pglib Documentation

Release 2.4.0

Michael Kleehammer

Jan 20, 2021

Contents

pglib is a Python 3.5+ module for working with PostgreSQL databases. It is a C extension that exposes the libpq API. It is designed to be small, fast, and as convenient as possible. It provides both synchronous and asynchronous APIs.

Unlike some libraries, it never modifies the SQL you pass. Parameters are passed using the official libpq protocol for parameters.

CHAPTER 1

Quick Start

1.1 Connecting

To connect, pass a libpq connection string to the `connect()` function.

```
import pglib
cnxn = pglib.connect('host=localhost dbname=test')
```

1.2 Basic Selecting

`Connection.execute` accepts a SQL statement and optional parameters. What it returns depends on the kind of SQL statement:

- A select statement will return `ResultSet` with all of the rows.
- An insert, update, or delete statement will return the number of rows modified.
- Any other statement (e.g. "create table") will return None.

If the SQL was a select statement, `ResultSet.columns` will be a tuple containing the column names selected.

The Row objects can be accessed by indexing into the ResultSet or iterating over it.

```
rset = cnxn.execute("select id, name from users")

print('columns:', rset.columns) # ('id', 'name')

print('count:', len(rset))
print('first:', rset[0])

for row in rset:
    print(row)
```

The PostgreSQL client library, libpq, stores all row data in memory while the ResultSet exists. This means that result sets can be iterated over multiple times, but it also means large result sets use a lot of memory and should be discarded as soon as possible.

Row objects are similar to tuples, but they also allow access to columns by name.

```
rset = cnxn.execute("select id, name from users limit 1")
row = rset[0]
print('id:', row[0]) # access id by column index
print('id:', row.id) # access id by column name
```

The SQL 'as' keyword makes it easy to set the name:

```
rset = cnxn.execute("select count(*) as cnt from users")
row = rset[0]
print(row.cnt)
```

The Connection.fetchrow method is a convenience method that returns the first result Row. If there are no results it returns None.

```
row = cnxn.fetchrow("select count(*) as cnt from users")
print(row.cnt)
```

The Connection.fetchval method, another convenience method, returns the first column of the first row. If there are no results it returns None.

```
count = cnxn.fetchval("select count(*) from users")
print(count)
```

Each row is a Python sequence, so it can be used in many places that a tuple or list can. To convert the values into a tuple use *tuple(row)*.

Finally, to make it convenient to pass a Row around to functions, the columns are also available from the row object. Note that a column actually named 'columns' will override this.

```
print('columns:', row.columns)

# Convert to dictionary:
d = dict(zip(row.columns, row))
```

1.3 Parameters

PostgreSQL supports parameters using $1, $2, etc. as a place holder in the SQL. Values for these are passed after the SQL. The first parameter passed is used for $1, the second for $2, etc.

```
cnxn.execute("""
             select id, name
               from users
              where id > $1
                    and bill_overdue = $2
             """, 100, 1)  # 100 -> $1, 1 -> $2
```

CHAPTER 2

Asynchronous Quick Start

Most of the API is the same as the synchronous one, but any that communicate with the server require the *await* keyword.

2.1 Connecting

To connect, pass a libpq connection string to the `async_connect()` function.

```
import asyncio
import pglib
cnxn = await pglib.connect_async('host=localhost dbname=test')
```

2.2 Selecting

There are asynchronous versions of *execute*, *fetchrow*, and *fetchval*:

```
rset  = await cnxn.execute("select id, name from users")
row   = await cnxn.fetchrow("select count(\*) as cnt from users")
count = await cnxn.fetchval("select count(\*) from users")
```

The ResultSet and Row objects don't require `await`.

CHAPTER 3

Building

Before you build, you may want to try *pip install pglib* to see if pre-build binaries are already available. Binary wheels are provided for OS X.

Otherwise you'll need:

- Python 3.3 or greater
- the pglib source
- the compiler Python was built with
- PostgreSQL header files and lib files

Once these installed and paths are configured correctly, building is (supposed to be) as simple as running `python3 setup.py build` in the pglib source directory.

3.1 OS/X

Install PostgreSQL. The Postgres.app installation is particularly convenient.

Ensure that `pg_config` is in the path so setup.py can use it to determine the include and lib directories. If you are using Postgres.app, you can add the following to your ~/.bashrc file:

```
export PATH=$PATH:/Applications/Postgres.app/Contents/MacOS/bin
```

You will also need Xcode installed *and* you'll need to download the Xcode command line tools. Some versions of Xcode provide a Downloads tab on the preferences. For later versions, run *xcode-select –install* on the command line.

The setup script will use the latest SDK directory to find the header and library files.

Once you have built pglib, you can install it into your global Python site directory using `sudo python3 setup.py install` or into your venv directory using `python3 setup.py install`.

3.2 Linux

You will need some header files and pg_config. The binary package names differ by distribution.

Component	RedHat	Debian
Python headers and libs	python-devel	python-dev
PostreSQL headers and libs	postgresql-devel	postgres-dev

Make sure pg_config is in your path before building. On RedHat-based systems, you'll find it in /usr/pgsql-*version*/bin.

Once you have the necessary files and a compiler installed, download the pglib source and run `python3 setup.py build` in the pglib source directory. Install into your site directory using `sudo python3 setup.py install` or into your venv directory using `python3 setup.py install`.

3.3 Windows

This library has only been tested with 64-bit Python, not 32-bit Python on 64-bit Windows. It *may* work but it won't be tested until someone requests it.

There are two complications on Windows not present on the other operating systems:

- The PostgreSQL header files have an old copy of the Python header files.
- The resulting library uses libpq.dll which must be in the path to import.

First you will need Visual Studio 10 if building for Python 3.3. The setup.py scripts always look for the version of Visual Studio they were built with.

The PostgreSQL files can come from an installed version or by downloading a zipped version. There will be 3 directories you will need: pgsql\bin, pgsql\lib, and pgsql\include.

Unfortunately, as of PostgreSQL 9.3, the PostgreSQL download includes the Python 2.7 header files. If these are used you will get lots of errors about undefined items. To fix this, you must manually force setup.py to include the Python 3.3 include files *before* including the PostgreSQL include files. The Python header files are in the 'include' directory under the Python installation.

To do this, create a setup.cfg file in the pglib directory and configure it with your directories:

```
[build_ext]
include_dirs=c:\bin\python33-64\include;c:\bin\pgsql\include
library-dirs=c:\bin\pgsql\lib
```

It is very important that the Python include directory is before the pgsql include directory.

Once this is done, make sure Python 3.3 is in your path and run: `python setup.py build install`. If successful, a pglib.pyd file will have been created.

Since pglib dynamically links to libpq.dll, you will need the DLL in your path and the DLLs that it needs. This means you will need the files from both pgsql\lib and pgsql\bin in your path.

CHAPTER 4

Tips

4.1 Where X In Parameters

You should never embed user-provided data in a SQL statement and most of the time you can use a simple parameter. To perform a query like "where x in $1" use the following form:

```
rset = cnxn.execute("select * from t where id = any($1)", [1,2,3])
```

CHAPTER 5

API

5.1 pglib

pglib.**version**

The pglib module version as a string, such as "5.2.0".

pglib.**hstore**

Wrap a dictionary with this type to pass as an hstore parameter. A raw dictionary is assumed to be JSON:

```
cnxn.execute("create table h(hcol hstore)")

value = {'one': 1, 'two': 2}
cnxn.execute("insert into h($1)", value)          # <-- WRONG - this passes JSON
cnxn.execute("insert into h($1)", hstore(value))  # <-- OK
```

Note that reading hstore data will return a raw dictionary:

```
result = cnxn.fetchval("select hcol from h limit 1")
# result is {'one': 1, 'two': 2}
```

connect(conninfo : string) --> Connection

Accepts a connection string and returns a new *Connection*. Raises an *Error* if an error occurs.

```
cnxn = pglib.connect('host=localhost dbname=test')
```

connect_async(conninfo : string) --> Connection

A coroutine that accepts a connection string and returns a new asynchronous *Connection*. Raises an *Error* if an error occurs.

```
cnxn = yield from pglib.connect_async('host=localhost dbname=test')
```

defaults() --> dict

Returns a dictionary of default connection string values.

PQTRANS_*

Constants returned by *Connection.transaction_status()*:

- PQTRANS_ACTIVE
- PQTRANS_INTRANS
- PQTRANS_INERROR
- PQTRANS_UNKNOWN

5.2 Connection

class pglib.**Connection**

Represents a connection to the database. Internally this wraps a `PGconn*`. The database connection is closed when the Connection object is destroyed.

`Connection.`**client_encoding**
The client encoding as a string such as "UTF8".

`Connection.`**pid**
The integer backend PID of the connection.

`Connection.`**protocol_version**
An integer representing the protocol version returned by PQprotocolVersion.

`Connection.`**server_version**
An integer representing the server version returned by PQserverVersion.

`Connection.`**server_encoding**
The server encoding as a string such as "UTF8".

`Connection.`**status**
True if the connection is valid and False otherwise.

Accessing this property calls PQstatus and returns True if the status is CONNECTION_OK and False otherwise. Note that this returns the last status of the connection but does not actually test the connection. If you are caching connections, consider executing something like 'select 1;' to test an old connection.

`Connection.`**transaction_status**
Returns the current in-transaction status of the server via PQtransactionStatus as one of PQTRANS_IDLE, PQTRANS_ACTIVE, PQTRANS_INTRANS, PQTRANS_INERROR, or PQTRANS_UNKNOWN.

`Connection.`**copy_from_csv**(*table: str*, *source*, *header=False*) → int
Copies a CSV file to the given table. Returns the number of rows copied.

`table` is the name of the table to copy to. You can also provide the columns to be populated:

```
count = cnxn.copy_from_csv("t1(b, a)", '"one",1\n"two",2')
```

The `source` can be a file-like object (not a filename) or the actual text of a CSV.

The `header` parameter is a flag. Pass True if your CSV has a header.

Connection.execute(sql [, param, ...]) --> ResultSet | int | None
Submits a command to the server and waits for the result. If the connection is asynchronous, you must use `await` with this method.

If the command returns rows, such as a selecto statement or one using the returning keyword, the result will be a *ResultSet*:

```
rset = cnxn.execute(
           """
           select id, name
             from users
            where id > $1
                  and bill_overdue = $2
           """, 100, 1)  # 100 -> $1, 1 -> $2
for row in rset:
    print('user id=', row.id, 'name=', row.name)
```

If the command is an UPDATE or DELETE statement, the result is the number of rows affected:

```
count = cnxn.execute("delete from articles where expired <= now()")
print('Articles deleted:', count)
```

Otherwise, None is returned.

```
cnxn.execute("create table t1(a int)") # returns None
```

Parameters may be passed as arguments after the SQL statement. Use $1, $2, etc. as markers for these in the SQL. Parameters must be Python types that pglib can convert to appropriate SQL types. See *Parameter Types*.

Parameters are always passed to the server separately from the SQL statement using PQexecParams and pglib *never* modifies the SQL passed to it. You should *always* pass parameters separately to protect against SQL injection attacks.

Connection.listen(channel [, channel, ...]) --> None

Executes a LISTEN command for each channel.

This is only available for asynchronous connections.

Connection.notify(channel [, payload]) --> None

A convenience method that issues a NOTIFY command using "select pg_notify(channel, payload)".

Note that `pg_notify` does *not* lowercase the channel name but executing the NOTIFY command via SQL will unless you put the channel name in double quotes. For example `cnxn.execute('NOTIFY TESTING')` will actually use the channel "testing" but both `cnxn.execute('NOTIFY "TESTING"')` and `cnxn.notify('TESTING')` will use the channel "TESTING".

Connection.notifications(timeout=None) --> (channel, payload) | None

Returns a list of notifications. Each notification is a tuple containing (channel, payload).

To use this, first issue one or more LISTEN statements: `cnxn.execute('LISTEN channel')`. Note that if you don't put the channel name in double quotes it will be lowercased by the server.

Notifications will always contain two elements and the PostgreSQL documentation seems to indicate the payload will be an empty string and never None (NULL), but I have not confirmed this.

Connection.fetchall(sql [, param, ...]) --> ResultSet

Executes the SQL and returns a result set. This is identical to *execute* except it will raise an error if the SQL does not return results:

```
rset = cnxn.fetchall("select name from users")
for row in rset:
    print('name:', row.name)
```

Connection.fetchrow(sql [, param, ...]) --> Row | None

A convenience method that submits a command and returns the first row of the result. If the result has no rows, None is returned. If the connection is asynchronous, you must use `await` with this method.:

```
row = cnxn.fetchrow("select name from users where id = $1", userid)
if row:
    print('name:', row.name)
else:
    print('There is no user with this id', userid)
```

Connection.fetchval(sql [, param, ...]) --> value

A convenience method that submits a command and returns the first column of the first row of the result. If there are no rows, None is returned. If the connection is asynchronous, you must use `await` with this method.

```
name = cnxn.fetchval("select name from users where id = $1", userid)
if name:
    print('name:', name)
else:
    print('There is no user with this id', userid)
```

Connection.fetchvals(sql, [, param, ...]) --> List[object]

A convenience method that submits a command and returns the first column of every row of the result. If there are no rows, an empty list is returned.

names = cnxn.fetchval("select name from users")

5.3 ResultSet

class pglib.**ResultSet**

Holds the results of a select statement: the column names and a collection of *Row* objects. ResultSets behave as simple sequences, so the number of rows it contains can be determined using `len(rset)` and individual rows can be accessed by index: `row = rset[3]`.

ResultSets can also be iterated over:

```
rset = cnxn.execute("select user_id, user_name from users")
for row in rset:
    print(row)
```

A ResultSet is a wrapper around a `PGresult` pointer and contains data for *all* of the rows selected in PostgreSQL's raw, binary format. Iterating over the rows converts the raw data into Python objects and returns them as *Row* objects, but does not "use up" the raw data. The `PGresult` memory is not freed until the ResultSet is freed.

ResultSet.**columns**

The column names from the select statement. Each *Row* from the result set will have one element for each column.

ResultSet.**colinfos**

A tuple of column information structures. Each element is a named tuple with the following fields:

name The column name.

type The OID (int) of the column.

mod The type modifier (int). See PQfmod. This is often the precision of a field. It is usually -1 for variable length field types like "text".

size The size of the column in bytes.

5.4 Row

class pglib.**Row**

Row objects are sequence objects that hold query results. All rows from the same result set will have the same number of columns, one for each column in the result set's `columns` attribute. Values are converted from PostgreSQL's raw format to Python objects as they are accessed. See *Result Types*.

Rows are similar to tuples; `len` returns the number of columns and they can be indexed into and iterated over:

```
row = rset[0]
print('col count:', len(row))
print('first col:', row[0])
for index, value in enumerate(row):
    print('value', index, 'is', value)
```

Columns can also be accessed by name. (Non-alphanumeric characters are replaced with an underscore.) Use the SQL *as* keyword to change a column's name

```
rset = cnxn.execute("select cust_id, cust_name from cust limit 1")
row = rset[0]
print(row.cust_id, row.cust_name)

rset = cnxn.execute("select count(*) as total from cust")
print(rset[0].total)
```

Unlike tuples, Row values can be replaced. This is particularly handy for "fixing up" values after fetching them.

```
row.ctime = row.ctime.replace(tzinfo=timezone)
```

Row.**columns**

A tuple of column names in the Row, shared with the ResultSet that the Row is from.

If you select a column actually named "columns", the column will override this attribute.

To create a dictionary of column names and values, use zip:

```
obj = dict(zip(row.columns, row))
```

5.5 Error

class pglib.**Error**

The error class raised for all errors.

Errors generated by pglib itself are rare, but only contain a message.

Errors reported by the database will contain a message with the format "[sqlstate] database message" and the following attributes:

attribute	libpq field code
severity	PG_DIAG_SEVERITY
sqlstate	PG_DIAG_SQLSTATE
detail	PG_DIAG_MESSAGE_DETAIL
hint	PG_DIAG_MESSAGE_HINT
position	PG_DIAG_STATEMENT_POSITION
internal_position	PG_DIAG_INTERNAL_POSITION
internal_query	PG_DIAG_INTERNAL_QUERY
context	PG_DIAG_CONTEXT
file	PG_DIAG_SOURCE_FILE
line	PG_DIAG_SOURCE_LINE
function	PG_DIAG_SOURCE_FUNCTION

The most most useful attribute for processing errors is usually the SQLSTATE.

CHAPTER 6

Data Types

Right now there is a limited set of data types and text is always encoded as UTF-8. Feel free to open an issue to request new ones.

6.1 Parameter Types

Parameters of the following types are accepted:

Python Type	SQL Type
None	NULL
bool	boolean
bytes	bytea
bytearray	bytea
datetime.date	date
datetime.datetime	timestamp
datetime.time	time
datetime.timedelta	interval
decimal.Decimal	numeric
float	float8
int	int64 or numeric
str	text (UTF-8)
uuid.UUID	uuid
tuple<int>, list<int>	int[]
tuple<str>, list<str>	str[]
tuple<date>, list<date>	date[]
hstore(dict)	hstore
dict	json

Arrays can only contain one type, so tuples and lists must contain elements of all of the same type. Only strings and integers are supported at this time. Note that a list or tuple can contain None, but it must contain at least one string or integer so the type of array can be determined.

To use hstore, you must first tell pglib what OID was assigned to hstore.

6.2 Result Types

The following data types are recognized in results:

SQL Type	Python Type
NULL	None
boolean	bool
bytea	bytes
char, varchar, text	str (UTF-8)
float4, float8	float
smallint, integer, bigint	int
money	decimal.Decimal
numeric	decimal.Decimal
date	datetime.date
time	datetime.time
timestamp / timestamptz	datetime.datetime
uuid	uuid.UUID
int[]	list<int>
text[]	list<str>
date[]	list<date>
hstore	dict
json & jsonb	dict

Python's `timedelta` only stores days, seconds, and microseconds internally, so intervals with year and month are not supported.

Python Module Index

p

Index