In this chapter, we discuss how to communicate with databases by using Python. Files or simplistic persistent storage can meet the needs of smaller applications, but larger server or high-data-volume applications might require a full-fledged database system, instead. Thus, we cover both relational and non-relational databases as well as Object-Relational Mappers (ORMs).

6.1 Introduction

This opening section will discuss the need for databases, present the Structured Query Language (SQL), and introduce readers to Python's database application programming interface (API).

6.1.1 Persistent Storage

In any application, there is a need for persistent storage. Generally, there are three basic storage mechanisms: files, a database system, or some sort of hybrid, such as an API that sits on top of one of those existing systems, an ORM, file manager, spreadsheet, configuration file, etc.

In the Files chapter of *Core Python Language Fundamentals* or *Core Python Programming*, we discussed persistent storage using both plain file access as well as a Python and database manager (DBM), which is an old Unix persistent storage mechanism, overlay on top of files, that is, `*dbm`, `dbhash`/`bsddb` files, `shelve` (combination of `pickle` and DBM), and using their dictionary-like object interface.

This chapter will focus on using databases for the times when files or creating your own data storage system does not suffice for larger projects. In such cases, you will have many decisions to make. Thus, the goal of this chapter is to introduce you to the basics and show you as many of your options as possible (and how to work with them from within Python) so that you can make the right decision. We start off with SQL and relational databases first, because they are still the prevailing form of persistent storage.

6.1.2 Basic Database Operations and SQL

Before we dig into databases and how to use them with Python, we want to present a quick introduction (or review if you have some experience) to some elementary database concepts and SQL.

Underlying Storage

Databases usually have a fundamental persistent storage that uses the file system, that is, normal operating system files, special operating system files, and even raw disk partitions.

User Interface

Most database systems provide a command-line tool with which to issue SQL commands or queries. There are also some GUI tools that use the command-line clients or the database client library, affording users a much more comfortable interface.

Databases

A relational database management system (RDBMS) can usually manage multiple databases, such as sales, marketing, customer support, etc., all on the same server (if the RDBMS is server-based; simpler systems are usually not). In the examples we will look at in this chapter, MySQL demonstrates a server-based RDBMS because there is a server process running continuously, waiting for commands; neither SQLite nor Gadfly have running servers.

Components

The *table* is the storage abstraction for databases. Each *row* of data will have fields that correspond to database *columns*. The set of table definitions of columns and data types per table all put together define the database *schema*.

Databases are *created* and *dropped*. The same is true for tables. Adding new rows to a database is called *inserting*; changing existing rows in a table is called *updating*; and removing existing rows in a table is called *deleting*. These actions are usually referred to as database *commands* or *operations*. Requesting rows from a database with optional criteria is called *querying*.

When you query a database, you can *fetch* all of the results (rows) at once, or just iterate slowly over each resulting row. Some databases use the concept of a *cursor* for issuing SQL commands, queries, and grabbing results, either all at once or one row at a time.

SQL

Database commands and queries are given to a database via SQL. Not all databases use SQL, but the majority of relational databases do. Following are some examples of SQL commands. Note that most databases are configured to be case-insensitive, especially database commands. The accepted style is to use CAPS for database keywords. Most command-line programs require a trailing semicolon (;) to terminate a SQL statement.

Creating a Database

```
CREATE DATABASE test;
GRANT ALL ON test.* to user(s);
```

The first line creates a database named "test," and assuming that you are a database administrator, the second line can be used to grant permissions to specific users (or all of them) so that they can perform the database operations that follow.

Using a Database

```
USE test;
```

If you logged into a database system without choosing which database you want to use, this simple statement allows you to specify one with which to perform database operations.

Dropping a Database

```
DROP DATABASE test;
```

This simple statement removes all the tables and data from the database and deletes it from the system.

Creating a Table

```
CREATE TABLE users (login VARCHAR(8), userid INT, projid INT);
```

This statement creates a new table with a string column `login` and a pair of integer fields, `userid` and `projid`.

Dropping a Table

```
DROP TABLE users;
```

This simple statement drops a database table, along with all its data.

Inserting a Row

```
INSERT INTO users VALUES('leanna', 2111, 1);
```

You can insert a new row in a database by using the `INSERT` statement. You specify the table and the values that go into each field. For our example, the string `'leanna'` goes into the `login` field, and `2111` and `1` to `userid` and `projid`, respectively.

Updating a Row

```
UPDATE users SET projid=4 WHERE projid=2;
UPDATE users SET projid=1 WHERE userid=311;
```

To change existing table rows, you use the `UPDATE` statement. Use `SET` for the columns that are changing and provide any criteria for determining which rows should change. In the first example, all users with a "project ID" (or `projid`) of 2 will be moved to project #4. In the second example, we take one user (with a `UID` of `311`) and move him to project #1.

Deleting a Row

```
DELETE FROM users WHERE projid=%d;
DELETE FROM users;
```

To delete a table row, use the `DELETE FROM` command, specify the table from which you want to delete rows, and any optional criteria. Without it, as in the second example, all rows will be deleted.

Now that you are up to speed on basic database concepts, it should make following the rest of the chapter and its examples much easier. If you need additional help, there are plenty of database tutorial books available that can do the trick.

6.1.3 Databases and Python

We are going to cover the Python database API and look at how to access relational databases from Python—either directly through a database interface, or via an ORM—and how you can accomplish the same task but without necessarily having to give explicit commands in SQL.

Topics such as database principles, concurrency, schema, atomicity, integrity, recovery, proper complex left JOINs, triggers, query optimization, transactions, stored procedures, etc., are all beyond the scope of this text, and we will not be discussing them in this chapter other than direct use from a Python application. Rather, we will present how to store and retrieve data to and from RDBMSs while playing within a Python framework. You can then decide which is best for your current project or application and be able to study sample code that can get you started instantly. The goal is to get you on top of things as quickly as possible if you need to integrate your Python application with some sort of database system.

We are also breaking out of our mode of covering only the "batteries included" features of the Python Standard Library. While our original goal was to play only in that arena, it has become clear that being able to work with databases is really a core component of everyday application development in the Python world.

As a software engineer, you can probably only make it so far in your career without having to learn something about databases: how to use one (command-line and/or GUI interfaces), how to extract data by using the SQL, perhaps how to add or update information in a database, etc. If Python is your programming tool, then a lot of the hard work has already been done for you as you add database access to your Python universe. We first describe what the Python database API, or *DB-API* is, then give examples of database interfaces that conform to this standard.

We will show some examples using popular open-source RDBMSs. However, we will not include discussions of open-source versus commercial products. Adapting to those other RDBMS systems should be fairly straightforward. A special mention will be given to Aaron Watters's Gadfly database, a simple RDBMS written completely in Python.

The way to access a database from Python is via an *adapter*. An adapter is a Python module with which you can interface to a relational database's client library, usually in C. It is recommended that all Python adapters conform to the API of the Python database special interest group (DB-SIG). This is the first major topic of this chapter.

Figure 6-1 illustrates the layers involved in writing a Python database application, with and without an ORM. The figure demonstrates that the DB-API is your interface to the C libraries of the database client.

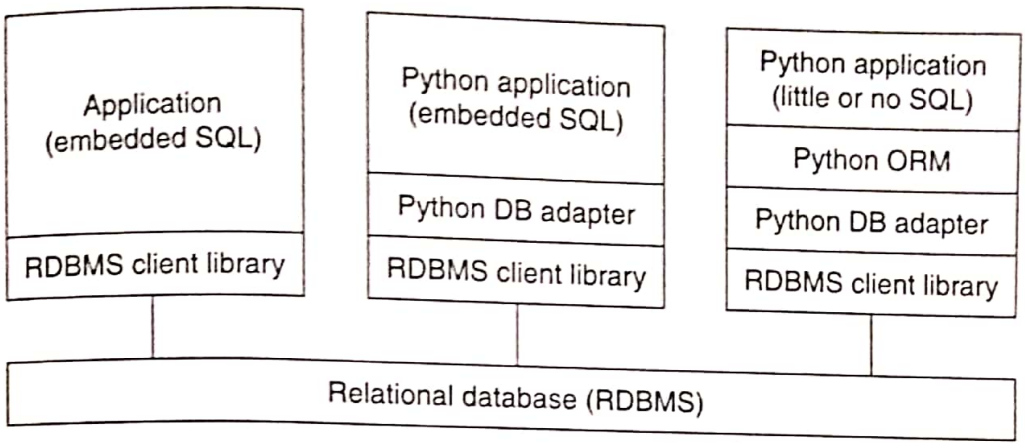

Figure 6-1 Multitiered communication between application and database. The first box is generally a C/C++ program, whereas DB-API-compliant adapters let you program applications in Python. ORMs can simplify an application by handling all of the database-specific details.

6.2 The Python DB-API

Where can one find the interfaces necessary to talk to a database? Simple. Just go to the database topics section at the main Python Web site. There you will find links to the full and current DB-API (version 2.0), existing database modules, documentation, the special interest group, etc. Since its inception, the DB-API has been moved into PEP 249. (This PEP supersedes the old DB-API 1.0 specification, which is PEP 248.) What is the DB-API?

The API is a specification that states a set of required objects and database access mechanisms to provide consistent access across the various database adapters and underlying database systems. Like most community-based efforts, the API was driven by strong need.

In the "old days," we had a scenario of many databases and many people implementing their own database adapters. It was a wheel that was being reinvented over and over again. These databases and adapters were implemented at different times by different people without any consistency of functionality. Unfortunately, this meant that application code using such interfaces also had to be customized to which database module they chose to use, and any changes to that interface also meant updates were needed in the application code.

SIG for Python database connectivity was formed, and eventually, an API was born: the DB-API version 1.0. The API provides for a consistent interface to a variety of relational databases, and porting code between different databases is much simpler, usually only requiring tweaking several lines of code. You will see an example of this later on in this chapter.

6.2.1 Module Attributes

The DB-API specification mandates that the features and attributes listed below must be supplied. A DB-API-compliant module must define the global attributes as shown in Table 6-1.

Table 6-1 DB-API Module Attributes

Attribute	Description
`apilevel`	The version of the DB-API with which an adapter is compliant
`threadsafety`	Level of thread safety of this module
`paramstyle`	SQL statement parameter style of this module
`connect()`	`Connect()` function
(Various exceptions)	*(SeeTable 6-4)*

Data Attributes

apilevel

This string (not float) indicates the highest version of the DB-API with which the module is compliant, for example, 1.0, 2.0, etc. If absent, 1.0 should be assumed as the default value.

threadsafety

This an integer that can take the following possible values:

- 0: Not threadsafe, so threads should not share the module at all
- 1: Minimally threadsafe: threads can share the module but not connections
- 2: Moderately threadsafe: threads can share the module and connections but not cursors
- 3: Fully threadsafe: threads can share the module, connections, and cursors

If a resource is shared, a synchronization primitive such as a spin lock or semaphore is required for atomic-locking purposes. Disk files and global variables are not reliable for this purpose and can interfere with standard mutex operation. See the `threading` module or go back to Chapter 4, "Multithreaded Programming," for more information on how to use a lock.

paramstyle

The API supports a variety of ways to indicate how parameters should be integrated into an SQL statement that is eventually sent to the server for execution. This argument is just a string that specifies the form of string substitution you will use when building rows for a query or command (see Table 6-2).

Table 6-2 `paramstyle` Database Parameter Styles

Parameter Style	Description	Example
`numeric`	Numeric positional style	`WHERE name=:1`
`named`	Named style	`WHERE name=:name`
`pyformat`	Python dictionary `printf()` format conversion	`WHERE name=%(name)s`
`qmark`	Question mark style	`WHERE name=?`
`format`	ANSI C `printf()` format conversion	`WHERE name=%s`

Function Attribute(s)

`connect()` Function access to the database is made available through `Connection` objects. A compliant module must implement a `connect()` function, which creates and returns a `Connection` object. Table 6-3 shows the arguments to `connect()`.

Table 6-3 `connect()` Function Attributes

Parameter	Description
`user`	Username
`password`	Password
`host`	Hostname
`database`	Database name
`dsn`	Data source name

You can pass in database connection information as a string with multiple parameters (DSN) or individual parameters passed as positional arguments (if you know the exact order), or more likely, keyword arguments. Here is an example of using `connect()` from PEP 249:

```
connect(dsn='myhost:MYDB',user='guido',password='234$')
```

The use of DSN versus individual parameters is based primarily on the system to which you are connecting. For example, if you are using an API like Open Database Connectivity (ODBC) or Java DataBase Connectivity (JDBC), you would likely be using a DSN, whereas if you are working directly with a database, then you are more likely to issue separate login parameters. Another reason for this is that most database adapters have not implemented support for DSN. The following are some examples of non-DSN `connect()` calls. Note that not all adapters have implemented the specification exactly, e.g., `MySQLdb` uses `db` instead of `database`.

- `MySQLdb.connect(host='dbserv', db='inv', user='smith')`
- `PgSQL.connect(database='sales')`
- `psycopg.connect(database='template1', user='pgsql')`
- `gadfly.dbapi20.connect('csrDB', '/usr/local/database')`
- `sqlite3.connect('marketing/test')`

Exceptions

Exceptions that should also be included in the compliant module as globals are shown in Table 6-4.

Table 6-4 DB-API Exception Classes

Exception	Description
`Warning`	Root warning exception class
`Error`	Root error exception class
`InterfaceError`	Database interface (not database) error
`DatabaseError`	Database error
`DataError`	Problems with the processed data
`OperationalError`	Error during database operation execution
`IntegrityError`	Database relational integrity error
`InternalError`	Error that occurs within the database
`ProgrammingError`	SQL command failed
`NotSupportedError`	Unsupported operation occurred

6.2.2 Connection Objects

Connections are how your application communicates with the database. They represent the fundamental mechanism by which commands are sent to the server and results returned. Once a connection has been established (or a pool of connections), you create cursors to send requests to and receive replies from the database.

Connection Object Methods

`Connection` objects are not required to have any data attributes but should define the methods shown in Table 6-5.

Table 6-5 `Connection` Object Methods

Method Name	Description
`close()`	Close database connection
`commit()`	Commit current transaction
`rollback()`	Cancel current transaction
`cursor()`	Create (and return) a cursor or cursor-like object using this connection
`errorhandler(cxn, cur, errcls, errval)`	Serves as a handler for given connection cursor

When `close()` is used, the same connection cannot be used again without running into an exception.

The `commit()` method is irrelevant if the database does not support transactions or if it has an auto-commit feature that has been enabled. You can implement separate methods to turn auto-commit off or on if you wish. Since this method is required as part of the API, databases that do not support transactions should just implement "pass" for this method.

Like `commit()`, `rollback()` only makes sense if transactions are supported in the database. After execution, `rollback()` should leave the database in the same state as it was when the transaction began. According to PEP 249, *"Closing a connection without committing the changes first will cause an implicit rollback to be performed."*

If the RDBMS does not support cursors, `cursor()` should still return an object that faithfully emulates or imitates a real cursor object. These are just the minimum requirements. Each individual adapter developer can always add special attributes specifically for their interface or database.

It is also recommended but not required for adapter writers to make all database module exceptions (see earlier) available via a connection. If not, then it is assumed that `Connection` objects will throw the corresponding module-level exception. Once you have completed using your connection and cursors are closed, you should `commit()` any operations and `close()` your connection.

6.2.3 Cursor Objects

Once you have a connection, you can begin communicating with the database. As we mentioned earlier in the introductory section, a cursor lets a user issue database commands and retrieve rows resulting from queries. A Python DB-API cursor object functions as a cursor for you, even if cursors are not supported in the database. In this case, if you are creating a database adapter, you must implement `cursor` objects so that they act like cursors. This keeps your Python code consistent when you switch between database systems that support or do not support cursors.

Once you have created a cursor, you can execute a query or command (or multiple queries and commands) and retrieve one or more rows from the results set. Table 6-6 presents `Cursor` object data attributes and methods.

Table 6-6 `Cursor` Object Attributes

Object Attribute	Description
`arraysize`	Number of rows to fetch at a time with `fetchmany()`; default is 1
`connection`	Connection that created this cursor (optional)
`description`	Returns cursor activity (7-item tuples): `(name, type_code, display_size, internal_ size, precision, scale, null_ok)`; only *name* and `type_code` are required
`lastrowid`	Row ID of last modified row (optional; if row IDs not supported, default to None)
`rowcount`	Number of rows that the last `execute*()` produced or affected
`callproc(`*`func`*`[, `*`args`*`])`	Call a stored procedure
`close()`	Close cursor
`execute(`*`op`*`[, `*`args`*`])`	Execute a database query or command
`executemany(`*`op, args`*`)`	Like `execute()` and `map()` combined; prepare and execute a database query or command over given arguments

(Continued)

Table 6-6 Cursor Object Attributes (Continued)

Object Attribute	Description
`fetchone()`	Fetch next row of query result
`fetchmany([size=cursor.arraysize])`	Fetch next size rows of query result
`fetchall()`	Fetch all (remaining) rows of a query result
`__iter__()`	Create iterator object from this cursor (optional; also see `next()`)
`messages`	List of messages (set of tuples) received from the database for cursor execution (optional)
`next()`	Used by iterator to fetch next row of query result (optional; like `fetchone()`, also see `__iter__()`)
`nextset()`	Move to next results set (if supported)
`rownumber`	Index of cursor (by row, 0-based) in current result set (optional)
`setinputsizes(sizes)`	Set maximum input size allowed (required but implementation optional)
`setoutputsize(size[,col])`	Set maximum buffer size for large column fetches (required but implementation optional)

The most critical attributes of cursor objects are the `execute*()` and the `fetch*()` methods; all service requests to the database are performed by these. The `arraysize` data attribute is useful in setting a default size for `fetchmany()`. Of course, closing the cursor is a good thing, and if your database supports stored procedures, then you will be using `callproc()`.

6.2.4 Type Objects and Constructors

Oftentimes, the interface between two different systems are the most fragile. This is seen when converting Python objects to C types and vice versa. Similarly, there is also a fine line between Python objects and native database objects. As a programmer writing to Python's DB-API, the parameters you send to a database are given as strings, but the database

might need to convert it to a variety of different, supported data types that are correct for any particular query.

For example, should the Python string be converted to a VARCHAR, a TEXT, a BLOB, or a raw BINARY object, or perhaps a DATE or TIME object if that is what the string is supposed to be? Care must be taken to provide database input in the expected format; therefore, another requirement of the DB-API is to create constructors that build special objects that can easily be converted to the appropriate database objects. Table 6-7 describes classes that can be used for this purpose. SQL NULL values are mapped to and from Python's NULL object, `None`.

Table 6-7 Type Objects and Constructors

Type Object	Description
`Date(yr,mo,dy)`	Object for a date value
`Time(hr,min,sec)`	Object for a time value
`Timestamp (yr,mo,dy,hr,min,sec)`	Object for a timestamp value
`DateFromTicks(ticks)`	Date object, given in number of seconds since the epoch
`TimeFromTicks(ticks)`	Time object, given in number of seconds since the epoch
`TimestampFromTicks(ticks)`	Timestamp object, given in number of seconds since the epoch
`Binary(string)`	Object for a binary (long) string value
`STRING`	Object describing string-based columns, for example, VARCHAR
`BINARY`	Object describing (long) binary columns, for example, RAW, BLOB
`NUMBER`	Object describing numeric columns
`DATETIME`	Object describing date/time columns
`ROWID`	Object describing "row ID" columns

Changes to API Between Versions

Several important changes were made when the DB-API was revised from version 1.0 (1996) to 2.0 (1999):

- The required `dbi` module was removed from the API.
- Type objects were updated.
- New attributes were added to provide better database bindings.
- `callproc()` semantics and the return value of `execute()` were redefined.
- Conversion to class-based exceptions.

Since version 2.0 was published, some of the additional, optional DB-API extensions that you just read about were added in 2002. There have been no other significant changes to the API since it was published. Continuing discussions of the API occur on the DB-SIG mailing list. Among the topics brought up over the last five years include the possibilities for the next version of the DB-API, tentatively named DB-API 3.0. These include the following:

- Better return value for `nextset()` when there is a new result set.
- Switch from `float` to `Decimal`.
- Improved flexibility and support for parameter styles.
- Prepared statements or statement caching.
- Refine the transaction model.
- State the role of API with respect to portability.
- Add unit testing.

If you have strong feelings about the API or its future, feel free to participate and join in the discussion. Here are some references that you might find handy.

- http://python.org/topics/database
- http://linuxjournal.com/article/2605 (outdated but historical)
- http://wiki.python.org/moin/DbApi3

6.2.5 Relational Databases

So, you are now ready to go, but you probably have one burning question: "which interfaces to database systems are available to me in Python?" That inquiry is similar to, "which platforms is Python available for?" The answer is, "Pretty much all of them." Following is a broad (but not exhaustive) list of interfaces:

Commercial RDBMSs

- IBM Informix
- Sybase
- Oracle
- Microsoft SQL Server
- IBM DB2
- SAP
- Embarcadero Interbase
- Ingres

Open-Source RDBMSs

- MySQL
- PostgreSQL
- SQLite
- Gadfly

Database APIs

- JDBC
- ODBC

Non-Relational Databases

- MongoDB
- Redis
- Cassandra
- SimpleDB

- Tokyo Cabinet
- CouchDB
- Bigtable (via Google App Engine Datastore API)

To find an updated (but not necessarily the most recent) list of what databases are supported, go to the following Web site:

http://wiki.python.org/moin/DatabaseInterfaces

6.2.6 Databases and Python: Adapters

For each of the databases supported, there exists one or more adapters that let you connect to the target database system from Python. Some databases, such as Sybase, SAP, Oracle, and SQLServer, have more than one adapter available. The best thing to do is to determine which ones best fit your needs. Your questions for each candidate might include: how good is its performance, how useful is its documentation and/or Web site, whether it has an active community or not, what is the overall quality and stability of the driver, etc. You have to keep in mind that most adapters provide just the basic necessities to get you connected to the database. It is the extras that you might be looking for. Keep in mind that you are responsible for higher-level code like threading and thread management as well as management of database connection pools, etc.

If you are squeamish and want less hands-on interaction—for example, if you prefer to do as little SQL or database administration as possible—then you might want to consider ORMs, which are covered later in this chapter.

Let's now look at some examples of how to use an adapter module to communicate with a relational database. The real secret is in setting up the connection. Once you have this and use the DB-API objects, attributes, and object methods, your core code should be pretty much the same, regardless of which adapter and RDBMS you use.

6.2.7 Examples of Using Database Adapters

First, let's look at a some sample code, from creating a database to creating a table and using it. We present examples that use MySQL, PostgreSQL, and SQLite.

MySQL

We will use MySQL as the example here, along with the most well-known MySQL Python adapter: MySQLdb, a.k.a. MySQL-python—we'll discuss the other MySQL adapter, MySQL Connector/Python, when our conversation turns to Python 3. In the various bits of code that follow, we'll also expose you (deliberately) to examples of error situations so that you have an idea of what to expect, and for which you might want to create handlers.

We first log in as an administrator to create a database and grant permissions, then log back in as a normal client, as shown here:

```
>>> import MySQLdb
>>> cxn = MySQLdb.connect(user='root')
>>> cxn.query('DROP DATABASE test')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
_mysql_exceptions.OperationalError: (1008, "Can't drop, database
'test'; database doesn't exist")
>>> cxn.query('CREATE DATABASE test')
>>> cxn.query("GRANT ALL ON test.* to ''@'localhost'")
>>> cxn.commit()
>>> cxn.close()
```

In the preceding code, we did not use a cursor. Some adapters have `Connection` objects, which can execute SQL queries with the `query()` method, but not all. We recommend you either not use it or check your adapter to ensure that it is available.

The `commit()` was optional for us because auto-commit is turned on by default in MySQL. We then connect back to the new database as a regular user, create a table, and then perform the usual queries and commands by using SQL to get our job done via Python. This time we use cursors and their `execute()` method.

The next set of interactions shows us creating a table. An attempt to create it again (without first dropping it) results in an error:

```
>>> cxn = MySQLdb.connect(db='test')
>>> cur = cxn.cursor()
>>> cur.execute('CREATE TABLE users(login VARCHAR(8), userid INT)')
0L
```

Now we will insert a few rows into the database and query them out:

```
>>> cur.execute("INSERT INTO users VALUES('john', 7000)")
1L
>>> cur.execute("INSERT INTO users VALUES('jane', 7001)")
1L
>>> cur.execute("INSERT INTO users VALUES('bob', 7200)")
1L
```

```
>>> cur.execute("SELECT * FROM users WHERE login LIKE 'j%'")
2L
>>> for data in cur.fetchall():
...  print '%s\t%s' % data
...
john    7000
jane    7001
```

The last bit features updating the table, either by updating or deleting rows:

```
>>> cur.execute("UPDATE users SET userid=7100 WHERE userid=7001")
1L
>>> cur.execute("SELECT * FROM users")
3L
>>> for data in cur.fetchall():
...  print '%s\t%s' % data
...
john    7000
jane    7100
bob     7200
>>> cur.execute('DELETE FROM users WHERE login="bob"')
1L
>>> cur.execute('DROP TABLE users')
0L
>>> cur.close()
>>> cxn.commit()
>>> cxn.close()
```

MySQL is one of the most popular open-source databases in the world, and it is no surprise that a Python adapter is available for it.

PostgreSQL

Another popular open-source database is PostgreSQL. Unlike MySQL, there are no less than three Python adapters available for Postgres: psycopg, PyPgSQL, and PyGreSQL. A fourth, PoPy, is now defunct, having contributed its project to combine with that of PyGreSQL in 2003. Each of the three remaining adapters has its own characteristics, strengths, and weaknesses, so it would be a good idea to practice due diligence to determine which is right for you.

Note that while we demonstrate the use of each of these, PyPgSQL has not been actively developed since 2006, whereas PyGreSQL released its most recent version (4.0) in 2009. This inactivity clearly leaves psycopg as the sole leader of the PostgreSQL adapters, and this will be the final version of this book featuring examples of those adapters. psycopg is on its second version, meaning that even though our examples use the version 1 psycopg module, when you download it today, you'll be using psycopg2, instead.

The good news is that the interfaces are similar enough that you can create an application that, for example, measures the performance between all three (if that is a metric that is important to you). The following presents the setup code to get a `Connection` object for each adapter.

psycopg

```
>>> import psycopg
>>> cxn = psycopg.connect(user='pgsql')
```

PyPgSQL

```
>>> from pyPgSQL import PgSQL
>>> cxn = PgSQL.connect(user='pgsql')
```

PyGreSQL

```
>>> import pgdb
>>> cxn = pgdb.connect(user='pgsql')
```

Here is some generic code that will work for all three adapters:

```
>>> cur = cxn.cursor()
>>> cur.execute('SELECT * FROM pg_database')
>>> rows = cur.fetchall()
>>> for i in rows:
...  print i
>>> cur.close()
>>> cxn.commit()
>>> cxn.close()
```

Finally, you can see how the output from each adapter is slightly different from one another.

PyPgSQL

```
sales
template1
template0
```

psycopg

```
('sales', 1, 0, 0, 1, 17140, '140626', '3221366099', '', None, None)
('template1', 1, 0, 1, 1, 17140, '462', '462', '', None, '{pgsql=C*T*/
pgsql}')
('template0', 1, 0, 1, 0, 17140, '462', '462', '', None, '{pgsql=C*T*/
pgsql}')
```

PyGreSQL

```
['sales', 1, 0, False, True, 17140L, '140626', '3221366099', '', None,
None]
```

```
['template1', 1, 0, True, True, 17140L, '462', '462', '', None,
'{pgsql-C*T*/pgsql}']
['template0', 1, 0, True, False, 17140L, '462', '462', '', None,
'{pgsql-C*T*/pgsql}']
```

SQLite

For extremely simple applications, using files for persistent storage usually suffices, but the most complex and data-driven applications demand a full relational database. SQLite targets the intermediate systems, and indeed is a hybrid of the two. It is extremely lightweight and fast, plus it is serverless and requires little or no administration.

SQLite has experienced a rapid growth in popularity, and it is available on many platforms. With the introduction of the pysqlite database adapter in Python 2.5 as the `sqlite3` module, this marks the first time that the Python Standard Library has featured a database adapter in any release.

It was bundled with Python not because it was favored over other databases and adapters, but because it is simple, uses files (or memory) as its back-end store like the DBM modules do, does not require a server, and does not have licensing issues. It is simply an alternative to other similar persistent storage solutions included with Python but which happens to have a SQL interface.

Having a module like this in the standard library allows you to develop rapidly in Python by using SQLite, and then migrate to a more powerful RDBMS such as MySQL, PostgreSQL, Oracle, or SQL Server for production purposes, if this is your intention. If you don't need all that horsepower, `sqlite3` is a great solution.

Although the database adapter is now provided in the standard library, you still have to download the actual database software yourself. However, once you have installed it, all you need to do is start up Python (and import the adapter) to gain immediate access:

```
>>> import sqlite3
>>> cxn = sqlite3.connect('sqlite_test/test')
>>> cur = cxn.cursor()
>>> cur.execute('CREATE TABLE users(login VARCHAR(8),
      userid INTEGER)')
>>> cur.execute('INSERT INTO users VALUES("john", 100)')
>>> cur.execute('INSERT INTO users VALUES("jane", 110)')
>>> cur.execute('SELECT * FROM users')
>>> for eachUser in cur.fetchall():
...     print eachUser
...
(u'john', 100)
(u'jane', 110)
```

```
>>> cur.execute('DROP TABLE users')
<sqlite3.Cursor object at 0x3d4320>
>>> cur.close()
>>> cxn.commit()
>>> cxn.close()
```

Okay, enough of the small examples. Next, we look at an application similar to our earlier example with MySQL, but which does a few more things:

- Creates a database (if necessary)
- Creates a table
- Inserts rows into the table
- Updates rows in the table
- Deletes rows from the table
- Drops the table

For this example, we will use two other open-source databases. SQLite has become quite popular of late. It is very small, lightweight, and extremely fast for all of the most common database functions. Another database involved in this example is Gadfly, a mostly SQL-compliant RDBMS written entirely in Python. (Some of the key data structures have a C module available, but Gadfly can run without it [slower, of course].)

Some notes before we get to the code. Both SQLite and Gadfly require that you specify the location to store database files (MySQL has a default area and does not require this information). The most current incarnation of Gadfly is not yet fully DB-API 2.0 compliant, and as a result, it is missing some functionality, most notably the cursor attribute, `rowcount`, in our example.

6.2.8 A Database Adapter Example Application

In the example that follows, we demonstrate how to use Python to access a database. For the sake of variety and exposing you to as much code as possible, we added support for three different database systems: Gadfly, SQLite, and MySQL. To mix things up even further, we're first going to dump out the entire Python 2.x source, without a line-by-line explanation.

The application works in exactly the same ways as described via the bullet points in the previous subsection. You should be able to understand its functionality without a full explanation—just start with the `main()` function at the bottom. (To keep things simple, for a full system such as