SQLian Documentation

Release 0.1.0.dev

Tzu-ping Chung

Sep 07, 2017

Contents

1 The Basics **3**

2 Table of Contents **5**

2.1 Installation Guide 5

2.2 Quickstart 5

2.3 Inspirations 7

2.4 Indices and Tables 8

SQLian is an all-in-one library that "shephards" you through interaction with SQL. Unlike an ORM (like the ones come with Django, SQLAlchemy, etc.), it does not try to hide the fact you're writing SQL, but on the other hand still:

- Frees you from handling pesky SQL syntax oddities, so you can better debug your SQL like your Python code.
- Provides a unified interface to connect to different database implementations.
- Automatic cursor handling anda better way to interact with the data returned by the SQL database.

CHAPTER 1

The Basics

Connect to a database:

```
import sqlian
db = sqlian.connect('postgresql://...')
```

and perform a query:

```
rows = db.select(
    'name', 'occupation',
    from_='person',
    where={'main_language': 'Python'},
)
```

Now you can access the data directly:

```
>>> rows[0]
<Record {"name": "Mosky", "occupation": "Pinkoi"}>
```

or iterate over them:

```
for r in rows:
    print('{} works at {}'.format(r.name, r.occupation))
```

Interested? Read on!

CHAPTER 2

Table of Contents

Installation Guide

I highly recommend you check out Pipenv by Kenneth Reitz to handle you project dependencies. With Pipenv, you can insall SQLian simply with:

```
$ pipenv install sqlian
```

The "non-modern" way to install SQLian is from the PyPI, through Pip:

```
$ pip install sqlian
```

The source code is also available at GitHub. You can download release packages directly on it, or use Pip and Git to install the in-development snapshot:

```
$ pip install git+https://github.com/uranusjr/sqlian.git
```

Or even just clone and install manually yourself:

```
$ git clone https://github.com/uranusjr/sqlian.git
$ cd sqlian
$ python setup.py install
```

Quickstart

SQLian is composed of three main parts:

- **Connectors** connect to a database.
- **Queries** take native objects and convert them to a SQL command. The command can then be executed by the connector in the database.

- **Records** are wrappers providing a clean, nice interface to interact with the database cursor, and the data it retrieves.

Let's do a quick walk through on them one by one.

Connecting to a database

SQLian uses the 12factor-inspired database URL syntax to describe a database. This syntax is compatible with popular tools, including DJ-Database-URL, SQLAlchemy, and everything that builds on top of them. Which means, like, everything?

As an example, let's connect to a PostgreSQL database:

```
import sqlian
db = sqlian.connect('postgresql:://user:pa55@localhost/contactbook')
```

SQLian has a few connectors built-in. Some of them requires extra dependencies to actually connect to, like `psycopg2` for PostgreSQL. You can also build your own connectors if SQLian doesn't have them built-in, but we'll save that discussion for later.

The *connect()* function returns a *Connection* object, which conforms to the DB-API 2.0 specification (PEP 249), so you can get to work directly if you know your way around. But there's a better way to do it.

Making queries

Aside from the DB-API 2.0-compatible stuff, the *Connection* object also provides a rich set of "query builders" that frees you from formatting SQL yourself, and convert native Python objects more easily for SQL usage.

Let's insert some data first:

```
db.insert('person', values={
    'name': 'Mosky',
    'occupation': 'Pinkoi',
    'main_language': 'Python',
})
```

This roughly translates to:

```
INSERT INTO "person" ("name", "occupation", "main_language")
VALUES ('Mosky', 'Pinkoi', 'Python')
```

but saves you from dealing with column and value clauses and all those `%(name)s` stuff.

You can still use column name and value sequences if you have them already:

```
db.insert(
    'person',
    columns=('name', 'occupation', 'main_language'),
    values=[
        ('Tim', 'GilaCloud', 'Python'),
        ('Adam', 'Pinkoi', 'JavaScript'),
    ],
)
```

Did I mention you can insert multiple rows at one go? Yeah, you can.

It's easy to update data as well:

```
db.update('person', where={'name': 'Adam'}, set={'main_language': 'CSS'})
```

Notice the key order does not matter. Remember that time you forget a *WHERE* clause and mistakenly wipe the whole table? Put it first so you don't miss it next time.

You'd guess how deletion works by now, so let's add a little twist:

```
db.delete('person', where={'occupation !=': 'Pinkoi'})
```

The query build automatically parse trailing operators and do the right thing.

Handling results

Some queries produce data. For every query, SQLian returns an iterable object so you can handle those data.

```
>>> rows = db.select('person')
>>> rows
<RecordCollection (pending)>
```

Accessing the content in any way automatically resolve it:

```
>>> rows[0]
<Record {"name": "Mosky", "occupation": "Pinkoi", "main_language": "Python"}>
>>> rows
<RecordCollection (1+ rows, pending)>
```

```
>>> for row in rows:
...     print(row)
<Record {"name": "Mosky", "occupation": "Pinkoi", "main_language": "Python"}>
<Record {"name": "Adam", "occupation": "Pinkoi", "main_language": "CSS"}>
>>> rows
<RecordCollection (2 rows)>
```

A record can be accessed like a sequence, mapping, or even object:

```
>>> row = rows[0]
>>> row[0]
'Mosky'
>>> row['occupation']
Pinkoi
>>> row.main_language
Python
```

And in fact, it conforms completely to the Sequence and Mapping ABCs, so you can freely treat them as such, and easily convert a record to built-in Python type.

Inspirations

Few ideas in SQLian is original. Special thanks to the following projects and their contributors:

MoSQL by Mosky Liu for the idea of building SQL from Python constructs, and coming up with how most of the function calls should look like.

Records by Kenneth Reitz for the record API.

SQLAlchemy by Michael Bayer for the database connector API.

DJ-Database-URL by Kenneth Reitz for the *urlparse*-based database URL parsing logic.

Indices and Tables

- genindex
- modindex
- search