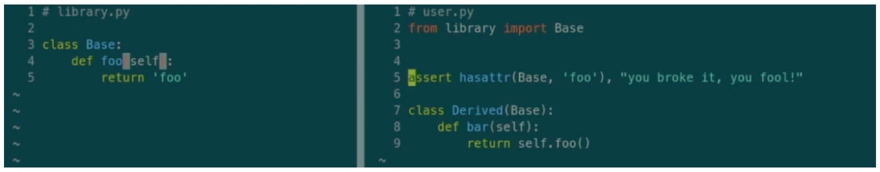
He describes this mechanism very nicely—the underscore function describes a protocol and there is always some high level operation that triggers this protocol (new, +, len etc). Usually the protocol just delegates the same protocol to the internal components of the class—we see this pattern all the time.

Read more on documentation.

Second aspect—Base/Derived class constraints

The speaker talks about a scenario where your code derives from a library your do not own.

Scenario 1—You are on the "right" side of the two-sided screen below. You wrote your code with certain preconditions —for example, that 'foo' method exists in the *Base* class you derive from. You can be sure if you **assert** those preconditions before declaring your actual code.

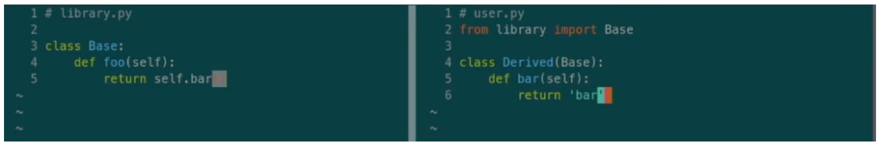


```python
# library.py

class Base:
    def foo(self):
        return 'foo'
```

```python
# user.py
from library import Base


assert hasattr(Base, 'foo'), "you broke it, you fool!"

class Derived(Base):
    def bar(self):
        return self.foo()
```

assert hasattr(Base, 'foo'), "you broke it…"

Scenario 2—You are on the "left" side of the screen below and you depend on the completeness of the classes deriving from you (you **are** the Base class now). There are several solutions for that.



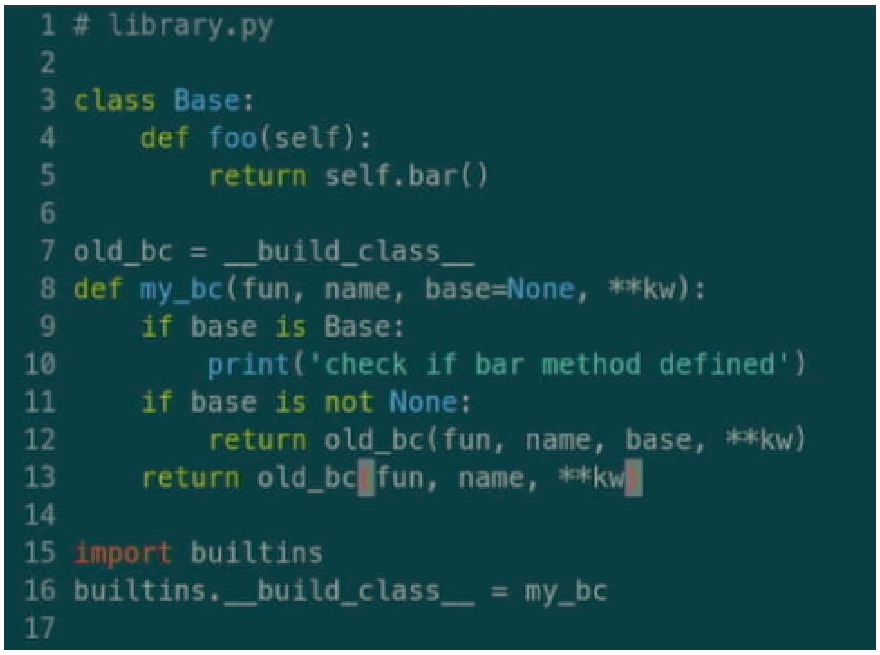
```python
# library.py

class Base:
    def foo(self):
        return self.bar
```

```python
# user.py
from library import Base

class Derived(Base):
    def bar(self):
        return 'bar'
```

Base uses self.bar from the Derived

Obviously you can't assert preconditions regarding Derive because you have no access to it before actual runtime.

option #1—overload the default **__build_class__** function that is called every time you build a class in runtime. In that function you get all the data you need to concur the completeness of the Derived class.

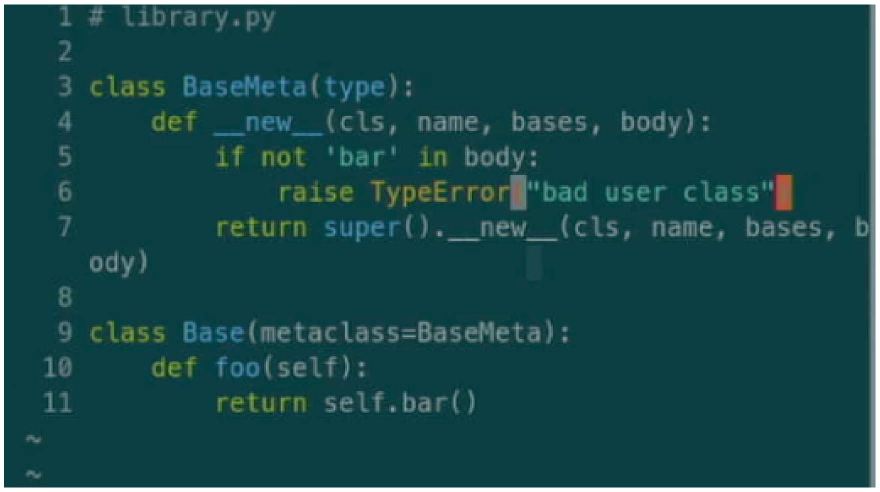


```python
# library.py

class Base:
    def foo(self):
        return self.bar()

old_bc = __build_class__
def my_bc(fun, name, base=None, **kw):
    if base is Base:
        print('check if bar method defined')
    if base is not None:
        return old_bc(fun, name, base, **kw)
    return old_bc(fun, name, **kw)

import builtins
builtins.__build_class__ = my_bc

```

Forcing constraints from Base class to Derived class before any actual business logic is applied

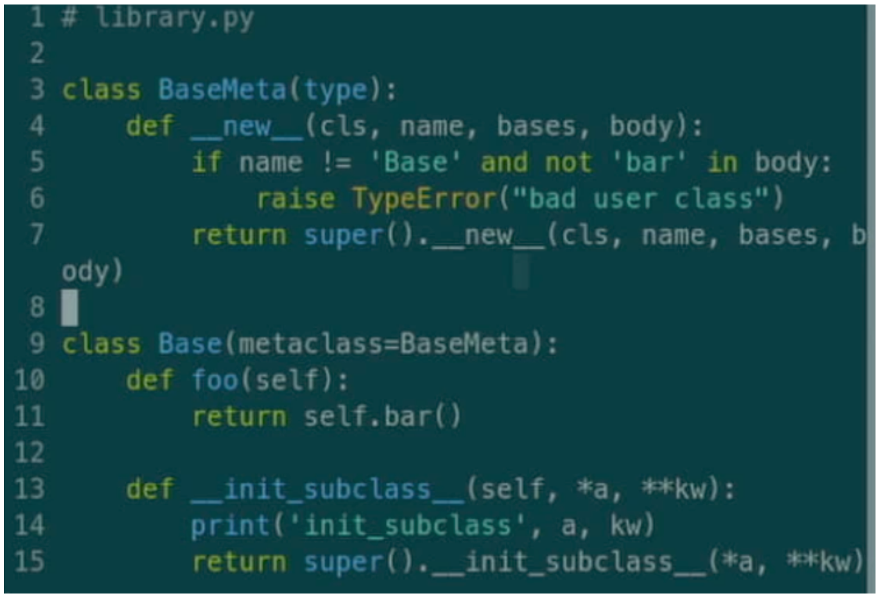
This is because python is a protocol-based class and everything can be done in runtime. This is why we have the __build_class__ function available for us and we can hook too it directly.

option #2—**Metaclass**; A straightforward approach to make sure that you can **enforce constraints down the class hierarchy** from a Base class to a Derived class.



```
# library.py

class BaseMeta(type):
    def __new__(cls, name, bases, body):
        if not 'bar' in body:
            raise TypeError "bad user class"
        return super().__new__(cls, name, bases, b
ody)

class Base(metaclass=BaseMeta):
    def foo(self):
        return self.bar()
~
~
```

option #3—(Py3) __init_subclass__ —simple customization of class creation—there is an introductory video you can watch. He’s not going into details in the lecture—but it’s important to understand the concept.

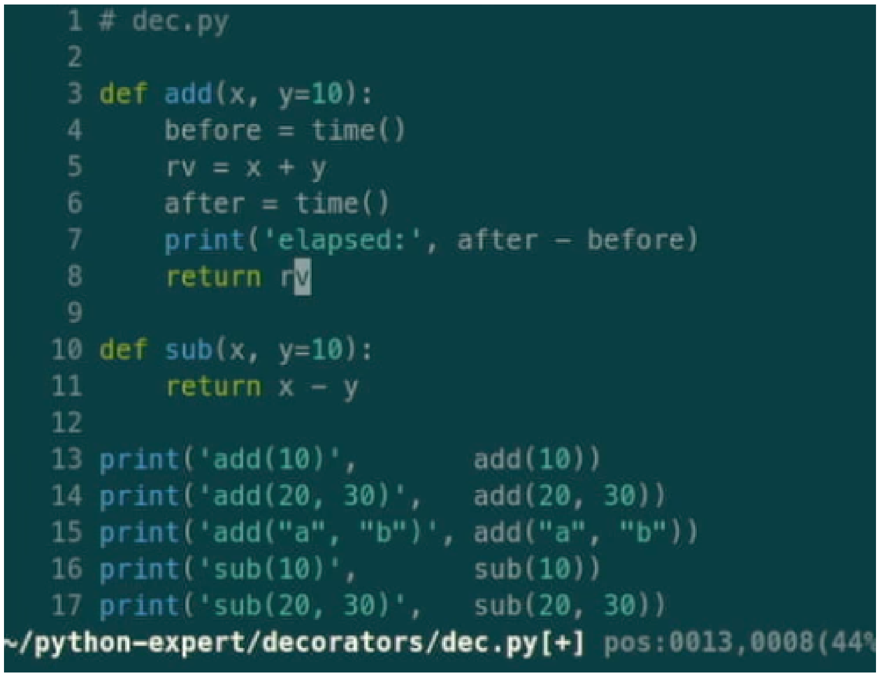


```
# library.py

class BaseMeta(type):
    def __new__(cls, name, bases, body):
        if name != 'Base' and not 'bar' in body:
            raise TypeError("bad user class")
        return super().__new__(cls, name, bases, b
ody)

class Base(metaclass=BaseMeta):
    def foo(self):
        return self.bar()

    def __init_subclass__(self, *a, **kw):
        print('init_subclass', a, kw)
        return super().__init_subclass__(*a, **kw)
```

Third aspect—Decorators

Everything in python is accessible in runtime and you can ask it questions in runtime. Things like address, name, source code, what file are you in , what module are you defined in, what like are you defined in.

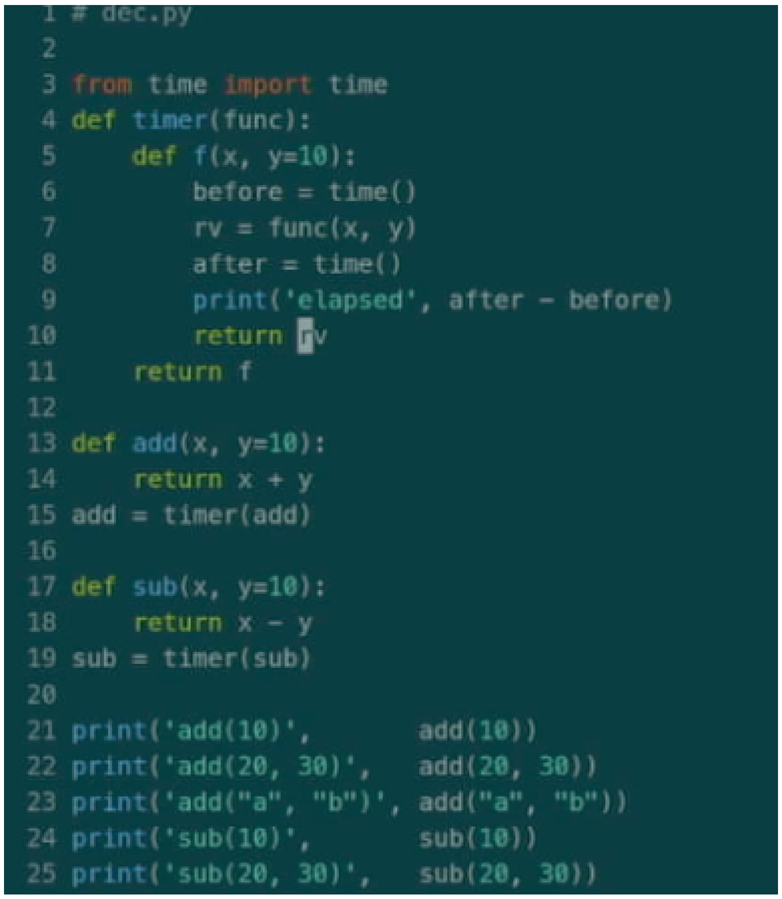
He speaks of a common scenario where you want a recurring functionality attached to a function. For example, to time the execution of function(s).



```
# dec.py

def add(x, y=10):
    before = time()
    rv = x + y
    after = time()
    print('elapsed:', after - before)
    return rv

def sub(x, y=10):
    return x - y

print('add(10)',         add(10))
print('add(20, 30)',     add(20, 30))
print('add("a", "b")', add("a", "b"))
print('sub(10)',         sub(10))
print('sub(20, 30)',     sub(20, 30))
~/python-expert/decorators/dec.py[+] pos:0013,0008(44
```

Do we need to add the execution duration tracking code to each function??

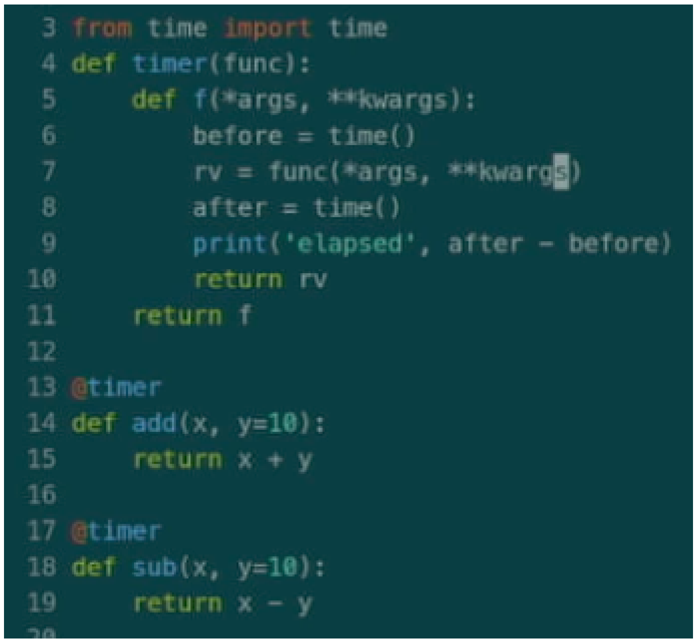
A solution would be to create a new function wrapper at runtime that return a new function with the wrapped functionality you want to attach your core functions.



```python
# dec.py

from time import time
def timer(func):
    def f(x, y=10):
        before = time()
        rv = func(x, y)
        after = time()
        print('elapsed', after - before)
        return rv
    return f

def add(x, y=10):
    return x + y
add = timer(add)

def sub(x, y=10):
    return x - y
sub = timer(sub)

print('add(10)',       add(10))
print('add(20, 30)',   add(20, 30))
print('add("a", "b")', add("a", "b"))
print('sub(10)',       sub(10))
print('sub(20, 30)',   sub(20, 30))
```

a function to return a new function. this feels very much like a factory design pattern…

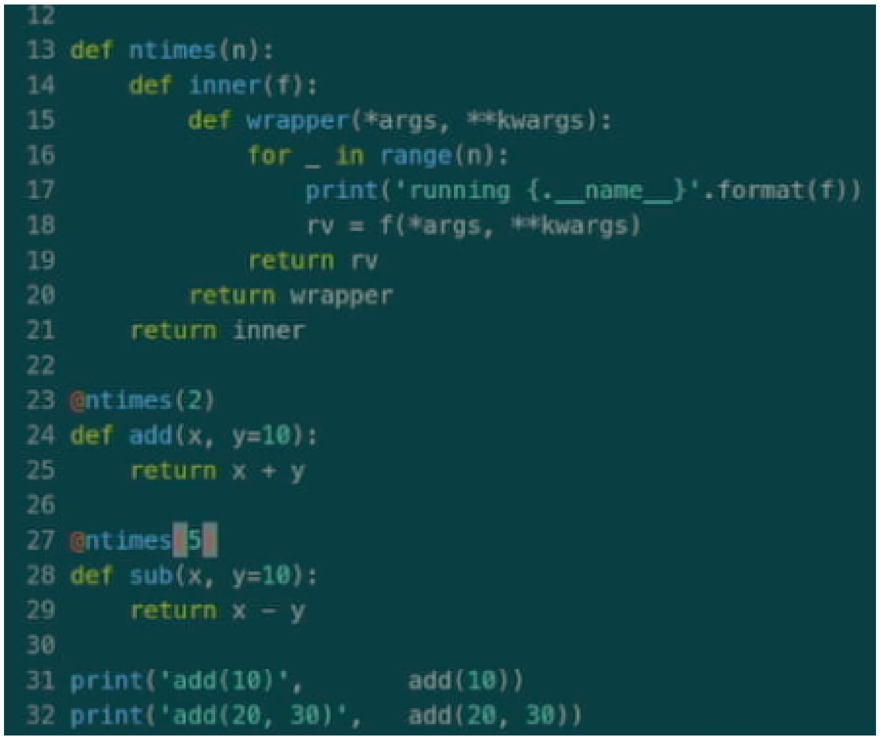
Python takes this design pattern and make it available as **Decorators**.



```python
from time import time
def timer(func):
    def f(*args, **kwargs):
        before = time()
        rv = func(*args, **kwargs)
        after = time()
        print('elapsed', after - before)
        return rv
    return f

@timer
def add(x, y=10):
    return x + y

@timer
def sub(x, y=10):
    return x - y
```

This code will do the same things as before.

Than he speaks of higher-order decorators… so cool



```python
def ntimes(n):
    def inner(f):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                print('running {.__name__}'.format(f))
                rv = f(*args, **kwargs)
            return rv
        return wrapper
    return inner

@ntimes(2)
def add(x, y=10):
    return x + y

@ntimes(5)
def sub(x, y=10):
    return x - y

print('add(10)',       add(10))
print('add(20, 30)',   add(20, 30))
```

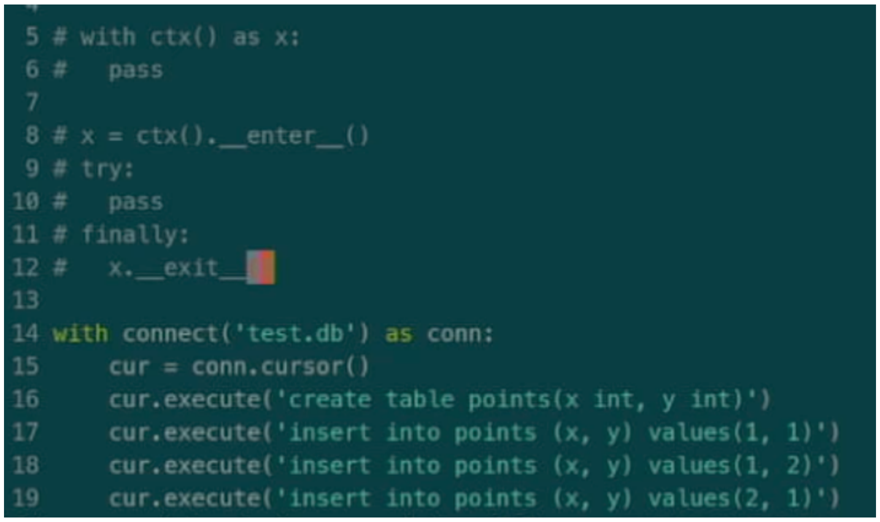
Its a double wrapper, one to take the arguments of the decorator and to return a new WRAPPED decorator. We can go as deep as we want.

Forth aspect — Generators

Hmmm he didn’t say something especially important about that. But he says this is the correct way to approach concurrency.

Fifth aspect — Context Managers / Transactional support — Setup & Tare

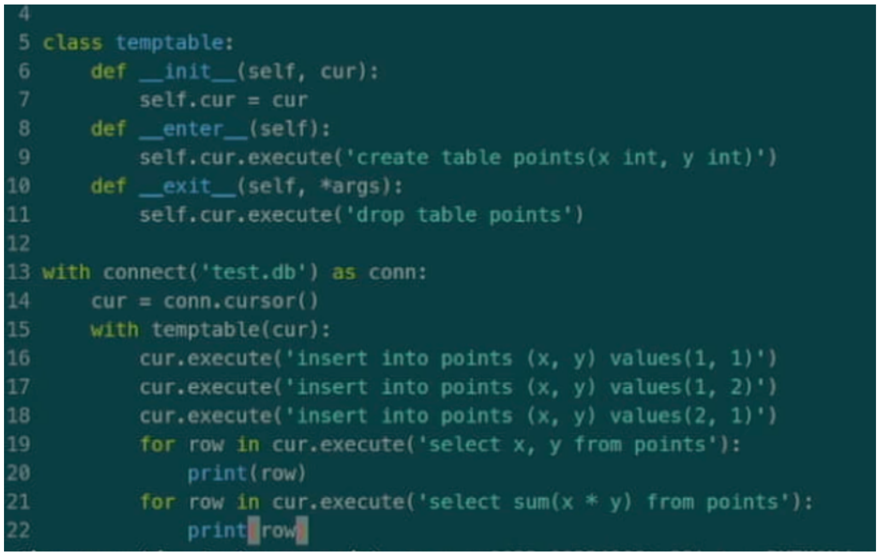
The Setup&Tare mechanism is implemented as a protocol in python.



```
# with ctx() as x:
#   pass

# x = ctx().__enter__()
# try:
#   pass
# finally:
#   x.__exit__

with connect('test.db') as conn:
    cur = conn.cursor()
    cur.execute('create table points(x int, y int)')
    cur.execute('insert into points (x, y) values(1, 1)')
    cur.execute('insert into points (x, y) values(1, 2)')
    cur.execute('insert into points (x, y) values(2, 1)')
```

what happens behind the scenes

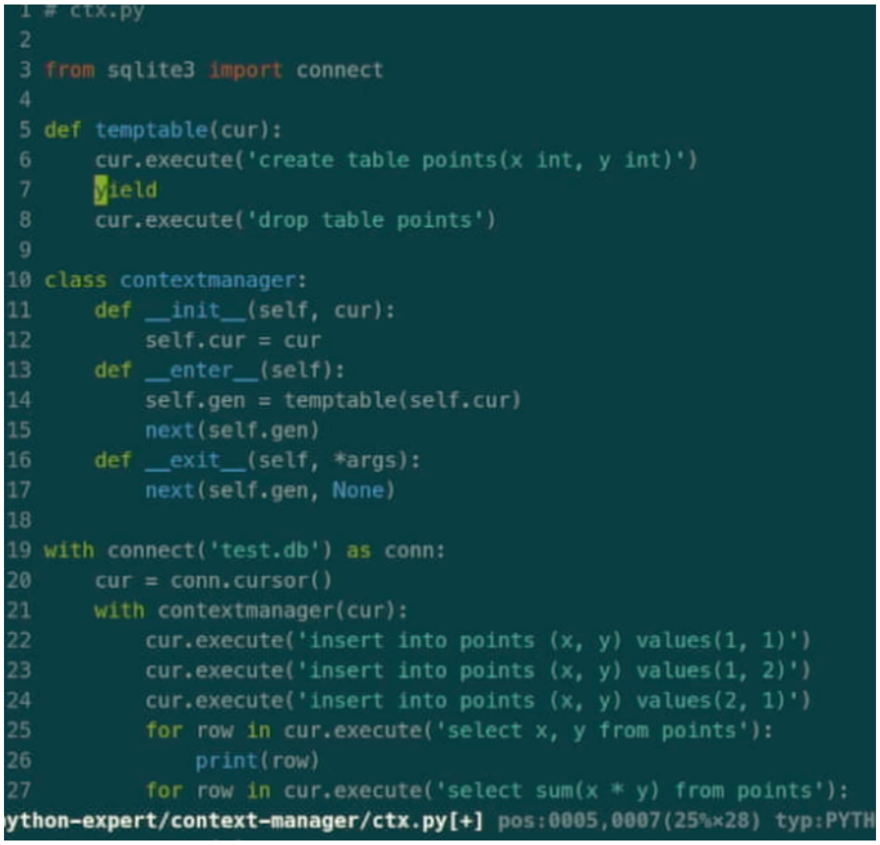
Implement __enter and __exit__ and you can use this syntax with your own classes like a pro. In the following example we create and destroy a sqlite table with this mechanism:



```
class temptable:
    def __init__(self, cur):
        self.cur = cur
    def __enter__(self):
        self.cur.execute('create table points(x int, y int)')
    def __exit__(self, *args):
        self.cur.execute('drop table points')

with connect('test.db') as conn:
    cur = conn.cursor()
    with temptable(cur):
        cur.execute('insert into points (x, y) values(1, 1)')
        cur.execute('insert into points (x, y) values(1, 2)')
        cur.execute('insert into points (x, y) values(2, 1)')
        for row in cur.execute('select x, y from points'):
            print(row)
        for row in cur.execute('select sum(x * y) from points'):
            print(row)
```

Example Content Manager

Because the sequence of __enter__ before __exit__ is important, one can use Generator to validate this behavior and write a more robust code:



```
# ctx.py

from sqlite3 import connect

def temptable(cur):
    cur.execute('create table points(x int, y int)')
    yield
    cur.execute('drop table points')

class contextmanager:
    def __init__(self, cur):
        self.cur = cur
    def __enter__(self):
        self.gen = temptable(self.cur)
        next(self.gen)
    def __exit__(self, *args):
        next(self.gen, None)

with connect('test.db') as conn:
    cur = conn.cursor()
    with contextmanager(cur):
        cur.execute('insert into points (x, y) values(1, 1)')
        cur.execute('insert into points (x, y) values(1, 2)')
        cur.execute('insert into points (x, y) values(2, 1)')
        for row in cur.execute('select x, y from points'):
            print(row)
        for row in cur.execute('select sum(x * y) from points'):
ython-expert/context-manager/ctx.py[+] pos:0005,0007(25%×28) typ:PYTH
```

Content Manager with Generator

And this brings him to talk about Contextlib. Contextlib provides you the entire class to Setup and Tare. Only you need to provide is the generator function that the contextlib works on when __enter__ing and __exit__ing.

Another good example is in Fun With `contextlib`.

> *”An Export level code in Python is code that has a certain clarity to where and when a feature should be used. It’s code that doesn’t waste the time of the person writing it”*

This was an amazing talk! Thank you James Powell!