SQLite - Python

Installation

SQLite3 can be integrated with Python using sqlite3 module, which was written by Gerhard Haring. It provides an SQL interface compliant with the DB-API 2.0 specification described by PEP 249. You do not need to install this module separately because it is shipped by default along with Python version 2.5.x onwards.

To use sqlite3 module, you must first create a connection object that represents the database and then optionally you can create a cursor object, which will help you in executing all the SQL statements.

Python sqlite3 module APIs

Following are important sqlite3 module routines, which can suffice your requirement to work with SQLite database from your Python program. If you are looking for a more sophisticated application, then you can look into Python sqlite3 module's official documentation.

Sr.No.	API & Description
1	**sqlite3.connect(database [,timeout ,other optional arguments])** This API opens a connection to the SQLite database file. You can use ":memory:" to open a database connection to a database that resides in RAM instead of on disk. If database is opened successfully, it returns a connection object. When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The timeout parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds). If the given database name does not exist then this call will create the database. You can specify filename with the required path as well if you want to create a database anywhere else except in the current directory.
2	**connection.cursor([cursorClass])** This routine creates a **cursor** which will be used throughout of your database programming with Python. This method accepts a single optional parameter cursorClass. If supplied, this must be a custom cursor class that extends sqlite3.Cursor.
3	**cursor.execute(sql [, optional parameters])** This routine executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks and named placeholders (named style).

	For example − cursor.execute("insert into people values (?, ?)", (who, age))
4	**connection.execute(sql [, optional parameters])** This routine is a shortcut of the above execute method provided by the cursor object and it creates an intermediate cursor object by calling the cursor method, then calls the cursor's execute method with the parameters given.
5	**cursor.executemany(sql, seq_of_parameters)** This routine executes an SQL command against all parameter sequences or mappings found in the sequence sql.
6	**connection.executemany(sql[, parameters])** This routine is a shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor.s executemany method with the parameters given.
7	**cursor.executescript(sql_script)** This routine executes multiple SQL statements at once provided in the form of script. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter. All the SQL statements should be separated by a semi colon (;).
8	**connection.executescript(sql_script)** This routine is a shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's executescript method with the parameters given.
9	**connection.total_changes()** This routine returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.
10	**connection.commit()** This method commits the current transaction. If you don't call this method, anything you did since the last call to commit() is not visible from other database connections.
11	**connection.rollback()** This method rolls back any changes to the database since the last call to commit().
12	**connection.close()**

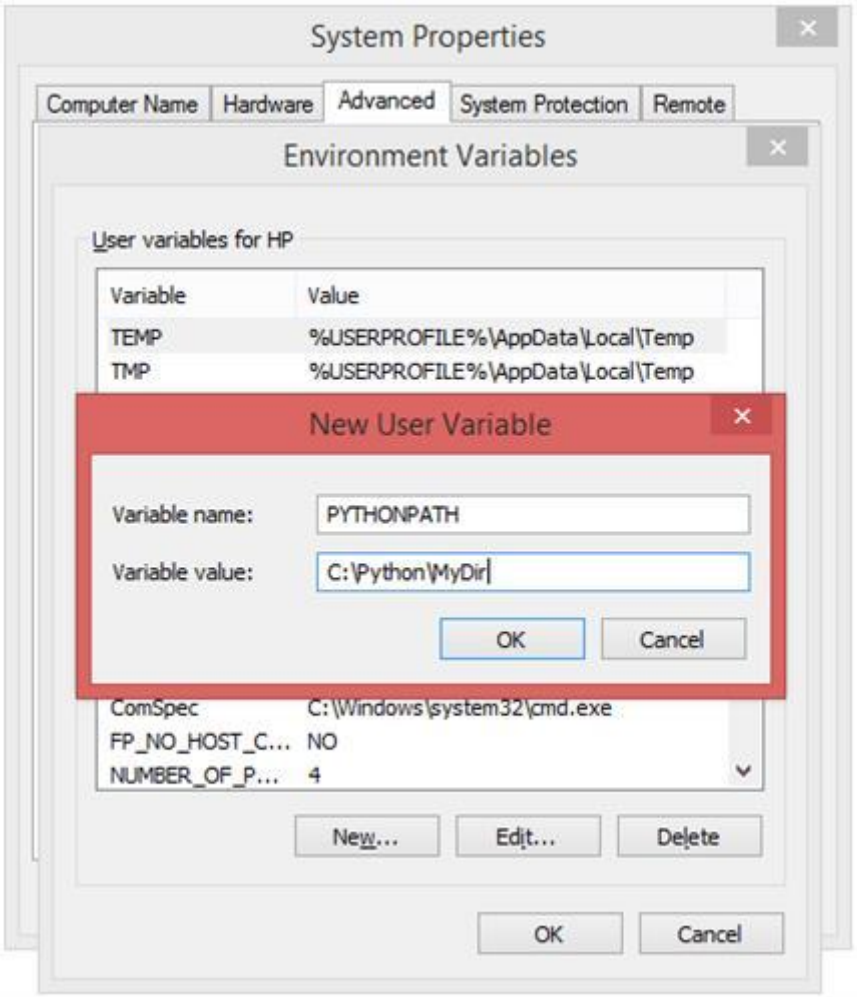
	This method closes the database connection. Note that this does not automatically call commit(). If you just close your database connection without calling commit() first, your changes will be lost!
13	**cursor.fetchone()** This method fetches the next row of a query result set, returning a single sequence, or None when no more data is available.
14	**cursor.fetchmany([size = cursor.arraysize])** This routine fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available. The method tries to fetch as many rows as indicated by the size parameter.
15	**cursor.fetchall()** This routine fetches all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available.

What is PYTHONPATH?

PYTHONPATH is an environment variable which the user can set to add additional directories that the user wants Python to add to the sys.path directory list. In short, we can say that it is an environment variable that you set before running the Python interpreter. Mostly you should not set these variables as they are not needed for Python to execute normal programs because it knows where its standard library is to be found. PYTHONPATH is used to help in importing the modules. So, when you import modules in your Python scripts, PYTHONPATH is also checked to see which directories might contain the imported module.

How to add to the PYTHONPATH in windows?

- My Computer > Properties > Advanced System Settings > Environment Variables >
- Click the “New” button in the top half of the dialog, to make a new user variable.
- Give the variable name as PYTHONPATH and the value is the path to the code directory.



- **C**lick OK and then OK to save this variable.

To confirm **PYTHONPATH,** open Command Prompt and type :

- Echo %PYTHONPATH%

NOTE:- Don't confuse it with Python PATH environment variable. That is used to assist OS in calling an executable from anywhere which means if you just type Python on your Command Window, the system will look into the PATH to see which directories might contain an executable named python.

Setting the path for Python:

Windows allows environment variables to be configured permanently at both User level as well as the System level, or temporarily in a command prompt.

To run Python properly from a Command Prompt, you might choose to change some default environment variables in windows.

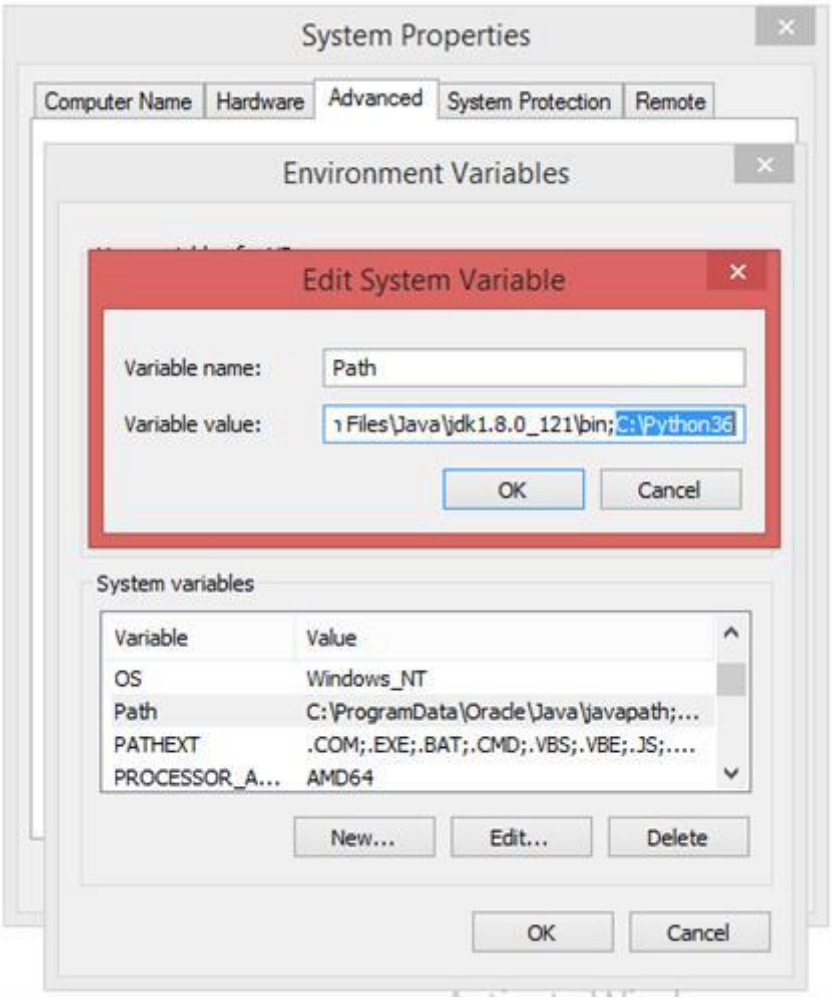
- To temporarily set **environment variables**, open Command Prompt and use the "set" command:

  C:\> set PATH=" Directory of your python folder in C: drive"; %path%

- To permanently modify the default **environment variables:**

How to set the Python path in windows:

- My Computer > Properties > Advanced System Settings > Environment Variables > Edit.
- Add python's path to the end of the given directory ending with a semicolon (;)



How to set Python path in Unix or Linux:

To add the Python directory to the path for a particular session in Unix or Linux

- In the cshshell : type setenv PATH "$PATH:/usr/local/bin/python" and press Enter.
- In the bash shell (Linux) :type export PATH="$PATH:/usr/local/bin/python" and press Enter.
- In the sh or kshshell :type PATH="$PATH:/usr/local/bin/python" and press Enter.

NOTE:- /usr/local/bin/python is the path of the Python directory.

Namespaces in Python

A namespace is a collection of currently defined symbolic names along with information about the object that each name references. You can think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves. Each key-value pair maps a name to its corresponding object.

> Namespaces are one honking great idea—let's do more of those!
>
> — *The Zen of Python, by Tim Peters*

As Tim Peters suggests, namespaces aren't just great. They're *honking* great, and Python uses them extensively. In a Python program, there are four types of namespaces:

1. Built-In
2. Global
3. Enclosing
4. Local

These have differing lifetimes. As Python executes a program, it creates namespaces as necessary and deletes them when they're no longer needed. Typically, many namespaces will exist at any given time.

Variable Scope

The existence of multiple, distinct namespaces means several different instances of a particular name can exist simultaneously while a Python program runs. As long as each instance is in a different namespace, they're all maintained separately and won't interfere with one another.

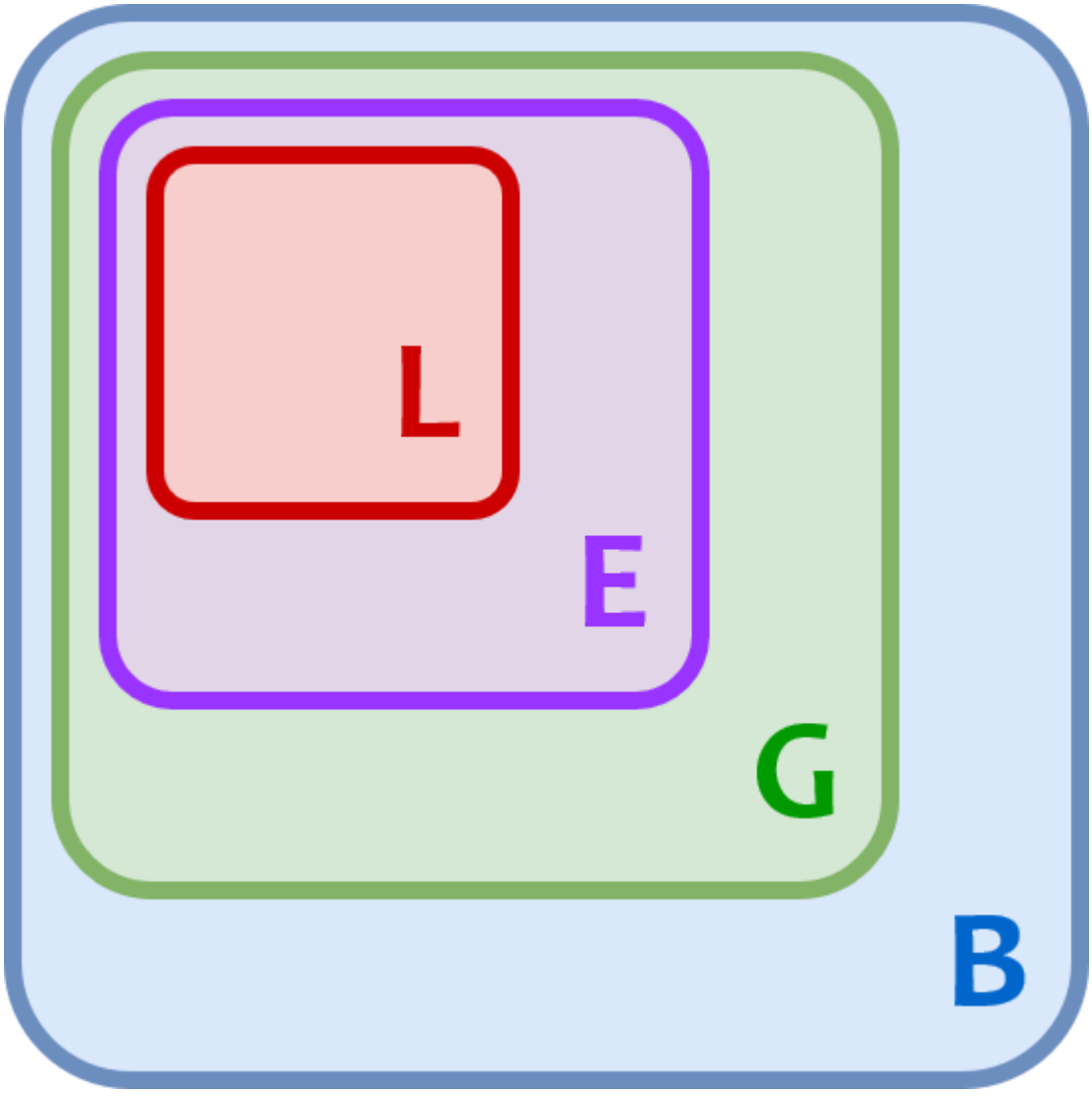
But that raises a question: Suppose you refer to the name `x` in your code, and `x` exists in several namespaces. How does Python know which one you mean?

The answer lies in the concept of **scope**. The scope of a name is the region of a program in which that name has meaning. The interpreter determines this at runtime based on where the name definition occurs and where in the code the name is referenced.

To return to the above question, if your code refers to the name `x`, then Python searches for `x` in the following namespaces in the order shown:

1. **Local**: If you refer to `x` inside a function, then the interpreter first searches for it in the innermost scope that's local to that function.
2. **Enclosing**: If `x` isn't in the local scope but appears in a function that resides inside another function, then the interpreter searches in the enclosing function's scope.
3. **Global**: If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
4. **Built-in**: If it can't find `x` anywhere else, then the interpreter tries the built-in scope.

This is the **LEGB rule** as it's commonly called in Python literature (although the term doesn't actually appear in the Python documentation). The interpreter searches for a name from the inside out, looking in the **l**ocal, **e**nclosing, **g**lobal, and finally the **b**uilt-in scope:



If the interpreter doesn't find the name in any of these locations, then Python raises a `NameError` exception.

What are Python packages?

Educative Answers Team

A **python package** is a collection of modules. Modules that are related to each other are mainly put in the same package. When a module from an external package is required in a program, that package can be imported and its modules can be put to use.

Any Python file, whose name is the module's name property without the **.py** extension, is a **module**.

A package is a directory of Python modules that contains an additional `__init__.py` file, which distinguishes a package from a directory that is supposed to contain multiple Python scripts. Packages can be nested to multiple depths if each corresponding directory contains its own `__init__.py` file.

When you import a module or a package, the object created by Python is always of type module.

When you import a package, only the methods and the classes in the `__init__.py` file of that package are directly visible.

Code

For example, let's take the `datetime` module, which has a submodule called `date`. When `datetime` is imported, it'll result in an error, as shown below:

The commit() method is one among the various methods in Python which is used to make the database transactions.

Here, we will discuss about the commit() method. The commit() method is used to confirm the changes made by the user to the database. Whenever any change is made to the database using update or any other statements, it is necessary to commit the changes. If we donot use the commit() method after making any changes to the database, the database will not not be updated and changes will not be reflected.

Syntax

```
db.commit()
```

db refers to the database connection object.

Given below is an example to update value in a table and commit the changes to the database.

Steps invloved to update data and commit change made in a table using MySQL in python

- import MySQL connector
- establish connection with the connector using connect()
- create the cursor object using cursor() method
- create a query using the appropriate mysql statements
- execute the SQL query using execute() method
- commit the changes made using commit() method
- close the connection

Suppose we have a table named “Student” as follows −

```
+----------+---------+-----------+------------+
|  Name  |  Class |   City   |   Marks   |
+----------+---------+-----------+------------+
|  Karan |   4    | Amritsar |   95     |
|  Sahil |   6    | Amritsar |   93     |
|  Kriti |   3    | Batala   |   88     |
| Khushi |   9    | Delhi    |   90     |
|  Kirat |   5    | Delhi    |   85     |
+----------+---------+-----------+------------+
```

Example

Suppose, we have the above table of students and we want to update the city of Kriti from Batala to Kolkata. And commit the changes to the database.

```
import mysql.connector
db=mysql.connector.connect(host="your host", user="your username", password="your
```

```
password",database="database_name")

cursor=db.cursor()

query="UPDATE Students SET City='Kolkata' WHERE Name='Kriti'"
cursor.execute(query)
db.commit()

query="SELECT * FROM Students"
cursor.execute(query)

for row in cursor:
  print(row)
db.close()
```

The above code updates the city name of Kriti and commits this change to the database.

Output

```
('Karan', 4 ,'Amritsar' , 95)
('Sahil' , 6 , 'Amritsar' ,93)
('Kriti' , 3 , 'Kolkata' ,88)
('Amit' , 9 , 'Delhi' , 90)
('Priya' , 5 , 'Delhi' ,85)
```

NOTE

The **db.commit()** in the above code is important. It is used to commit the changes made to the table. Without using commit(), no changes will be made in the table.

Connect To Database

Following Python code shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')

print "Opened database successfully";
```

Here, you can also supply database name as the special name **:memory:** to create a database in RAM. Now, let's run the above program to create our database **test.db** in the current directory. You can change your path as per your requirement. Keep the above code in sqlite.py file and execute it as shown below. If the database is successfully created, then it will display the following message.

```
$chmod +x sqlite.py
$./sqlite.py
Open database successfully
```

Create a Table

Following Python program will be used to create a table in the previously created database.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute('''CREATE TABLE COMPANY
       (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL);''')
print "Table created successfully";

conn.close()
```

When the above program is executed, it will create the COMPANY table in your **test.db** and it will display the following messages −

```
Opened database successfully
Table created successfully
```

INSERT Operation

Following Python program shows how to create records in the COMPANY table created in the above example.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
    VALUES (1, 'Paul', 32, 'California', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
    VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
    VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
    VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");

conn.commit()
print "Records created successfully";
conn.close()
```

When the above program is executed, it will create the given records in the COMPANY table and it will display the following two lines −

```
Opened database successfully
Records created successfully
```

SELECT Operation

Following Python program shows how to fetch and display records from the COMPANY table created in the above example.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
```

```
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 20000.0

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

UPDATE Operation

Following Python code shows how to use UPDATE statement to update any record and then fetch and display the updated records from the COMPANY table.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("UPDATE COMPANY set SALARY = 25000.00 where ID = 1")
conn.commit()
print "Total number of rows updated :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
```

```
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
Total number of rows updated : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 25000.0

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

DELETE Operation

Following Python code shows how to use DELETE statement to delete any record and then fetch and display the remaining records from the COMPANY table.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("DELETE from COMPANY where ID = 2;")
conn.commit()
print "Total number of rows deleted :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
```

```
  print "ADDRESS = ", row[2]
  print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
Total number of rows deleted : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 20000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```