Python - Acesso ao banco de dados

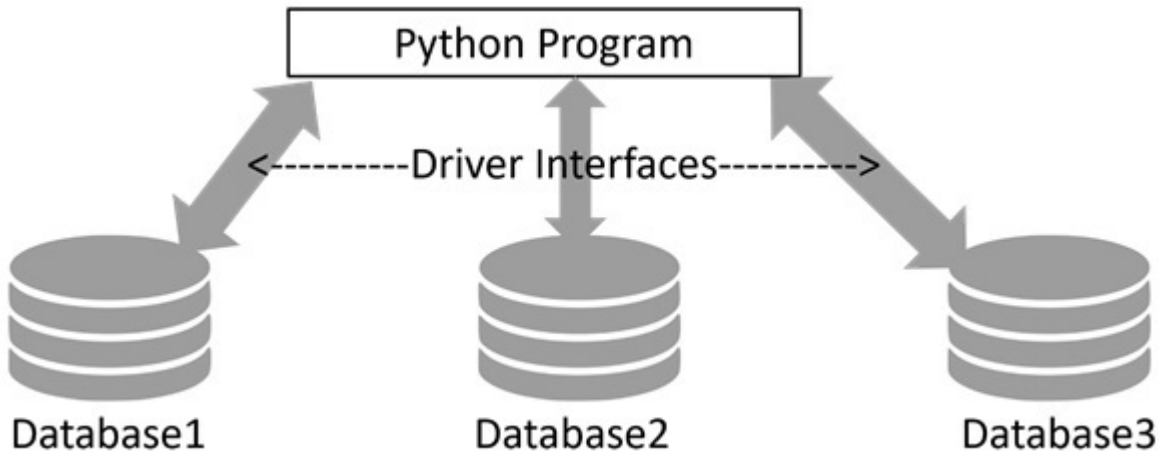
Os dados inseridos e gerados durante a execução de um programa são armazenados na RAM. Se for armazenado de forma persistente, ele precisará ser armazenado em tabelas de banco de dados. Existem vários sistemas de gerenciamento de banco de dados relacional (RDBMS) disponíveis.

- GadFly
- MySQL
- PostgreSQL
- Servidor SQL da Microsoft
- Informix
- Oráculo
- Sybase
- SQLite
- e muitos mais...

Neste capítulo, aprenderemos como acessar o banco de dados usando Python, como armazenar dados de objetos Python em um banco de dados SQLite e como recuperar dados do banco de dados SQLite e processá-los usando o programa Python.

Os bancos de dados relacionais usam SQL (Structured Query Language) para realizar operações INSERT/DELETE/UPDATE nas tabelas do banco de dados. No entanto, a implementação do SQL varia de um tipo de banco de dados para outro. Isso levanta problemas de incompatibilidade. As instruções SQL para um banco de dados não correspondem a outro.

Para superar essa incompatibilidade, uma interface comum foi proposta no PEP (Python Enhancement Proposal) 249. Esta proposta é chamada DB-API e exige que um programa de driver de banco de dados usado para interagir com Python seja compatível com DB-API.



A biblioteca padrão do Python inclui o módulo sqlite3, que é um driver compatível com DB_API para banco de dados SQLite3, e também é uma implementação de referência do DB-API.

Como a interface DB-API necessária está integrada, podemos facilmente usar o banco de dados SQLite com um aplicativo Python. Para outros tipos de bancos de dados, você terá que instalar o pacote Python relevante.

Base de dados	Pacote Python
Oráculo	cx_oracle,pyodbc
servidor SQL	pymssql, pyodbc
PostgreSQL	psicopg2
MySQL	Conector MySQL/Python, pymysql

Um módulo DB-API como sqlite3 contém classes de conexão e cursor. O objeto de conexão é obtido com o método connect() fornecendo as credenciais de conexão necessárias, como nome do servidor e número da porta, e nome de usuário e senha, se aplicável. O objeto de conexão trata da abertura e fechamento do banco de dados e do mecanismo de controle de transação para confirmar ou reverter uma transação.

O objeto cursor, obtido do objeto conexão, atua como identificador do banco de dados ao realizar todas as operações CRUD.

O Módulo sqlite3

SQLite é um banco de dados relacional transacional leve, sem servidor e baseado em arquivo. Não requer nenhuma instalação e nenhuma credencial como nome de usuário e senha é necessária para acessar o banco de dados.

O módulo sqlite3 do Python contém implementação DB-API para banco de dados SQLite. Foi escrito por Gerhard Häring. Vamos aprender como usar o módulo sqlite3 para acesso ao banco de dados com Python.

Vamos começar importando o sqlite3 e verificar sua versão.

```
>>> import sqlite3
>>> sqlite3.sqlite_version
'3.39.4'
```

O objeto de conexão

Um objeto de conexão é configurado pela função connect() no módulo sqlite3. O primeiro argumento posicional para esta função é uma string que representa o caminho (relativo ou

absoluto) para um arquivo de banco de dados SQLite. A função retorna um objeto de conexão referente ao banco de dados.

```
>>> conn=sqlite3.connect('testdb.sqlite3')
>>> type(conn)
<class 'sqlite3.Connection'>
```

Vários métodos são definidos na classe de conexão. Um deles é o método cursor() que retorna um objeto cursor, sobre o qual conheceremos na próxima seção. O controle de transação é obtido pelos métodos commit() e rollback() do objeto de conexão. A classe Connection possui métodos importantes para definir funções customizadas e agregações a serem usadas em consultas SQL.

O objeto Cursor

A seguir, precisamos obter o objeto cursor do objeto de conexão. É o seu identificador para o banco de dados ao executar qualquer operação CRUD no banco de dados. O método cursor() no objeto de conexão retorna o objeto cursor.

```
>>> cur=conn.cursor()
>>> type(cur)
<class 'sqlite3.Cursor'>
```

Agora podemos realizar todas as operações de consulta SQL, com a ajuda de seu método execute() disponível para o objeto cursor. Este método precisa de um argumento de string que deve ser uma instrução SQL válida.

Criando uma tabela de banco de dados

Agora adicionaremos a tabela Employee em nosso banco de dados 'testdb.sqlite3' recém-criado. No script a seguir, chamamos o método execute() do objeto cursor, fornecendo a ele uma string com a instrução CREATE TABLE dentro.

```
import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry='''
CREATE TABLE Employee (
EmpID INTEGER PRIMARY KEY AUTOINCREMENT,
FIRST_NAME TEXT (20),
LAST_NAME TEXT(20),
AGE INTEGER,
SEX TEXT(1),
INCOME FLOAT
);
'''
```

```
try:
   cur.execute(qry)
   print ('Table created successfully')
except:
   print ('error in creating table')
conn.close()
```

Quando o programa acima é executado, o banco de dados com a tabela Employee é criado no diretório de trabalho atual.

Podemos verificar listando as tabelas deste banco de dados no console SQLite.

```
sqlite> .open mydb.sqlite
sqlite> .tables
Employee
```


Operação INSERIR

A operação INSERT é necessária quando você deseja criar seus registros em uma tabela de banco de dados.

Exemplo

O exemplo a seguir executa a instrução SQL INSERT para criar um registro na tabela EMPLOYEE -

```
import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="""INSERT INTO EMPLOYEE(FIRST_NAME,
   LAST_NAME, AGE, SEX, INCOME)
   VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
   cur.execute(qry)
   conn.commit()
   print ('Record inserted successfully')
except:
   conn.rollback()
print ('error in INSERT operation')
conn.close()
```

Você também pode usar a técnica de substituição de parâmetros para executar a consulta INSERT da seguinte forma -

```
import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="""INSERT INTO EMPLOYEE(FIRST_NAME,
   LAST_NAME, AGE, SEX, INCOME)
   VALUES (?, ?, ?, ?, ?)"""
try:
   cur.execute(qry, ('Makrand', 'Mohan', 21, 'M', 5000))
   conn.commit()
   print ('Record inserted successfully')
except Exception as e:
   conn.rollback()
   print ('error in INSERT operation')
conn.close()
```


LER Operação

A operação READ em qualquer banco de dados significa buscar algumas informações úteis do banco de dados.

Depois que a conexão com o banco de dados for estabelecida, você estará pronto para fazer uma consulta nesse banco de dados. Você pode usar o método fetchone() para buscar um único registro ou o método fetchall() para buscar vários valores de uma tabela de banco de dados.

- **fetchone()** - Busca a próxima linha de um conjunto de resultados de consulta. Um conjunto de resultados é um objeto retornado quando um objeto cursor é usado para consultar uma tabela.
- **fetchall()** - Busca todas as linhas em um conjunto de resultados. Se algumas linhas já tiverem sido extraídas do conjunto de resultados, ele recuperará as linhas restantes do conjunto de resultados.
- **rowcount** - Este é um atributo somente leitura e retorna o número de linhas que foram afetadas por um método execute().

Exemplo

No código a seguir, o objeto cursor executa a consulta SELECT * FROM EMPLOYEE. O conjunto de resultados é obtido com o método fetchall(). Imprimimos todos os registros no conjunto de resultados com um loop **for** .

```
import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="SELECT * FROM EMPLOYEE"

try:
   # Execute the SQL command
   cur.execute(qry)
   # Fetch all the rows in a list of lists.
   results = cur.fetchall()
   for row in results:
      fname = row[1]
      lname = row[2]
      age = row[3]
      sex = row[4]
      income = row[5]
      # Now print fetched result
      print ("fname={},lname={},age={},sex={},income={}".format(fname, lname, age, se
except Exception as e:
   print (e)
   print ("Error: unable to fecth data")

conn.close()
```

Ele produzirá a seguinte **saída** -

```
fname=Mac,lname=Mohan,age=20,sex=M,income=2000.0
fname=Makrand,lname=Mohan,age=21,sex=M,income=5000.0
```


Operação de atualização

UPDATE A operação em qualquer banco de dados significa atualizar um ou mais registros, que já estão disponíveis no banco de dados.

O procedimento a seguir atualiza todos os registros com renda = 2.000. Aqui, aumentamos a receita em 1000.

```
import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="UPDATE EMPLOYEE SET INCOME = INCOME+1000 WHERE INCOME=?"

try:
   # Execute the SQL command
   cur.execute(qry, (1000,))
   # Fetch all the rows in a list of lists.
   conn.commit()
   print ("Records updated")
except Exception as e:
   print ("Error: unable to update data")
conn.close()
```

Operação EXCLUIR

A operação DELETE é necessária quando você deseja excluir alguns registros do seu banco de dados. A seguir está o procedimento para excluir todos os registros de EMPLOYEE onde INCOME for inferior a 2.000.

```
import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="DELETE FROM EMPLOYEE WHERE INCOME<?"

try:
   # Execute the SQL command
   cur.execute(qry, (2000,))
   # Fetch all the rows in a list of lists.
   conn.commit()
   print ("Records deleted")
except Exception as e:
   print ("Error: unable to delete data")

conn.close()
```

Realizando transações

As transações são um mecanismo que garante a consistência dos dados. As transações têm as seguintes quatro propriedades -

- **Atomicity** - Ou uma transação é concluída ou nada acontece.

- **Consistency** - Uma transação deve começar em um estado consistente e deixar o sistema em um estado consistente.
- **Isolation** - Os resultados intermediários de uma transação não são visíveis fora da transação atual.
- **Durability** - Depois que uma transação é confirmada, os efeitos são persistentes, mesmo após uma falha do sistema.

A API Python DB 2.0 fornece dois métodos para confirmar ou reverter uma transação.

Exemplo

Você já sabe como implementar transações. Aqui está um exemplo semelhante -

```
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > ?"
try:
   # Execute the SQL command
   cursor.execute(sql, (20,))
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
```

Operação COMMIT

Commit é uma operação que dá um sinal verde ao banco de dados para finalizar as alterações, e após esta operação nenhuma alteração pode ser revertida.

Aqui está um exemplo simples para chamar o método commit.

```
db.commit()
```

Operação de ROLLBACK

Se você não estiver satisfeito com uma ou mais alterações e quiser revertê-las completamente, use o método rollback().

Aqui está um exemplo simples para chamar o método rollback().

```
db.rollback()
```

O Módulo PyMySQL

PyMySQL é uma interface para conexão com um servidor de banco de dados MySQL do Python. Ele implementa a API de banco de dados Python v2.0 e contém uma biblioteca cliente MySQL Python pura. O objetivo do PyMySQL é ser um substituto imediato para o MySQLdb.

Instalando PyMySQL

Antes de prosseguir, certifique-se de ter o PyMySQL instalado em sua máquina. Basta digitar o seguinte em seu script Python e executá-lo -

```
import PyMySQL
```

Se produzir o seguinte resultado, significa que o módulo MySQLdb não está instalado -

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

A última versão estável está disponível no PyPI e pode ser instalada com pip -

```
pip install PyMySQL
```

Note - Certifique-se de ter privilégios de root para instalar o módulo acima.

Conexão de banco de dados MySQL

Antes de conectar-se a um banco de dados MySQL, certifique-se dos seguintes pontos -

- Você criou um banco de dados TESTDB.
- Você criou uma tabela EMPLOYEE em TESTDB.
- Esta tabela possui os campos FIRST_NAME, LAST_NAME, AGE, SEX e INCOME.
- O ID do usuário "testuser" e a senha "test123" estão definidos para acessar o TESTDB.
- O módulo Python PyMySQL está instalado corretamente em sua máquina.
- Você passou pelo tutorial do MySQL para entender os fundamentos do MySQL.

Exemplo

Para usar o banco de dados MySQL em vez do banco de dados SQLite nos exemplos anteriores, precisamos alterar a função connect() da seguinte forma -

```
import PyMySQL
# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )
```

Além dessa mudança, todas as operações do banco de dados podem ser realizadas sem dificuldade.

Tratamento de erros

Existem muitas fontes de erros. Alguns exemplos são um erro de sintaxe em uma instrução SQL executada, uma falha de conexão ou a chamada do método fetch para um identificador de instrução já cancelado ou concluído.

A API do banco de dados define uma série de erros que devem existir em cada módulo do banco de dados. A tabela a seguir lista essas exceções.

Sr. Não.	Exceção e descrição
1	**Aviso** Usado para problemas não fatais. Deve subclassificar StandardError.
2	**Erro** Classe base para erros. Deve subclassificar StandardError.
3	**Erro de interface** Usado para erros no módulo de banco de dados, não no próprio banco de dados. Deve subclassificar Error.
4	**Erro de banco de dados** Usado para erros no banco de dados. Deve subclassificar Error.
5	**Erro de dados** Subclasse de DatabaseError que se refere a erros nos dados.
6	**Erro Operacional** Subclasse de DatabaseError que se refere a erros como perda de conexão com o banco de dados. Esses erros geralmente estão fora do controle do criador de scripts Python.
7	**Erro de integridade** Subclasse de DatabaseError para situações que prejudicariam a integridade relacional, como restrições de exclusividade ou chaves estrangeiras.

8	**Erro interno** Subclasse de DatabaseError que se refere a erros internos ao módulo de banco de dados, como um cursor que não está mais ativo.
9	**Erro de programação** Subclasse de DatabaseError que se refere a erros como nome de tabela incorreto e outras coisas que podem ser atribuídas a você com segurança.
10	**Erro NotSupported** Subclasse de DatabaseError que se refere à tentativa de chamar uma funcionalidade não suportada.