PyGreSQL Documentation

Release 4.2

The PyGreSQL team

Nov 24, 2017

Contents

CHAPTER 1

About PyGreSQL

PyGreSQL is an *open-source* Python module that interfaces to a PostgreSQL database. It embeds the PostgreSQL query library to allow easy use of the powerful PostgreSQL features from a Python script.

PostgreSQL is a highly scalable, SQL compliant, open source object-relational database management system. With more than 20 years of development history, it is quickly becoming the de facto database for enterprise level open source solutions. Best of all, PostgreSQL's source code is available under the most liberal open source license: the BSD license.

Python Python is an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme or Java. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface. The Python implementation is copyrighted but freely usable and distributable, even for commercial use.

PyGreSQL is a Python module that interfaces to a PostgreSQL database. It embeds the PostgreSQL query library to allow easy use of the powerful PostgreSQL features from a Python script.

PyGreSQL is developed and tested on a NetBSD system, but it also runs on most other platforms where PostgreSQL and Python is running. It is based on the PyGres95 code written by Pascal Andre (andre@chimay.via.ecp.fr). D'Arcy (darcy@druid.net) renamed it to PyGreSQL starting with version 2.0 and serves as the "BDFL" of PyGreSQL.

The current version PyGreSQL 4.2 needs PostgreSQL 8.3 or newer and Python 2.5 to 2.7. If you are using Python 3.x, you will need PyGreSQL 5.0 or newer.

CHAPTER 2

Copyright notice

Written by D'Arcy J.M. Cain (darcy@druid.net)

Based heavily on code written by Pascal Andre (andre@chimay.via.ecp.fr)

CHAPTER 3

PyGreSQL Announcements

3.1 Release of PyGreSQL version 4.2

Release 4.2 of PyGreSQL.

It is available at: http://pygresql.org/files/PyGreSQL-4.2.tgz.

If you are running NetBSD, look in the packages directory under databases. There is also a package in the FreeBSD ports collection.

Please refer to changelog.txt for things that have changed in this version.

Please refer to readme.txt for general information.

This version has been built and unit tested on:

- NetBSD
- FreeBSD
- openSUSE
- Ubuntu
- Windows 7 with both MinGW and Visual Studio
- PostgreSQL 8.4 and 9.3 64bit
- Python 2.4, 2.5, 2.6 and 2.7 32 and 64bit

D'Arcy J.M. Cain
darcy@PyGreSQL.org
and the PyGreSQL team

CHAPTER 4

Download information

4.1 Current PyGreSQL versions

You can find PyGreSQL on the Python Package Index at

- http://pypi.python.org/pypi/PyGreSQL/

The released version of the source code is available at

- http://pygresql.org/files/PyGreSQL.tar.gz

You can also check the latest pre-release version at

- http://pygresql.org/files/PyGreSQL-beta.tar.gz

A Linux RPM can be picked up from

- http://pygresql.org/files/pygresql.i386.rpm

A NetBSD package is available in their pkgsrc collection

- ftp://ftp.netbsd.org/pub/NetBSD/packages/pkgsrc/databases/py-postgresql/README.html

A FreeBSD package is available in their ports collection

- http://www.freebsd.org/cgi/cvsweb.cgi/ports/databases/py-PyGreSQL/

An openSUSE package is available through their build service at

- https://software.opensuse.org/package/PyGreSQL?search_term=pygresql

A Win32 installer for Python 2.6 and 2.7 is available at

- http://pygresql.org/files/PyGreSQL-4.2.2.win-amd64-py2.6.exe
- http://pygresql.org/files/PyGreSQL-4.2.2.win-amd64-py2.7.exe

4.2 Older PyGreSQL versions

You can look for older PyGreSQL versions at

- http://pygresql.org/files/

4.3 News, Changes and Future Development

See the *PyGreSQL Announcements* for current news.

For a list of all changes in the current version 4.2 and in past versions, have a look at the *ChangeLog*.

The section on *PyGreSQL Development and Support* lists ideas for future developments and ways to participate.

4.4 Installation

Please read the chapter on *Installation* in our documentation.

4.5 Distribution files

pgmodule.c	the C Python module (_pg)
pgfs.h	PostgreSQL definitions for large objects
pgtypes.h	PostgreSQL type definitions
pg.py	the "classic" PyGreSQL module
pgdb.py	a DB-SIG DB-API 2.0 compliant API wrapper for PyGreSQL
setup.py	the Python setup script To install PyGreSQL, you can run "python setup.py install".
setup.cfg	the Python setup configuration
docs/	documentation directory The documentation has been created with Sphinx. All text files are in ReST format; a HTML version of the documentation can be created with the command "make html" or "gmake html".
tests/	a suite of unit tests for PyGreSQL

4.6 Project home sites

Python: http://www.python.org

PostgreSQL: http://www.postgresql.org

PyGreSQL: http://www.pygresql.org

CHAPTER 5

The PyGreSQL documentation

5.1 Contents

5.1.1 Installation

General

You must first have installed Python and PostgreSQL on your system. If you want to access remote databases only, you don't need to install the full PostgreSQL server, but only the C interface (libpq). If you are on Windows, make sure that the directory with libpq.dll is in your `PATH` environment variable.

The current version of PyGreSQL has been tested with Python 2.7 and PostGreSQL 9.3. Older version should work as well, but you will need at least Python 2.4 and PostgreSQL 8.3.

PyGreSQL will be installed as three modules, a dynamic module called _pg.pyd, and two pure Python wrapper modules called pg.py and pgdb.py. All three files will be installed directly into the Python site-packages directory. To uninstall PyGreSQL, simply remove these three files again.

Installing with Pip

This is the most easy way to install PyGreSQL if you have "pip" installed on your computer. Just run the following command in your terminal:

```
pip install PyGreSQL
```

This will automatically try to find and download a distribution on the Python Package Index that matches your operating system and Python version and install it on your computer.

Installing from a Binary Distribution

If you don't want to use "pip", or "pip" doesn't find an appropriate distribution for your computer, you can also try to manually download and install a distribution.

When you download the source distribution, you will need to compile the C extensions, for which you need a C compiler installed on your computer. If you don't want to install a C compiler or avoid possible problems with the compilation, you can search for a pre-compiled binary distribution of PyGreSQL on the Python Package Index or the PyGreSQL homepage.

You can currently download PyGreSQL as Linux RPM, NetBSD package and Windows installer. Make sure the required Python version of the binary package matches the Python version you have installed.

Install the package as usual on your system.

Note that the documentation is currently only included in the source package.

Installing from Source

If you want to install PyGreSQL from Source, or there is no binary package available for your platform, follow these instructions.

Make sure the Python header files and PostgreSQL client and server header files are installed. These come usually with the "devel" packages on Unix systems and the installer executables on Windows systems.

If you are using a precompiled PostgreSQL, you will also need the pg_config tool. This is usually also part of the "devel" package on Unix, and will be installed as part of the database server feature on Windows systems.

Building and installing with Distutils

You can build and install PyGreSQL using Distutils.

Download and unpack the PyGreSQL source tarball if you haven't already done so.

Type the following commands to build and install PyGreSQL:

```
python setup.py build
python setup.py install
```

Now you should be ready to use PyGreSQL.

Compiling Manually

The source file for compiling the dynamic module is called pgmodule.c. You have two options. You can compile PyGreSQL as a stand-alone module or you can build it into the Python interpreter.

Stand-Alone

- In the directory containing `pgmodule.c`, run the following command:

  ```
  cc -fpic -shared -o _pg.so -I$PYINC -I$PGINC -I$PSINC -L$PGLIB -lpq pgmodule.c
  ```

  where you have to set:

  ```
  PYINC = path to the Python include files
          (usually something like /usr/include/python)
  PGINC = path to the PostgreSQL client include files
          (something like /usr/include/pgsql or /usr/include/postgresql)
  PSINC = path to the PostgreSQL server include files
  ```

```
        (like /usr/include/pgsql/server or /usr/include/postgresql/server)
PGLIB = path to the PostgreSQL object code libraries (usually /usr/lib)
```

If you are not sure about the above paths, try something like:

```
PYINC=`find /usr -name Python.h`
PGINC=`find /usr -name libpq-fe.h`
PSINC=`find /usr -name postgres.h`
PGLIB=`find /usr -name libpq.so`
```

If you have the `pg_config` tool installed, you can set:

```
PGINC=`pg_config --includedir`
PSINC=`pg_config --includedir-server`
PGLIB=`pg_config --libdir`
```

Some options may be added to this line:

```
-DNO_DEF_VAR   no default variables support
-DNO_DIRECT    no direct access methods
-DNO_LARGE     no large object support
-DNO_PQSOCKET  if running an older PostgreSQL
```

On some systems you may need to include `-lcrypt` in the list of libraries to make it compile.

- Test the new module. Something like the following should work:

```
$ python

>>> import _pg
>>> db = _pg.connect('thilo','localhost')
>>> db.query("INSERT INTO test VALUES ('ping','pong')")
18304
>>> db.query("SELECT * FROM test")
eins|zwei
----+----
ping|pong
(1 row)
```

- Finally, move the `_pg.so`, `pg.py`, and `pgdb.py` to a directory in your `PYTHONPATH`. A good place would be `/usr/lib/python/site-packages` if your Python modules are in `/usr/lib/python`.

Built-in to Python interpreter

- Find the directory where your `Setup` file lives (usually in the `Modules` subdirectory) in the Python source hierarchy and copy or symlink the `pgmodule.c` file there.
- Add the following line to your 'Setup' file:

```
_pg  pgmodule.c -I$PGINC -I$PSINC -L$PGLIB -lpq
```

where:

```
PGINC = path to the PostgreSQL client include files (see above)
PSINC = path to the PostgreSQL server include files (see above)
PGLIB = path to the PostgreSQL object code libraries (see above)
```

Some options may be added to this line:

```
-DNO_DEF_VAR   no default variables support
-DNO_DIRECT    no direct access methods
-DNO_LARGE     no large object support
-DNO_PQSOCKET  if running an older PostgreSQL (see above)
```

On some systems you may need to include `-lcrypt` in the list of libraries to make it compile.

- If you want a shared module, make sure that the `shared` keyword is uncommented and add the above line below it. You used to need to install your shared modules with `make sharedinstall` but this no longer seems to be true.
- Copy `pg.py` to the lib directory where the rest of your modules are. For example, that's `/usr/local/lib/Python` on my system.
- Rebuild Python from the root directory of the Python source hierarchy by running `make -f Makefile.pre.in boot` and `make && make install`.
- For more details read the documentation at the top of `Makefile.pre.in`.

5.1.2 ChangeLog

Version 4.2.2 (2016-03-18)

- The get_relations() and get_tables() methods now also return system views and tables if you set the optional "system" parameter to True.
- Fixed a regression when using temporary tables with DB wrapper methods (thanks to Patrick TJ McPhee for reporting).

Version 4.2.1 (2016-02-18)

- Fixed a small bug when setting the notice receiver.
- Some more minor fixes and re-packaging with proper permissions.

Version 4.2 (2016-01-21)

- The supported Python versions are 2.4 to 2.7.
- PostgreSQL is supported in all versions from 8.3 to 9.5.
- Set a better default for the user option "escaping-funcs".
- Force build to compile with no errors.
- New methods get_parameters() and set_parameters() in the classic interface which can be used to get or set run-time parameters.
- New method truncate() in the classic interface that can be used to quickly empty a table or a set of tables.
- Fix decimal point handling.
- Add option to return boolean values as bool objects.
- Add option to return money values as string.
- get_tables() does not list information schema tables any more.

- Fix notification handler (Thanks Patrick TJ McPhee).
- Fix a small issue with large objects.
- Minor improvements in the NotificationHandler.
- Converted documentation to Sphinx and added many missing parts.
- The tutorial files have become a chapter in the documentation.
- Greatly improved unit testing, tests run with Python 2.4 to 2.7 again.

Version 4.1.1 (2013-01-08)

- Add NotificationHandler class and method. Replaces need for pgnotify.
- Sharpen test for inserting current_timestamp.
- Add more quote tests. False and 0 should evaluate to NULL.
- More tests - Any number other than 0 is True.
- Do not use positional parameters internally. This restores backward compatibility with version 4.0.
- Add methods for changing the decimal point.

Version 4.1 (2013-01-01)

- Dropped support for Python below 2.5 and PostgreSQL below 8.3.
- Added support for Python up to 2.7 and PostgreSQL up to 9.2.
- Particularly, support PQescapeLiteral() and PQescapeIdentifier().
- The query method of the classic API now supports positional parameters. This an effective way to pass arbitrary or unknown data without worrying about SQL injection or syntax errors (contribution by Patrick TJ McPhee).
- The classic API now supports a method namedresult() in addition to getresult() and dictresult(), which returns the rows of the result as named tuples if these are supported (Python 2.6 or higher).
- The classic API has got the new methods begin(), commit(), rollback(), savepoint() and release() for handling transactions.
- Both classic and DBAPI 2 connections can now be used as context managers for encapsulating transactions.
- The execute() and executemany() methods now return the cursor object, so you can now write statements like "for row in cursor.execute(...)" (as suggested by Adam Frederick).
- Binary objects are now automatically escaped and unescaped.
- Bug in money quoting fixed. Amounts of $0.00 handled correctly.
- Proper handling of date and time objects as input.
- Proper handling of floats with 'nan' or 'inf' values as input.
- Fixed the set_decimal() function.
- All DatabaseError instances now have a sqlstate attribute.
- The getnotify() method can now also return payload strings (#15).
- Better support for notice processing with the new methods set_notice_receiver() and get_notice_receiver() (as suggested by Michael Filonenko, see #37).
- Open transactions are rolled back when pgdb connections are closed (as suggested by Peter Harris, see #46).

- Connections and cursors can now be used with the "with" statement (as suggested by Peter Harris, see #46).
- New method use_regtypes() that can be called to let getattnames() return regular type names instead of the simplified classic types (#44).

Version 4.0 (2009-01-01)

- Dropped support for Python below 2.3 and PostgreSQL below 7.4.
- Improved performance of fetchall() for large result sets by speeding up the type casts (as suggested by Peter Schuller).
- Exposed exceptions as attributes of the connection object.
- Exposed connection as attribute of the cursor object.
- Cursors now support the iteration protocol.
- Added new method to get parameter settings.
- Added customizable row_factory as suggested by Simon Pamies.
- Separated between mandatory and additional type objects.
- Added keyword args to insert, update and delete methods.
- Added exception handling for direct copy.
- Start transactions only when necessary, not after every commit().
- Release the GIL while making a connection (as suggested by Peter Schuller).
- If available, use decimal.Decimal for numeric types.
- Allow DB wrapper to be used with DB-API 2 connections (as suggested by Chris Hilton).
- Made private attributes of DB wrapper accessible.
- Dropped dependence on mx.DateTime module.
- Support for PQescapeStringConn() and PQescapeByteaConn(); these are now also used by the internal _quote() functions.
- Added 'int8' to INTEGER types. New SMALLINT type.
- Added a way to find the number of rows affected by a query() with the classic pg module by returning it as a string. For single inserts, query() still returns the oid as an integer. The pgdb module already provides the "rowcount" cursor attribute for the same purpose.
- Improved getnotify() by calling PQconsumeInput() instead of submitting an empty command.
- Removed compatibility code for old OID munging style.
- The insert() and update() methods now use the "returning" clause if possible to get all changed values, and they also check in advance whether a subsequent select is possible, so that ongoing transactions won't break if there is no select privilege.
- Added "protocol_version" and "server_version" attributes.
- Revived the "user" attribute.
- The pg module now works correctly with composite primary keys; these are represented as frozensets.
- Removed the undocumented and actually unnecessary "view" parameter from the get() method.
- get() raises a nicer ProgrammingError instead of a KeyError if no primary key was found.

- delete() now also works based on the primary key if no oid available and returns whether the row existed or not.

Version 3.8.1 (2006-06-05)

- Use string methods instead of deprecated string functions.
- Only use SQL-standard way of escaping quotes.
- Added the functions escape_string() and escape/unescape_bytea() (as suggested by Charlie Dyson and Kavous Bojnourdi a long time ago).
- Reverted code in clear() method that set date to current.
- Added code for backwards compatibility in OID munging code.
- Reorder attnames tests so that "interval" is checked for before "int."
- If caller supplies key dictionary, make sure that all has a namespace.

Version 3.8 (2006-02-17)

- Installed new favicon.ico from Matthew Sporleder <mspo@mspo.com>
- Replaced snprintf by PyOS_snprintf.
- Removed NO_SNPRINTF switch which is not needed any longer
- Clean up some variable names and namespace
- Add get_relations() method to get any type of relation
- Rewrite get_tables() to use get_relations()
- Use new method in get_attnames method to get attributes of views as well
- Add Binary type
- Number of rows is now -1 after executing no-result statements
- Fix some number handling
- Non-simple types do not raise an error any more
- Improvements to documentation framework
- Take into account that nowadays not every table must have an oid column
- Simplification and improvement of the inserttable() function
- Fix up unit tests
- The usual assortment of minor fixes and enhancements

Version 3.7 (2005-09-07)

Improvement of pgdb module:

- Use Python standard *datetime* if *mxDateTime* is not available

Major improvements and clean-up in classic pg module:

- All members of the underlying connection directly available in *DB*
- Fixes to quoting function

- Add checks for valid database connection to methods
- Improved namespace support, handle *search_path* correctly
- Removed old dust and unnessesary imports, added docstrings
- Internal sql statements as one-liners, smoothed out ugly code

Version 3.6.2 (2005-02-23)

- Further fixes to namespace handling

Version 3.6.1 (2005-01-11)

- Fixes to namespace handling

Version 3.6 (2004-12-17)

- Better DB-API 2.0 compliance
- Exception hierarchy moved into C module and made available to both APIs
- Fix error in update method that caused false exceptions
- Moved to standard exception hierarchy in classic API
- Added new method to get transaction state
- Use proper Python constants where appropriate
- Use Python versions of strtol, etc. Allows Win32 build.
- Bug fixes and cleanups

Version 3.5 (2004-08-29)

Fixes and enhancements:

- Add interval to list of data types
- fix up method wrapping especially close()
- retry pkeys once if table missing in case it was just added
- wrap query method separately to handle debug better
- use isinstance instead of type
- fix free/PQfreemem issue - finally
- miscellaneous cleanups and formatting

Version 3.4 (2004-06-02)

Some cleanups and fixes. This is the first version where PyGreSQL is moved back out of the PostgreSQL tree. A lot of the changes mentioned below were actually made while in the PostgreSQL tree since their last release.

- Allow for larger integer returns
- Return proper strings for true and false

- Cleanup convenience method creation
- Enhance debugging method
- Add reopen method
- Allow programs to preload field names for speedup
- Move OID handling so that it returns long instead of int
- Miscellaneous cleanups and formatting

Version 3.3 (2001-12-03)

A few cleanups. Mostly there was some confusion about the latest version and so I am bumping the number to keep it straight.

- Added NUMERICOID to list of returned types. This fixes a bug when returning aggregates in the latest version of PostgreSQL.

Version 3.2 (2001-06-20)

Note that there are very few changes to PyGreSQL between 3.1 and 3.2. The main reason for the release is the move into the PostgreSQL development tree. Even the WIN32 changes are pretty minor.

- Add Win32 support (gerhard@bigfoot.de)
- Fix some DB-API quoting problems (niall.smart@ebeon.com)
- Moved development into PostgreSQL development tree.

Version 3.1 (2000-11-06)

- Fix some quoting functions. In particular handle NULLs better.
- Use a method to add primary key information rather than direct manipulation of the class structures
- Break decimal out in *_quote* (in pg.py) and treat it as float
- Treat timestamp like date for quoting purposes
- Remove a redundant SELECT from the *get* method speeding it, and *insert* (since it calls *get*) up a little.
- Add test for BOOL type in typecast method to *pgdbTypeCache* class (tv@beamnet.de)
- Fix pgdb.py to send port as integer to lower level function (dildog@l0pht.com)
- Change pg.py to speed up some operations
- Allow updates on tables with no primary keys

Version 3.0 (2000-05-30)

- Remove strlen() call from pglarge_write() and get size from object (Richard@Bouska.cz)
- Add a little more error checking to the quote function in the wrapper
- Add extra checking in *_quote* function
- Wrap query in pg.py for debugging
- Add DB-API 2.0 support to pgmodule.c (andre@via.ecp.fr)

- Add DB-API 2.0 wrapper pgdb.py (andre@via.ecp.fr)
- Correct keyword clash (temp) in tutorial
- Clean up layout of tutorial
- Return NULL values as None (rlawrence@lastfoot.com) (WARNING: This will cause backwards compatibility issues)
- Change None to NULL in insert and update
- Change hash-bang lines to use /usr/bin/env
- Clearing date should be blank (NULL) not TODAY
- Quote backslashes in strings in *_quote* (brian@CSUA.Berkeley.EDU)
- Expanded and clarified build instructions (tbryan@starship.python.net)
- Make code thread safe (Jerome.Alet@unice.fr)
- Add README.distutils (mwa@gate.net & jeremy@cnri.reston.va.us)
- Many fixes and increased DB-API compliance by chifungfan@yahoo.com, tony@printra.net, jeremy@alum.mit.edu and others to get the final version ready to release.

Version 2.4 (1999-06-15)

- Insert returns None if the user doesn't have select permissions on the table. It can (and does) happen that one has insert but not select permissions on a table.
- Added ntuples() method to query object (brit@druid.net)
- Corrected a bug related to getresult() and the money type
- Corrected a bug related to negative money amounts
- Allow update based on primary key if munged oid not available and table has a primary key
- Add many __doc__ strings (andre@via.ecp.fr)
- Get method works with views if key specified

Version 2.3 (1999-04-17)

- connect.host returns "localhost" when connected to Unix socket (torppa@tuhnu.cutery.fi)
- Use *PyArg_ParseTupleAndKeywords* in connect() (torppa@tuhnu.cutery.fi)
- fixes and cleanups (torppa@tuhnu.cutery.fi)
- Fixed memory leak in dictresult() (terekhov@emc.com)
- Deprecated pgext.py - functionality now in pg.py
- More cleanups to the tutorial
- Added fileno() method - terekhov@emc.com (Mikhail Terekhov)
- added money type to quoting function
- Compiles cleanly with more warnings turned on
- Returns PostgreSQL error message on error
- Init accepts keywords (Jarkko Torppa)

- Convenience functions can be overridden (Jarkko Torppa)
- added close() method

Version 2.2 (1998-12-21)

- Added user and password support thanks to Ng Pheng Siong (ngps@post1.com)
- Insert queries return the inserted oid
- Add new *pg* wrapper (C module renamed to _pg)
- Wrapped database connection in a class
- Cleaned up some of the tutorial. (More work needed.)
- Added *version* and *__version__*. Thanks to thilo@eevolute.com for the suggestion.

Version 2.1 (1998-03-07)

- return fields as proper Python objects for field type
- Cleaned up pgext.py
- Added dictresult method

Version 2.0 (1997-12-23)

- Updated code for PostgreSQL 6.2.1 and Python 1.5
- Reformatted code and converted to use full ANSI style prototypes
- Changed name to PyGreSQL (from PyGres95)
- Changed order of arguments to connect function
- Created new type *pgqueryobject* and moved certain methods to it
- Added a print function for pgqueryobject
- Various code changes - mostly stylistic

Version 1.0b (1995-11-04)

- Keyword support for connect function moved from library file to C code and taken away from library
- Rewrote documentation
- Bug fix in connect function
- Enhancements in large objects interface methods

Version 1.0a (1995-10-30)

A limited release.

- Module adapted to standard Python syntax
- Keyword support for connect function in library file

- Rewrote default parameters interface (internal use of strings)
- Fixed minor bugs in module interface
- Redefinition of error messages

Version 0.9b (1995-10-10)

The first public release.

- Large objects implementation
- Many bug fixes, enhancements, ...

Version 0.1a (1995-10-07)

- Basic libpq functions (SQL access)

5.1.3 General PyGreSQL Programming Information

PyGreSQL consists of two parts: the "classic" PyGreSQL interface provided by the *pg* module and the newer DB-API 2.0 compliant interface provided by the *pgdb* module.

If you use only the standard features of the DB-API 2.0 interface, it will be easier to switch from PostgreSQL to another database for which a DB-API 2.0 compliant interface exists.

The "classic" interface may be easier to use for beginners, and it provides some higher-level and PostgreSQL specific convenience methods.

See also:

DB-API 2.0 (Python Database API Specification v2.0) is a specification for connecting to databases (not only PostGreSQL) from Python that has been developed by the Python DB-SIG in 1999. The authoritative programming information for the DB-API is **PEP 0249**.

Both Python modules utilize the same lower level C extension module that serves as a wrapper for the C API to PostgreSQL that is available in form of the so-called "libpq" library.

This means you must have the libpq library installed as a shared library on your client computer, in a version that is supported by PyGreSQL. Depending on the client platform, you may have to set environment variables like *PATH* or *LD_LIBRARY_PATH* so that PyGreSQL can find the library.

Warning: Note that PyGreSQL is not thread-safe on the connection level. Therefore we recommend using DBUtils for multi-threaded environments, which supports both PyGreSQL interfaces.

5.1.4 First Steps with PyGreSQL

In this small tutorial we show you the basic operations you can perform with both flavors of the PyGreSQL interface. Please choose your flavor:

- *First Steps with the classic PyGreSQL Interface*
- *First Steps with the DB-API 2.0 Interface*

First Steps with the classic PyGreSQL Interface

The first thing you need to do anything with your PostgreSQL database is to create a database connection.

To do this, simply import the *DB* wrapper class and create an instance of it, passing the necessary connection parameters, like this:

```
>>> from pg import DB
>>> db = DB(dbname='testdb', host='pgserver', port=5432,
...     user='scott', passwd='tiger')
```

You can omit one or even all parameters if you want to use their default values. PostgreSQL will use the name of the current operating system user as the login and the database name, and will try to connect to the local host on port 5432 if nothing else is specified.

The *db* object has all methods of the lower-level *pgobject* class plus some more convenience methods provided by the *DB* wrapper.

You can now execute database queries using the *DB.query()* method:

```
>>> db.query("create table fruits(id serial primary key, name varchar)")
```

You can list all database tables with the *DB.get_tables()* method:

```
>>> db.get_tables()
['public.fruits']
```

To get the attributes of the *fruits* table, use *DB.get_attnames()*:

```
>>> db.get_attnames('fruits')
{'id': 'int', 'name': 'text'}
```

Verify that you can insert into the newly created *fruits* table:

```
>>> db.has_table_privilege('fruits', 'insert')
True
```

You can insert a new row into the table using the *DB.insert()* method, for example:

```
>>> db.insert('fruits', name='apple')
{'name': 'apple', 'id': 1}
```

Note how this method returns the full row as a dictionary including its *id* column that has been generated automatically by a database sequence. You can also pass a dictionary to the *DB.insert()* method instead of or in addition to using keyword arguments.

Let's add another row to the table:

```
>>> banana = db.insert('fruits', name='banana')
```

Or, you can add a whole bunch of fruits at the same time using the `DB.inserttable()` method. Note that this method uses the COPY command of PostgreSQL to insert all data in one operation, which is faster than sending many INSERT commands:

```
>>> more_fruits = 'cherimaya durian eggfruit fig grapefruit'.split()
>>> data = list(enumerate(more_fruits, start=3))
>>> db.inserttable('fruits', data)
```

We can now query the database for all rows that have been inserted into the *fruits* table:

```
>>> print db.query('select * from fruits')
id|   name
--+----------
 1|apple
 2|banana
 3|cherimaya
 4|durian
 5|eggfruit
 6|fig
 7|grapefruit
(7 rows)
```

Instead of simply printing the *pgqueryobject* instance that has been returned by this query, we can also request the data as list of tuples:

```
>>> q = db.query('select * from fruits')
>>> q.getresult()
... [(1, 'apple'), ..., (7, 'grapefruit')]
```

Instead of a list of tuples, we can also request a list of dicts:

```
>>> q.dictresult()
[{'id': 1, 'name': 'apple'}, ..., {'id': 7, 'name': 'grapefruit'}]
```

And with Python 2.5 or higher, you can also return the rows as named tuples:

```
>>> rows = q.namedresult()
>>> rows[3].name
'durian'
```

To change a single row in the database, you can use the *DB.update()* method. For instance, if you want to capitalize the name 'banana':

```
>>> db.update('fruits', banana, name=banana['name'].capitalize())
{'id': 2, 'name': 'Banana'}
>>> print db.query('select * from fruits where id between 1 and 3')
id|  name
--+---------
 1|apple
 2|Banana
 3|cherimaya
(3 rows)
```

Let's also capitalize the other names in the database:

```
>>> db.query('update fruits set name=initcap(name)')
'7'
```

The returned string *'7'* tells us the number of updated rows. It is returned as a string to discern it from an OID which will be returned as an integer, if a new row has been inserted into a table with an OID column.

To delete a single row from the database, use the *DB.delete()* method:

```
>>> db.delete('fruits', banana)
1
```

The returned integer value *1* tells us that one row has been deleted. If we try it again, the method returns the integer value *0*. Naturally, this method can only return 0 or 1:

```
>>> db.delete('fruits', banana)
0
```

Of course, we can insert the row back again:

```
>>> db.insert('fruits', banana)
{'id': 2, 'name': 'Banana'}
```

If we want to change a different row, we can get its current state with:

```
>>> apple = db.get('fruits', 1)
>>> apple
{'name': 'Apple', 'id': 1}
```

We can duplicate the row like this:

```
   >>> db.insert('fruits', apple, id=8)
   {'id': 8, 'name': 'Apple'}

To remove the duplicated row, we can do::

   >>> db.delete('fruits', id=8)
   1
```

Finally, to remove the table from the database and close the connection:

```
>>> db.query("drop table fruits")
>>> db.close()
```

For more advanced features and details, see the reference: *pg — The Classic PyGreSQL Interface*

First Steps with the DB-API 2.0 Interface

As with the classic interface, the first thing you need to do is to create a database connection. To do this, use the function *pgdb.connect()* in the *pgdb* module, passing the connection parameters:

```
>>> from pgdb import connect
>>> con = connect(database='testdb', host='pgserver:5432',
...     user='scott', password='tiger')
```

Note that like in the classic interface, you can omit parameters if they are the default values used by PostgreSQL.

To do anything with the connection, you need to request a cursor object from it, which is thought of as the Python representation of a database cursor. The connection has a method that lets you get a cursor:

```
>>> cursor = con.cursor()
```

The cursor now has a method that lets you execute database queries:

```
>>> cursor.execute("create table fruits("
...     "id serial primary key, name varchar)")
```

To insert data into the table, also can also use this method:

```
>>> cursor.execute("insert into fruits (name) values ('apple')")
```

You can pass parameters in a safe way:

```
>>> cursor.execute("insert into fruits (name) values (%s)", ('banana',))
```

For inserting multiple rows at once, you can use the following method:

```
>>> more_fruits = 'cherimaya durian eggfruit fig grapefruit'.split()
>>> parameters = [(name,) for name in more_fruits]
>>> cursor.executemany("insert into fruits (name) values (%s)", parameters)
```

Note that the DB API 2.0 interface does not have an autocommit as you may be used from PostgreSQL. So in order to make these inserts permanent, you need to commit them to the database first:

```
>>> con.commit()
```

If you end the program without calling the commit method of the connection, or if you call the rollback method of the connection, then all the changes will be discarded.

In a similar way, you can also update or delete rows in the database, executing UPDATE or DELETE statements instead of INSERT statements.

To fetch rows from the database, execute a SELECT statement first. Then you can use one of several fetch methods to retrieve the results. For instance, to request a single row:

```
>>> cursor.execute('select * from fruits where id=1')
>>> cursor.fetchone()
[1, 'apple']
```

The output is a single list that represents the row.

To fetch all rows of the query, use this method instead:

```
>>> cursor.execute('select * from fruits')
>>> cursor.fetchall()
[[1, 'apple'], ..., [7, 'grapefruit']]
```

The output is a list with 7 items which are lists of 2 items.

If you want to fetch only a limited number of rows from the query:

```
>>> cursor.execute('select * from fruits')
>>> cursor.fetchmany(2)
[[1, 'apple'], [2, 'banana']]
```

Finally, to remove the table from the database and close the connection:

```
>>> db.execute("drop table fruits")
>>> cur.close()
>>> db.close()
```

For more advanced features and details, see the reference: *pgdb — The DB-API Compliant Interface*

5.1.5 pg — The Classic PyGreSQL Interface

Contents

Introduction

You may either choose to use the "classic" PyGreSQL interface provided by the *pg* module or else the newer DB-API 2.0 compliant interface provided by the *pgdb* module.

The following part of the documentation covers only the older *pg* API.

The *pg* module handles three types of objects,

- the pgobject, which handles the connection and all the requests to the database,
- the pglarge object, which handles all the accesses to PostgreSQL large objects,
- the pgqueryobject that handles query results

and it provides a convenient wrapper class DB for the pgobject.

See also:

If you want to see a simple example of the use of some of these functions, see the *Examples* page.

Module functions and constants

The *pg* module defines a few functions that allow to connect to a database and to define "default variables" that override the environment variables used by PostgreSQL.

These "default variables" were designed to allow you to handle general connection parameters without heavy code in your programs. You can prompt the user for a value, put it in the default variable, and forget it, without having to modify your environment. The support for default variables can be disabled by setting the -DNO_DEF_VAR option in the Python setup file. Methods relative to this are specified by the tag [DV].

All variables are set to None at module initialization, specifying that standard environment variables should be used.

connect – Open a PostgreSQL connection

pg.**connect**([*dbname*][, *host*][, *port*][, *opt*][, *tty*][, *user*][, *passwd*])

Open a *pg* connection

Parameters

- **dbname** – name of connected database (*None* = defbase)
- **host** (*str or None*) – name of the server host (*None* = defhost)
- **port** (*int*) – port used by the database server (-1 = defport)
- **opt** (*str or None*) – connection options (*None* = defopt)
- **tty** (*str or None*) – debug terminal (*None* = deftty)
- **user** (*str or None*) – PostgreSQL user (*None* = defuser)
- **passwd** (*str or None*) – password for user (*None* = defpasswd)

Returns If successful, the *pgobject* handling the connection

Return type *pgobject*

Raises

- **TypeError** – bad argument type, or too many arguments
- **SyntaxError** – duplicate argument definition
- **pg.InternalError** – some error occurred during pg connection definition
- **Exception** – (all exceptions relative to object allocation)

This function opens a connection to a specified database on a given PostgreSQL server. You can use keywords here, as described in the Python tutorial. The names of the keywords are the name of the parameters given in the syntax line. For a precise description of the parameters, please refer to the PostgreSQL user manual.

Example:

```
import pg

con1 = pg.connect('testdb', 'myhost', 5432, None, None, 'bob', None)
con2 = pg.connect(dbname='testdb', host='localhost', user='bob')
```

get/set_defhost – default server host [DV]

pg.**get_defhost**(*host*)
Get the default host

Returns the current default host specification

Return type str or None

Raises **TypeError** – too many arguments

This method returns the current default host specification, or `None` if the environment variables should be used. Environment variables won't be looked up.

pg.**set_defhost**(*host*)
Set the default host

Parameters **host** (*str or None*) – the new default host specification

Returns the previous default host specification

Return type str or None

Raises **TypeError** – bad argument type, or too many arguments

This methods sets the default host value for new connections. If `None` is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default host.

get/set_defport – default server port [DV]

pg.**get_defport**()
Get the default port

Returns the current default port specification

Return type int

Raises **TypeError** – too many arguments

This method returns the current default port specification, or `None` if the environment variables should be used. Environment variables won't be looked up.

pg.**set_defport**(*port*)
Set the default port

Parameters **port** (*int*) – the new default port

Returns previous default port specification

Return type int or None

This methods sets the default port value for new connections. If -1 is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default port.

get/set_defopt – default connection options [DV]

pg.**get_defopt**()
Get the default connection options

Returns the current default options specification

Return type str or None

Raises **TypeError** – too many arguments

This method returns the current default connection options specification, or `None` if the environment variables should be used. Environment variables won't be looked up.

pg.**set_defopt**(*options*)
Set the default connection options

Parameters **options** (*str or None*) – the new default connection options

Returns previous default options specification

Return type str or None

Raises **TypeError** – bad argument type, or too many arguments

This methods sets the default connection options value for new connections. If `None` is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default options.

get/set_deftty – default debug tty [DV]

pg.**get_deftty**()
Get the default debug terminal

Returns the current default debug terminal specification

Return type str or None

Raises **TypeError** – too many arguments

This method returns the current default debug terminal specification, or `None` if the environment variables should be used. Environment variables won't be looked up. Note that this is ignored in newer PostgreSQL versions.

pg.**set_deftty**(*terminal*)
Set the default debug terminal

Parameters **terminal** (*str or None*) – the new default debug terminal

Returns the previous default debug terminal specification

Return type str or None

Raises **TypeError** – bad argument type, or too many arguments

This methods sets the default debug terminal value for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default terminal. Note that this is ignored in newer PostgreSQL versions.

get/set_defbase – default database name [DV]

pg.**get_defbase**()
Get the default database name

Returns the current default database name specification

Return type str or None

Raises **TypeError** – too many arguments

This method returns the current default database name specification, or None if the environment variables should be used. Environment variables won't be looked up.

pg.**set_defbase**(*base*)
Set the default database name

Parameters **base** (*str or None*) – the new default base name

Returns the previous default database name specification

Return type str or None

Raises **TypeError** – bad argument type, or too many arguments

This method sets the default database name value for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default host.

get/set_defuser – default database user [DV]

pg.**get_defuser**()
Get the default database user

Returns the current default database user specification

Return type str or None

Raises **TypeError** – too many arguments

This method returns the current default database user specification, or None if the environment variables should be used. Environment variables won't be looked up.

pg.**set_defuser**(*user*)
Set the default database user

Parameters **user** – the new default database user

Returns the previous default database user specification

Return type str or None

Raises **TypeError** – bad argument type, or too many arguments

This method sets the default database user name for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default host.

get/set_defpasswd – default database password [DV]

pg.**get_defpasswd**()
Get the default database password

Returns the current default database password specification

Return type str or None

Raises **TypeError** – too many arguments

This method returns the current default database password specification, or `None` if the environment variables should be used. Environment variables won't be looked up.

pg.**set_defpasswd**(*passwd*)
Set the default database password

Parameters **passwd** – the new default database password

Returns the previous default database password specification

Return type str or None

Raises **TypeError** – bad argument type, or too many arguments

This method sets the default database password for new connections. If `None` is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default host.

escape_string – escape a string for use within SQL

pg.**escape_string**(*string*)
Escape a string for use within SQL

Parameters **string** (*str*) – the string that is to be escaped

Returns the escaped string

Return type str

Raises **TypeError** – bad argument type, or too many arguments

This function escapes a string for use within an SQL command. This is useful when inserting data values as literal constants in SQL commands. Certain characters (such as quotes and backslashes) must be escaped to prevent them from being interpreted specially by the SQL parser. *escape_string()* performs this operation. Note that there is also a *pgobject* method with the same name which takes connection properties into account.

Note: It is especially important to do proper escaping when handling strings that were received from an untrustworthy source. Otherwise there is a security risk: you are vulnerable to "SQL injection" attacks wherein unwanted SQL commands are fed to your database.

Example:

```
name = raw_input("Name? ")
phone = con.query("select phone from employees where name='%s'"
    % escape_string(name)).getresult()
```

escape_bytea – escape binary data for use within SQL

pg.**escape_bytea**(*datastring*)
escape binary data for use within SQL as type bytea

Parameters **datastring** (*str*) – string containing the binary data that is to be escaped

Returns the escaped string

Return type str

Raises **TypeError** – bad argument type, or too many arguments

Escapes binary data for use within an SQL command with the type bytea. As with *escape_string()*, this is only used when inserting data directly into an SQL command string. Note that there is also a *pgobject* method with the same name which takes connection properties into account.

Example:

```
picture = open('garfield.gif', 'rb').read()
con.query("update pictures set img='%s' where name='Garfield'"
    % escape_bytea(picture))
```

unescape_bytea – unescape data that has been retrieved as text

pg.**unescape_bytea**(*string*)
Unescape bytea data that has been retrieved as text

Parameters **datastring** (*str*) – the bytea data string that has been retrieved as text

Returns byte string containing the binary data

Return type str

Raises **TypeError** – bad argument type, or too many arguments

Converts an escaped string representation of binary data into binary data – the reverse of *escape_bytea()*. This is needed when retrieving bytea data with one of the *pgqueryobject.getresult()*, *pgqueryobject.dictresult()* or *pgqueryobject.namedresult()* methods.

Example:

```
picture = unescape_bytea(con.query(
      "select img from pictures where name='Garfield'").getresult[0][0])
open('garfield.gif', 'wb').write(picture)
```

get/set_decimal – decimal type to be used for numeric values

pg.**get_decimal**()
Get the decimal type to be used for numeric values

Returns the Python class used for PostgreSQL numeric values

Return type class

This function returns the Python class that is used by PyGreSQL to hold PostgreSQL numeric values. The default class is decimal.Decimal if available, otherwise the float type is used.

pg.**set_decimal**(*cls*)
Set a decimal type to be used for numeric values

Parameters **cls** (*class*) – the Python class to be used for PostgreSQL numeric values

This function can be used to specify the Python class that shall be used by PyGreSQL to hold PostgreSQL numeric values. The default class is `decimal.Decimal` if available, otherwise the `float` type is used.

get/set_decimal_point – decimal mark used for monetary values

pg.**get_decimal_point**()
Get the decimal mark used for monetary values

Returns string with one character representing the decimal mark

Return type str

This function returns the decimal mark used by PyGreSQL to interpret PostgreSQL monetary values when converting them to decimal numbers. The default setting is `'.'` as a decimal point. This setting is not adapted automatically to the locale used by PostGreSQL, but you can use `set_decimal()` to set a different decimal mark manually. A return value of `None` means monetary values are not interpreted as decimal numbers, but returned as strings including the formatting and currency.

New in version 4.1.1.

pg.**set_decimal_point**(*string*)
Specify which decimal mark is used for interpreting monetary values

Parameters **string** (*str*) – string with one character representing the decimal mark

This function can be used to specify the decimal mark used by PyGreSQL to interpret PostgreSQL monetary values. The default value is '.' as a decimal point. This value is not adapted automatically to the locale used by PostGreSQL, so if you are dealing with a database set to a locale that uses a `','` instead of `'.'` as the decimal point, then you need to call `set_decimal(',')` to have PyGreSQL interpret monetary values correctly. If you don't want money values to be converted to decimal numbers, then you can call `set_decimal(None)`, which will cause PyGreSQL to return monetary values as strings including their formatting and currency.

New in version 4.1.1.

get/set_bool – whether boolean values are returned as bool objects

pg.**get_bool**()
Check whether boolean values are returned as bool objects

Returns whether or not bool objects will be returned

Return type bool

This function checks whether PyGreSQL returns PostgreSQL boolean values converted to Python bool objects, or as `'f'` and `'t'` strings which are the values used internally by PostgreSQL. By default, conversion to bool objects is not activated, but you can enable this with the `set_bool()` method.

New in version 4.2.

pg.**set_bool**(*on*)
Set whether boolean values are returned as bool objects

Parameters **on** – whether or not bool objects shall be returned

This function can be used to specify whether PyGreSQL shall return PostgreSQL boolean values converted to Python bool objects, or as `'f'` and `'t'` strings which are the values used internally by PostgreSQL. By default, conversion to bool objects is not activated, but you can enable this by calling `set_bool(True)`.

New in version 4.2.

get/set_namedresult – conversion to named tuples

pg.**get_namedresult**()
Get the function that converts to named tuples

This returns the function used by PyGreSQL to construct the result of the *pgqueryobject.namedresult()* method.

New in version 4.1.

pg.**set_namedresult**(*func*)
Set a function that will convert to named tuples

Parameters **func** – the function to be used to convert results to named tuples

You can use this if you want to create different kinds of named tuples returned by the *pgqueryobject.namedresult()* method. If you set this function to *None*, then it will become equal to *pgqueryobject.getresult()*.

New in version 4.1.

Module constants

Some constants are defined in the module dictionary. They are intended to be used as parameters for methods calls. You should refer to the libpq description in the PostgreSQL user manual for more information about them. These constants are:

version, __version__
constants that give the current version

INV_READ, INV_WRITE
large objects access modes, used by *pgobject.locreate()* and *pglarge.open()*

SEEK_SET, SEEK_CUR, SEEK_END:
positional flags, used by *pglarge.seek()*

pgobject – The connection object

class pg.**pgobject**

This object handles a connection to a PostgreSQL database. It embeds and hides all the parameters that define this connection, thus just leaving really significant parameters in function calls.

Note: Some methods give direct access to the connection socket. *Do not use them unless you really know what you are doing.* If you prefer disabling them, set the -DNO_DIRECT option in the Python setup file. These methods are specified by the tag [DA].

Note: Some other methods give access to large objects (refer to PostgreSQL user manual for more information about these). If you want to forbid access to these from the module, set the -DNO_LARGE option in the Python setup file. These methods are specified by the tag [LO].

---

query – execute a SQL command string

pgobject.**query**(*command*[, *args*])
Execute a SQL command string

Parameters

- **command** (*str*) – SQL command
- **args** – optional positional arguments

Returns result values

Return type *pgqueryobject*, None

Raises

- **TypeError** – bad argument type, or too many arguments
- **TypeError** – invalid connection
- **ValueError** – empty SQL query or lost connection
- **pg.ProgrammingError** – error in query
- **pg.InternalError** – error during query processing

This method simply sends a SQL query to the database. If the query is an insert statement that inserted exactly one row into a table that has OIDs, the return value is the OID of the newly inserted row. If the query is an update or delete statement, or an insert statement that did not insert exactly one row in a table with OIDs, then the number of rows affected is returned as a string. If it is a statement that returns rows as a result (usually a select statement, but maybe also an "`insert/update ... returning`" statement), this method returns a *pgqueryobject* that can be accessed via the *pgqueryobject.getresult()*, *pgqueryobject.dictresult()* or *pgqueryobject.namedresult()* methods or simply printed. Otherwise, it returns `None`.

The query may optionally contain positional parameters of the form `$1`, `$2`, etc instead of literal data, and the values supplied as a tuple. The values are substituted by the database in such a way that they don't need to be escaped, making this an effective way to pass arbitrary or unknown data without worrying about SQL injection or syntax errors.

When the database could not process the query, a `pg.ProgrammingError` or a `pg.InternalError` is raised. You can check the `SQLSTATE` error code of this error by reading its `sqlstate` attribute.

Example:

```
name = raw_input("Name? ")
phone = con.query("select phone from employees where name=$1",
    (name,)).getresult()
```

reset – reset the connection

pgobject.**reset**()
Reset the *pg* connection

Return type None

Raises

- **TypeError** – too many (any) arguments
- **TypeError** – invalid connection

This method resets the current database connection.

cancel – abandon processing of current SQL command

pgobject.**cancel**()

Return type None

Raises

- **TypeError** – too many (any) arguments
- **TypeError** – invalid connection

This method requests that the server abandon processing of the current SQL command.

close – close the database connection

pgobject.**close**()

Close the *pg* connection

Return type None

Raises **TypeError** – too many (any) arguments

This method closes the database connection. The connection will be closed in any case when the connection is deleted but this allows you to explicitly close it. It is mainly here to allow the DB-SIG API wrapper to implement a close function.

fileno – returns the socket used to connect to the database

pgobject.**fileno**()

Return the socket used to connect to the database

Returns the socket id of the database connection

Return type int

Raises

- **TypeError** – too many (any) arguments
- **TypeError** – invalid connection

This method returns the underlying socket id used to connect to the database. This is useful for use in select calls, etc.

getnotify – get the last notify from the server

pgobject.**getnotify**()

Get the last notify from the server

Returns last notify from server

Return type tuple, None

Raises

- **TypeError** – too many parameters
- **TypeError** – invalid connection

This method tries to get a notify from the server (from the SQL statement NOTIFY). If the server returns no notify, the methods returns None. Otherwise, it returns a tuple (triplet) *(relname, pid, extra)*, where *relname* is the name of the notify, *pid* is the process id of the connection that triggered the notify, and *extra* is a payload string that has been sent with the notification. Remember to do a listen query first, otherwise *pgobject.getnotify()* will always return None.

Changed in version 4.1: Support for payload strings was added in version 4.1.

inserttable – insert a list into a table

pgobject.**inserttable**(*table*, *values*)
Insert a Python list into a database table

Parameters

- **table** (*str*) – the table name
- **values** (*list*) – list of rows values

Return type None

Raises

- **TypeError** – invalid connection, bad argument type, or too many arguments
- **MemoryError** – insert buffer could not be allocated
- **ValueError** – unsupported values

This method allows to *quickly* insert large blocks of data in a table: It inserts the whole values list into the given table. Internally, it uses the COPY command of the PostgreSQL database. The list is a list of tuples/lists that define the values for each inserted row. The rows values may contain string, integer, long or double (real) values.

Warning: This method doesn't type check the fields according to the table definition; it just look whether or not it knows how to handle such types.

get/set_notice_receiver – custom notice receiver

pgobject.**get_notice_receiver**()
Get the current notice receiver

Returns the current notice receiver callable

Return type callable, None

Raises **TypeError** – too many (any) arguments

This method gets the custom notice receiver callback function that has been set with *pgobject.set_notice_receiver()*, or None if no custom notice receiver has ever been set on the connection.

New in version 4.1.

pgobject.**set_notice_receiver**(*proc*)
Set a custom notice receiver

Parameters **proc** – the custom notice receiver callback function

Return type None

Raises **TypeError** – the specified notice receiver is not callable

This method allows setting a custom notice receiver callback function. When a notice or warning message is received from the server, or generated internally by libpq, and the message level is below the one set with `client_min_messages`, the specified notice receiver function will be called. This function must take one parameter, the `pgnotice` object, which provides the following read-only attributes:

`pgnotice.`**`pgcnx`**
the connection

`pgnotice.`**`message`**
the full message with a trailing newline

`pgnotice.`**`severity`**
the level of the message, e.g. 'NOTICE' or 'WARNING'

`pgnotice.`**`primary`**
the primary human-readable error message

`pgnotice.`**`detail`**
an optional secondary error message

`pgnotice.`**`hint`**
an optional suggestion what to do about the problem

New in version 4.1.

putline – write a line to the server socket [DA]

`pgobject.`**`putline`**(*line*)
Write a line to the server socket

Parameters **`line`** (*str*) – line to be written

Return type None

Raises **`TypeError`** – invalid connection, bad parameter type, or too many parameters

This method allows to directly write a string to the server socket.

getline – get a line from server socket [DA]

`pgobject.`**`getline`**()
Get a line from server socket

Returns the line read

Return type str

Raises

- **`TypeError`** – invalid connection
- **`TypeError`** – too many parameters
- **`MemoryError`** – buffer overflow

This method allows to directly read a string from the server socket.

endcopy – synchronize client and server [DA]

pgobject.**endcopy**()
Synchronize client and server

Return type None

Raises

- **TypeError** – invalid connection
- **TypeError** – too many parameters

The use of direct access methods may desynchronize client and server. This method ensure that client and server will be synchronized.

locreate – create a large object in the database [LO]

pgobject.**locreate**(*mode*)
Create a large object in the database

Parameters **mode** (*int*) – large object create mode

Returns object handling the PostGreSQL large object

Return type *pglarge*

Raises

- **TypeError** – invalid connection, bad parameter type, or too many parameters
- **pg.OperationalError** – creation error

This method creates a large object in the database. The mode can be defined by OR-ing the constants defined in the *pg* module (INV_READ, INV_WRITE and INV_ARCHIVE). Please refer to PostgreSQL user manual for a description of the mode values.

getlo – build a large object from given oid [LO]

pgobject.**getlo**(*oid*)
Create a large object in the database

Parameters **oid** (*int*) – OID of the existing large object

Returns object handling the PostGreSQL large object

Return type *pglarge*

Raises

- **TypeError** – invalid connection, bad parameter type, or too many parameters
- **ValueError** – bad OID value (0 is invalid_oid)

This method allows to reuse a formerly created large object through the *pglarge* interface, providing the user have its OID.

loimport – import a file to a large object [LO]

pgobject.**loimport**(*name*)
Import a file to a large object

Parameters **name** (*str*) – the name of the file to be imported

Returns object handling the PostGreSQL large object

Return type *pglarge*

Raises

- **TypeError** – invalid connection, bad argument type, or too many arguments
- **pg.OperationalError** – error during file import

This methods allows to create large objects in a very simple way. You just give the name of a file containing the data to be used.

Object attributes

Every *pgobject* defines a set of read-only attributes that describe the connection and its status. These attributes are:

pgobject.**host**
the host name of the server (str)

pgobject.**port**
the port of the server (int)

pgobject.**db**
the selected database (str)

pgobject.**options**
the connection options (str)

pgobject.**tty**
the connection debug terminal (str)

pgobject.**user**
user name on the database system (str)

pgobject.**protocol_version**
the frontend/backend protocol being used (int)

New in version 4.0.

pgobject.**server_version**
the backend version (int, e.g. 80305 for 8.3.5)

New in version 4.0.

pgobject.**status**
the status of the connection (int: 1 = OK, 0 = bad)

pgobject.**error**
the last warning/error message from the server (str)

The DB wrapper class

class pg.**DB**

The *pgobject* methods are wrapped in the class *DB*. The preferred way to use this module is as follows:

```
import pg

db = pg.DB(...)  # see below

for r in db.query(  # just for example
    """SELECT foo,bar
     FROM foo_bar_table
     WHERE foo !~ bar"""
    ).dictresult():

    print '%(foo)s %(bar)s' % r
```

This class can be subclassed as in this example:

```
import pg

class DB_ride(pg.DB):
    """Ride database wrapper

    This class encapsulates the database functions and the specific
    methods for the ride database."""

def __init__(self):
    """Open a database connection to the rides database"""
    pg.DB.__init__(self, dbname='ride')
    self.query("SET DATESTYLE TO 'ISO'")

[Add or override methods here]
```

The following describes the methods and variables of this class.

Initialization

The *DB* class is initialized with the same arguments as the *connect()* function described above. It also initializes a few internal variables. The statement db = DB() will open the local database with the name of the user just like connect() does.

You can also initialize the DB class with an existing *pg* or *pgdb* connection. Pass this connection as a single unnamed parameter, or as a single parameter named db. This allows you to use all of the methods of the DB class with a DB-API 2 compliant connection. Note that the *pgobject.close()* and pgobject.reopen() methods are inoperative in this case.

pkey – return the primary key of a table

DB.**pkey**(*table*)

Return the primary key of a table

Parameters **table** (*str*) – name of table

Returns Name of the field which is the primary key of the table

Return type str

Return type str

Raises **KeyError** – the table does not have a primary key

This method returns the primary key of a table. For composite primary keys, the return value will be a frozenset. Note that this raises a KeyError if the table does not have a primary key.

get_databases – get list of databases in the system

DB.**get_databases**()
Get the list of databases in the system

Returns all databases in the system

Return type list

Although you can do this with a simple select, it is added here for convenience.

get_relations – get list of relations in connected database

DB.**get_relations**([*kinds*][, *system*])
Get the list of relations in connected database

Parameters

- **kinds** (*str*) – a string or sequence of type letters
- **system** (*bool*) – whether system relations should be returned

Returns all relations of the given kinds in the database

Return type list

This method returns the list of relations in the connected database. Although you can do this with a simple select, it is added here for convenience. You can select which kinds of relations you are interested in by passing type letters in the *kinds* parameter. The type letters are `r` = ordinary table, `i` = index, `S` = sequence, `v` = view, `c` = composite type, `s` = special, `t` = TOAST table. If *kinds* is None or an empty string, all relations are returned (this is also the default). If *system* is set to *True*, then system tables and views (temporary tables, toast tables, catalog vies and tables) will be returned as well, otherwise they will be ignored.

get_tables – get list of tables in connected database

DB.**get_tables**([*system*])
Get the list of tables in connected database

Parameters **system** (*bool*) – whether system tables should be returned

Returns all tables in connected database

Return type list

This is a shortcut for `get_relations('r', system)` that has been added for convenience.

get_attnames – get the attribute names of a table

DB.**get_attnames**(*table*)
Get the attribute names of a table

Parameters **table** (*str*) – name of table

Returns a dictionary mapping attribute names to type names

Given the name of a table, digs out the set of attribute names.

Returns a dictionary of attribute names (the names are the keys, the values are the names of the attributes' types).

By default, only a limited number of simple types will be returned. You can get the regular types after enabling this by calling the *DB.use_regtypes()* method.

get/set_parameter – get or set run-time parameters

DB.**get_parameter**(*parameter*)
Get the value of run-time parameters

Parameters **parameter** – the run-time parameter(s) to get

Returns the current value(s) of the run-time parameter(s)

Return type str, list or dict

Raises

- **TypeError** – Invalid parameter type(s)
- **pg.ProgrammingError** – Invalid parameter name(s)

If the parameter is a string, the return value will also be a string that is the current setting of the run-time parameter with that name.

You can get several parameters at once by passing a list, set or dict. When passing a list of parameter names, the return value will be a corresponding list of parameter settings. When passing a set of parameter names, a new dict will be returned, mapping these parameter names to their settings. Finally, if you pass a dict as parameter, its values will be set to the current parameter settings corresponding to its keys.

By passing the special name *'all'* as the parameter, you can get a dict of all existing configuration parameters.

New in version 4.2.

DB.**set_parameter**(*parameter*[, *value*][, *local*])
Set the value of run-time parameters

Parameters

- **parameter** – the run-time parameter(s) to set
- **value** – the value to set

Raises

- **TypeError** – Invalid parameter type(s)
- **ValueError** – Invalid value argument(s)
- **pg.ProgrammingError** – Invalid parameter name(s) or values

If the parameter and the value are strings, the run-time parameter will be set to that value. If no value or *None* is passed as a value, then the run-time parameter will be restored to its default value.

You can set several parameters at once by passing a list of parameter names, together with a single value that all parameters should be set to or with a corresponding list of values. You can also pass the parameters as a set if you only provide a single value. Finally, you can pass a dict with parameter names as keys. In this case, you should not pass a value, since the values for the parameters will be taken from the dict.

By passing the special name *'all'* as the parameter, you can reset all existing settable run-time parameters to their default values.

If you set *local* to *True*, then the command takes effect for only the current transaction. After `DB.commit()` or `DB.rollback()`, the session-level setting takes effect again. Setting *local* to *True* will appear to have no effect if it is executed outside a transaction, since the transaction will end immediately.

New in version 4.2.

has_table_privilege – check table privilege

DB.**has_table_privilege**(*table*, *privilege*)

Check whether current user has specified table privilege

Parameters

- **table** (*str*) – the name of the table
- **privilege** (*str*) – privilege to be checked – default is 'select'

Returns whether current user has specified table privilege

Return type bool

Returns True if the current user has the specified privilege for the table.

New in version 4.0.

begin/commit/rollback/savepoint/release – transaction handling

DB.**begin**([*mode*])

Begin a transaction

Parameters **mode** (*str*) – an optional transaction mode such as 'READ ONLY'

This initiates a transaction block, that is, all following queries will be executed in a single transaction until `DB.commit()` or `DB.rollback()` is called.

New in version 4.1.

DB.**start**()

This is the same as the `DB.begin()` method.

DB.**commit**()

Commit a transaction

This commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

DB.**end**()

This is the same as the `DB.commit()` method.

New in version 4.1.

DB.**rollback**([*name*])
Roll back a transaction

Parameters **name** (*str*) – optionally, roll back to the specified savepoint

This rolls back the current transaction and causes all the updates made by the transaction to be discarded.

New in version 4.1.

DB.**abort**()
This is the same as the *DB.rollback()* method.

New in version 4.2.

DB.**savepoint**(*name*)
Define a new savepoint

Parameters **name** (*str*) – the name to give to the new savepoint

This establishes a new savepoint within the current transaction.

New in version 4.1.

DB.**release**(*name*)
Destroy a savepoint

Parameters **name** (*str*) – the name of the savepoint to destroy

This destroys a savepoint previously defined in the current transaction.

New in version 4.1.

get – get a row from a database table or view

DB.**get**(*table*, *arg*[, *keyname*])
Get a row from a database table or view

Parameters

- **table** (*str*) – name of table or view
- **arg** – either a dictionary or the value to be looked up
- **keyname** (*str*) – name of field to use as key (optional)

Returns A dictionary - the keys are the attribute names, the values are the row values.

Raises **pg.ProgrammingError** – no primary key or missing privilege

This method is the basic mechanism to get a single row. It assumes that the key specifies a unique row. If *keyname* is not specified, then the primary key for the table is used. If *arg* is a dictionary then the value for the key is taken from it and it is modified to include the new values, replacing existing values where necessary. For a composite key, *keyname* can also be a sequence of key names. The OID is also put into the dictionary if the table has one, but in order to allow the caller to work with multiple tables, it is munged as `oid(schema.table)`.

insert – insert a row into a database table

DB.**insert**(*table*[, *d*][, *key*=*val*, ...])
Insert a row into a database table

Parameters

- **table** (*str*) – name of table

- **d** (*dict*) – optional dictionary of values

Returns the inserted values in the database

Return type dict

Raises **pg.ProgrammingError** – missing privilege or conflict

This method inserts a row into a table. If the optional dictionary is not supplied then the required values must be included as keyword/value pairs. If a dictionary is supplied then any keywords provided will be added to or replace the entry in the dictionary.

The dictionary is then, if possible, reloaded with the values actually inserted in order to pick up values modified by rules, triggers, etc.

update – update a row in a database table

DB.**update**(*table*[, *d*][, *key*=*val*, ...])
Update a row in a database table

Parameters

- **table** (*str*) – name of table
- **d** (*dict*) – optional dictionary of values

Returns the new row in the database

Return type dict

Raises **pg.ProgrammingError** – no primary key or missing privilege

Similar to insert but updates an existing row. The update is based on the OID value as munged by *DB.get()* or passed as keyword, or on the primary key of the table. The dictionary is modified, if possible, to reflect any changes caused by the update due to triggers, rules, default values, etc.

Like insert, the dictionary is optional and updates will be performed on the fields in the keywords. There must be an OID or primary key either in the dictionary where the OID must be munged, or in the keywords where it can be simply the string `'oid'`.

query – execute a SQL command string

DB.**query**(*command*[, *arg1*[, *arg2*, ...]])
Execute a SQL command string

Parameters

- **command** (*str*) – SQL command
- **arg*** – optional positional arguments

Returns result values

Return type *pgqueryobject*, None

Raises

- **TypeError** – bad argument type, or too many arguments
- **TypeError** – invalid connection
- **ValueError** – empty SQL query or lost connection

- **pg.ProgrammingError** – error in query
- **pg.InternalError** – error during query processing

Similar to the *pgobject* function with the same name, except that positional arguments can be passed either as a single list or tuple, or as individual positional arguments.

Example:

```
name = raw_input("Name? ")
phone = raw_input("Phone? ")
rows = db.query("update employees set phone=$2 where name=$1",
    (name, phone)).getresult()[0][0]
# or
rows = db.query("update employees set phone=$2 where name=$1",
     name, phone).getresult()[0][0]
```

clear – clear row values in memory

DB.**clear**(*table*[, *d*])
Clear row values in memory

Parameters

- **table** (*str*) – name of table
- **d** (*dict*) – optional dictionary of values

Returns an empty row

Return type dict

This method clears all the attributes to values determined by the types. Numeric types are set to 0, Booleans are set to `'f'`, and everything else is set to the empty string. If the optional dictionary is present, it is used as the row and any entries matching attribute names are cleared with everything else left unchanged.

If the dictionary is not supplied a new one is created.

delete – delete a row from a database table

DB.**delete**(*table*[, *d*][, *key*=*val*, ...])
Delete a row from a database table

Parameters

- **table** (*str*) – name of table
- **d** (*dict*) – optional dictionary of values

Return type None

Raises **pg.ProgrammingError** – table has no primary key, row is still referenced or missing privilege

This method deletes the row from a table. It deletes based on the OID value as munged by *DB.get()* or passed as keyword, or on the primary key of the table. The return value is the number of deleted rows (i.e. 0 if the row did not exist and 1 if the row was deleted).

truncate – quickly empty database tables

DB.**truncate**(*table*[, *restart*][, *cascade*][, *only*])
Empty a table or set of tables

Parameters

- **table** (*str, list or set*) – the name of the table(s)
- **restart** (*bool*) – whether table sequences should be restarted
- **cascade** (*bool*) – whether referenced tables should also be truncated
- **only** (*bool or list*) – whether only parent tables should be truncated

This method quickly removes all rows from the given table or set of tables. It has the same effect as an unqualified DELETE on each table, but since it does not actually scan the tables it is faster. Furthermore, it reclaims disk space immediately, rather than requiring a subsequent VACUUM operation. This is most useful on large tables.

If *restart* is set to *True*, sequences owned by columns of the truncated table(s) are automatically restarted. If *cascade* is set to *True*, it also truncates all tables that have foreign-key references to any of the named tables. If the parameter *only* is not set to *True*, all the descendant tables (if any) will also be truncated. Optionally, a * can be specified after the table name to explicitly indicate that descendant tables are included. If the parameter *table* is a list, the parameter *only* can also be a list of corresponding boolean values.

New in version 4.2.

escape_literal/identifier/string/bytea – escape for SQL

The following methods escape text or binary strings so that they can be inserted directly into an SQL command. Except for `DB.escape_byte()`, you don't need to call these methods for the strings passed as parameters to *DB.query()*. You also don't need to call any of these methods when storing data using *DB.insert()* and similar.

DB.**escape_literal**(*string*)
Escape a string for use within SQL as a literal constant

Parameters **string** (*str*) – the string that is to be escaped

Returns the escaped string

Return type str

This method escapes a string for use within an SQL command. This is useful when inserting data values as literal constants in SQL commands. Certain characters (such as quotes and backslashes) must be escaped to prevent them from being interpreted specially by the SQL parser.

New in version 4.1.

DB.**escape_identifier**(*string*)
Escape a string for use within SQL as an identifier

Parameters **string** (*str*) – the string that is to be escaped

Returns the escaped string

Return type str

This method escapes a string for use as an SQL identifier, such as a table, column, or function name. This is useful when a user-supplied identifier might contain special characters that would otherwise not be interpreted as part of the identifier by the SQL parser, or when the identifier might contain upper case characters whose case should be preserved.

New in version 4.1.

DB.**escape_bytea**(*datastring*)
Escape binary data for use within SQL as type bytea

Parameters **datastring** (*str*) – string containing the binary data that is to be escaped

Returns the escaped string

Return type str

Similar to the module function *pg.escape_string()* with the same name, but the behavior of this method is adjusted depending on the connection properties (such as character encoding).

unescape_bytea – unescape data retrieved from the database

DB.**unescape_bytea**(*string*)
Unescape bytea data that has been retrieved as text

Parameters **datastring** – the bytea data string that has been retrieved as text

Returns byte string containing the binary data

Return type str

See the module function *pg.unescape_bytea()* with the same name.

use_regtypes – determine use of regular type names

DB.**use_regtypes**([*regtypes*])
Determine whether regular type names shall be used

Parameters **regtypes** (*bool*) – if passed, set whether regular type names shall be used

Returns whether regular type names are used

The *DB.get_attnames()* method can return either simplified "classic" type names (the default) or more specific "regular" type names. Which kind of type names is used can be changed by calling DB.get_regtypes(). If you pass a boolean, it sets whether regular type names shall be used. The method can also be used to check through its return value whether currently regular type names are used.

New in version 4.1.

notification_handler – create a notification handler

class DB.**notification_handler**(*event*, *callback*[, *arg_dict*][, *timeout*][, *stop_event*])
Create a notification handler instance

Parameters

- **event** (*str*) – the name of an event to listen for
- **callback** – a callback function
- **arg_dict** (*dict*) – an optional dictionary for passing arguments
- **timeout** (*int, float or None*) – the time-out when waiting for notifications
- **stop_event** (*str*) – an optional different name to be used as stop event

This method creates a *pg.NotificationHandler* object using the *DB* connection as explained under *The Notification Handler*.

New in version 4.1.1.

pgqueryobject methods

class pg.**pgqueryobject**

The *pgqueryobject* returned by *pgobject.query()* and *DB.query()* provides the following methods for accessing the results of the query:

getresult – get query values as list of tuples

pgqueryobject.**getresult**()
Get query values as list of tuples

Returns result values as a list of tuples

Return type list

Raises

- **TypeError** – too many (any) parameters
- **MemoryError** – internal memory error

This method returns the list of the values returned by the query. More information about this result may be accessed using *pgqueryobject.listfields()*, *pgqueryobject.fieldname()* and *pgqueryobject.fieldnum()* methods.

dictresult – get query values as list of dictionaries

pgqueryobject.**dictresult**()
Get query values as list of dictionaries

Returns result values as a list of dictionaries

Return type list

Raises

- **TypeError** – too many (any) parameters
- **MemoryError** – internal memory error

This method returns the list of the values returned by the query with each tuple returned as a dictionary with the field names used as the dictionary index.

namedresult – get query values as list of named tuples

pgqueryobject.**namedresult**()
Get query values as list of named tuples

Returns result values as a list of named tuples

Return type list

Raises

- **TypeError** – too many (any) parameters
- **TypeError** – named tuples not supported
- **MemoryError** – internal memory error

This method returns the list of the values returned by the query with each row returned as a named tuple with proper field names.

New in version 4.1.

listfields – list fields names of previous query result

pgqueryobject.**listfields**()
List fields names of previous query result

Returns field names

Return type list

Raises **TypeError** – too many parameters

This method returns the list of names of the fields defined for the query result. The fields are in the same order as the result values.

fieldname, fieldnum – field name/number conversion

pgqueryobject.**fieldname**(*num*)
Get field name from its number

Parameters **num** (*int*) – field number

Returns field name

Return type str

Raises

- **TypeError** – invalid connection, bad parameter type, or too many parameters
- **ValueError** – invalid field number

This method allows to find a field name from its rank number. It can be useful for displaying a result. The fields are in the same order as the result values.

pgqueryobject.**fieldnum**(*name*)
Get field number from its name

Parameters **name** (*str*) – field name

Returns field number

Return type int

Raises

- **TypeError** – invalid connection, bad parameter type, or too many parameters
- **ValueError** – unknown field name

This method returns a field number from its name. It can be used to build a function that converts result list strings to their correct type, using a hardcoded table definition. The number returned is the field rank in the result values list.

ntuples – return number of tuples in query object

pgqueryobject.**ntuples**()
Return number of tuples in query object

Returns number of tuples in *pgqueryobject*

Return type int

Raises **TypeError** – Too many arguments.

This method returns the number of tuples found in a query.

pglarge – Large Objects

class pg.**pglarge**

Objects that are instances of the class *pglarge* are used to handle all the requests concerning a PostgreSQL large object. These objects embed and hide all the "recurrent" variables (object OID and connection), exactly in the same way *pgobject* instances do, thus only keeping significant parameters in function calls. The *pglarge* object keeps a reference to the *pgobject* used for its creation, sending requests though with its parameters. Any modification but dereferencing the *pgobject* will thus affect the *pglarge* object. Dereferencing the initial *pgobject* is not a problem since Python won't deallocate it before the *pglarge* object dereferences it. All functions return a generic error message on call error, whatever the exact error was. The error attribute of the object allows to get the exact error message.

See also the PostgreSQL programmer's guide for more information about the large object interface.

open – open a large object

pglarge.**open**(*mode*)
Open a large object

Parameters **mode** (*int*) – open mode definition

Return type None

Raises

- **TypeError** – invalid connection, bad parameter type, or too many parameters
- **IOError** – already opened object, or open error

This method opens a large object for reading/writing, in the same way than the Unix open() function. The mode value can be obtained by OR-ing the constants defined in the *pg* module (INV_READ, INV_WRITE).

close – close a large object

pglarge.**close**()
Close a large object

Return type None

Raises

- **TypeError** – invalid connection
- **TypeError** – too many parameters

- **IOError** – object is not opened, or close error

This method closes a previously opened large object, in the same way than the Unix close() function.

read, write, tell, seek, unlink – file-like large object handling

pglarge.**read**(*size*)
Read data from large object

Parameters **size** (*int*) – maximal size of the buffer to be read

Returns the read buffer

Return type str

Raises

- **TypeError** – invalid connection, invalid object, bad parameter type, or too many parameters
- **ValueError** – if *size* is negative
- **IOError** – object is not opened, or read error

This function allows to read data from a large object, starting at current position.

pglarge.**write**(*string*)
Read data to large object

Parameters **string** (*str*) – string buffer to be written

Return type None

Raises

- **TypeError** – invalid connection, bad parameter type, or too many parameters
- **IOError** – object is not opened, or write error

This function allows to write data to a large object, starting at current position.

pglarge.**seek**(*offset*, *whence*)
Change current position in large object

Parameters

- **offset** (*int*) – position offset
- **whence** (*int*) – positional parameter

Returns new position in object

Return type int

Raises

- **TypeError** – invalid connection or invalid object, bad parameter type, or too many parameters
- **IOError** – object is not opened, or seek error

This method allows to move the position cursor in the large object. The valid values for the whence parameter are defined as constants in the *pg* module (SEEK_SET, SEEK_CUR, SEEK_END).

pglarge.**tell**()
Return current position in large object

Returns current position in large object

Return type int

Raises

- **TypeError** – invalid connection or invalid object
- **TypeError** – too many parameters
- **IOError** – object is not opened, or seek error

This method allows to get the current position in the large object.

pglarge.**unlink**()
Delete large object

Return type None

Raises

- **TypeError** – invalid connection or invalid object
- **TypeError** – too many parameters
- **IOError** – object is not closed, or unlink error

This methods unlinks (deletes) the PostgreSQL large object.

size – get the large object size

pglarge.**size**()
Return the large object size

Returns the large object size

Return type int

Raises

- **TypeError** – invalid connection or invalid object
- **TypeError** – too many parameters
- **IOError** – object is not opened, or seek/tell error

This (composite) method allows to get the size of a large object. It was implemented because this function is very useful for a web interfaced database. Currently, the large object needs to be opened first.

export – save a large object to a file

pglarge.**export**(*name*)
Export a large object to a file

Parameters **name** (*str*) – file to be created

Return type None

Raises

- **TypeError** – invalid connection or invalid object, bad parameter type, or too many parameters
- **IOError** – object is not closed, or export error

This methods allows to dump the content of a large object in a very simple way. The exported file is created on the host of the program, not the server host.

Object attributes

pglarge objects define a read-only set of attributes that allow to get some information about it. These attributes are:

pglarge.**oid**
: the OID associated with the object (int)

pglarge.**pgcnx**
: the *pgobject* associated with the object

pglarge.**error**
: the last warning/error message of the connection

Warning: In multi-threaded environments, *pglarge.error* may be modified by another thread using the same *pgobject*. Remember these object are shared, not duplicated. You should provide some locking to be able if you want to check this. The *pglarge.oid* attribute is very interesting, because it allows you to reuse the OID later, creating the *pglarge* object with a *pgobject.getlo()* method call.

The Notification Handler

PyGreSQL comes with a client-side asynchronous notification handler that was based on the pgnotify module written by Ng Pheng Siong.

New in version 4.1.1.

Instantiating the notification handler

class pg.**NotificationHandler**(*db*, *event*, *callback*[, *arg_dict*][, *timeout*][, *stop_event*])
: Create an instance of the notification handler

Parameters

- **db** (Connection) – the database connection
- **event** (*str*) – the name of an event to listen for
- **callback** – a callback function
- **arg_dict** (*dict*) – an optional dictionary for passing arguments
- **timeout** (*int, float or None*) – the time-out when waiting for notifications
- **stop_event** (*str*) – an optional different name to be used as stop event

You can also create an instance of the NotificationHandler using the DB.connection_handler method. In this case you don't need to pass a database connection because the *DB* connection itself will be used as the datebase connection for the notification handler.

You must always pass the name of an *event* (notification channel) to listen for and a *callback* function.

You can also specify a dictionary *arg_dict* that will be passed as the single argument to the callback function, and a *timeout* value in seconds (a floating point number denotes fractions of seconds). If it is absent or *None*, the callers will

never time out. If the time-out is reached, the callback function will be called with a single argument that is *None*. If you set the *timeout* to `0`, the handler will poll notifications synchronously and return.

You can specify the name of the event that will be used to signal the handler to stop listening as *stop_event*. By default, it will be the event name prefixed with `'stop_'`.

All of the parameters will be also available as attributes of the created notification handler object.

Invoking the notification handler

To invoke the notification handler, just call the instance without passing any parameters.

The handler is a loop that listens for notifications on the event and stop event channels. When either of these notifications are received, its associated *pid*, *event* and *extra* (the payload passed with the notification) are inserted into its *arg_dict* dictionary and the callback is invoked with this dictionary as a single argument. When the handler receives a stop event, it stops listening to both events and return.

In the special case that the timeout of the handler has been set to `0`, the handler will poll all events synchronously and return. If will keep listening until it receives a stop event.

Warning: If you run this loop in another thread, don't use the same database connection for database operations in the main thread.

Sending notifications

You can send notifications by either running `NOTIFY` commands on the database directly, or using the following method:

`NotificationHandler.`**`notify`**`(`[*db*][, *stop*][, *payload*]`)`
Generate a notification

Parameters

- **db** (`Connection`) – the database connection for sending the notification
- **stop** (*bool*) – whether to produce a normal event or a stop event
- **payload** (*str*) – an optional payload to be sent with the notification

This method sends a notification event together with an optional *payload*. If you set the *stop* flag, a stop notification will be sent instead of a normal notification. This will cause the handler to stop listening.

Warning: If the notification handler is running in another thread, you must pass a different database connection since PyGreSQL database connections are not thread-safe.

Auxiliary methods

`NotificationHandler.`**`listen`**`()`
Start listening for the event and the stop event

This method is called implicitly when the handler is invoked.

`NotificationHandler.`**`unlisten`**`()`
Stop listening for the event and the stop event

This method is called implicitly when the handler receives a stop event or when it is closed or deleted.

NotificationHandler.**close**()
Stop listening and close the database connection

You can call this method instead of *NotificationHandler.unlisten()* if you want to close not only the handler, but also the database connection it was created with.

5.1.6 pgdb — The DB-API Compliant Interface

Contents

Introduction

You may either choose to use the "classic" PyGreSQL interface provided by the *pg* module or else the newer DB-API 2.0 compliant interface provided by the *pgdb* module.

The following part of the documentation covers only the newer *pgdb* API.

DB-API 2.0 (Python Database API Specification v2.0) is a specification for connecting to databases (not only PostGreSQL) from Python that has been developed by the Python DB-SIG in 1999. The authoritative programming information for the DB-API is **PEP 0249**.

See also:

A useful tutorial-like introduction to the DB-API has been written by Andrew M. Kuchling for the LINUX Journal in 1998.

Module functions and constants

The *pgdb* module defines a *connect()* function that allows to connect to a database, some global constants describing the capabilities of the module as well as several exception classes.

connect – Open a PostgreSQL connection

pgdb.**connect**([*dsn*][, *user*][, *password*][, *host*][, *database*])
Return a new connection to the database

Parameters

- **dsn** (*str*) – data source name as string
- **user** (*str*) – the database user name
- **password** (*str*) – the database password
- **host** (*str*) – the hostname of the database
- **database** – the name of the database

Returns a connection object

Return type *pgdbCnx*

Raises ***pgdb.OperationalError*** – error connecting to the database

This function takes parameters specifying how to connect to a PostgreSQL database and returns a *pgdbCnx* object using these parameters. If specified, the *dsn* parameter must be a string with the format `'host:base:user:passwd:opt:tty'`. All of the parts specified in the *dsn* are optional. You can also specify the parameters individually using keyword arguments, which always take precedence. The *host* can also contain a port if specified in the format `'host:port'`. In the *opt* part of the *dsn* you can pass command-line options to the server, the *tty* part is used to send server debug output.

Example:

```
con = connect(dsn='myhost:mydb', user='guido', password='234$')
```

Module constants

`pgdb.`**`apilevel`**
The string constant `'2.0'`, stating that the module is DB-API 2.0 level compliant.

`pgdb.`**`threadsafety`**
The integer constant 1, stating that the module itself is thread-safe, but the connections are not thread-safe, and therefore must be protected with a lock if you want to use them from different threads.

`pgdb.`**`paramstyle`**
The string constant `pyformat`, stating that parameters should be passed using Python extended format codes, e.g. `" ... WHERE name=%(name)s"`.

Errors raised by this module

The errors that can be raised by the *pgdb* module are the following:

exception `pgdb.`**`Warning`**
Exception raised for important warnings like data truncations while inserting.

exception `pgdb.`**`Error`**
Exception that is the base class of all other error exceptions. You can use this to catch all errors with one single except statement. Warnings are not considered errors and thus do not use this class as base.

exception `pgdb.`**`InterfaceError`**
Exception raised for errors that are related to the database interface rather than the database itself.

exception `pgdb.`**`DatabaseError`**
Exception raised for errors that are related to the database.

In PyGreSQL, this also has a `DatabaseError.sqlstate` attribute that contains the `SQLSTATE` error code of this error.

exception `pgdb.`**`DataError`**
Exception raised for errors that are due to problems with the processed data like division by zero or numeric value out of range.

exception `pgdb.`**`OperationalError`**
Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, or a memory allocation error occurred during processing.

exception `pgdb.`**`IntegrityError`**
Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails.

exception pgdb.**ProgrammingError**
Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement or wrong number of parameters specified.

exception pgdb.**NotSupportedError**
Exception raised in case a method or database API was used which is not supported by the database.

pgdbCnx – The connection object

class pgdb.**pgdbCnx**

These connection objects respond to the following methods.

Note that pgdb.pgdbCnx objects also implement the context manager protocol, i.e. you can use them in a with statement.

close – close the connection

pgdbCnx.**close**()
Close the connection now (rather than whenever it is deleted)

Return type None

The connection will be unusable from this point forward; an *Error* (or subclass) exception will be raised if any operation is attempted with the connection. The same applies to all cursor objects trying to use the connection. Note that closing a connection without committing the changes first will cause an implicit rollback to be performed.

commit – commit the connection

pgdbCnx.**commit**()
Commit any pending transaction to the database

Return type None

Note that connections always use a transaction, there is no auto-commit.

rollback – roll back the connection

pgdbCnx.**rollback**()
Roll back any pending transaction to the database

Return type None

This method causes the database to roll back to the start of any pending transaction. Closing a connection without committing the changes first will cause an implicit rollback to be performed.

cursor – return a new cursor object

pgdbCnx.**cursor**()
Return a new cursor object using the connection

Returns a connection object

Return type *pgdbCursor*

This method returns a new *pgdbCursor* object that can be used to operate on the database in the way described in the next section.

pgdbCursor – The cursor object

class pgdb.**pgdbCursor**

These objects represent a database cursor, which is used to manage the context of a fetch operation. Cursors created from the same connection are not isolated, i.e., any changes done to the database by a cursor are immediately visible by the other cursors. Cursors created from different connections can or can not be isolated, depending on the level of transaction isolation. The default PostgreSQL transaction isolation level is "read committed".

Cursor objects respond to the following methods and attributes.

Note that pgdbCursor objects also implement both the iterator and the context manager protocol, i.e. you can iterate over them and you can use them in a with statement.

description – details regarding the result columns

pgdbCursor.**description**

This read-only attribute is a sequence of 7-item tuples.

Each of these tuples contains information describing one result column:

- *name*
- *type_code*
- *display_size*
- *internal_size*
- *precision*
- *scale*
- *null_ok*

Note that *display_size*, *precision*, *scale* and *null_ok* are not implemented.

This attribute will be None for operations that do not return rows or if the cursor has not had an operation invoked via the *pgdbCursor.execute()* or *pgdbCursor.executemany()* method yet.

rowcount – number of rows of the result

pgdbCursor.**rowcount**

This read-only attribute specifies the number of rows that the last *pgdbCursor.execute()* or *pgdbCursor.executemany()* call produced (for DQL statements like SELECT) or affected (for DML statements like UPDATE or INSERT). The attribute is -1 in case no such method call has been performed on the cursor or the rowcount of the last operation cannot be determined by the interface.

close – close the cursor

pgdbCursor.**close**()

Close the cursor now (rather than whenever it is deleted)

Return type None

The cursor will be unusable from this point forward; an *Error* (or subclass) exception will be raised if any operation is attempted with the cursor.

execute – execute a database operation

pgdbCursor.**execute**(*operation*[, *parameters*])
Prepare and execute a database operation (query or command)

Parameters

- **operation** (*str*) – the database operation
- **parameters** – a sequence or mapping of parameters

Returns the cursor, so you can chain commands

Parameters may be provided as sequence or mapping and will be bound to variables in the operation. Variables are specified using Python extended format codes, e.g. `" ... WHERE name=%(name)s"`.

A reference to the operation will be retained by the cursor. If the same operation object is passed in again, then the cursor can optimize its behavior. This is most effective for algorithms where the same operation is used, but different parameters are bound to it (many times).

The parameters may also be specified as list of tuples to e.g. insert multiple rows in a single operation, but this kind of usage is deprecated: *pgdbCursor.executemany()* should be used instead.

Note that in case this method raises a *DatabaseError*, you can get information about the error condition that has occurred by introspecting its `DatabaseError.sqlstate` attribute, which will be the `SQLSTATE` error code associated with the error. Applications that need to know which error condition has occurred should usually test the error code, rather than looking at the textual error message.

executemany – execute many similar database operations

pgdbCursor.**executemany**(*operation*[, *seq_of_parameters*])
Prepare and execute many similar database operations (queries or commands)

Parameters

- **operation** (*str*) – the database operation
- **seq_of_parameters** – a sequence or mapping of parameter tuples or mappings

Returns the cursor, so you can chain commands

Prepare a database operation (query or command) and then execute it against all parameter tuples or mappings found in the sequence *seq_of_parameters*.

Parameters are bounded to the query using Python extended format codes, e.g. `" ... WHERE name=%(name)s"`.

fetchone – fetch next row of the query result

pgdbCursor.**fetchone**()
Fetch the next row of a query result set

Returns the next row of the query result set

Return type list or None

Fetch the next row of a query result set, returning a single list, or None when no more data is available.

An *Error* (or subclass) exception is raised if the previous call to *pgdbCursor.execute()* or *pgdbCursor.executemany()* did not produce any result set or no call was issued yet.

fetchmany – fetch next set of rows of the query result

pgdbCursor.**fetchmany**([*size=None*][, *keep=False*])
Fetch the next set of rows of a query result

Parameters

- **size** (*int or None*) – the number of rows to be fetched
- **keep** – if set to true, will keep the passed arraysize

Tpye keep bool

Returns the next set of rows of the query result

Return type list of lists

Fetch the next set of rows of a query result, returning a list of lists. An empty sequence is returned when no more rows are available.

The number of rows to fetch per call is specified by the *size* parameter. If it is not given, the cursor's arraysize determines the number of rows to be fetched. If you set the *keep* parameter to True, this is kept as new arraysize.

The method tries to fetch as many rows as indicated by the *size* parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.

An *Error* (or subclass) exception is raised if the previous call to *pgdbCursor.execute()* or *pgdbCursor.executemany()* did not produce any result set or no call was issued yet.

Note there are performance considerations involved with the *size* parameter. For optimal performance, it is usually best to use the arraysize attribute. If the *size* parameter is used, then it is best for it to retain the same value from one *pgdbCursor.fetchmany()* call to the next.

fetchall – fetch all rows of the query result

pgdbCursor.**fetchall**()
Fetch all (remaining) rows of a query result

Returns the set of all rows of the query result

Return type list of list

Fetch all (remaining) rows of a query result, returning them as list of lists. Note that the cursor's arraysize attribute can affect the performance of this operation.

row_factory – process a row of the query result

pgdbCursor.**row_factory**(*row*)
Process rows before they are returned

Parameters **row** (*list*) – the currently processed row of the result set

Returns the transformed row that the fetch methods shall return

Note: This method is not part of the DB-API 2 standard.

You can overwrite this method with a custom row factory, e.g. if you want to return rows as dicts instead of lists:

```
class DictCursor(pgdb.pgdbCursor):

    def row_factory(self, row):
        return dict((d[0], v) for d, v in zip(self.description, row))

cur = DictCursor(con)
```

New in version 4.0.

arraysize - the number of rows to fetch at a time

pgdbCursor.**arraysize**
The number of rows to fetch at a time

This read/write attribute specifies the number of rows to fetch at a time with *pgdbCursor.fetchmany()*. It defaults to 1 meaning to fetch a single row at a time.

pgdbType – Type objects and constructors

Type constructors

For binding to an operation's input parameters, PostgreSQL needs to have the input in a particular format. However, from the parameters to the *pgdbCursor.execute()* and *pgdbCursor.executemany()* methods it is not always obvious as which PostgreSQL data types they shall be bound. For instance, a Python string could be bound as a simple `char` value, or also as a `date` or a `time`. To make the intention clear in such cases, you can wrap the parameters in type helper objects. PyGreSQL provides the constructors defined below to create such objects that can hold special values. When passed to the cursor methods, PyGreSQL can then detect the proper type of the input parameter and bind it accordingly.

The *pgdb* module exports the following type constructors as part of the DB-API 2 standard:

pgdb.**Date**(*year*, *month*, *day*)
Construct an object holding a date value

pgdb.**Time**(*hour*, *minute=0*, *second=0*, *microsecond=0*)
Construct an object holding a time value

pgdb.**Timestamp**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*)
Construct an object holding a time stamp value

pgdb.**DateFromTicks**(*ticks*)
Construct an object holding a date value from the given *ticks* value

pgdb.**TimeFromTicks**(*ticks*)
Construct an object holding a time value from the given *ticks* value

pgdb.**TimestampFromTicks**(*ticks*)
Construct an object holding a time stamp from the given *ticks* value

pgdb.**Binary**(*bytes*)
Construct an object capable of holding a (long) binary string value

Note: SQL NULL values are always represented by the Python *None* singleton on input and output.

Type objects

class pgdb.**pgdbType**

The *pgdbCursor.description* attribute returns information about each of the result columns of a query. The *type_code* must compare equal to one of the *pgdbType* objects defined below. Type objects can be equal to more than one type code (e.g. DATETIME is equal to the type codes for date, time and timestamp columns).

The pgdb module exports the following Type objects as part of the DB-API 2 standard:

STRING
Used to describe columns that are string-based (e.g. char, varchar, text)

BINARY
Used to describe (long) binary columns (bytea)

NUMBER
Used to describe numeric columns (e.g. int, float, numeric, money)

DATETIME
Used to describe date/time columns (e.g. date, time, timestamp, interval)

ROWID
Used to describe the oid column of PostgreSQL database tables

Note: The following more specific type objects are not part of the DB-API 2 standard.

BOOL
Used to describe boolean columns

SMALLINT
Used to describe smallint columns

INTEGER
Used to describe integer columns

LONG
Used to describe bigint columns

FLOAT
Used to describe float columns

NUMERIC
Used to describe numeric columns

MONEY
Used to describe money columns

DATE
Used to describe date columns

TIME
Used to describe time columns

TIMESTAMP
Used to describe timestamp columns

INTERVAL
Used to describe date and time `interval` columns

5.1.7 A PostgreSQL Primer

The examples in this chapter of the documentation have been taken from the PostgreSQL manual. They demonstrate some PostgreSQL features using the classic PyGreSQL interface. They can serve as an introduction to PostgreSQL, but not so much as examples for the use of PyGreSQL.

Contents

Basic examples

In this section, we demonstrate how to use some of the very basic features of PostgreSQL using the classic PyGreSQL interface.

Creating a connection to the database

We start by creating a **connection** to the PostgreSQL database:

```
>>> from pg import DB
>>> db = DB()
```

If you pass no parameters when creating the *DB* instance, then PyGreSQL will try to connect to the database on the local host that has the same name as the current user, and also use that name for login.

You can also pass the database name, host, port and login information as parameters when creating the *DB* instance:

```
>>> db = DB(dbname='testdb', host='pgserver', port=5432,
...     user='scott', passwd='tiger')
```

The *DB* class of which `db` is an object is a wrapper around the lower level *pgobject* class of the *pg* module. The most important method of such connection objects is the `query` method that allows you to send SQL commands to the database.

Creating tables

The first thing you would want to do in an empty database is creating a table. To do this, you need to send a **CREATE TABLE** command to the database. PostgreSQL has its own set of built-in types that can be used for the table columns. Let us create two tables "weather" and "cities":

```
>>> db.query("""CREATE TABLE weather (
...     city varchar(80),
...     temp_lo int, temp_hi int,
...     prcp float8,
...     date date)""")
>>> db.query("""CREATE TABLE cities (
...     name varchar(80),
...     location point)""")
```

Note: Keywords are case-insensitive but identifiers are case-sensitive.

You can get a list of all tables in the database with:

```
>>> db.get_tables()
['public.cities', 'public.weather']
```

Insert data

Now we want to fill our tables with data. An **INSERT** statement is used to insert a new row into a table. There are several ways you can specify what columns the data should go to.

Let us insert a row into each of these tables. The simplest case is when the list of values corresponds to the order of the columns specified in the CREATE TABLE command:

```
>>> db.query("""INSERT INTO weather
...     VALUES ('San Francisco', 46, 50, 0.25, '11/27/1994')""")
>>> db.query("""INSERT INTO cities
...     VALUES ('San Francisco', '(-194.0, 53.0)')""")
```

You can also specify what column the values correspond to. The columns can be specified in any order. You may also omit any number of columns, unknown precipitation below:

```
>>> db.query("""INSERT INTO weather (date, city, temp_hi, temp_lo)
...     VALUES ('11/29/1994', 'Hayward', 54, 37)""")
```

If you get errors regarding the format of the date values, your database is probably set to a different date style. In this case you must change the date style like this:

```
>>> db.query("set datestyle = MDY")
```

Instead of explicitly writing the INSERT statement and sending it to the database with the *DB.query()* method, you can also use the more convenient *DB.insert()* method that does the same under the hood:

```
>>> db.insert('weather',
...     date='11/29/1994', city='Hayward', temp_hi=54, temp_lo=37)
```

And instead of using keyword parameters, you can also pass the values to the *DB.insert()* method in a single Python dictionary.

If you have a Python list with many rows that shall be used to fill a database table quickly, you can use the `DB.inserttable()` method.

Retrieving data

After having entered some data into our tables, let's see how we can get the data out again. A **SELECT** statement is used for retrieving data. The basic syntax is:

```
SELECT columns FROM tables WHERE predicates
```

A simple one would be the following query:

```
>>> q = db.query("SELECT * FROM weather")
>>> print q
    city     |temp_lo|temp_hi|prcp|   date
-------------+-------+-------+----+----------
San Francisco|     46|     50|0.25|1994-11-27
Hayward      |     37|     54|    |1994-11-29
(2 rows)
```

You may also specify expressions in the target list. (The 'AS column' specifies the column name of the result. It is optional.)

```
>>> print db.query("""SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date
...     FROM weather""")
    city     |temp_avg|   date
-------------+--------+----------
San Francisco|      48|1994-11-27
Hayward      |      45|1994-11-29
(2 rows)
```

If you want to retrieve rows that satisfy certain condition (i.e. a restriction), specify the condition in a WHERE clause. The following retrieves the weather of San Francisco on rainy days:

```
>>> print db.query("""SELECT * FROM weather
...     WHERE city = 'San Francisco' AND prcp > 0.0""")
    city     |temp_lo|temp_hi|prcp|   date
-------------+-------+-------+----+----------
San Francisco|     46|     50|0.25|1994-11-27
(1 row)
```

Here is a more complicated one. Duplicates are removed when DISTINCT is specified. ORDER BY specifies the column to sort on. (Just to make sure the following won't confuse you, DISTINCT and ORDER BY can be used separately.)

```
>>> print db.query("SELECT DISTINCT city FROM weather ORDER BY city")
    city
-------------
Hayward
San Francisco
(2 rows)
```

So far we have only printed the output of a SELECT query. The object that is returned by the query is an instance of the *pgqueryobject* class that can print itself in the nicely formatted way we saw above. But you can also retrieve the results as a list of tuples, by using the *pgqueryobject.getresult()* method:

```
>>> from pprint import pprint
>>> q = db.query("SELECT * FROM weather")
>>> pprint(q.getresult())
[('San Francisco', 46, 50, 0.25, '1994-11-27'),
 ('Hayward', 37, 54, None, '1994-11-29')]
```

Here we used pprint to print out the returned list in a nicely formatted way.

If you want to retrieve the results as a list of dictionaries instead of tuples, use the *pgqueryobject.dictresult()* method instead:

```
>>> pprint(q.dictresult())
[{'city': 'San Francisco',
  'date': '1994-11-27',
```

```
  'prcp': 0.25,
  'temp_hi': 50,
  'temp_lo': 46},
 {'city': 'Hayward',
  'date': '1994-11-29',
  'prcp': None,
  'temp_hi': 54,
  'temp_lo': 37}]
```

Finally, in Python 2.5 and above you can also retrieve the results as a list of named tuples, using the *pgqueryobject.namedresult()* method. This can be a good compromise between simple tuples and the more memory intensive dictionaries:

```
>>> for row in q.namedresult():
...     print row.city, row.date
...
San Francisco 1994-11-27
Hayward 1994-11-29
```

If you only want to retrieve a single row of data, you can use the more convenient *DB.get()* method that does the same under the hood:

```
>>> d = dict(city='Hayward')
>>> db.get('weather', d, 'city')
>>> pprint(d)
{'city': 'Hayward',
 'date': '1994-11-29',
 'prcp': None,
 'temp_hi': 54,
 'temp_lo': 37}
```

As you see, the *DB.get()* method returns a dictionary with the column names as keys. In the third parameter you can specify which column should be looked up in the WHERE statement of the SELECT statement that is executed by the *DB.get()* method. You normally don't need it when the table was created with a primary key.

Retrieving data into other tables

A SELECT ... INTO statement can be used to retrieve data into another table:

```
>>> db.query("""SELECT * INTO TEMPORARY TABLE temptab FROM weather
...     WHERE city = 'San Francisco' and prcp > 0.0""")
```

This fills a temporary table "temptab" with a subset of the data in the original "weather" table. It can be listed with:

```
>>> print db.query("SELECT * from temptab")
    city     |temp_lo|temp_hi|prcp|   date
-------------+-------+-------+----+----------
San Francisco|     46|     50|0.25|1994-11-27
(1 row)
```

Aggregates

Let's try the following query:

```
>>> print db.query("SELECT max(temp_lo) FROM weather")
max
---
 46
(1 row)
```

You can also use aggregates with the GROUP BY clause:

```
>>> print db.query("SELECT city, max(temp_lo) FROM weather GROUP BY city")
    city     |max
-------------+---
Hayward      | 37
San Francisco| 46
(2 rows)
```

Joining tables

Queries can access multiple tables at once or access the same table in such a way that multiple instances of the table are being processed at the same time.

Suppose we want to find all the records that are in the temperature range of other records. W1 and W2 are aliases for weather. We can use the following query to achieve that:

```
>>> print db.query("""SELECT W1.city, W1.temp_lo, W1.temp_hi,
...     W2.city, W2.temp_lo, W2.temp_hi FROM weather W1, weather W2
...     WHERE W1.temp_lo < W2.temp_lo and W1.temp_hi > W2.temp_hi""")
 city  |temp_lo|temp_hi|    city     |temp_lo|temp_hi
-------+-------+-------+-------------+-------+-------
Hayward|     37|     54|San Francisco|     46|     50
(1 row)
```

Now let's join two tables. The following joins the "weather" table and the "cities" table:

```
>>> print db.query("""SELECT city, location, prcp, date FROM weather, cities
...     WHERE name = city""")
    city     |location |prcp|   date
-------------+---------+----+----------
San Francisco|(-194,53)|0.25|1994-11-27
(1 row)
```

Since the column names are all different, we don't have to specify the table name. If you want to be clear, you can do the following. They give identical results, of course:

```
>>> print db.query("""SELECT w.city, c.location, w.prcp, w.date
...     FROM weather w, cities c WHERE c.name = w.city""")
    city     |location |prcp|   date
-------------+---------+----+----------
San Francisco|(-194,53)|0.25|1994-11-27
(1 row)
```

Updating data

It you want to change the data that has already been inserted into a database table, you will need the **UPDATE** statement.

Suppose you discover the temperature readings are all off by 2 degrees as of Nov 28, you may update the data as follow:

```
>>> db.query("""UPDATE weather
...     SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
...     WHERE date > '11/28/1994'""")
'1'
>>> print db.query("SELECT * from weather")
    city     |temp_lo|temp_hi|prcp|   date
-------------+-------+-------+----+----------
San Francisco|     46|     50|0.25|1994-11-27
Hayward      |     35|     52|    |1994-11-29
(2 rows)
```

Note that the UPDATE statement returned the string `'1'`, indicating that exactly one row of data has been affected by the update.

If you retrieved one row of data as a dictionary using the *DB.get()* method, then you can also update that row with the *DB.update()* method.

Deleting data

To delete rows from a table, a **DELETE** statement can be used.

Suppose you are no longer interested in the weather of Hayward, you can do the following to delete those rows from the table:

```
>>> db.query("DELETE FROM weather WHERE city = 'Hayward'")
'1'
```

Again, you get the string `'1'` as return value, indicating that exactly one row of data has been deleted.

You can also delete all the rows in a table by doing the following. This is different from DROP TABLE which removes the table itself in addition to the removing the rows, as explained in the next section.

```
>>> db.query("DELETE FROM weather")
'1'
>>> print db.query("SELECT * from weather")
city|temp_lo|temp_hi|prcp|date
----+-------+-------+----+----
(0 rows)
```

Since only one row was left in the table, the DELETE query again returns the string `'1'`. The SELECT query now gives an empty result.

If you retrieved a row of data as a dictionary using the *DB.get()* method, then you can also delete that row with the *DB.delete()* method.

Removing the tables

The **DROP TABLE** command is used to remove tables. After you have done this, you can no longer use those tables:

```
>>> db.query("DROP TABLE weather, cities")
>>> db.query("select * from weather")
pg.ProgrammingError: Error:  Relation "weather" does not exist
```

Examples for advanced features

In this section, we show how to use some advanced features of PostgreSQL using the classic PyGreSQL interface.

We assume that you have already created a connection to the PostgreSQL database, as explained in the *Basic examples*:

```
>>> from pg import DB
>>> db = DB()
>>> query = query
```

Inheritance

A table can inherit from zero or more tables. A query can reference either all rows of a table or all rows of a table plus all of its descendants.

For example, the capitals table inherits from cities table (it inherits all data fields from cities):

```
>>> data = [('cities', [
...         "'San Francisco', 7.24E+5, 63",
...         "'Las Vegas', 2.583E+5, 2174",
...         "'Mariposa', 1200, 1953"]),
...     ('capitals', [
...         "'Sacramento',3.694E+5,30,'CA'",
...         "'Madison', 1.913E+5, 845, 'WI'"])]
```

Now, let's populate the tables:

```
>>> data = ['cities', [
...         "'San Francisco', 7.24E+5, 63"
...         "'Las Vegas', 2.583E+5, 2174"
...         "'Mariposa', 1200, 1953"],
...     'capitals', [
...         "'Sacramento',3.694E+5,30,'CA'",
...         "'Madison', 1.913E+5, 845, 'WI'"]]
>>> for table, rows in data:
...     for row in rows:
...         query("INSERT INTO %s VALUES (%s)" % (table, row))
>>> print query("SELECT * FROM cities")
    name     |population|altitude
-------------+----------+--------
San Francisco|    724000|      63
Las Vegas    |    258300|    2174
Mariposa     |      1200|    1953
Sacramento   |    369400|      30
Madison      |    191300|     845
(5 rows)
>>> print query("SELECT * FROM capitals")
   name    |population|altitude|state
-----------+----------+--------+-----
Sacramento|    369400|      30|CA
Madison   |    191300|     845|WI
(2 rows)
```

You can find all cities, including capitals, that are located at an altitude of 500 feet or higher by:

```
>>> print query("""SELECT c.name, c.altitude
...     FROM cities
```

```
...     WHERE altitude > 500""")
  name   |altitude
---------+--------
Las Vegas|    2174
Mariposa |    1953
Madison  |     845
(3 rows)
```

On the other hand, the following query references rows of the base table only, i.e. it finds all cities that are not state capitals and are situated at an altitude of 500 feet or higher:

```
>>> print query("""SELECT name, altitude
...     FROM ONLY cities
...     WHERE altitude > 500""")
  name   |altitude
---------+--------
Las Vegas|    2174
Mariposa |    1953
(2 rows)
```

Arrays

Attributes can be arrays of base types or user-defined types:

```
>>> query("""CREATE TABLE sal_emp (
...         name                 text,
...         pay_by_quarter       int4[],
...         pay_by_extra_quarter  int8[],
...         schedule             text[][])""")
```

Insert instances with array attributes. Note the use of braces:

```
>>> query("""INSERT INTO sal_emp VALUES (
...     'Bill', '{10000,10000,10000,10000}',
...     '{9223372036854775800,9223372036854775800,9223372036854775800}',
...     '{{"meeting", "lunch"}, {"training", "presentation"}}')""")
>>> query("""INSERT INTO sal_emp VALUES (
...     'Carol', '{20000,25000,25000,25000}',
...      '{9223372036854775807,9223372036854775807,9223372036854775807}',
...      '{{"breakfast", "consulting"}, {"meeting", "lunch"}}')""")
```

Queries on array attributes:

```
>>> query("""SELECT name FROM sal_emp WHERE
...     sal_emp.pay_by_quarter[1] != sal_emp.pay_by_quarter[2]""")
name
-----
Carol
(1 row)
```

Retrieve third quarter pay of all employees:

```
>>> query("SELECT sal_emp.pay_by_quarter[3] FROM sal_emp")
pay_by_quarter
--------------
         10000
```

```
         25000
(2 rows)
```

Retrieve third quarter extra pay of all employees:

```
>>> query("SELECT sal_emp.pay_by_extra_quarter[3] FROM sal_emp")
pay_by_extra_quarter
--------------------
 9223372036854775800
 9223372036854775807
(2 rows)
```

Retrieve first two quarters of extra quarter pay of all employees:

```
>>> query("SELECT sal_emp.pay_by_extra_quarter[1:2] FROM sal_emp")
          pay_by_extra_quarter
---------------------------------------
{9223372036854775800,9223372036854775800}
{9223372036854775807,9223372036854775807}
(2 rows)
```

Select subarrays:

```
>>> query("""SELECT sal_emp.schedule[1:2][1:1] FROM sal_emp
...     WHERE sal_emp.name = 'Bill'""")
       schedule
----------------------
{{meeting},{training}}
(1 row)
```

Examples for using SQL functions

We assume that you have already created a connection to the PostgreSQL database, as explained in the *Basic examples*:

```
>>> from pg import DB
>>> db = DB()
>>> query = db.query
```

Creating SQL Functions on Base Types

A **CREATE FUNCTION** statement lets you create a new function that can be used in expressions (in SELECT, INSERT, etc.). We will start with functions that return values of base types.

Let's create a simple SQL function that takes no arguments and returns 1:

```
>>> query("""CREATE FUNCTION one() RETURNS int4
...     AS 'SELECT 1 as ONE' LANGUAGE SQL""")
```

Functions can be used in any expressions (eg. in the target"list or qualifications):

```
>>> print db.query("SELECT one() AS answer")
answer
------
     1
(1 row)
```

Here's how you create a function that takes arguments. The following function returns the sum of its two arguments:

```
>>> query("""CREATE FUNCTION add_em(int4, int4) RETURNS int4
...     AS $$ SELECT $1 + $2 $$ LANGUAGE SQL""")
>>> print query("SELECT add_em(1, 2) AS answer")
answer
------
     3
(1 row)
```

Creating SQL Functions on Composite Types

It is also possible to create functions that return values of composite types.

Before we create more sophisticated functions, let's populate an EMP table:

```
>>> query("""CREATE TABLE EMP (
...     name   text,
...     salary int4,
...     age f   int4,
...     dept   varchar(16))""")
>>> emps = ["'Sam', 1200, 16, 'toy'",
...     "'Claire', 5000, 32, 'shoe'",
...     "'Andy', -1000, 2, 'candy'",
...     "'Bill', 4200, 36, 'shoe'",
...     "'Ginger', 4800, 30, 'candy'"]
>>> for emp in emps:
...     query("INSERT INTO EMP VALUES (%s)" % emp)
```

Every INSERT statement will return a '1' indicating that it has inserted one row into the EMP table.

The argument of a function can also be a tuple. For instance, *double_salary* takes a tuple of the EMP table:

```
>>> query("""CREATE FUNCTION double_salary(EMP) RETURNS int4
...     AS $$ SELECT $1.salary * 2 AS salary $$ LANGUAGE SQL""")
>>> print query("""SELECT name, double_salary(EMP) AS dream
...     FROM EMP WHERE EMP.dept = 'toy'""")
name|dream
----+-----
Sam | 2400
(1 row)
```

The return value of a function can also be a tuple. However, make sure that the expressions in the target list are in the same order as the columns of EMP:

```
>>> query("""CREATE FUNCTION new_emp() RETURNS EMP AS $$
...     SELECT 'None'::text AS name,
...         1000 AS salary,
...         25 AS age,
...         'None'::varchar(16) AS dept
...     $$ LANGUAGE SQL""")
```

You can then project a column out of resulting the tuple by using the "function notation" for projection columns (i.e. `bar(foo)` is equivalent to `foo.bar`). Note that `new_emp().name` isn't supported:

```
>>> print query("SELECT name(new_emp()) AS nobody")
nobody
------
None
(1 row)
```

Let's try one more function that returns tuples:

```
>>> query("""CREATE FUNCTION high_pay() RETURNS setof EMP
...          AS 'SELECT * FROM EMP where salary > 1500'
...      LANGUAGE SQL""")
>>> query("SELECT name(high_pay()) AS overpaid")
overpaid
--------
Claire
Bill
Ginger
(3 rows)
```

Creating SQL Functions with multiple SQL statements

You can also create functions that do more than just a SELECT.

You may have noticed that Andy has a negative salary. We'll create a function that removes employees with negative salaries:

```
>>> query("SELECT * FROM EMP")
 name |salary|age|dept
------+------+---+-----
Sam   |  1200| 16|toy
Claire|  5000| 32|shoe
Andy  | -1000|  2|candy
Bill  |  4200| 36|shoe
Ginger|  4800| 30|candy
(5 rows)
>>> query("""CREATE FUNCTION clean_EMP () RETURNS int4 AS
...          'DELETE FROM EMP WHERE EMP.salary <= 0;
...           SELECT 1 AS ignore_this'
...      LANGUAGE SQL""")
>>> query("SELECT clean_EMP()")
clean_emp
---------
        1
(1 row)
>>> query("SELECT * FROM EMP")
 name |salary|age|dept
------+------+---+-----
Sam   |  1200| 16|toy
Claire|  5000| 32|shoe
Bill  |  4200| 36|shoe
Ginger|  4800| 30|candy
(4 rows)
```

Remove functions that were created in this example

We can remove the functions that we have created in this example and the table EMP, by using the DROP command:

```
query("DROP FUNCTION clean_EMP()")
query("DROP FUNCTION high_pay()")
query("DROP FUNCTION new_emp()")
query("DROP FUNCTION add_em(int4, int4)")
query("DROP FUNCTION one()")
query("DROP TABLE EMP CASCADE")
```

Examples for using the system catalogs

The system catalogs are regular tables where PostgreSQL stores schema metadata, such as information about tables and columns, and internal bookkeeping information. You can drop and recreate the tables, add columns, insert and update values, and severely mess up your system that way. Normally, one should not change the system catalogs by hand, there are always SQL commands to do that. For example, CREATE DATABASE inserts a row into the *pg_database* catalog — and actually creates the database on disk.

It this section we want to show examples for how to parse some of the system catalogs, making queries with the classic PyGreSQL interface.

We assume that you have already created a connection to the PostgreSQL database, as explained in the *Basic examples*:

```
>>> from pg import DB
>>> db = DB()
>>> query = query
```

Lists indices

This query lists all simple indices in the database:

```
print query("""SELECT bc.relname AS class_name,
        ic.relname AS index_name, a.attname
    FROM pg_class bc, pg_class ic, pg_index i, pg_attribute a
    WHERE i.indrelid = bc.oid AND i.indexrelid = ic.oid
        AND i.indkey[0] = a.attnum AND a.attrelid = bc.oid
        AND NOT a.attisdropped
    ORDER BY class_name, index_name, attname""")
```

List user defined attributes

This query lists all user defined attributes and their type in user-defined classes:

```
print query("""SELECT c.relname, a.attname, t.typname
    FROM pg_class c, pg_attribute a, pg_type t
    WHERE c.relkind = 'r' and c.relname !~ '^pg_'
        AND c.relname !~ '^Inv' and a.attnum > 0
        AND a.attrelid = c.oid and a.atttypid = t.oid
        AND NOT a.attisdropped
    ORDER BY relname, attname""")
```

List user defined base types

This query lists all user defined base types:

```
print query("""SELECT r.rolname, t.typname
    FROM pg_type t, pg_authid r
    WHERE r.oid = t.typowner
        AND t.typrelid = '0'::oid and t.typelem = '0'::oid
        AND r.rolname != 'postgres'
    ORDER BY rolname, typname""")
```

List operators

This query lists all right-unary operators:

```
print query("""SELECT o.oprname AS right_unary,
        lt.typname AS operand, result.typname AS return_type
    FROM pg_operator o, pg_type lt, pg_type result
    WHERE o.oprkind='r' and o.oprleft = lt.oid
        AND o.oprresult = result.oid
    ORDER BY operand""")
```

This query lists all left-unary operators:

```
print query("""SELECT o.oprname AS left_unary,
        rt.typname AS operand, result.typname AS return_type
    FROM pg_operator o, pg_type rt, pg_type result
    WHERE o.oprkind='l' AND o.oprright = rt.oid
        AND o.oprresult = result.oid
    ORDER BY operand""")
```

And this one lists all of the binary operators:

```
print query("""SELECT o.oprname AS binary_op,
        rt.typname AS right_opr, lt.typname AS left_opr,
        result.typname AS return_type
    FROM pg_operator o, pg_type rt, pg_type lt, pg_type result
    WHERE o.oprkind = 'b' AND o.oprright = rt.oid
        AND o.oprleft = lt.oid AND o.oprresult = result.oid""")
```

List functions of a language

Given a programming language, this query returns the name, args and return type from all functions of a language:

```
language = 'sql'
print query("""SELECT p.proname, p.pronargs, t.typname
    FROM pg_proc p, pg_language l, pg_type t
    WHERE p.prolang = l.oid AND p.prorettype = t.oid
        AND l.lanname = $1
    ORDER BY proname""", (language,))
```

List aggregate functions

This query lists all of the aggregate functions and the type to which they can be applied:

```
print query("""SELECT p.proname, t.typname
    FROM pg_aggregate a, pg_proc p, pg_type t
    WHERE a.aggfnoid = p.oid
        and p.proargtypes[0] = t.oid
    ORDER BY proname, typname""")
```

List operator families

The following query lists all defined operator families and all the operators included in each family:

```
print query("""SELECT am.amname, opf.opfname, amop.amopopr::regoperator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid
        AND amop.amopfamily = opf.oid
    ORDER BY amname, opfname, amopopr""")
```

5.1.8 Examples

I am starting to collect examples of applications that use PyGreSQL. So far I only have a few but if you have an example for me, you can either send me the files or the URL for me to point to.

The *tutorial* directory that is part of the PyGreSQL distribution shows some examples of using PostgreSQL with PyGreSQL.

Here is a list of motorcycle rides in Ontario that uses a PostgreSQL database to store the rides. There is a link at the bottom of the page to view the source code.

Oleg Broytmann has written a simple example RGB database demo

5.2 Indices and tables

- genindex
- modindex
- search

CHAPTER 6

PyGreSQL Development and Support

PyGreSQL is an open-source project created by a group of volunteers. The project and the development infrastructure are currently maintained by D'Arcy J.M. Cain. We would be glad to welcome more contributors so that PyGreSQL can be further developed, modernized and improved.

6.1 Mailing list

You can join the mailing list to discuss future development of the PyGreSQL interface or if you have questions or problems with PyGreSQL that are not covered in the *documentation*.

This is usually a low volume list except when there are new features being added.

6.2 Access to the source repository

We are using a central Subversion source code repository for PyGreSQL.

The current trunk of the repository can be checked out with the command:

```
svn co svn://svn.pygresql.org/pygresql/trunk
```

You can also browse through the repository using the PyGreSQL Trac browser.

6.3 Bug Tracker

We are using Trac as an issue tracker.

Track tickets are usually entered after discussion on the mailing list, but you may also request an account for the issue tracker and add or process tickets if you want to get more involved into the development of the project. You can use the following links to get an overview:

- PyGreSQL Issues Tracker
- Timeline with all changes
- Roadmap of the project
- Lists of active tickets
- PyGreSQL Trac browser

6.4 Support

Python: see http://www.python.org/community/

PostgreSQL: see http://www.postgresql.org/support/

PyGreSQL: Join the PyGreSQL mailing list if you need help regarding PyGreSQL.

Please also send context diffs there, if you would like to proposes changes.

Please note that messages to individual developers will generally not be answered directly. All questions, comments and code changes must be submitted to the mailing list for peer review and archiving purposes.

6.5 Project home sites

Python: http://www.python.org

PostgreSQL: http://www.postgresql.org

PyGreSQL: http://www.pygresql.org

Python Module Index

p

Index

W