UNIT-5

Database Programming

Introduction

The Database is a collection of organized information that can easily be used, managed, update.

Most Python's database interface remains to Python's DB-API standard & most of the databases have ODBC support. Other than that Java database usually support JDBC & programmers can work with that from Jython.

Benefits of Python Database programming:

→ Programming in Python is considerably simple & efficient with compared to other languages, so, as the database programming.

→ Python database is portable & the program is also portable so both can give an advantage in case of portability.

→ Python supports SQL cursors.

→ It also supports Relational Database systems.

→ The API of Python for the database is Compatible with other databases also.

→ It is platform Independent.

Python Database Application Programmer's Interface (DB-API)

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

You Can choose the right database for your application. Python DB-API supports a wide range of database servers such as -

→ Gadfly
→ mSQL ⇒ Mini SQL (It is a light weight database management system)
→ MySQL
→ PostgreSQL
→ Microsoft SQL server 2000
→ Informix
→ Interbase
→ Oracle
→ Sybase

You must download a separate DB-API module for each database you need to access. For example, if you need to access an oracle database as well as MySQL database, you must download both oracle & MySQL database modules.

The DB-API provides a minimal standard for working with databases using Python structures & syntax wherever possible.

This API includes the following—

- Importing the API module.
- Acquiring a Connection with the database
- Issuing SQL statements & stored procedures
- Closing the connection.

MySQLdb

It is an interface for associating SQL database server from Python & uses the DB-API of Python to work.

How to install MySQLdb:

Before proceeding, you make sure you have MySQLdb installed on your machine. Just type the following in your Python script & execute it.

```
import MySQLdb
```

If it produces the following result, then it means MySQLdb module is not installed.

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    import MySQLdb
ImportError: No module named MySQLdb
```

To install MySQLdb module, use the following Command,

```
pip install MySQL-Python
```

Eg:

```
import MySQLdb
#open database connection
db=MySQLdb.connect("localhost","user","pwd","testdb")
#prepare a cursor object using cursor() method
cursor=db.cursor()
#execute SQL query using execute() method
cursor.execute("SELECT VERSION()")
#Fetch a single row using fetchone() method
data=cursor.fetchone()
```

Eg: userId = user
Password = pwd
Database = testdb

```
print("Database version : %s" %data)
#disconnect from server
db.close()
```

→ Creating database

```
import MySQLdb
db=MySQLdb.connect("localhost", "user", "pwd")
#creating cursor
cursor = db.cursor()
#create database
cursor.execute("CREATE DATABASE my_first_db")
#get list of all databases
cursor.execute("SHOW DATABASES")
#print all databases
for db in cursor:
    print(db)
```

→ Creating Table

: Let's create a simple 'student' table.

```
import MySQLdb
db=MySQLdb.connect("localhost", "user", "pwd",
                   "my_first_db")
cursor=db.cursor()
cursor.execute("CREATE TABLE student(sid INT,
  Name VARCHAR(20))")
```

```
cursor.execute("SHOW TABLES")
for table in cursor:
    print(table)
```

→ ALTER Table

Add Primary key to student table.

```
import MySQLdb
db = MySQLdb.Connect("localhost", "user", "pwd",
                     "my_first_db")
cursor = db.cursor()
#Altering student table.
cursor.execute("ALTER TABLE student MODIFY
    id INT PRIMARY KEY")
```

→ Insert operation

```
import MySQLdb
db = MySQLdb.Connect("localhost", "user", "pwd",
                     "my_first_db")
cursor = db.cursor()
sql_query = "INSERT INTO student (sid, Name)
    VALUES(01, 'Arjun')"
cursor.execute(sql_query)
db.commit()
print(cursor.rowcount, "Record Inserted")
```

→ Creating employee table

```
import MySQLdb
db=MySQLdb.Connect("localhost","user","pwd",
                   "my_first_db")
Cursor=db.cursor()
#drop table if already exist using execute() method
Cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")
#Create table as per requirement
Sql="""CREATE TABLE EMPLOYEE(Eid INT
PRIMARYKEY, Name CHAR(10), Salary FLOAT)"""
Cursor.execute(sql)
db.close().
```

→ INSERT values into employee table

```
import MySQLdb
db=MySQLdb.connect("localhost","user","pwd","my_first_db")
Cursor=db.cursor()
Sql="""INSERT INTO EMPLOYEE(Eid,Name,Salary)
VALUES(01,'Arjun',27000)"""
try:
    Cursor.execute(sql)
    db.commit()
except:
    #Rollback in case there is any error
    db.rollback()
db.close()
```

READ operation:

- fetchone() - It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- fetchall() - It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
- rowcount() - This is a read-only attribute & returns the no. of rows that were affected by an execute() method.

→ UPDATE operation

```
import MySQLdb
db=MySQLdb.connect("localhost","user","pwd",
                   "my-first_db")
cursor=db.cursor()
sql="""UPDATE EMPLOYEE SET Salary=70000
 WHERE Eid=01 """
try:
    cursor.execute(sql)
    db.commit()
except:
    db.rollback()
db.close()
```

→ DELETE operation

```
import MySQLdb
db = MySQLdb.connect("localhost", "user", "pwd", "my_first_db")
cursor = db.cursor()
sql = "DELETE FROM EMPLOYEE WHERE Age > 30"
try:
    cursor.execute(sql)
    db.commit()
except:
    db.rollback()
db.close().
```

Object Relational Mappers (ORMs)

An object-relational mapper (ORM) is a code library that automates the transfer of data stored in relational databases tables into objects that are more commonly used in application code.

Relational database

Id	F_Name	L_Name	PhoneNo
1	John	Connor	+160555123
2	Matt	Makai	+120555689
3	Sasah	Smith	+19755551 2

Python objects

```
Class Person:
F_Name = "John"
L_Name = "Connor"
PhoneNO = "+160555123"
```

```
Class Person:
F_Name = "Matt"
L_Name = "Makai"
PhoneNO = "+12055689"
```

```
Class Person:
F_Name = "Sasah"
L_Name = "Smith"
PhoneNO = "+19755551 2"
```

ORMs provided a bridge between relational database tables, relationships of fields and Python objects

ORMs provide a high-level abstraction upon a relational database that allows a developer to write Python code instead of SQL to create, read, update & delete data and schemas in their database. Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures.

For example, without ORM a developer would write the following SQL statement to retrieve every row in the USERS table where the zip_code column is 94107:

```
SELECT * FROM USERS WHERE zip_code=94107;
```

The equivalent Django ORM query would instead look like the following Python code:

```
#obtain everyone in the 94107 zip_code & assign to users variable.
users = users.objects.filter(zip_code=94107)
```

The ability to write Python code instead of SQL can speed up web application development, especially at the beginning of a project.

ORMs also make it theoretically possible to switch an application between various relational databases. For example, a developer could use SQLite for local development & MySQL in production. A production application could be switched from MySQL to PostgreSQL with minimal code modifications.

[Note: In practice however, it's best to use same database for local development as is used in production. Otherwise unexpected errors could hit in production that were not seen in a local development environment. Also, its rare that a project would switch from one database in production to another one unless there was a pressing reason.]

Python ORM libraries are not required for accessing relational databases. Infact, the low-level access is typically provided by another library called a database connector, such as psycopg (for PostgreSQL) or MySQL-Python (for MySQL)

Python ORM Implementations:

There are numerous ORM implementations written in Python,

1) SQLAlchemy ORM
2) ~~Peewee~~ ORM
   peewee
3) Django ORM
4) Pony ORM
5) SQLObject ORM
6) Tortoise ORM

1) SQLAlchemy ORM:

SQLAlchemy is a well-regarded Python ORM because it gets the abstraction level "just right" & seems to make complex database queries easier to write than the Django ORM in most cases.

It provides a generalized interface for creating & executing database-agnostic code without needing to write SQL statements.

2> peewee ORM :

peewee is a Python ORM implementation that is written to be "simpler, smaller & more hackable" than SQLAlchemy. peewee is good for pulling data out of relational database in a script or Jupyter notebook.

3> Django ORM:

The Django web framework comes with its own built-in ORM module, generally referred as "the django ORM".

It works well for simple & medium-complexity database operations.

4> Pony ORM:

Pony ORM is another Python ORM available as open source, under Apache 2.0 license.

5> SQLObject ORM:

SQLObject is an ORM that has been under active open source development for over many years.

SQLObject includes a Python-object-based query language that makes SQL more abstract & provides substantial database independence for applications.

6) Tortoise ORM:

Tortoise ORM is an easy-to-use asyncio (asynchronous I/O, is a library to write concurrent code using async/await syntax) ORM inspired by Django.

Tortoise ORM is supported on CPython >= 3.6 for SQLite, MySQL & PostgreSQL, and PyPy 3.6 >= 7.1 for SQLite & MySQL only.

Tortoise ORM is designed to be functional, yet familiar, to ease the migration of developers wishing to switch to asyncio.

Downsides of using ORM:

1) Impedance Mismatch - Difficulties that occur when moving data between relational tables & application objects. The problem is that the way a developer uses objects is different from how data is stored & joined in relational tables.

2) Potential for reduced performance — with ORMs, the performance hit comes from the translation of application code into a corresponding SQL statement which may not be tuned properly.

3) Shifting complexity from database into app code — The code for working with an application's data has to live somewhere. Before ORMs were common, database stored procedures were used to encapsulate the database logic. With an ORM, the data manipulation code instead lives within the applications Python codebase. It increases the total amount of Python code instead of splitting code between the application & the database stored procedures.