Using the sqlite3 database

Although there are Python modules for many databases, for the following examples we'll look at the one that comes included with Python: sqlite3. Although not suited for large, high-traffic applications, sqlite3 has two advantages: first, because it's part of the standard library it can be used anywhere you need a database, without worrying about adding dependencies; second, sqlite3 stores all of its records in a local file, so it doesn't need both a client and server, like MySQL or other common databases. These features make sqlite3 a handy option for both smaller applications and quick prototypes.
To use a sqlite3 database, the first thing you need is a connection object. Getting a connection object requires only calling the connect function with the name of file that will be used to store the data:

```
>>> import sqlite3
>>> conn = sqlite3.connect("datafile")
```

It's also possible to hold the data in memory by using ":memory:" as the filename. For storing Python integers, strings, and floats, nothing more is needed. If you want sqlite3 to automatically convert query results for some columns into other types, it's useful to include the detect_types parameter set to sqlite3.PARSE_DECLTYPES| sqlite3.PARSE_COLNAMES, which will direct the connection object to parse the name and types of columns in queries and attempt to match them with converters you've already defined.
The second step is to create a cursor object from the connection:

```
>>> cursor = conn.cursor()
>>> cursor
<sqlite3.Cursor object at 0xb7a12980>
```

At this point, you're able to make queries against the database. In our current situation, because there are no tables or records in the database yet, we need to create one and insert a couple of records:

```
>>> cursor.execute("create table test (name text, count integer)")
>>> cursor.execute("insert into test (name, count) values ('Bob', 1)")
>>> cursor.execute("insert into test (name, count) values (?, ?)",
...         ("Jill", 15))
```

The last insert query illustrates the preferred way to make a query with variables; rather than constructing the query string, it's more secure to use a ? for each variable and then pass the variables as a tuple parameter to the execute method. The advantage is that you don't need to worry about incorrectly escaping a value; sqlite3 takes care of it for you.
You can also use variable names prefixed with : in the query and pass in a corresponding dictionary with the values to be inserted:

```
>>> cursor.execute("insert into test (name, count) values (:username, \
:usercount)", {"username": "Joe", "usercount": 10})
```

After a table is populated, you can query the data using SQL commands, again using either ? for variable binding or names and dictionaries:

```
>>> result = cursor.execute("select * from test")
>>> print(result.fetchall())
[('Bob', 1), ('Jill', 15), ('Joe', 10)]
>>> result = cursor.execute("select * from test where name like :name",
...         {"name": "bob"})
>>> print(result.fetchall()) [('Bob', 1)]
>>> cursor.execute("update test set count=? where name=?", (20, "Jill"))
>>> result = cursor.execute("select * from test")
>>> print(result.fetchall())
[('Bob', 1), ('Jill', 20), ('Joe', 10)]
```

In addition to the fetchall method, the fetchone method gets one row of the result and fetchmany returns an arbitrary number of rows. For convenience, it's also possible to iterate over a cursor object's rows similar to iterating over a file:

```
>>> result = cursor.execute("select * from test")
>>> for row in result:
...         print(row)
... ('Bob', 1)
('Jill', 20)
('Joe', 10)
```

Finally, by default, sqlite3 doesn't immediately commit transactions. That means you have the option of rolling back a transaction if it fails, but it also means you need to use the connection object's commit method to ensure that any changes made have been saved. This is a particularly good idea before you close a connection to a database, because the close method doesn't automatically commit any active transactions:

```
>>> cursor.execute("update test set count=? where name=?", (20, "Jill"))
>>> conn.commit()
>>> conn.close()
```

Table 8.1 gives an overview of the most common operations on a sqlite3 database. These operations are usually all you need to manipulate a sqlite3 database. Of course, several options let you control their precise behavior; see the Python documentation for more information.

Table 8.1 Common database operations

Operation	Sqlite3 command

Create a connection to a database	`conn = sqlite3.connect(filename)`
Create a cursor for a connection	`Cursor = conn.cursor()`
Execute a query with the cursor	`cursor.execute(query)`

Return the results of a query	`cursor.fetchall(),cursor.fetchmany(num_rows), cursor.fetchone()`
	`for row in cursor:`
	`    ....`

Commit a transaction to a database	`conn.commit()`
Close a connection	`conn.close()`