CHAPTER 13

Database Support

Using simple, plain-text files can get you only so far. Yes, they *can* get you *very* far, but at some point, you may need some extra functionality. You may want some automated serialization, and you can turn to `shelve` (see Chapter 10) and `pickle` (a close relative of `shelve`). But you may want features that go beyond even this. For example, you might want to have automated support for concurrent access to your data; that is, to allow several users to read from and write to your disk-based data without causing any corrupted files or the like. Or you may want to be able to perform complex searches using many data fields or properties at the same time, rather than the simple single-key lookup of `shelve`. There are plenty of solutions to choose from, but if you want this to scale to large amounts of data and you want the solution to be easily understandable by other programmers, choosing a relatively standard form of *database* is probably a good idea.

This chapter discusses the Python Database API, a standardized way of connecting to SQL databases, and demonstrates how to execute some basic SQL using this API. The last section also discusses some alternative database technology.

I won't be giving you a tutorial on relational databases or the SQL language. The documentation for most databases (such as PostgreSQL or MySQL, or, the one used in this chapter, SQLite) should cover what you need to know. If you haven't used relational databases before, you might want to check out `http://www.sqlcourse.com` (or just do a Web search on the subject) or *Beginning SQL Queries* by Clare Churcher (Apress, 2008).

The simple database used throughout this chapter (SQLite) is, of course, not the only choice—by far. There are several popular commercial choices (such as Oracle or Microsoft SQL Server), as well as some solid and widespread open source databases (such as MySQL, PostgreSQL, and Firebird). Chapter 26 uses PostgreSQL and has some instructions for MySQL and SQLite. For a list of some other databases supported by Python packages, check out `http://www.python.org/topics/database/` or visit the Database category of Vaults of Parnassus (`http://www.vex.net/parnassus`).

Relational (SQL) databases aren't the only kind around, of course. There are object databases such as the Zope Object Database (ZODB, `http://wiki.zope.org/ZODB`), compact table-based ones such as Metakit (`http://www.equi4.com/metakit/python.html`), or even simpler *key-value* databases, such as BSD DB (`http://docs.python.org/lib/module-bsddb.html`).

While this chapter focuses on rather low-level database interaction, you can find several high-level libraries to help you abstract away some of the grind (see, for example, `http://www.sqlalchemy.org` or `http://www.sqlobject.org`, or search the Web for other so-called object-relational mappers for Python).

The Python Database API

As I've mentioned, you can choose from various SQL databases, and many of them have corresponding client modules in Python (some databases even have several). Most of the basic functionality of all the databases is the same, so a program written to use one of them might easily—in theory—be used with another. The problem with switching between different modules that provide the same functionality (more or less) is usually that their interfaces (APIs) are different. In order to solve this problem for database modules in Python, a standard Database API (DB API) has been agreed upon. The current version of the API (2.0) is defined in PEP 249, Python Database API Specification v2.0 (available from `http://python.org/peps/pep-0249.html`).

This section gives you an overview of the basics. I won't cover the optional parts of the API, because they don't apply to all databases. You can find more information in the PEP mentioned, or in the database programming guide in the official Python Wiki (available from `http://wiki.python.org/moin/DatabaseProgramming`). If you're not really interested in all the API details, you can skip this section.

Global Variables

Any compliant database module (compliant, that is, with the DB API, version 2.0) must have three global variables, which describe the peculiarities of the module. The reason for this is that the API is designed to be very flexible and to work with several different underlying mechanisms without too much wrapping. If you want your program to work with several different databases, this can be a nuisance, because you need to cover many different possibilities. A more realistic course of action, in many cases, would be to simply check these variables to see that a given database module is acceptable to your program. If it isn't, you could simply exit with an appropriate error message, for example, or raise some exception. The global variables are summarized in Table 13-1.

Table 13-1. *The Module Properties of the Python DB API*

Variable Name	Use
`apilevel`	The version of the Python DB API in use
`threadsafety`	How thread-safe the module is
`paramstyle`	Which parameter style is used in the SQL queries

The API level (`apilevel`) is simply a string constant, giving the API version in use. According to the DB API version 2.0, it may either have the value `'1.0'` or the value `'2.0'`. If the variable isn't there, the module is not 2.0-compliant, and you should (according to the API) assume that the DB API version 1.0 is in effect. It also probably wouldn't hurt to write your code to allow other values here (who knows when, say, version 3.0 of the DB API will come out?).

The thread-safety level (`threadsafety`) is an integer ranging from 0 to 3, inclusive. 0 means that threads may not share the module at all, and 3 means that the module is completely thread-safe. A value of 1 means that threads may share the module itself, but not connections

(see "Connections and Cursors," later in this chapter), and 2 means that threads may share modules and connections, but not cursors. If you don't use threads (which, most of the time, you probably won't), you don't have to worry about this variable at all.

The parameter style (`paramstyle`) indicates how parameters are spliced into SQL queries when you make the database perform multiple similar queries. The value `'format'` indicates standard string formatting (using basic format codes), so you insert `%s` where you want to splice in parameters, for example. The value `'pyformat'` indicates extended format codes, as used with dictionary splicing, such as `%(foo)s`. In addition to these Pythonic styles, there are three ways of writing the splicing fields: `'qmark'` means that question marks are used, `'numeric'` means fields of the form `:1` or `:2` (where the numbers are the numbers of the parameters), and `'named'` means fields like `:foobar`, where `foobar` is a parameter name. If parameter styles seem confusing, don't worry. For basic programs, you won't need them, and if you need to understand how a specific database interface deals with parameters, the relevant documentation will probably explain it.

Exceptions

The API defines several exceptions, to make fine-grained error handling possible. However, they're defined in a hierarchy, so you can also catch several types of exceptions with a single `except` block. (Of course, if you expect everything to work nicely, and you don't mind having your program shut down in the unlikely event of something going wrong, you can just ignore the exceptions altogether.)

The exception hierarchy is shown in Table 13-2. The exceptions should be available globally in the given database module. For more in-depth descriptions of these exceptions, see the API specification (the PEP mentioned previously).

Table 13-2. *Exceptions Specified in the Python DB API*

Exception	Superclass	Description
`StandardError`		Generic superclass of all exceptions
`Warning`	`StandardError`	Raised if a nonfatal problem occurs
`Error`	`StandardError`	Generic superclass of all error conditions
`InterfaceError`	`Error`	Errors relating to the interface, not the database
`DatabaseError`	`Error`	Superclass for errors relating to the database
`DataError`	`DatabaseError`	Problems related to the data; e.g., values out of range
`OperationalError`	`DatabaseError`	Errors internal to the operation of the database
`IntegrityError`	`DatabaseError`	Relational integrity compromised; e.g., key check fails
`InternalError`	`DatabaseError`	Internal errors in the database; e.g., invalid cursor
`ProgrammingError`	`DatabaseError`	User programming error; e.g., table not found
`NotSupportedError`	`DatabaseError`	An unsupported feature (e.g., rollback) requested

Connections and Cursors

In order to use the underlying database system, you must first *connect* to it. For this you use the aptly named function `connect`. It takes several parameters; exactly which depends on the database. The API defines the parameters in Table 13-3 as a guideline. It recommends that they be usable as keyword arguments, and that they follow the order given in the table. The arguments should all be strings.

Table 13-3. *Common Parameters of the connect Function*

Parameter Name	Description	Optional?
`dsn`	Data source name. Specific meaning database dependent.	No
`user`	User name	Yes
`password`	User password	Yes
`host`	Host name	Yes
`database`	Database name	Yes

You'll see specific examples of using the `connect` function in the section "Getting Started" later in this chapter, as well as in Chapter 26.

The `connect` function returns a connection object. This represents your current session with the database. Connection objects support the methods shown in Table 13-4.

Table 13-4. *Connection Object Methods*

Method Name	Description
`close()`	Closes the connection. Connection object and its cursors are now unusable.
`commit()`	Commits pending transactions, if supported; otherwise does nothing.
`rollback()`	Rolls back pending transactions (may not be available).
`cursor()`	Returns a cursor object for the connection.

The `rollback` method may not be available, because not all databases support transactions. (*Transactions* are just sequences of actions.) If it exists, it will "undo" any transactions that have not been committed.

The `commit` method is always available, but if the database doesn't support transactions, it doesn't actually do anything. If you close a connection and there are still transactions that have not been committed, they will implicitly be rolled back—but only if the database supports rollbacks! So if you don't want to rely on this, you should always commit before you close your connection. If you commit, you probably don't need to worry too much about closing your connection; it's automatically closed when it's garbage-collected. If you want to be on the safe side, though, a call to `close` won't cost you that many keystrokes.

The `cursor` method leads us to another topic: cursor objects. You use cursors to execute SQL queries and to examine the results. Cursors support more methods than connections, and

probably will be quite a bit more prominent in your programs. Table 13-5 gives an overview of the cursor methods, and Table 13-6 gives an overview of the attributes.

Table 13-5. *Cursor Object Methods*

Name	Description
`callproc(name[, params])`	Calls a named database procedure with given name and parameters (optional).
`close()`	Closes the cursor. Cursor is now unusable.
`execute(oper[, params])`	Executes a SQL operation, possibly with parameters.
`executemany(oper, pseq)`	Executes a SQL operation for each parameter set in a sequence.
`fetchone()`	Fetches the next row of a query result set as a sequence, or `None`.
`fetchmany([size])`	Fetches several rows of a query result set. Default size is `arraysize`.
`fetchall()`	Fetches all (remaining) rows as a sequence of sequences.
`nextset()`	Skips to the next available result set (optional).
`setinputsizes(sizes)`	Used to predefine memory areas for parameters.
`setoutputsize(size[, col])`	Sets a buffer size for fetching big data values.

Table 13-6. *Cursor Object Attributes*

Name	Description
`description`	Sequence of result column descriptions. Read-only.
`rowcount`	The number of rows in the result. Read-only.
`arraysize`	How many rows to return in `fetchmany`. Default is 1.

Some of these methods will be explained in more detail in the upcoming text, while some (such as `setinputsizes` and `setoutputsizes`) will not be discussed. Consult the PEP for more details.

Types

In order to interoperate properly with the underlying SQL databases, which may place various requirements on the values inserted into columns of certain types, the DB API defines certain constructors and constants (singletons) used for special types and values. For example, if you want to add a date to a database, it should be constructed with (for example) the `Date` constructor of the corresponding database connectivity module. That allows the connectivity module to perform any necessary transformations behind the scenes. Each module is required to implement the constructors and special values shown in Table 13-7. Some modules may not be entirely compliant. For example, the `sqlite3` module (discussed next) does not export the special values (`STRING` through `ROWID`) in Table 13-7.

Table 13-7. *DB API Constructors and Special Values*

Name	Description
`Date(year, month, day)`	Creates an object holding a date value
`Time(hour, minute, second)`	Creates an object holding a time value
`Timestamp(y, mon, d, h, min, s)`	Creates an object holding a timestamp value
`DateFromTicks(ticks)`	Creates an object holding a date value from ticks since epoch
`TimeFromTicks(ticks)`	Creates an object holding a time value from ticks
`TimestampFromTicks(ticks)`	Creates an object holding a timestamp value from ticks
`Binary(string)`	Creates an object holding a binary string value
`STRING`	Describes string-based column types (such as `CHAR`)
`BINARY`	Describes binary columns (such as `LONG` or `RAW`)
`NUMBER`	Describes numeric columns
`DATETIME`	Describes date/time columns
`ROWID`	Describes row ID columns

SQLite and PySQLite

As mentioned previously, many SQL database engines are available, with corresponding Python modules. Most of these database engines are meant to be run as server programs, and require administrator privileges even to install them. In order to lower the threshold for playing around with the Python DB API, I've chosen to use a tiny database engine called SQLite, which doesn't need to be run as a stand-alone server, and which can work directly on local files, instead of with some centralized database storage mechanism.

In recent Python versions (from 2.5) SQLite has the advantage that a wrapper for it (PySQLite) is included in the standard library. Unless you're compiling Python from source yourself, chances are that the database itself is also included. You might want to just try the program snippets in the section "Getting Started." If they work, you don't need to bother with installing PySQLite and SQLite separately.

■ **Note** If you're not using the standard library version of PySQLite, you may need to modify the `import` statement. Refer to the relevant documentation for more information.

GETTING PYSQLITE

If you are using an older version of Python, you will need to install PySQLite before you can use the SQLite database. You can download it from the official web page, `http://pysqlite.org`.

For Linux systems with package manager systems, chances are you can get PySQLite and SQLite directly from the package manager.

The Windows binaries for PySQLite actually *include* the database engine itself (that is, SQLite), so all you need to do is to download the PySQLite installer corresponding to your Python version, run it, and you're all set.

If you're not using Windows, and your operating system does not have a package manager where you can find PySQLite and SQLite, you will need to get the source packages for PySQLite and SQLite and compile them yourself.

If you're using a recent version of Python, you will most certainly have PySQLite. If anything is missing, it will be the database itself, SQLite (but again, that will probably be available as well). You can get the sources from the SQLite web page, `http://sqlite.org`. (Make sure you get one of the source packages where automatic code generation has already been performed.) Compiling SQLite is basically a matter of following the instructions in the included README file. When subsequently compiling PySQLite, you need to make sure that the compilation process can access the SQLite libraries and include files. If you've installed SQLite in some standard location, it may well be that the setup script in the PySQLite distribution can find it on its own. In that case, you simply need to execute the following commands:

```
python setup.py build
python setup.py install
```

You could simply use the latter command, which will perform the build automatically. If this gives you heaps of error messages, chances are the installation script didn't find the required files. Make sure you know where the include files and libraries are installed, and supply them explicitly to the install script. Let's say I compiled SQLite in place in a directory called `/home/mlh/sqlite/current`; then the header files could be found in `/home/mlh/sqlite/current/src` and the library in `/home/mlh/sqlite/current/build/lib`. In order to let the installation process use these paths, edit the setup script, `setup.py`. In this file you'll want to set the variables `include_dirs` and `library_dirs`:

```
include_dirs = ['/home/mlh/sqlite/current/src']
library_dirs = ['/home/mlh/sqlite/current/build/lib']
```

After rebinding these variables, the install procedure described earlier should work without errors.

Getting Started

You can import SQLite as a module, under the name `sqlite3` (if you are using the one in the Python standard library). You can then create a connection directly to a database file—which will be created if it does not exist—by supplying a file name (which can be a relative or absolute path to the file):

```
>>> import sqlite3
>>> conn = sqlite3.connect('somedatabase.db')
```

You can then get a cursor from this connection:

```
>>> curs = conn.cursor()
```

This cursor can then be used to execute SQL queries. Once you're finished, if you've made any changes, make sure you commit them, so they're actually saved to the file:

```
>>> conn.commit()
```

You can (and should) commit each time you've modified the database, not just when you're ready to close it. When you *are* ready to close it, just use the `close` method:

```
>>> conn.close()
```

A Sample Database Application

As an example, I'll demonstrate how to construct a little nutrient database, based on data from the United States Department of Agriculture (USDA) Nutrient Data Laboratory (`http://www.ars.usda.gov/nutrientdata`). On their web page, follow the link to the USDA National Nutrient Database for Standard Reference. There, you should find a lot of different data files in plain-text (ASCII) format, just the way we like it. Follow the Download link, and download the zip file referenced by the ASCII link under the heading "Abbreviated." You should now get a zip file containing a text file named `ABBREV.txt`, along with a PDF file describing its contents.[1] If you have trouble finding this particular file, any old data will do. Just modify the source code to suit.

The data in the `ABBREV.txt` file has one data record per line, with the fields separated by caret (^) characters. The numeric fields contain numbers directly, while the textual fields have their string values "quoted" with a tilde (~) on each side. Here is a sample line, with parts deleted for brevity:

```
~07276~^~HORMEL SPAM ... PORK W/ HAM  MINCED  CND~^ ... ^~1 serving~^^~~^0
```

Parsing such a line into individual fields is a simple as using `line.split('^')`. If a field starts with a tilde, you know it's a string and can use `field.strip('~')` to get its contents. For the other (numeric) fields, `float(field)` should do the trick, except, of course, when the field is empty. The program developed in the following sections will transfer the data in this ASCII file into your SQL database, and let you perform some (semi-)interesting queries on them.

1. At the time of writing, you can get this file from `http://www.nal.usda.gov/fnic/foodcomp/Data/SR20/dnload/sr20abbr.zip`.

■ **Note** This sample program is intentionally simple. For a slightly more advanced example of database use in Python, see Chapter 26.

Creating and Populating Tables

To actually create the tables of the database and populate them, writing a completely separate one-shot program is probably the easiest solution. You can run this program once, and then forget about both it and the original data source (the `ABBREV.txt` file), although keeping them around is probably a good idea.

The program shown in Listing 13-1 creates a table called `food` with some appropriate fields, reads the file `ABBREV.txt`, parses it (by splitting the lines and converting the individual fields using a utility function, `convert`), and inserts values read from the text field into the database using a SQL `INSERT` statement in a call to `curs.execute`.

■ **Note** It would have been possible to use `curs.executemany`, supplying a list of all the rows extracted from the data file. This would have given a minor speedup in this case, but might have given a more substantial speedup if a networked client/server SQL system were used.

Listing 13-1. *Importing Data into the Database (importdata.py)*

```
import sqlite3

def convert(value):
    if value.startswith('~'):
        return value.strip('~')
    if not value:
        value = '0'
    return float(value)

conn = sqlite3.connect('food.db')
curs = conn.cursor()

curs.execute('''
CREATE TABLE food (
  id         TEXT       PRIMARY KEY,
  desc       TEXT,
  water      FLOAT,
  kcal       FLOAT,
  protein    FLOAT,
  fat        FLOAT,
  ash        FLOAT,
  carbs      FLOAT,
```

```
  fiber      FLOAT,
  sugar      FLOAT
)
''')

query = 'INSERT INTO food VALUES (?,?,?,?,?,?,?,?,?,?)'

for line in open('ABBREV.txt'):
    fields = line.split('^')
    vals = [convert(f) for f in fields[:field_count]]
    curs.execute(query, vals)

conn.commit()
conn.close()
```

■ **Note** In Listing 13-1, I use the "qmark" version of `paramstyle`; that is, a question mark as a field marker. If you're using an older version of PySQLite, you may need to use `%` characters instead.

When you run this program (with `ABBREV.txt` in the same directory), it will create a new file called `food.db`, containing all the data of the database.

I encourage you to play around with this example, using other inputs, adding `print` statements, and the like.

Searching and Dealing with Results

Using the database is really simple. Again, you create a connection and get a cursor from that connection. Execute the SQL query with the `execute` method and extract the results with, for example, the `fetchall` method. Listing 13-2 shows a tiny program that takes a SQL `SELECT` condition as a command-line argument and prints out the returned rows in a record format. You could try it out with a command line like the following:

```
$ python food_query.py "kcal <= 100 AND fiber >= 10 ORDER BY sugar"
```

You may notice a problem when you run this. The first row, raw orange peel, seems to have no sugar at all. That's because the field is missing in the data file. You could improve the import script to detect this condition, and insert `None` instead of a real value, to indicate missing data. Then you could use a condition such as the following:

```
"kcal <= 100 AND fiber >= 10 AND sugar ORDER BY sugar"
```

requiring the sugar field to have real data in any returned rows. As it happens, this strategy will work with the current database, as well, where this condition will discard rows where the sugar level is zero.

■ **Caution** You might want to try a condition that searches for a specific food item, using an ID, such as 08323 for Cocoa Pebbles. The problem is that SQLite handles its values in a rather nonstandard fashion. Internally, all values are, in fact, strings, and some conversion and checking goes on between the database and the Python API. Usually, this works just fine, but this is an example of where you might run into trouble. If you supply the value `08323`, it will be interpreted as the number `8323`, and subsequently converted into the string `"8323"`—an ID that doesn't exist. One might have expected an error message here, rather than this surprising and rather unhelpful behavior, but if you are careful, and use the string `"08323"` in the first place, you'll be fine.

Listing 13-2. *Food Database Query Program (food_query.py)*

```
import sqlite3, sys

conn = sqlite3.connect('food.db')
curs = conn.cursor()

query = 'SELECT * FROM food WHERE %s' % sys.argv[1]
print query
curs.execute(query)
names = [f[0] for f in curs.description]
for row in curs.fetchall():
    for pair in zip(names, row):
        print '%s: %s' % pair
    print
```

A Quick Summary

This chapter has given a rather brief introduction to making Python programs interact with relational databases. It's brief because, if you master Python and SQL, the coupling between the two, in the form of the Python DB API, is quite easy to master. Here are some of the concepts covered in this chapter:

The Python DB API: This API provides a simple, standardized interface to which database wrapper modules should conform, to make it easier to write programs that will work with several different databases.

Connections: A connection object represents the communication link with the SQL database. From it, you can get individual cursors, using the `cursor` method. You also use the connection object to commit or roll back transactions. After you're finished with the database, the connection can be closed.

Cursors: A cursor is used to execute queries and to examine the results. Resulting rows can be retrieved one by one, or many (or all) at once.

Types and special values: The DB API specifies the names of a set of constructors and special values. The constructors deal with date and time objects, as well as binary data objects. The special values represent the types of the relational database, such as `STRING`, `NUMBER`, and `DATETIME`.

SQLite: This is a small, embedded SQL database, whose Python wrapper is called PySQLite. It's fast and simple to use, and does not require a separate server to be set up.

New Functions in This Chapter

Function	Description
`connect(...)`	Connect to a database and return a connection object[2]

What Now?

Persistence and database handling are important parts of many, if not most, big program systems. Another component shared by a great number of such systems is a network, which is dealt with in the next chapter.

2. The parameters to the `connect` function are database dependent.