Using SQLite 3 with Flask

In Flask you can easily implement the opening of database connections on demand and closing them when the context dies (usually at the end of the request).

Here is a simple example of how you can use SQLite 3 with Flask:

```
import sqlite3
from flask import g

DATABASE = '/path/to/database.db'

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
    return db

@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()
```

Now, to use the database, the application must either have an active application context (which is always true if there is a request in flight) or create an application context itself. At that point the `get_db` function can be used to get the current database connection. Whenever the context is destroyed the database connection will be terminated.

Note: if you use Flask 0.9 or older you need to use `flask._app_ctx_stack.top` instead of `g` as the **`flask.g`** object was bound to the request and not application context.

Example:

```
@app.route('/')
def index():
    cur = get_db().cursor()
    ...
```

Note:

Please keep in mind that the teardown request and appcontext functions are always executed, even if a before-request handler failed or was never executed. Because of this we have to make sure here that the database is there before we close it.

Connect on Demand

The upside of this approach (connecting on first use) is that this will only open the connection if truly necessary. If you want to use this code outside a request context you can use it in a Python shell by opening the application context by hand:

```
with app.app_context():
    # now you can use get_db()
```

Easy Querying

Now in each request handling function you can access *get_db()* to get the current open database connection. To simplify working with SQLite, a row factory function is useful. It is executed for every result returned from the database to convert the result. For instance, in order to get dictionaries instead of tuples, this could be inserted into the `get_db` function we created above:

```
def make_dicts(cursor, row):
    return dict((cursor.description[idx][0], value)
                for idx, value in enumerate(row))

db.row_factory = make_dicts
```

This will make the sqlite3 module return dicts for this database connection, which are much nicer to deal with. Even more simply, we could place this in `get_db` instead:

```
db.row_factory = sqlite3.Row
```

This would use Row objects rather than dicts to return the results of queries. These are `namedtuple` s, so we can access them either by index or by key. For example, assuming we have a `sqlite3.Row` called `r` for the rows `id`, `FirstName`, `LastName`, and `MiddleInitial`:

```
>>> # You can get values based on the row's name
>>> r['FirstName']
John
>>> # Or, you can get them based on index
>>> r[1]
John
# Row objects are also iterable:
```

```
>>> for value in r:
...     print(value)
1
John
Doe
M
```

Additionally, it is a good idea to provide a query function that combines getting the cursor, executing and fetching the results:

```
def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv
```

This handy little function, in combination with a row factory, makes working with the database much more pleasant than it is by just using the raw cursor and connection objects.

Here is how you can use it:

```
for user in query_db('select * from users'):
    print user['username'], 'has the id', user['user_id']
```

Or if you just want a single result:

```
user = query_db('select * from users where username = ?',
                [the_username], one=True)
if user is None:
    print 'No such user'
else:
    print the_username, 'has the id', user['user_id']
```

To pass variable parts to the SQL statement, use a question mark in the statement and pass in the arguments as a list. Never directly add them to the SQL statement with string formatting because this makes it possible to attack the application using SQL Injections.

Initial Schemas

Relational databases need schemas, so applications often ship a *schema.sql* file that creates the database. It's a good idea to provide a function that creates the database based on that schema. This function can do that for you:

```
def init_db():
    with app.app_context():
        db = get_db()
        with app.open_resource('schema.sql', mode='r') as f:
            db.cursor().executescript(f.read())
        db.commit()
```

You can then create such a database from the Python shell:

```
>>> from yourapplication import init_db
>>> init_db()
```