pysqlw Documentation

Release 1.3.0

plausibility

January 26, 2013

CONTENTS

1 Documentation **3**
1.1 Usage 3
1.2 pysqlw wrappers 6

pysqlw is a wrapping library for Python to make creating and executing SQL queries a breeze.

CHAPTER

ONE

DOCUMENTATION

1.1 Usage

1.1.1 Install & setup

What good is pysqlw if you can't actually use it, right? Thankfully, pysqlw is a breeze to install and setup!

pysqlw is on pypi, so you just have to `pip install pysqlw`, and pip does the rest! Then you just have to `import pysqlw`, and you're set.

Setup

Generally, you'll simply create a pysqlw object by making an instance of `pysqlw.pysqlw`. The actual things you pass to the initialisation will differ based on what type of database you want to work with.

```
import pysqlw
p = pysqlw.pysqlw(...)
```

Setup for sqlite

```
import pysqlw

# Pass in an absolute path to your sqlite3 database file.
p = pysqlw.pysqlw(db_type="sqlite", db_path="/home/user/pysqlw.db")
```

Setup for MySQL

```
import pysqlw

# Since we're using MySQL, we have to pass in relevant information; host, user, pass, database
conf = {
    "db_type": "mysql",
    "db_host": "localhost",
    "db_user": "foo",
    "db_pass": "bar",
    "db_name": "example"
}
p = pysqlw.pysqlw(**conf)
```

1.1.2 Querying your database

For reference, the `p` object we're working with is from `p = pysqlw.pysqlw(...)`. `table_name` is just a fictional table in a non-existant database. Swap it out for your tables real name.

SELECT

```
# SELECT all rows from the table
rows = p.get('table_name')

# SELECT only the first five rows
rows = p.get('table_name', 5)

# SELECT only the rows where: `foo` = 'bar'
rows = p.where('foo', 'bar').get('table_name')
```

INSERT

```
# This is the data we're going to INSERT into the table.
# Keys relate to table column names, values are what we're inserting there.
data = {
    "foo": "bar",
    "baz": "qux",
    "moo": 1
}
if p.insert('table_name', data):
    # Success!
```

UPDATE

You **must** call `where()` before you can update.

```
# k/v relates to column-name/value, as above
data = {
    "foo": "baz",
    "moo": 7
}

# UPDATE all rows where: `name` = 'John'
if p.where('name', 'John').update('table_name', data):
    # Success!

# UPDATE only a single row where: `name` = 'Pete'
if p.where('name', 'Pete').update('table_name', data, 1):
    # Success!
```

DELETE

You **must** call `where()` before you can delete.

```
# DELETE any row where: `name` = 'John'
if p.where('name', 'Pete').delete('table_name'):
    # Success!
# DELETE only a single row where: `id` = 1
if p.where('id', 1).delete('table_name'):
    # Success!
```

WHERE

This adds a WHERE query to your SQL. You can dictate what rows and columns to operate on with this. There are two ways to work with your `where()` call.

```
# On separate lines:
p.where('id', 1)
p.where('foo', 'bar')
rows = p.get('table_name')

# Chained together
rows = p.where('id', 1).where('foo', 'bar').get('table_name')
```

Affected rows

Want to know how many rows you modified with the last executed query? This will show you just that.

```
data = {
    "surname": "Smith"
}
if p.where('name', 'John').update('table_name', data):
    # Success!
    print 'Affected rows:', p.affected_rows()
```

Escape unsafe data

Due to how bound queries work, the data you pass in is actually transparently escaped for you; you don't have to do anything to be safe. If for some reason you still want to escape data, use the `escape(var)` method. It passes through your information to the database's escape method, so we can't guarantee that it's secure.

```
# OH NO, THIS PERSON HAS SOME SCARY QUOTES!
user = "Some'Dangerous'Username"
# Not today, hacker scum!
safe_user = p.escape(user)
# Now you're safe from the menaces of society.
print safe_user
```

Your own query

Warning: This doesn't do any behind-the-scenes binding, escaping, or anything of the sort. It's **your** job to keep it safe.

If for some reason, you wish to execute a manual query (joins, union selects, other query wizardry), you'll have to use the `query(q)` method. Poor you!

```
data = p.query('SELECT `this` FROM `that` UNION SELECT `this` FROM `other`')
# Data is whatever your query might return.
```

Close your connections

Warning: Once you call `p.close()`, the object is useless. There is no way to reconnect, you **have** to recreate your object. If you try and interact with it once it's been closed, all sorts of nasty errors could crop up. Don't do it!

If you know you're done with your queries, or you don't need the object anymore, you can simply call `p.close()` to kill off connections and null out left over references.

```
rows = p.get('table_name')
p.close()
```

1.1.3 Making it simpler

Some neato functions are included, just to make life with pysqlw easier.

with pysqlw as p

New in version 1.3.0. pysqlw supports the python `with` statement. This makes it much easier to visually see where you're working with a pysqlw instance.

```
with pysqlw.pysqlw(...) as p:
    rows = p.get('table_name')
    do_something_with(rows)
```

This calls the close statement and all, so you're good to go without any dead resources.

raw wrapper access

If for whatever reason you need access to the underlying database connection, or the databases' cursor, they're easy to find, under the `p.wrapper` object.

`p.wrapper.dbc` is your database connection, and `p.wrapper.cursor` is the databases' cursor. Simple, right?

```
# execute our schema: CREATE TABLE IF ...
with pysqlw.pysqlw(...) as p:
    p.wrapper.cursor.executescript(my_schema_file())
```

1.2 pysqlw wrappers

Note: These wrappers are more like *meta-wrappers*. They're just an easy way to create and connect to a wrapped database. The sqlite meta-wrapper just wraps the `sqlite3` module, the mysql wrapper just wraps (if you have it) the `MySQLdb` module.

Currently, there are only two supported meta-wrappers, and these are `sqlite` and `mysql`.

1.2.1 Wrapper structure

Meta-wrappers are fairly easy to implement yourself, they have three functions/properties. These are `required`, `connect()` and `format(item)`. They're fairly self explanatory, but will be documented regardless.

All meta-wrappers should subclass the `sqltype` class of pysqlw.

Naming

A meta-wrapper is required to be in a file called `<wrapper>w.py`, with a class by the same name inside. *e.g.*, `sqlitew.py` -> `class sqlitew` This simply allows the wrapper types to be imported dynamically at runtime, without any hardcoding or hackery.

When a script creates a pysqlw object, they pass in the `db_type`, which is the meta-wrapper name, minus the trailing `w`. You have the file `sqlitew.py`, and the inner-class `class sqlitew`, so you create the object like: `pysqlw.pysqlw(db_type="sqlite")`; easy!

required()

This property returns a list of variables the meta-wrapper needs to be passed in with the kwargs.

For example, if this returned: `['db_host', 'db_port']`, the script would have to specify: `pysqlw.pysqlw(db_type="...", db_host="..." db_port="...")`

connect()

Note: This should return a boolean value based on the success of the connection.

This connects to the database, and stores the database connection object, aswell as the database cursor, to execute queries.

This method **has** to store the database connection to `self.dbc`, and the database cursor to `self.cursor`, or else pysqlw will muck up.

format()

This returns a string value, which the database uses to bind queries to; for example sqlite uses `?`, MySQL uses `%s`

1.2.2 Wrapper example

This is the mysql meta-wrapper that pysqlw provides.

```python
from sqltype import sqltype

class mysqlw(sqltype):
    @property
    def required(self):
        return ['db_host', 'db_user', 'db_pass', 'db_name']

    def connect(self):
        try:
            import MySQLdb
            self.dbc = MySQLdb.connect(self.args.get('db_host'), self.args.get('db_user'), self.args
            self.cursor = self.dbc.cursor(MySQLdb.cursors.DictCursor)
        except Exception as e:
            return False
        return True

    def format(self, item):
        return '%s'
```

See how easy it is to meta-wrap?

Notes

- Arguments passed in are stored in the `self.args` dictionary. Use this to access params given to `pysqlw.pysqlw(...)`.