Context manager

Context managers are very simple metaphore and common metaphore we see all over the place.

The basic context manager looks like this:

In [ ]:

```
# with open('ctx.py') as f:
#     pass
```

The reason for using context manager here is because there is corresponding setup and teardown. If we open the file we got to close the file. For example on windows if you don't close the file you might not be able to delete it later. This is especially true if the file is backed by some storage where it is not necessarily automatically flushed. If we open file and write to it we want to make sure there is a flush to disk because we don't want to loose the data.

Fundamentally there is the idea here that we have some setup action and teardown action and we want to match them together.

Here is an example of context manager wrapping some database operations in SQLite (the connect method is the context manager itself):

In [11]:

```
from sqlite3 import connect

with connect('test.db') as conn:
    # we connect to some database and have some cursor on that database
    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)')
    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)
    cur.execute('drop table points')
```

```
(1, 1)
(1, 2)
(2, 1)
(5,)
```

We created a table and we dropped a table. And let's assume we don't have transactional support and we have to be in charge of this paring (create and drop). We want to make sure they are both get done irrespective of some error that might pop in the middle.

We created a table and we dropped a table. And let's assume we don't have transactional support and we have to be in charge of this paring (create and drop). We want to make sure they are both get done irrespective of some error that might pop in the middle.

There is always in python some top-level syntax or some function and some underscore method that implements it.

In [12]:

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

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

So that is how we write a context manager. We implement __enter__ and __exit__. There are some arguments that should be passed into it. We will create tamptable context manager.

In [18]:

```
class temptable:
    # in needs to be initialized with a cursor
    def __init__(self, cur):
        self.cur = cur
    def __enter__(self):
        print('__enter__')
        # the enter just executes one statement
        self.cur.execute('create table points(x int, y int)')
    def __exit__(self, *args):
        print('__exit__')
         # the exit just executes one statement
        self.cur.execute('drop table points')
```

And thats it.

In [19]:

```
with connect('test.db') as conn:
    # we connect to some database and have some cursor on that database
    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)
```

```
__enter__
(1, 1)
(1, 2)
(2, 1)
(5,)
__exit__
```

That code works every time because we destroy the table every time.

So content managers have very clear and unambigious metaphore behind them. But what if we want to exit the call before the enter? We shouldn't, the enter should always be called before the exit, so we see some sequencing, so that offers us a generator.

In [24]:

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

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

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'):
            print(row)
```

```
__enter__
created table
(1, 1)
(1, 2)
(2, 1)
(5,)
dropped table
__exit__
```

We can generalize it.

In [26]:

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

class contextmanager:
    def __init__(self, gen):
        self.gen = gen
    def __call__(self, *args, **kwargs):
        self.args, self.kwargs = args, kwargs
        return self
    def __enter__(self):
        self.gen_inst = self.gen(*self.args, **self.kwargs)
        print('__enter__')
        next(self.gen_inst)
    def __exit__(self, *args):
        next(self.gen_inst, None)
        print('__exit__')

with connect('test.db') as conn:
    cur = conn.cursor()
    # this line looks ugly
    with contextmanager(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)
```

```
__enter__
created table
(1, 1)
(1, 2)
(2, 1)
(5,)
dropped table
__exit__
```

In [28]:

```
class contextmanager:
    def __init__(self, gen):
        self.gen = gen
    def __call__(self, *args, **kwargs):
        self.args, self.kwargs = args, kwargs
        return self
    def __enter__(self):
        self.gen_inst = self.gen(*self.args, **self.kwargs)
        print('__enter__')
        next(self.gen_inst)
    def __exit__(self, *args):
        next(self.gen_inst, None)
        print('__exit__')

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

# wee took our generator and wrapped in some object
temptable = contextmanager(temptable)


with connect('test.db') as conn:
    cur = conn.cursor()
    # this line looks ugly
    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)
```

```
__enter__
created table
(1, 1)
(1, 2)
(2, 1)
(5,)
dropped table
__exit__
```

But we just use the decorator syntax!

In [30]:

```
class contextmanager:
    def __init__(self, gen):
        self.gen = gen
    def __call__(self, *args, **kwargs):
        self.args, self.kwargs = args, kwargs
        return self
    def __enter__(self):
        self.gen_inst = self.gen(*self.args, **self.kwargs)
        print('__enter__')
        next(self.gen_inst)
    def __exit__(self, *args):
        next(self.gen_inst, None)
        print('__exit__')

@contextmanager
def temptable(cur):
    cur.execute('create table points(x int, y int)')
    print('created table')
    yield
    cur.execute('drop table points')
    print('dropped table')

with connect('test.db') as conn:
    cur = conn.cursor()
    # this line looks ugly
    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)
```

```
__enter__
created table
(1, 1)
(1, 2)
(2, 1)
(5,)
dropped table
__exit__
```

It turns out that we don't have to write class contextmanager at all. It sits in the library *contextlib*.

In [31]:

```
from sqlite3 import connect
from contextlib import contextmanager

@contextmanager
def temptable(cur):
    cur.execute('create table points(x int, y int)')
    print('created table')
    yield
    cur.execute('drop table points')
    print('dropped table')

with connect('test.db') as conn:
    cur = conn.cursor()
    # this line looks ugly
    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)
```

```
created table
(1, 1)
(1, 2)
(2, 1)
(5,)
dropped table
```

@contextmanager is just a decorator that turns a generator into a context manager.

In [32]:

```
from sqlite3 import connect
from contextlib import contextmanager

@contextmanager
def temptable(cur):
    cur.execute('create table points(x int, y int)')
    try:
        yield
    finally:
        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)
```

```
(1, 1)
(1, 2)
(2, 1)
(5,)
```

This is the example that combines three core features of python together: generators, context managers and decorators. And this three features with very clear conceptual meaning where each piece of its conceptual meaning fits together. A context manager is merely some piece of code that pairs setup and teardwon actions. A generator is merely some form of syntax that allows us to enforce sequencing and enterliving. And the context manager requires interliving because the setup is interlived with the actual action you do in the block. There is a sequencing where the teardown has to be done before the action, so it makes sense to have a generator here as well. Finally, we need something to adapt the generator to adapt this generator to the data model we looked at the very beginning. We have this underscore methods and we have to find some way to take how the generator works and fit it into those underscore methods. One of the things we need to do is take this generator object and wrap it in some function. That wrapping is core to how python works. However, there does happen to be a feature called decorators that allows us a nice and convinient syntax for doing that exactly.

In python expert level code is a code that has a certain clarity to where and when the feature should be used, it is a code that does not waste a time of the person who writes this code Because they say to themselves 'I have this pattern, python has this mechanism, I fit them together and everything just seamlessly works.

The syntax and implementation details are secondary to the core conceptual understanding of what this feature means.