Postgres.py Documentation

Release 1.1.0

Gittip, LLC

December 18, 2013

Contents

postgres is a high-value abstraction over psycopg2.

Installation

postgres is available on GitHub and on PyPI:

```
$ pip install postgres
```

We test against Python 2.6, 2.7, 3.2, and 3.3. We don't yet have a testing matrix for different versions of psycopg2 or PostgreSQL.

postgres is in the public domain.

Tutorial

Instantiate a `Postgres` object when your application starts:

```
>>> from postgres import Postgres
>>> db = Postgres("postgres://jrandom@localhost/testdb")
```

Use `run` to run SQL statements:

```
>>> db.run("CREATE TABLE foo (bar text)")
>>> db.run("INSERT INTO foo VALUES ('baz')")
>>> db.run("INSERT INTO foo VALUES ('buz')")
```

Use `all` to fetch all results:

```
>>> db.all("SELECT * FROM foo ORDER BY bar")
[{'bar': 'baz'}, {'bar': 'buz'}]
```

Use `one_or_zero` to fetch one result or `None`:

```
>>> db.one_or_zero("SELECT * FROM foo WHERE bar='baz'")
{'bar': 'baz'}
>>> db.one_or_zero("SELECT * FROM foo WHERE bar='blam'")
```

2.1 Bind Parameters

In case you're not familiar with bind parameters in DB-API 2.0, the basic idea is that you put `%(foo)s` in your SQL strings, and then pass in a second argument, a `dict`, containing parameters that `psycopg2` (as an implementation of DB-API 2.0) will bind to the query in a way that is safe against SQL injection. (This is inspired by old-style Python string formatting, but it is not the same.)

```
>>> db.one("SELECT * FROM foo WHERE bar=%(bar)s", {"bar": "baz"})
{'bar': 'baz'}
```

Never build SQL strings out of user input!

Always pass user input as bind parameters!

2.2 Context Managers

Eighty percent of your database usage should be covered by the simple run, all, one_or_zero API introduced above. For the other 20%, postgres provides context managers for working at increasingly lower levels of abstraction. The lowest level of abstraction in postgres is a psycopg2 connection pool that we configure and manage for you. Everything in postgres, both the simple API and the context managers, uses this connection pool.

Here's how to work directly with a psycogpg2 cursor while still taking advantage of connection pooling:

```
>>> with db.get_cursor() as cursor:
...     cursor.execute("SELECT * FROM foo ORDER BY bar")
...     cursor.fetchall()
...
[{'bar': 'baz'}, {'bar': 'buz'}]
```

A cursor you get from get_cursor has autocommit turned on for its connection, so every call you make using such a cursor will be isolated in a separate transaction. Need to include multiple calls in a single transaction? Use the get_transaction context manager:

```
>>> with db.get_transaction() as txn:
...     txn.execute("INSERT INTO foo VALUES ('blam')")
...     txn.execute("SELECT * FROM foo ORDER BY bar")
...     txn.fetchall()
...
[{'bar': 'baz'}, {'bar': 'blam'}, {'bar': 'buz'}]
```

Note that other calls won't see the changes on your transaction until the end of your code block, when the context manager commits the transaction for you:

```
>>> with db.get_transaction() as txn:
...     txn.execute("INSERT INTO foo VALUES ('blam')")
...     db.all("SELECT * FROM foo ORDER BY bar")
...
[{'bar': 'baz'}, {'bar': 'buz'}]
>>> db.all("SELECT * FROM foo ORDER BY bar")
[{'bar': 'baz'}, {'bar': 'blam'}, {'bar': 'buz'}]
```

The get_transaction manager gives you a cursor with autocommit turned off on its connection. If the block under management raises an exception, the connection is rolled back. Otherwise it's committed. Use this when you want a series of statements to be part of one transaction, but you don't need fine-grained control over the transaction. For fine-grained control, use get_connection to get a connection straight from the connection pool:

```
>>> with db.get_connection() as connection:
...     cursor = connection.cursor()
...     cursor.execute("SELECT * FROM foo ORDER BY bar")
...     cursor.fetchall()
...
[{'bar': 'baz'}, {'bar': 'buz'}]
```

A connection gotten in this way will have autocommit turned off, and it'll never be implicitly committed otherwise. It'll actually be rolled back when you're done with it, so it's up to you to explicitly commit as needed. This is the lowest-level abstraction that postgres provides, basically just a pre-configured connection pool from psycopg2.

CHAPTER 3

API

class postgres.**Postgres**(*url*, *minconn=1*, *maxconn=10*, *cursor_factory=<class 'psycopg2.extras.RealDictCursor'>*, *strict_one=None*)

Interact with a PostgreSQL database.

Parameters

- **url** (*unicode*) – A postgres:// URL or a PostgreSQL connection string
- **minconn** (*int*) – The minimum size of the connection pool
- **maxconn** (*int*) – The maximum size of the connection pool
- **cursor_factory** – Defaults to RealDictCursor
- **strict_one** (bool) – The default strict parameter for one

This is the main object that postgres provides, and you should have one instance per process for each PostgreSQL database your process wants to talk to using this library.

```
>>> import postgres
>>> db = postgres.Postgres("postgres://jrandom@localhost/test")
```

(Note that importing postgres under Python 2 will cause the registration of typecasters with psycopg2 to ensure that you get unicode instead of bytestrings for text data, according to this advice.)

When instantiated, this object creates a thread-safe connection pool, which opens minconn connections immediately, and up to maxconn according to demand. The fundamental value of a Postgres instance is that it runs everything through its connection pool.

Check the psycopg2 docs for additional cursor_factories, such as NamedTupleCursor.

The names in our simple API, run, all, and one_or_zero, were chosen to be short and memorable, and to not conflict with the DB-API 2.0 execute, fetchone, and fetchall methods, which have slightly different semantics (under DB-API 2.0 you call execute on a cursor and then call one of the fetch* methods on the same cursor to retrieve rows; with our simple API there is no second fetch step). See this ticket for more of the rationale behind these names. The context managers on this class are named starting with get_ to set them apart from the simple-case API. Note that when working inside a block under one of the context managers, you're using DB-API 2.0 (execute + fetch*), not our simple API (run / one / all).

run(*sql*, *parameters=None*)

Execute a query and discard any results.

Parameters

- **sql** (*unicode*) – the SQL statement to execute
- **parameters** (*dict or tuple*) – the bind parameters for the SQL statement

Returns `None`

```
>>> db.run("CREATE TABLE foo (bar text)")
>>> db.run("INSERT INTO foo VALUES (’baz’)")
>>> db.run("INSERT INTO foo VALUES (’buz’)")
```

`all`(*sql*, *parameters=None*)

Execute a query and return all results.

Parameters

- **sql** (*unicode*) – the SQL statement to execute
- **parameters** (*dict or tuple*) – the bind parameters for the SQL statement

Returns `list` of rows

```
>>> for row in db.all("SELECT bar FROM foo"):
...     print(row["bar"])
...
baz
buz
```

`one_or_zero`(*sql*, *parameters=None*)

Execute a query and return a single result or `None`.

Parameters

- **sql** (*unicode*) – the SQL statement to execute
- **parameters** (*dict or tuple*) – the bind parameters for the SQL statement

Returns a single row or `None`

Raises `TooFew` or `TooMany`

Use this for the common case where there should only be one record, but it may not exist yet.

```
>>> row = db.one_or_zero("SELECT * FROM foo WHERE bar=’blam’")
>>> if row is None:
...     print("No blam yet.")
...
No blam yet.
```

`one`(*sql*, *parameters=None*, *strict=None*)

Execute a query and return a single result.

Parameters

- **sql** (*unicode*) – the SQL statement to execute
- **parameters** (*dict or tuple*) – the bind parameters for the SQL statement
- **strict** (`bool`) – whether to raise when there isn’t exactly one result

Returns a single row or `None`

Raises `TooFew` or `TooMany`

By default, `strict` ends up evaluating to `True`, in which case we raise `postgres.TooFew` or `postgres.TooMany` if the number of rows returned isn’t exactly one (both are subclasses of `postgres.OutOfBounds`). You can override this behavior per-call with the `strict` argument here, or globally by passing `strict_one` to the `Postgres` constructor. If you use both, the `strict` argument here wins. If you pass `False` for `strict`, then we return `None` if there are no results, and the first if there is more than one.

```
>>> row = db.one("SELECT * FROM foo WHERE bar='baz'")
>>> print(row["bar"])
baz
```

get_cursor(**a*, ***kw*)

Return a `CursorContextManager` that uses our connection pool.

This gets you a cursor with `autocommit` turned on on its connection. The context manager closes the cursor when the block ends.

Use this when you want a simple cursor.

```
>>> with db.get_cursor() as cursor:
...     cursor.execute("SELECT * FROM foo")
...     cursor.rowcount
...
2
```

get_transaction(**a*, ***kw*)

Return a `TransactionContextManager` that uses our connection pool.

This gets you a cursor with `autocommit` turned off on its connection. If your code block inside the `with` statement raises an exception, the transaction will be rolled back. Otherwise, it'll be committed. The context manager closes the cursor when the block ends.

Use this when you want a series of statements to be part of one transaction, but you don't need fine-grained control over the transaction.

```
>>> with db.get_transaction() as txn:
...     txn.execute("SELECT * FROM foo")
...     txn.fetchall()
...
[{'bar': 'baz'}, {'bar': 'buz'}]
```

get_connection()

Return a `ConnectionContextManager` that uses our connection pool.

Use this when you want to take advantage of connection pooling, but otherwise need full control, for example, to do complex things with transactions.

```
>>> with db.get_connection() as connection:
...     cursor = connection.cursor()
...     cursor.execute("SELECT * FROM foo")
...     cursor.fetchall()
...
[{'bar': 'baz'}, {'bar': 'buz'}]
```

class `postgres.`**Connection**(**a*, ***kw*)

This is a subclass of `psycopg2.extensions.connection`.

`Postgres` uses this class as the `connection_factory` for its connection pool. Here are the differences from the base class:

- We set `autocommit` to `True`.
- We set the client encoding to `UTF-8`.
- We use `self.cursor_factory`.

class `postgres.`**`CursorContextManager`**(*pool*, **a*, ***kw*)

Instantiated once per `get_cursor` call.

The return value of `CursorContextManager.__enter__` is a `psycopg2.extras.RealDictCursor`. Any positional and keyword arguments to our constructor are passed through to the cursor constructor. The `Connection` underlying the cursor is checked out of the connection pool when the block starts, and checked back in when the block ends. Also when the block ends, the cursor is closed.

class `postgres.`**`TransactionContextManager`**(*pool*, **a*, ***kw*)

Instantiated once per `get_transaction` call.

The return value of `TransactionContextManager.__enter__` is a `psycopg2.extras.RealDictCursor`. Any positional and keyword arguments to our constructor are passed through to the cursor constructor. When the block starts, the `Connection` underlying the cursor is checked out of the connection pool and `autocommit` is set to `False`. If the block raises an exception, the `Connection` is rolled back. Otherwise it's committed. In either case, the cursor is closed, `autocommit` is restored to `True`, and the `Connection` is put back in the pool.

class `postgres.`**`ConnectionContextManager`**(*pool*)

Instantiated once per `get_connection` call.

The return value of `ConnectionContextManager.__enter__` is a `postgres.Connection`. When the block starts, a `Connection` is checked out of the connection pool and `autocommit` is set to `False`. When the block ends, `autocommit` is restored to `True` and the `Connection` is rolled back before being put back in the pool.

Python Module Index