Python

Using SQLite in Python
Contributed by Peyton McCullough
2006−04−12

SQLite is a small C library that implements a self−contained SQL database engine. This article examines pysqlite, one of a number of libraries in Python, that is used as an interface with SQLite.

SQLite is a small database engine that's gained quite a bit of popularity, and this popularity does appear to be deserved. SQLite stands alone and doesn't require any special dependencies. All the database interaction is done through whatever program needs to access the information, and the database itself is stored in a single file. Because of this, there is no confusing configuration involved, and SQLite is small and, in many applications, quite fast. It's also been released in the public domain.

Its presence in several languages is no secret, but what many people don't know is that there are also a few Python SQLite libraries. One of these libraries is pysqlite, which allows for painless database interaction. In this article, we'll explore SQLite and pysqlite and their capabilities. While prior database experience in Python is not required, you should be familiar with SQL in general.

Obtaining the Library

The pysqlite library may be obtained at its website:

http://initd.org/tracker/pysqlite

Windows users can simply download and run the pysqlite binary to install the product. Users of Linux and other similar operating systems will need to download the source distribution, and they will need SQLite and the SQLite development files installed. Many distributions already come with the library, so check for it. If not, you may obtain SQLite from its website:

http://sqlite.org/

The library is installed just like most other Python libraries. First, you must build the library:

```
python setup.py build
```

Next, simply install what you've just built. Make sure you have the proper privileges to do this:

```
python setup.py install
```

If everything went as it should, then pysqlite is now installed and ready for use.

Now we're ready to begin working with pysqlite. To start using the library, it, of course, needs to be imported. It can be found under the name of *pysqlite2*, but we'll need the *dbapi2* submodule. This should be imported under the name of *sqlite* for clarity:

```
>>> from pysqlite2 import dbapi2 as sqlite
```

Now we can begin using pysqlite. The first thing we'll do is create a new database to work with. This is done by creating a connection object and passing the name of a database file. If the database file does not exist, which, in our case, it doesn't, then it will be created. If it does exist, then it will be opened:

```
>>> connection = sqlite.connect('test.db')
```

It's also possible to create a temporary database that exists in memory:

```
>>> memoryConnection = sqlite.connect(':memory:')
```

Once we have a connection object, we must create a cursor object, which does the interaction for us. This is done by calling the *cursor* method of our connection object:

```
>>> cursor = connection.cursor()
```

We're ready to construct a table in our database now. To do this, we must call the *execute* method of our cursor, passing the necessary statement. We'll create a table that stores names and e−mail addresses, with each row possessing a unique number:

```
>>> cursor.execute('CREATE TABLE names (id INTEGER PRIMARY KEY,
name VARCHAR(50), email VARCHAR(50))')
```

Now that we've created our table, we can add some data using the *execute* method as well:

```
>>> cursor.execute('INSERT INTO names VALUES (null, "John Doe",
"jdoe@jdoe.zz")')
>>> cursor.execute('INSERT INTO names VALUES (null, "Mary Sue",
"msue@msue.yy")')
```

It's also possible to get the number of the last row from the cursor, should we need it:

```
>>> cursor.lastrowid
2
```

Of course, let's say we needed to insert a row with values that depend on user input. Obviously, user input is unsafe to use as it stands. A malicious user could easily pass in a value that manipulates the query and does some real damage to things. This is unacceptable, but, thankfully, pysqlite contains a way to ensure security. Let's pretend we have two variables containing the name and e−mail address of a person:

```
>>> name = "Luke Skywalker"
>>> email ="use@the.force"
```

Now, these are pretty safe values since we defined them ourselves, but it might not always be that way. To place these values in a query safely, we simply have to put question marks in their place, and pysqlite will take care of the rest:

```
>>> cursor.execute('INSERT INTO names VALUES (null, ?, ?)',
(name, email))
```

Now that we are done making our changes, we must save, or commit, them. This is done by calling the *commit* method of our connection:

```
>>> connection.commit()
```

If you attempt to close a connection that has been modified without a call to the commit method, then pysqlite will raise an error. This behavior may cause problems, however, if you don't want to save changes that you've made. This is where the *rollback* method comes in, which has the power to erase any unsaved changes. For example, let's add another row to our table:

```
>>> cursor.execute('INSERT INTO names VALUES (null, "Bobby John",
"bobby@john.qq")')
```

Now let's say we don't want the change to take effect. We can simply call the *rollback* method:

```
>>> connection.rollback()
```

There, the change has been erased.

We can now retrieve data from our table by querying our database with “SELECT.” Let's obtain all the values in our table:

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

The cursor now contains the data from the table, and there are several ways that we can retrieve that data. The first way is a list of every single row that the query returned using the *fetchall* method:

```
>>> cursor.execute('SELECT * FROM names')
>>> print cursor.fetchall()
[(1, u'John Doe', u'jdoe@jdoe.zz'), (2, u'Mary Sue',
u'msue@msue.yy'), (3, u'Luke Skywalker', u'use@the.force')]
```

Notice how it returns a list full of tuples, one tuple for each row. Obviously, it's pretty easy to loop through everything. All that's required is a for loop. However, I should note that an even easier way to loop through returned data is to just iterate through the cursor, like this:

```
>> cursor.execute('SELECT * FROM names')
>>> for row in cursor:
                print '-'*10
                print 'ID:', row[0]
                print 'Name:', row[1]
                print 'E-Mail:', row[2]
                print '-'*10

----------

ID: 1
Name: John Doe
E-Mail: jdoe@jdoe.zz

----------
```

```
----------

ID: 2
Name: Mary Sue
E-Mail: msue@msue.yy

----------

----------

ID: 3
Name: Luke Skywalker
E-Mail: use@the.force

----------
```

Iterating through the cursor works well if you don't plan to use the data later on. If you do intend to use it, it's best to just retrieve the list and then loop through that.

The *fetchall* method is very easy to work with, since you can get everything out of the cursor at once and put it all in a list, which can be used and manipulated in any way you see fit. Let's say, however, that you don't want to get all the rows that the cursor has returned all at once. In this case, the *fetchmany* method would be a better choice. It accepts a single integer that tells the cursor how many rows it needs to return. In the absence of such an integer, it simply returns one row at a time:

```
>>> cursor.execute('SELECT * FROM names')
>>> print cursor.fetchmany(2)
[(1, u'John Doe', u'jdoe@jdoe.zz'), (2, u'Mary Sue',
u'msue@msue.yy')]
```

We can then retrieve the next row when we need it:

```
>>> print cursor.fetchmany()
[(3, u'Luke Skywalker', u'use@the.force')]
```

Like the *fetchall* method, the *fetchmany* method returns a list of tuples, one for each row returned.

Finally, there's the *fetchone* method. This fetches one row from the cursor, but it differs from the *fetchmany* method in that it only returns the row's tuple. It doesn't return a list. It's useful for when you only want to return a single row, or perhaps only a single row at a time, but do not want to have the result contained in a list:

```
>>> cursor.execute('SELECT * FROM names')
>>> cursor.fetchone()
(1, u'John Doe', u'jdoe@jdoe.zz')
```

A method by the name of *next* also exists, and it works similar to *fetchone* in that it only returns one row at a time. However, instead of returning *None* when there's nothing left in the cursor, it raises a *StopIteration* error:

```
>>> cursor.execute('SELECT * FROM names')
>>> cursor.next()
(1, u'John Doe', u'jdoe@jdoe.zz')
```

```
>>> cursor.next()
(2, u'Mary Sue', u'msue@msue.yy')
>>> cursor.next()
(3, u'Luke Skywalker', u'use@the.force')
>>> cursor.next()
Traceback (most recent call last):
  File "<pyshell#201>", line 1, in −toplevel−
    cursor.next()
StopIteration
```

An interesting feature of pysqlite is the ability to easily store representations of other types of information inside of an SQLite database. For example, let's say we have a class whose instances we want to be able to store inside a database. SQLite will only hold a few types of information, but we can force instances of our class to adapt to one of SQLite's types. Then, when we want to retrieve the instance, we can convert it back.

Before we do anything, we have to close our connection and reopen it with instructions to recognize new types:

```
>>> cursor.close()
>>> connection.close()
>>> connection = sqlite.connect('test.db', detect_types =
sqlite.PARSE_DECLTYPES)
>>> cursor = connection.cursor()
```

Now, let's create a class for our new type. Continuing with names and emails, and for simplicity's sake, we'll create a *Person* class that stores a person's name and email. Note that pysqlite forces us to subclass *object* and create a new−style class:

```
>>> class Person (object):
                def __init__ (self, name, email):
                                self.name = name
                                self.email = email
```

Now, we'll create a table that stores a unique number and then an information field, which will use our new type:

```
>>> cursor.execute('CREATE TABLE people (id INTEGER PRIMARY KEY,
information Person)')
>>> cursor.commit()
```

Next, we have to create a way for our object to be converted into text, and we have to create a way to reverse the process. We can do this using two functions:

```
>>> def personAdapt (person):
                return person.name + "n" + person.email
>>> def personConvert (text):
                text = text.split("n")
                return Person(text[0], text[1])
```

The first function puts a linebreak character between the name and email, combining them into a single string. The second function splits the string between the character and then creates an object with the two resulting strings. It's very simple. We now have to register our two functions, one as an adapter and the other as a converter:

```
>>> sqlite.register_adapter(Person, personAdapt)
>>> sqlite.register_converter("Person", personConvert)
```

Now everything is set up properly. We are able to insert an instance of *Person*, and we are able to retrieve it. Let's create an instance and insert it first:

```
>>> socrates = Person('Socrates', 'socrates@ancient.gr')
>>> cursor.execute('INSERT INTO people VALUES (null, ?)',
(socrates,))
>>> connection.commit()
```

We'll now retrieve it and print out the object's information:

```
>>> cursor.execute('SELECT * FROM people')
>>> row = cursor.fetchone()
>>> print row[1].name, row[1].email
Socrates socrates@ancient.gr
```

Conclusion

SQLite is a pretty amazing database engine. Unlike other database engines which require a database server, SQLite functions as a library that uses normal files as databases. It also requires no dependencies and no configuration. This makes it extremely portable, which, combined with its small size, makes it ideal for projects of various sizes. The pysqlite library takes these benefits and allows Python developers to make use of them, allowing the benefits of both Python and SQLite to be used together.