anosql Documentation

Release 1.0.0

Honza Pokorny

Dec 12, 2018

Contents

A Python library for using SQL

Inspired by the excellent Yesql library by Kris Jenkins. In my mother tongue, *ano* means *yes*.

If you are on python3.6+ or need `anosql` to work with `asyncio` based database drivers. See the related project aiosql.

CHAPTER 1

Installation

```
$ pip install anosql
```

CHAPTER 2

Usage

2.1 Basics

Given a `queries.sql` file:

```
-- name: get-all-greetings
-- Get all the greetings in the database
SELECT * FROM greetings;
```

We can issue SQL queries, like so:

```
import anosql
import psycopg2
import sqlite3

# PostgreSQL
conn = psycopg2.connect('...')
queries = anosql.from_path('queries.sql', 'psycopg2')

# Or, Sqlite3...
conn = sqlite3.connect('cool.db')
queries = anosql.from_path('queries.sql', 'sqlite3)

queries.get_all_greetings(conn)
# => [(1, 'Hi')]

queries.get_all_greetings.__doc__
# => Get all the greetings in the database

queries.get_all_greetings.sql
# => SELECT * FROM greetings;

queries.available_queries
# => ['get_all_greetings']
```

2.2 Parameters

Often, you want to change parts of the query dynamically, particularly values in the WHERE clause. You can use parameters to do this:

```
-- name: get-greetings-for-language-and-length
-- Get all the greetings in the database
SELECT *
FROM greetings
WHERE lang = %s;
```

And they become positional parameters:

```
visitor_language = "en"
queries.get_all_greetings(conn, visitor_language)
```

2.3 Named Parameters

To make queries with many parameters more understandable and maintainable, you can give the parameters names:

```
-- name: get-greetings-for-language-and-length
-- Get all the greetings in the database
SELECT *
FROM greetings
WHERE lang = :lang
AND len(greeting) <= :length_limit;
```

If you were writing a Postgresql query, you could also format the parameters as `%s(lang)` and `%s(length_limit)`.

Then, call your queries like you would any Python function with named parameters:

```
visitor_language = "en"

greetings_for_texting = queries.get_all_greetings(conn, lang=visitor_language, length_
↪limit=140)
```

CHAPTER 3

Contents

3.1 Getting Started

Below is an example of a program which can print `"{greeting}, {world_name}!"` from data held in a minimal SQLite database containing greetings and worlds.

The SQL is in a `greetings.sql` file with `-- name:` definitions on each query to tell `anosql` under which name we would like to be able to execute them. For example, the query under the name `get-all-greetings` in the example below will be available to us after loading via `anosql.from_path` as a method `get_all_greetings(conn)`. Each method on an `anosql.Queries` object accepts a database connection to use in communicating with the database.

```
-- name: get-all-greetings
-- Get all the greetings in the database
select greeting_id, greeting from greetings;

-- name: get-worlds-by-name
-- Get all the world record from the database.
select world_id,
       world_name,
       location
  from worlds
 where world_name = :world_name;
```

By specifying `db_driver="sqlite3"` we can use the python stdlib `sqlite3` driver to execute these sql queries and get the results. We're also using the `sqlite3.Row` type for our records to make it easy to access our data via their column names rather than as tuple indices.

```
import sqlite3
import anosql

queries = anosql.from_path("greetings.sql", db_driver="sqlite3")
conn = sqlite3.connect("greetings.db")
conn.row_factory = sqlite3.Row
```

(continues on next page)

(continued from previous page)

```
greetings = queries.get_greetings(conn)
worlds = queries.get_worlds_by_name(conn, world_name="Earth")
# greetings = [
#     <Row greeting_id=1, greeting="Hi">,
#     <Row greeting_id=2, greeting="Aloha">,
#     <Row greeting_id=3, greeting="Hola">
# ]
# worlds = [<Row world_id=1, world_name="Earth">]

for world_row in worlds:
    for greeting_row in greetings:
        print(f"{greeting_row['greeting']}, {world_row['world_name']}!")
# Hi, Earth!
# Aloha, Earth!
# Hola, Earth!

conn.close()
```

3.2 Defining SQL Queries

3.2.1 Query Names & Comments

Name definitions are how `anosql` determines how to name the SQL code blocks which are loaded. A query name definition is a normal SQL comment starting with "– name:" and is followed by the name of the query. You can use - or _ in your query names, but the methods in python will always be valid python names using underscores.

```
-- name: get-all-blogs
select * from blogs;
```

The above example when loaded by `anosql.from_path` will return an object with a `.get_all_blogs(conn)` method.

Your SQL comments will be added to your methods as python documentation, and accessible by calling `help()` on them.

```
-- name: get-all-blogs
-- Fetch all fields for every blog in the database.
select * from blogs;
```

```
queries = anosql.from_path("blogs.sql", "sqlite3")
help(anosql.get_all_blogs)
```

output

```
Help on function get_user_blogs in module anosql.anosql:

get_all_blogs(conn, *args, **kwargs)
    Fetch all fields for every blog in the database.
```

3.2.2 Query Operations

Adding query operator symbols to the end of query names will inform `anosql` of how to execute and return results. In the above section the `get-all-blogs` name has no special operator characters trailing it. This lack of operator is actually the most basic operator which performs SQL `select` statements and returns a list of rows. When writing an application you will often need to perform other operations besides selects, like inserts, deletes, and bulk opearations. The operators detailed in this section let you declare in your SQL, how your code should be executed by the database driver.

Insert/Update/Delete with `!`

The `!` operator will execute SQL without returning any results. It is meant for use with `insert`, `update`, and `delete` statements for which returned data is not required.

```
-- name: publish-blog!
insert into blogs(userid, title, content) values (:userid, :title, :content);

-- name: remove-blog!
-- Remove a blog from the database
delete from blogs where blogid = :blogid;
```

The methods generated are:

- `publish_blog(conn, *args, **kwargs)`
- `remove_blog(conn, *args, **kwargs)`

Each of them can be run to alter the database, but both will return `None`.

Insert Returning with `<!`

Sometimes when performing an insert it is necessary to receive some information back about the newly created database row. The `<!` operator tells anosql to perform execute the insert query, but to also expect and return some data.

In SQLite this means the `cur.lastrowid` will be returned.

```
-- name: publish-blog<!
insert into blogs(userid, title, content) values (:userid, :title, :content);
```

Will return the `blogid` of the inserted row.

PostgreSQL however allows returning multiple values via the `returning` clause of insert queries.

```
-- name: publish-blog<!
insert into blogs (
    userid,
    title,
    content
)
values (
    :userid,
    :title,
    :content
)
returning blogid, title;
```

This will insert the new blog row and return both it's `blogid` and `title` value as follows:

```
queries = anosql.from_path("blogs.sql", "psycopg2")
blogid, title = queries.publish_blog(conn, userid=1, title="Hi", content="word.")
```

Insert/Update/Delete Many with *!

The DB-API 2.0 drivers like `sqlite3` and `psycopg2` have an `executemany` method which execute a SQL command against all parameter sequences or mappings found in a sequence. This is useful for bulk updates to the database. The below example is a PostgreSQL statement to insert many blog rows.

```
-- name: bulk-publish*!
-- Insert many blogs at once
insert into blogs (
    userid,
    title,
    content,
    published
)
values (
    :userid,
    :title,
    :content,
    :published
)
```

Applying this to a list of blogs in python:

```
queries = anosql.from_path("blogs.sql", "psycopg2")
blogs = [
    {"userid": 1, "title": "First Blog", "content": "...", published: datetime(2018,␣
↪1, 1)},
    {"userid": 1, "title": "Next Blog", "content": "...", published: datetime(2018, 1,
↪ 2)},
    {"userid": 2, "title": "Hey, Hey!", "content": "...", published: datetime(2018, 7,
↪ 28)},
]
queries.bulk_publish(conn, blogs)
```

Execute SQL script statements with

Executes some sql statements as a script. These methods don't do variable substitution, or return any rows. An example usecase is using data definition statements like create table in order to setup your database.

```
-- name: create-schema#
create table users (
    userid integer not null primary key,
    username text not null,
    firstname integer not null,
    lastname text not null
);

create table blogs (
    blogid integer not null primary key,
    userid integer not null,
```

(continues on next page)

(continued from previous page)

```
    title text not null,
    content text not null,
    published date not null default CURRENT_DATE,
    foreign key(userid) references users(userid)
);
```

From code:

```
queries = anosql.from_path("create_schema.sql", "sqlite3")
queries.create_schema(conn)
```

3.3 Extending anosql

3.3.1 Driver Adapters

Database driver adapters in `anosql` are a duck-typed class which follow the below interface.:

```
class MyDbAdapter():
    def process_sql(self, name, op_type, sql):
        pass

    def select(self, conn, sql, parameters):
        pass

    @contextmanager
    def select_cursor(self, conn, sql, parameters):
        pass

    def insert_update_delete(self, conn, sql, parameters):
        pass

    def insert_update_delete_many(self, conn, sql, parameters):
        pass

    def insert_returning(self, conn, sql, parameters):
        pass

    def execute_script(self, conn, sql):
        pass


anosql.register_driver_adapter("mydb", MyDbAdapter)
```

If your adapter constructor takes arguments you can register a function which can build your adapter instance:

```
def adapter_factory():
    return MyDbAdapter("foo", 42)

anosql.register_driver_adapter("mydb", adapter_factory)
```

Looking at the source of the builtin adapters/ is a great place to start seeing how you may write your own database driver adapter.

3.4 Upgrading

3.4.1 Upgrading from 0.x to 1.x

Changed `load_queries` and `load_queries_from_string`

These methods were changed, mostly for brevity. To load `anosql` queries, you should now use the `anosql.from_str` to load queries from a SQL string, and `anosql.from_path` to load queries from a SQL file, or directory of SQL files.

Removed the `$` "record" operator

Because most database drivers have more efficient, robust, and featureful ways of controlling the rows and records output, this feature was removed.

See:

- sqlite.Row
- psycopg2 - Connection and Cursor subclasses

SQLite example:

```
conn = sqlite3.connect("...")
conn.row_factory = sqlite3.Row
actual = queries.get_all_users(conn)

assert actual[0]["userid"] == 1
assert actual[0]["username"] == "bobsmith"
assert actual[0][2] == "Bob"
assert actual[0]["lastname" == "Smith"
```

PostgreSQL example:

```
with psycopg2.connect("...", cursor_factory=psycopg2.extras.RealDictCursor) as conn:
    actual = queries.get_all_users(conn)

assert actual[0] == {
    "userid": 1,
    "username": "bobsmith",
    "firstname": "Bob",
    "lastname": "Smith",
}
```

Driver Adapter classes instead of QueryLoader

I'm not aware of anyone who actually has made or distributed an extension for `anosql`, as it was only available in its current form for a few weeks. So this notice is really just for completeness.

For `0.3.x` versions of `anosql` in order to add a new database extensions you had to build a subclass of `anosql.QueryLoader`. This base class is no longer available, and driver adapters no longer have to extend from any class at all. They are duck-typed classes which are expected to adhere to a standard interface. For more information about this see *Extending anosql*.

3.4.2 New Things

Use the database driver `cursor` directly

All the queries with a *SELECT* type have a duplicate method suffixed by *_cursor* which is a context manager to the database cursor. So *get_all_blogs(conn)* can also be used as:

```
rows = queries.get_all_blogs(conn)
# [(1, "My Blog", "yadayada"), ...]

with queries.get_all_blogs_cursor(conn) as cur:
    # All the power of the underlying cursor object! Not limited to just a list of␣
↪rows.
    for row in cur:
        print(row)
```

New operator types for runnings scripts `#` and bulk-inserts `*!`

See *Query Operations*

3.5 anosql

3.5.1 anosql package

Subpackages

anosql.adapters package

Submodules

anosql.adapters.psycopg2 module

anosql.adapters.sqlite3 module

Module contents

Submodules

anosql.core module

anosql.exceptions module

anosql.patterns module

Module contents

CHAPTER 4

License

BSD, short and sweet