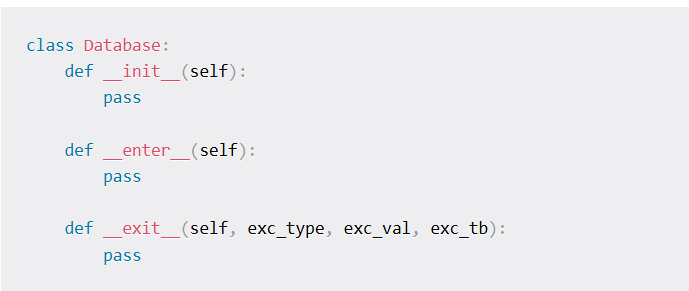
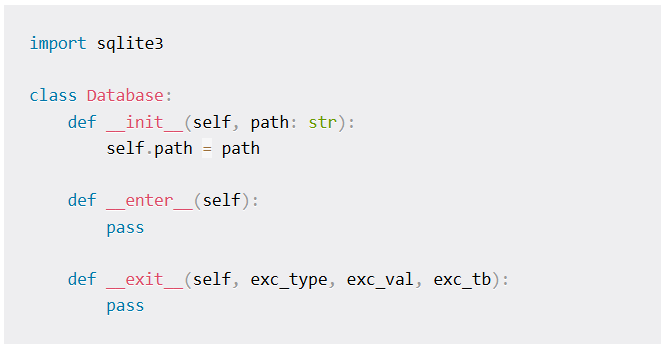
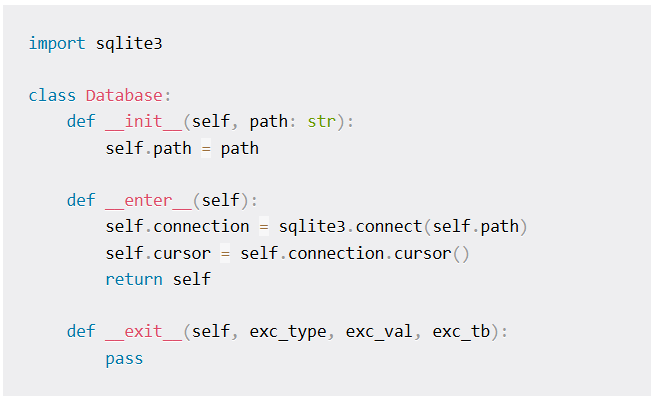
**How to Create a Class Based Context Manager in Python**

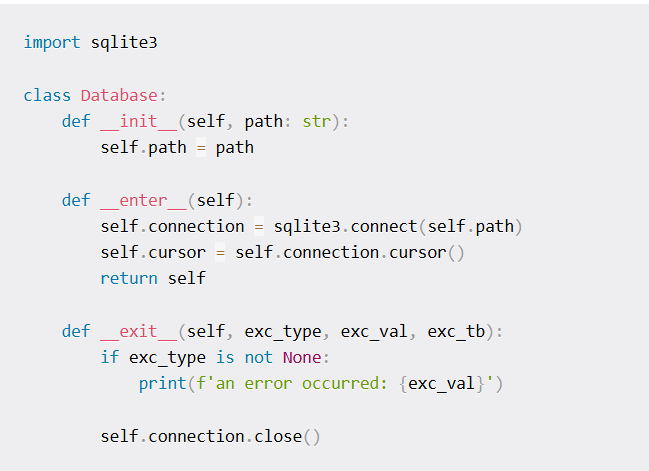
To write a class-based context manager in Python, you need to create an empty class with three specific methods:

The first one is obviously the class constructor that doesn't accept any parameter yet. It'll be responsible for accepting a database path:

The \_\_enter\_\_() method handles the task of setting up the resource. This is where you establish the connection and instantiate the cursor:

However you can not return two objects at once so you have to return the instance of the class itself.

Finally, the \_\_exit\_\_() method handles the task of closing the external resource in question.

You can use this context manager in conjunction with the with statement in your code as follows:

Evident from the output of the print() function call, the program has successfully stored and retrieved the given data from the database.

Without the with statement, Database is just a plain old class. However, the moment you put with infront of it, the three methods hop into action.

The \_\_init\_\_() method is the initializer and works identically to any other plain Python class's initializer method. It takes the path to the database.

The \_\_enter\_\_() method sets up the connection to the database and returns the instance of the context manager class to the target variable, db in this case.

This target variable is now encapsulating both the connection and the cursor objects. You can access them as db.connection and db.cursor respectively.

Once the code inside the with block finishes running, the \_\_exit\_\_() method will execute and close the active connection to the database.

You can handle any exception that may occur during the execution inside the \_\_exit\_\_() method. If there is an exception, exc\_type holds the type of the exception, exc\_val holds the value of the exception, exc\_tb holds the traceback.

If there is no exception, the three variables will have a value of None. I'll not get into the details of exception handling in this article since that can take on many forms depending on what you're dealing with.

To make this custom context manager accessible from anywhere in the program, you can put it into its own separate module or even package.

This is far better solution than the try...except...finally ladder you saw earlier. You don't have to repeat yourself and chances of a human error is lower.

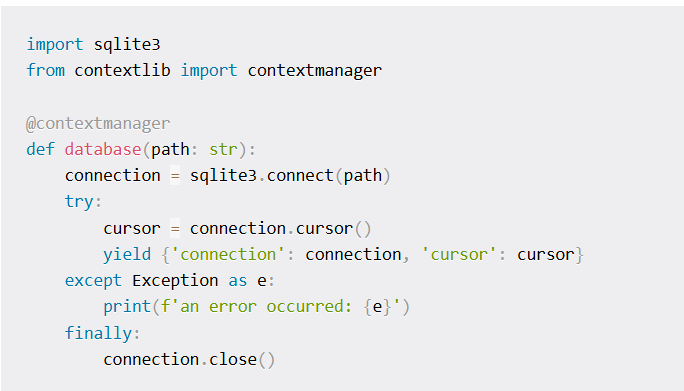
**How to Create a Generator Based Context Manager in Python**

Evident from the title of this section, this approach uses a generator instead of a class to implement a context manager.

Syntactically, generators are almost the same as normal functions, except that you need to use yield instead of return in a generator.

Writing a generator-based context manager requires less code but it also loses some of its readability.

You can write the generator-based equivalent of the class-based Database context manager as follows:

Instead of a class, you have a generator function here so there is no initializer. Instead, the function itself can accept the path to the database as a parameter.

Within a try block, you can establish a connection to the database, instantiate the cursor, and return both objects to the user.

You can write yield connection, cursor to return the two objects but in that case the generator will return them as a tuple.

I prefer to use strings over numbers as accessors and that's why I have put the two objects inside a dictionary with descriptive keys.

The except block will run in case of an exception. Feel free to implement any exception handling strategy that you see fit.

The finally block will run unconditionally and close the open connection at the end of the with block.

Since there are no \_\_enter\_\_() or \_\_exit\_\_() methods either, you need to decorate the generator with the @contextmanager decorator.

This decorator defines a factory function for with statement context managers, without needing to create a class or separate \_\_enter\_\_() and \_\_exit\_\_() methods.

Usage of this context manager is identical to its class-based conterpart except the capitalization of its name.

Since db is a dictionary instead of an object in this case, you will need to use square braces or the get() method to access the connection or cursor object.