Introduction

Persistent Storage

- In any application, there is a need for persistent storage.
- Generally, there are three basic storage mechanisms: files, a relational database system (RDBMS), or some sort of hybrid, i.e., an API (application programmer interface) that "sits on top of" one of those existing systems, an object relational mapper (ORM), file manager, spreadsheet, configuration file, etc.

Basic Database Operations and SQL

Underlying Storage

Databases usually have a fundamental persistent storage using the file system, i.e., 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, giving users a much nicer interface.

Databases

An RDBMS can usually manage multiple databases, e.g., 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 is an example of a server-based RDBMS because there is a server process running continuously waiting for commands while 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 by SQL. Not all databases use SQL, but the majority of relational databases do. Here are some examples of SQL commands. 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 below.

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), uid INT, prid INT);

This statement creates a new table with a string column login and a pair of integer fields uid and prid.

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', 311, 1);

You can insert a new row in a database with the INSERT statement. Specify the table and the values that go into each field. For our example, the string 'leanna' goes into the login field, and 311 and 1 to uid and prid, respectively.

Updating a Row

UPDATE users SET prid=4 WHERE prid=2;

UPDATE users SET prid=1 WHERE uid=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 prid of 2 will be moved to project #4. In the second example, we take one user (with a
UID of 311) and move them to project #1.

Deleting a Row

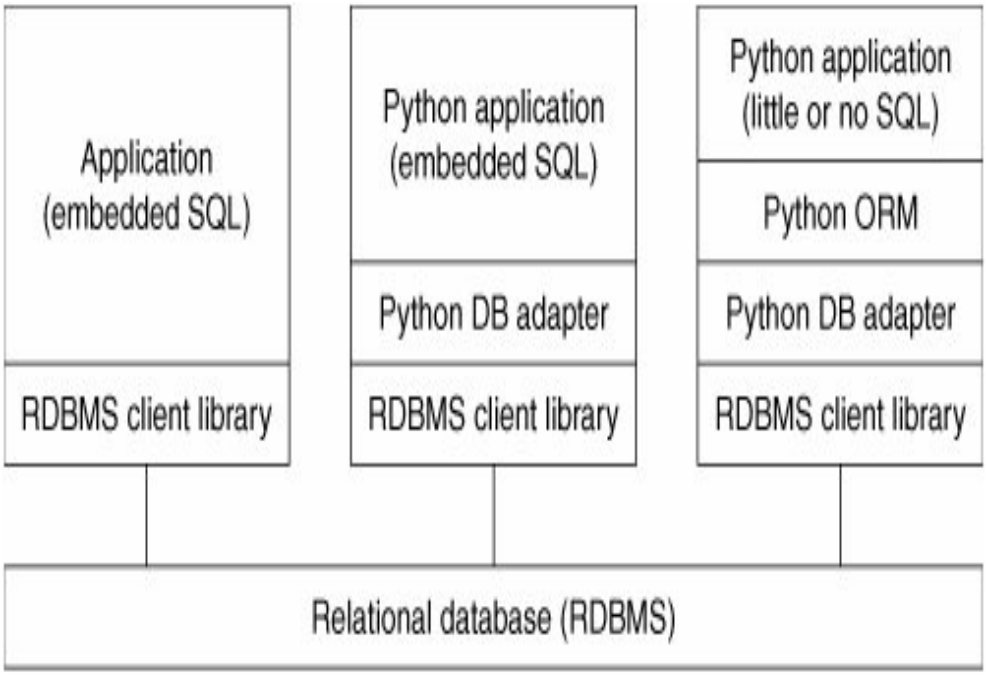
DELETE FROM users WHERE prid=%d;

DELETE FROM users;

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

Figure 21.1 illustrates the layers involved in writing a Python database application, with and without an ORM. As you can see, the DB-API is your interface to the C libraries of the database client.

Figure 21-1. Multitiered communication between application and database. The first box is generally a C/C++ program while DB-API compliant adapters let you program applications in Python. ORMs can simplify an application by handling all of the database-specific details.



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.
- A special interest group (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.

Module Attributes
The DB-API specification mandates that the features and attributes listed below must be supplied.
A DBAPI- compliant module must define the global attributes as shown in Table 21.1.

Table 21.1. DB-API Module Attributes

Attribute	***Description***
apilevel	Version of DB-API module is compliant with
threadsafety	Level of thread safety of this module
paramstyle	SQL statement parameter style of this module
Connect()	Connect() function

Data Attributes
apilevel
This string (not float) indicates the highest version of the DB-API the module is compliant with, i.e.,"1.0", "2.0", etc. If absent, "1.0" should be assumed as the default value.
threadsafety

This an integer with these 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 may interfere with standard mutex operation.

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 21.2).

Table 21.2. 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 has to implement a connect() function, which creates and returns a Connection object.

Table 21.3 shows the arguments to connect().

Table 21.3. 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, keyworded arguments.

Here is an example of using connect() from PEP 249:
connect(dsn='myhost:MYDB',user='guido',password='234$')

Examples:
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 21.4.

Table 21.4. 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

Connection Objects

- Connections are how your application gets to talk to the database.
- They represent the fundamental communication 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.

Methods

Connection objects are not required to have any data attributes but should define the methods shown in Table 21.5.

Table 21.5. 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 autocommit 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 have the concept of 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 animplicit 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 above) 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 closed, you should commit() any operations and close() your connection.

Cursor Objects

- Once you have a connection, you can start talking to the database.
- 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, the database adapter creator must implement CURSOR objects so that they act like cursors.
- This keeps your Python code consistent when you switch between database systems that have or do not have cursor support.
- 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 21.6 shows data attributes and methods that cursor objects have.

Table 21.6. Cursor Object Attributes

Object Attribute Description

arraysize Number of rows to fetch at a time with fetch many(); defaults to 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
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)
setinput-sizes(*sizes*) Set maximum input-size allowed (required but implementation optional)
setoutput size(*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 the 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().
- 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 21.7 describes classes that can be used for this purpose.
- SQL NULL values are mapped to and from Python's NULL object, None.

Table 21.7. 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 number of seconds since the epoch
TimeFromTicks(*ticks*) Time object given number of seconds since the epoch
TimestampFromTicks(*ticks*) Timestamp object given number of seconds since the epoch
Binary(*string*) Object for a binary (long) string value
STRING Object describing string-based columns, e.g., VARCHAR
BINARY Object describing (long) binary columns, i.e., 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):
● Required dbi module removed from API
● Type objects were updated
● New attributes added to provide better database bindings
● callproc() semantics and return value of execute() redefined
● Conversion to class-based exceptions

Since version 2.0 was published, some of the additional optional DB-API extensions that you read about above were added in 2002.
There have been no other significant changes to the API since it was published.

Among the topics brought up over the last 5 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

Relational Databases

Commercial RDBMSs
● Informix
● Sybase
● Oracle
● MS SQL Server
● DB/2
● SAP
● Interbase
● Ingres
Open Source RDBMSs
● MySQL
● PostgreSQL
● SQLite
● Gadfly
Database APIs
● JDBC
● ODBC

To find a current list of what databases are supported, check out:
http://python.org/topics/database/modules.html

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.

The Python programming language has powerful features for database programming.

Python supports various databases like MySQL, Oracle, Sybase, PostgreSQL, etc.

Python also supports Data Definition Language (DDL), Data Manipulation Language (DML) and Data Query Statements.

For database programming, the Python DB API is a widely used module that provides a database application programming interface.

Benefits of Python for database programming

There are many good reasons to use Python for programming database applications:

- Programming in Python is arguably more efficient and faster compared to other languages.
- Python is famous for its portability.
- It is platform independent.
- Python supports SQL cursors.
- In many programming languages, the application developer needs to take care of the open and closed connections of the database, to avoid further exceptions and errors. In Python, these connections are taken care of.
- Python supports relational database systems.
- Python database APIs are compatible with various databases, so it is very easy to migrate and port database application interfaces.

DB-API (SQL-API) for Python

Python DB-API is independent of any database engine, which enables you to write Python scripts to access any database engine.

- The Python DB API implementation for MySQL is MySQLdb.
- For PostgreSQL, it supports psycopg, PyGresQL and pyPgSQL modules.
- DB-API implementations for Oracle are dc_oracle2 and cx_oracle.
- Pydb2 is the DB-API implementation for DB2.
- Python's DB-API consists of connection objects, cursor objects, standard exceptions and some other module contents.

Connection objects

Connection objects create a connection with the database and these are further used for different transactions. These connection objects are also used as representatives of the database session.

A connection is created as follows:

```
>>>conn = MySQLdb.connect('library', user='suhas', password='python')
```

You can use a connection object for calling methods like commit(), rollback() and close() as shown below:

```
>>>cur = conn.cursor()  //creates new cursor object for executing SQL statements

>>>conn.commit()  //Commits the transactions

>>>conn.rollback()  //Roll back the transactions

>>>conn.close()  //closes the connection

>>>conn.callproc(proc,param)  //call stored procedure for execution

>>>conn.getsource(proc)  //fetches stored procedure code
```

Cursor objects

Cursor is one of the powerful features of SQL.

These are objects that are responsible for submitting various SQL statements to a database server.

There are several cursor classes in MySQLdb.cursors:

1. BaseCursor is the base class for Cursor objects.
2. Cursor is the default cursor class. CursorWarningMixIn, CursorStoreResultMixIn, CursorTupleRowsMixIn, and BaseCursor are some components of the cursor class.
3. CursorStoreResultMixIn uses the mysql_store_result() function to retrieve result sets from the executed query. These result sets are stored at the client side.
4. CursorUseResultMixIn uses the mysql_use_result() function to retrieve result sets from the executed query. These result sets are stored at the server side.

The following example illustrates the execution of SQL commands using cursor objects.

You can use execute to execute SQL commands like SELECT.

To commit all SQL operations you need to close the cursor as cursor.close().

```
>>>cursor.execute('SELECT * FROM books')

>>>cursor.execute('''SELECT * FROM books WHERE book_name = 'python' AND book_author = 'Mark Lutz' )

>>>cursor.close()
```

Error and exception handling in DB-API

Exception handling is very easy in the Python DB-API module.

We can place warnings and error handling messages in the programs.

Python DB-API has various options to handle this, like Warning, InterfaceError, DatabaseError, IntegrityError, InternalError, NotSupportedError, OperationalError and ProgrammingError.

Let's take a look at them one by one:

1. IntegrityError: Let's look at integrity error in detail. In the following example, we will try to enter duplicate records in the database. It will show an integrity error, _mysql_exceptions.IntegrityError, as shown below:

```
2. >>> cursor.execute('insert books values (%s,%s,%s,%s)',('Py9098','Programming With Perl',120,100))

3. Traceback (most recent call last):

4. File "<stdin>", line 1, in ?

5. File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 95, in execute

6. return self._execute(query, args)

7. File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 114, in _execute

8. self.errorhandler(self, exc, value)
```

```
raise errorclass, errorvalue _mysql_exceptions.IntegrityError: (1062, "Duplicate entry 'Py9098' for key 1")
```

10. OperationalError: If there are any operation errors like no databases selected, Python DB-API will handle this error as OperationalError, shown below:

```
>>> cursor.execute('Create database Library')

>>> q='select name from books where cost>=%s order by name'

>>>cursor.execute(q,[50])

Traceback (most recent call last):

File "<stdin>", line 1, in ?

File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 95, in execute

return self._execute(query, args)

File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 114, in _execute

self.errorhandler(self, exc, value)

File "/usr/lib/python2.3/site-packages/MySQLdb/connections.py", line 33, in defaulterrorhandler

raise errorclass, errorvalue
```

```
_mysql_exceptions.OperationalError: (1046, 'No Database Selected')
```

22. ProgrammingError: If there are any programming errors like duplicate database creations, Python DB-API will handle this error as ProgrammingError, shown below:

```
>>> cursor.execute('Create database Library')

Traceback (most recent call last):>>> cursor.execute('Create database Library')

Traceback (most recent call last):
```

```
26. File "<stdin>", line 1, in ?

27. File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 95, in execute

28. return self._execute(query, args)

29. File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 114, in _execute

30. self.errorhandler(self, exc, value)

31. File "/usr/lib/python2.3/site-packages/MySQLdb/connections.py", line 33, in defaulterrorhandler

32. raise errorclass, errorvalue
```

```
_mysql_exceptions.ProgrammingError: (1007, "Can't create database 'Library'. Database exists")
```

Python and MySQL

Python and MySQL are a good combination to develop database applications. After starting the MySQL service on Linux, you need to acquire MySQLdb, a Python DB-API for MySQL to perform database operations.

You can check whether the MySQLdb module is installed in your system with the following command:

```
>>>import MySQLdb
```

If this command runs successfully, you can now start writing scripts for your database.

To write database applications in Python, there are five steps to follow:

1. Import the SQL interface with the following command:

   ```
   >>> import MySQLdb
   ```

2. Establish a connection with the database with the following command:

```
>>> conn=MySQLdb.connect(host='localhost',user='root',passwd='')
```

…where host is the name of your host machine, followed by the username and password. In case of the root, there is no need to provide a password.

3. Create a cursor for the connection with the following command:

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

4. Execute any SQL query using this cursor as shown below—here the outputs in terms of 1L or 2L show a number of rows affected by this query:

```
>>> cursor.execute('Create database Library')

1L      // 1L Indicates how many rows affected

>>> cursor.execute('use Library')

>>>table='create table books(book_accno char(30) primary key, book_name

char(50),no_of_copies int(5),price int(5))'

>>> cursor.execute(table)
```

```
0L
```

11. Finally, fetch the result set and iterate over this result set. In this step, the user can fetch the result sets as shown below:

```
>>> cursor.execute('select * from books')

2L

>>> cursor.fetchall()
```

```
(('Py9098', 'Programming With Python', 100L, 50L), ('Py9099', 'Programming With Python', 100L, 50L))
```

In this example, the fetchall() function is used to fetch the result sets.

More SQL operations

We can perform all SQL operations with Python DB-API. Insert, delete, aggregate and update queries can be illustrated as follows.

1. Insert SQL Query

```
2. >>>cursor.execute('insert books values
   (%s,%s,%s,%s)',('Py9098','Programming With Python',100,50))

3. lL        // Rows affected.

4. >>> cursor.execute('insert books values
   (%s,%s,%s,%s)',('Py9099','Programming With Python',100,50))
```

```
1L    //Rows affected.
```

If the user wants to insert duplicate entries for a book's accession number, the Python DB-API will show an error as it is the primary key. The following example illustrates this:

```
>>> cursor.execute('insert books values
(%s,%s,%s,%s)',('Py9099','Programming With Python',100,50))

>>>cursor.execute('insert books values
(%s,%s,%s,%s)',('Py9098','Programming With Perl',120,100))

Traceback (most recent call last):

File "<stdin>", line 1, in ?

File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 95, in
execute

return self._execute(query, args)

File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 114, in
_execute

self.errorhandler(self, exc, value)
```

```
File "/usr/lib/python2.3/site-packages/MySQLdb/connections.py", line 33, in defaulterrorhandler

raise errorclass, errorvalue

_mysql_exceptions.IntegrityError: (1062, "Duplicate entry 'Py9098' for key 1")
```

5. The Update SQL query can be used to update existing records in the database as shown below:

```
6. >>> cursor.execute('update books set price=%s where no_of_copies<=%s',[60,101])

7. 2L

8. >>> cursor.execute('select * from books')

9. 2L

10. >>> cursor.fetchall()
```

```
(('Py9098', 'Programming With Python', 100L, 60L), ('Py9099', 'Programming With Python', 100L, 60L))
```

11. The Delete SQL query can be used to delete existing records in the database as shown below:

```
12. >>> cursor.execute('delete from books where no_of_copies<=%s',[101])

13. 2L

14. >>> cursor.execute('select * from books')

15. 0L

16. >>> cursor.fetchall()

17. ()

18.

19. >>> cursor.execute('select * from books')
```

```
20. 3L
```

```
>>> cursor.fetchall() (('Py9099', 'Python-Cookbook', 200L, 90L), ('Py9098', 'Programming With Python', 100L, 50L), ('Py9097', 'Python-Nut shell', 300L, 80L))
```

21. Aggregate functions can be used with Python DB-API in the database as shown below:

```
22. >>> cursor.execute('select * from books')

23. 4L

24. >>> cursor.fetchall()

25. (('Py9099', 'Python-Cookbook', 200L, 90L), ('Py9098', 'Programming With Python', 100L, 50L), ('Py9097', 'Python-Nut shell', 300L, 80L), ('Py9096', 'Python-Nut shell', 400L, 90L))

26. >>> cursor.execute("select sum(price),avg(price) from books where book_name='Python-Nut shell'")

27. 1L

28. >>> cursor.fetchall()
((170.0, 85.0),)
```