UNIT - V

The Python programming language has powerful features for database programming. Python supports various databases like **SQLite**, **MySQL**, **Oracle**, **Sybase**, **PostgreSQL**, etc. Python also supports **Data Definition Language (DDL)**, **Data Manipulation Language (DML)** and Data Query Statements.

The Python standard for database interfaces is the Python DB-API (application programming interface). Most Python database interfaces adhere to this standard.

Here is the list of available Python database interfaces: Python Database Interfaces and APIs(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:

1. **Commit:** a commit is the updating of a record in a database. In the context of a database transaction, a commit refers to the saving of data permanently after a set of tentative changes.

   A commit ends a transaction within a relational database and allows all other users to see the changes.

2. **Rollback:** In database technologies, a rollback is an operation that returns the database to some previous state.
   Rollbacks are important for database integrity because they mean that the database can be restored to a clean copy even after erroneous operations are performed.

```
>>>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. **CursorUseResultMixI**n 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()
```

These objects represent a database cursor, which is used to manage the context of a fetch operation.

Cursors created from the same connection are not isolated, i.e., any changes done to the database by a cursor are immediately visible by the other cursors.

Cursors created from different connections can or cannot be isolated, depending on how the transaction support is implemented.

Cursor Attributes
Cursor Objects should respond to the following methods and attributes.

This read-only attribute is a sequence of 7-item sequences. Each of these sequences contains information describing one result column:

- name
- type_code
- display_size
- internal_size
- precision
- scale
- null_ok

The first two items (name and type_code) are mandatory, the other five are optional and are set to None if no meaningful values can be provided.

Cursor Methods

This method is optional since not all databases provide stored procedures.
.callproc(procname [, parameters]): It calls a stored database procedure with the given name. The sequence of parameters must contain one entry for each argument that the procedure expects.
.close(): Close the cursor now. The cursor will be unusable from this point forward; an Error (or subclass) exception will be raised if any operation is attempted with the cursor.

.execute(operation [, parameters]): Prepare and execute a database operation (query or command). Parameters may be provided as sequence or mapping and will be bound to variables in the operation.

.fetchall(): Fetch all (remaining) rows of a query result, returning them as a sequence of sequences (e.g. a list of tuples). Note that the cursor's arraysize attribute can affect the performance of this operation. An Error (or subclass) exception is raised if the previous call to **.execute*()** did not produce any result set or no call was issued yet.

.nextset(): This method will make the cursor skip to the next available set, discarding any remaining rows from the current set. If there are no more sets, the method returns None. Otherwise, it returns a true value, and subsequent calls to the **.fetch*()** methods will return rows from the next result set.

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:

```
>>> 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)

raise errorclass, errorvalue
```

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

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

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

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

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

5. Traceback (most recent call last):

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

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

8. return self._execute(query, args)

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

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

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

12. raise errorclass, errorvalue
```

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

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

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

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

16. Traceback (most recent call last):

17. 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.ProgrammingError: (1007, "Can't create database 'Library'.
Database exist
```

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:

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

6. 1L      // 1L Indicates how many rows affected

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

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

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

10. >>> 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:

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

13. 2L

14. >>> 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. 1L          // 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),)
```

DB-API Module Attributes

Attribute Description	**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 compliantwith,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

Notcursors
●3: Fully threadsafe: threads can share the module, connections, and cursors

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.

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**	**pyformat**
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.

1. **connect() Function Attributes**

By using connect() function 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()

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