An extension to DBAPI 2.0 for easier SQL queries

Martin Blais

PyCon 2007

Introduction

DBAPI’s `Cursor.execute()` method interface is inconvenient to use.

With this work:

- Provide a simple extension that gets rid of the pitfalls
- Make it much easier to write queries
- A single pure Python module
- Support a number of DBAPI implementations
- Deals only with query writing

`Cursor.execute()` interface

General form:

```
cursor.execute(<string>, <tuple-or-map>)
```

For example:

```
cursor.execute("""
  INSERT INTO %s (%s, %s) VALUES (%%s, %%s)
  """ % ("Users", "username", "email"),
  (var_username, var_email))
```

`Cursor.execute()` interface

Be careful with escaping values, here is a common mistake:

```
cursor.execute(
  "INSERT INTO Users (username) VALUES (%s)" %
  var_username)
```

These are *also* incorrect:

```
cursor.execute(
  "INSERT INTO Users (username) VALUES ('%s')" %
  var_username)

cursor.execute(
  "INSERT INTO Users (username) VALUES (%s)" %
  repr(var_username))
```

`Cursor.execute()` interface

You *must* let DBAPI do its database-specific escaping of values:

```
cursor.execute(
  "INSERT INTO Users (username) VALUES (%s)",
  (var_username,))
```

Problems:

- Two lists of parameters is error-prone
- You have to provide a tuple or a dict
- It does not understand lists
- You can't mix positional and keyword arguments

`Cursor.execute()` interface

When you write real-world queries (instead of Mickey-mouse example presentation queries), it gets even messier:

```
cursor.execute("""
  SELECT %s FROM %s WHERE %s > %%s LIMIT %s
  """ % (','.join(columns), "Users", "age", 10)
  (18,))
```

- Because of string interpolation, you have to **double-escape** the format specifiers for the escaped values
- The parameters in the strings are in a **different order** than the function arguments (easy to make mistakes!)

New format specifier (%S)

We provide a new `execute()` method, which supports a format specifier for escaped arguments: `%S` (capital S)

```
cursor.execute_f(
  "INSERT INTO Users (username) VALUES (%S)",
  var_username)
```

You can now mix vanilla and escaped arguments:

```
cursor.execute_f(
  "INSERT INTO Users (%s) VALUES (%S)",
  "username", var_username)
```

Lists are understood

Lists are automatically joined with a comma:

```
columns = ["username", "email", "phone"]
cursor.execute_f(
  "INSERT INTO Users (%s) VALUES (...)",
  columns, ...)
```

This also works for escaped arguments:

```
values = [var_username, var_email, var_phone]
cursor.execute_f(
  "INSERT INTO Users (%s) VALUES (%S)",
  columns, values)
```

Lists are understood

Lists are automatically joined with a comma:

```
columns = ["username", "email", "phone"]
cursor.execute_f(
  "INSERT INTO Users (%s) VALUES (...)",
  columns, ...)
```

This also works for escaped arguments:

```
values = [var_username, var_email, var_phone]
cursor.execute_f(
  "INSERT INTO Users (%s) VALUES (%S)",
  columns, values)
```

Dictionaries are understood

Dictionaries are meant to be rendered appropriately for `UPDATE` statements:

- Comma-separated `<name> = <value>` pairs
- Values are DB-escaped automatically

```
values = {"id": 3,
          "brazil": "portuguese"}

cursor.execute_f("UPDATE languages SET %S",
                 values)
```

```
UPDATE languages
  SET id = 3, brazil = 'portuguese'
```

(Suggestion by D. Mertz)

Positional and Keywords Arguments

Positional and keyword arguments can be used simultaneously:

```
cursor.execute_f("""
   SELECT %(table)s FROM %s
     WHERE id = %(id)S
""", column_names, table=tablename, id=42)
```

- You can recycle arguments this way
  (i.e. a table name that occurs multiple times)

Performance and Remarks

- The extension only massages your query in a form that can be digested by DBAPI's `Cursor.execute()`
- I lied slightly in my examples, you have to use it like this:

```
    execute_f(cursor, """
        ...
```

- We cache as much of the preprocessing as possible (like `re`, `struct`)
  - You can cache your queries at load time with `qcompile()`.

Future work:

- Automatically parse the SQL query and determine which arguments should be quoted (a lot more work)

Performance and Remarks

- The extension only massages your query in a form that can be digested by DBAPI's `Cursor.execute()`
- I lied slightly in my examples, you have to use it like this:

```
    execute_f(cursor, """
        ...
```

- We cache as much of the preprocessing as possible (like `re`, `struct`)
    - You can cache your queries at load time with `qcompile()`.

Future work:

- Automatically parse the SQL query and determine which arguments should be quoted (a lot more work)

Performance and Remarks

- The extension only massages your query in a form that can be digested by DBAPI's `Cursor.execute()`
- I lied slightly in my examples, you have to use it like this:

```
execute_f(cursor, """
    ...
```

- We cache as much of the preprocessing as possible (like `re`, `struct`)
    - You can cache your queries at load time with `qcompile()`.

Future work:

- Automatically parse the SQL query and determine which arguments should be quoted (a lot more work)

Performance and Remarks

- The extension only massages your query in a form that can be digested by DBAPI's `Cursor.execute()`
- I lied slightly in my examples, you have to use it like this:

```
execute_f(cursor, """
    ...
```

- We cache as much of the preprocessing as possible (like `re`, `struct`)
    - You can cache your queries at load time with `qcompile()`.

Future work:

- Automatically parse the SQL query and determine which arguments should be quoted (a lot more work)

Questions

`dbapiext` is part of a package named `antiorm`

antiorm homepage:
`http://furius.ca/antiorm/`

Questions?