PyGreSQL Documentation

Release 5.1

The PyGreSQL team

Oct 04, 2019

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 or application.

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 5.1 needs PostgreSQL 9.0 to 9.6 or 10 to 12, and Python 2.6, 2.7 or 3.3 to 3.8. If you need to support older PostgreSQL versions or older Python 2.x versions, you can resort to the PyGreSQL 4.x versions that still support them.

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 5.1.1

Release 5.1.1 of PyGreSQL.

It is available at: http://pygresql.org/files/PyGreSQL-5.1.1.tar.gz.

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.

This version has been built and unit tested on:

- NetBSD
- FreeBSD
- openSUSE
- Ubuntu
- Windows 7 and 10 with both MinGW and Visual Studio
- PostgreSQL 9.0 to 9.6 and 10 tp 11 (32 and 64bit)
- Python 2.6, 2.7 and 3.3 to 3.8 (32 and 64bit)

D'Arcy J.M. Cain
darcy@PyGreSQL.org

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 various Python versions is available at

- http://pygresql.org/files/PyGreSQL-5.1.1.win-amd64-py2.6.exe
- http://pygresql.org/files/PyGreSQL-5.1.1.win-amd64-py2.7.exe
- http://pygresql.org/files/PyGreSQL-5.1.1.win-amd64-py3.4.exe
- http://pygresql.org/files/PyGreSQL-5.1.1.win-amd64-py3.5.exe
- http://pygresql.org/files/PyGreSQL-5.1.1.win-amd64-py3.6.exe

- http://pygresql.org/files/PyGreSQL-5.1.1.win-amd64-py3.7.exe
- http://pygresql.org/files/PyGreSQL-5.1.1.win-amd64-py3.8.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 5.1 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 main source file for the C extension module (_pg)
pgconn.c	the connection object
pginternal.c	internal functions
pglarge.c	large object support
pgnotice.c	the notice object
pgquery.c	the query object
pgsource.c	the source object
pgtypes.h	PostgreSQL type definitions
py3c.h	Python 2/3 compatibility layer for the C extension
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 "make 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 install 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 libpq C-interface library. If you are on Windows, make sure that the directory that contains libpq.dll is part of your `PATH` environment variable.

The current version of PyGreSQL has been tested with Python versions 2.6, 2.7 and 3.3 to 3.8, and PostgreSQL versions 9.0 to 9.6 and 10 to 12.

PyGreSQL will be installed as three modules, a shared library called _pg.so (on Linux) or a DLL called _pg.pyd (on Windows), 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.

Installing with Pip

This is the most easy way to install PyGreSQL if you have "pip" installed. 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.

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 extension, for which you need a C compiler installed. 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 C extension module is 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 5.1.1 (...)

- This version officially supports the new Python 3.8 and PostgreSQL 12.
- This version changes internal queries so that they cannot be exploited using a PostgreSQL security vulnerability described as CVE-2018-1058.

Version 5.1 (2019-05-17)

- **Changes to the classic PyGreSQL module (pg):**
  - Support for prepared statements (following a suggestion and first implementation by Justin Pryzby on the mailing list).
  - DB wrapper objects based on existing connections can now be closed and reopened properly (but the underlying connection will not be affected).
  - The query object can now be used as an iterator similar to query.getresult() and will then yield the rows as tuples. Thanks to Justin Pryzby for the proposal and most of the implementation.
  - Deprecated query.ntuples() in the classic API, since len(query) can now be used and returns the same number.
  - The i-th row of the result can now be accessed as *query[i]*.
  - New method query.scalarresult() that gets only the first field of each row as a list of scalar values.
  - New methods query.one(), query.onenamed(), query.onedict() and query.onescalar() that fetch only one row from the result or None if there are no more rows, similar to the cursor.fetchone() method in DB-API 2.
  - New methods query.single(), query.singlenamed(), query.singledict() and query.singlescalar() that fetch only one row from the result, and raise an error if the result does not have exactly one row.
  - New methods query.dictiter(), query.namediter() and query.scalariter() returning the same values as query.dictresult(), query.namedresult() and query.scalarresult(), but as iterables instead of lists. This avoids creating a Python list of all results and can be slightly more efficient.

  - Removed pg.get/set_namedresult. You can configure the named tuples factory with the pg.set_row_factory_size() function and change the implementation with pg.set_query_helpers(), but this is not recommended and this function is not part of the official API.
  - Added new connection attributes *socket*, *backend_pid*, *ssl_in_use* and *ssl_attributes* (the latter need PostgreSQL >= 9.5 on the client).

- **Changes to the DB-API 2 module (pgdb):**
  - Connections now have an *autocommit* attribute which is set to *False* by default but can be set to *True* to switch to autocommit mode where no transactions are started and calling commit() is not required. Note that this is not part of the DB-API 2 standard.

Vesion 5.0.7 (2019-05-17)

- This version officially supports the new PostgreSQL 11.
- Fixed a bug in parsing array subscript ranges (reported by Justin Pryzby).
- Fixed an issue when deleting a DB wrapper object with the underlying connection already closed (bug report by Jacob Champion).

Vesion 5.0.6 (2018-07-29)

- This version officially supports the new Python 3.7.
- Correct trove classifier for the PostgreSQL License.

Version 5.0.5 (2018-04-25)

- This version officially supports the new PostgreSQL 10.
- The memory for the string with the number of rows affected by a classic pg module query() was already freed (bug report and fix by Peifeng Qiu).

Version 5.0.4 (2017-07-23)

- This version officially supports the new Python 3.6 and PostgreSQL 9.6.
- query_formatted() can now be used without parameters.
- The automatic renaming of columns that are invalid as field names of named tuples now works more accurately in Python 2.6 and 3.0.
- Fixed error checks for unlink() and export() methods of large objects (bug report by Justin Pryzby).
- Fixed a compilation issue under OS X (bug report by Josh Johnston).

Version 5.0.3 (2016-12-10)

- It is now possible to use a custom array cast function by changing the type caster for the 'anyarray' type. For instance, by calling set_typecast('anyarray', lambda v, c: v) you can have arrays returned as strings instead of lists. Note that in the pg module, you can also call set_array(False) in order to return arrays as strings.

- The namedtuple classes used for the rows of query results are now cached and reused internally, since creating namedtuples classes in Python is a somewhat expensive operation. By default the cache has a size of 1024 entries, but this can be changed with the set_row_factory_size() function. In certain cases this change can notably improve the performance.
- The namedresult() method in the classic API now also tries to rename columns that would result in invalid field names.

Version 5.0.2 (2016-09-13)

- Fixed an infinite recursion problem in the DB wrapper class of the classic module that could occur when the underlying connection could not be properly opened (bug report by Justin Pryzby).

Version 5.0.1 (2016-08-18)

- The update() and delete() methods of the DB wrapper now use the OID instead of the primary key if both are provided. This restores backward compatibility with PyGreSQL 4.x and allows updating the primary key itself if an OID exists.
- The connect() function of the DB API 2.0 module now accepts additional keyword parameters such as "application_name" which will be passed on to PostgreSQL.
- PyGreSQL now adapts some queries to be able to access older PostgreSQL 8.x databases (as suggested on the mailing list by Andres Mejia). However, these old versions of PostgreSQL are not officially supported and tested any more.
- Fixed an issue with Postgres types that have an OID >= 0x80000000 (reported on the mailing list by Justin Pryzby).
- Allow extra values that are not used in the command in the parameter dict passed to the query_formatted() method (as suggested by Justin Pryzby).
- Improved handling of empty arrays in the classic module.
- Unused classic connections were not properly garbage collected which could cause memory leaks (reported by Justin Pryzby).
- Made C extension compatible with MSVC 9 again (this was needed to compile for Python 2 on Windows).

Version 5.0 (2016-03-20)

- This version now runs on both Python 2 and Python 3.
- The supported versions are Python 2.6 to 2.7, and 3.3 to 3.5.
- PostgreSQL is supported in all versions from 9.0 to 9.5.
- **Changes in the classic PyGreSQL module (pg):**
  - The classic interface got two new methods get_as_list() and get_as_dict() returning a database table as a Python list or dict. The amount of data returned can be controlled with various parameters.
  - A method upsert() has been added to the DB wrapper class that utilizes the "upsert" feature that is new in PostgreSQL 9.5. The new method nicely complements the existing get/insert/update/delete() methods.
  - When using insert/update/upsert(), you can now pass PostgreSQL arrays as lists and PostgreSQL records as tuples in the classic module.

- Conversely, when the query method returns a PostgreSQL array, it is passed to Python as a list. PostgreSQL records are converted to named tuples as well, but only if you use one of the get/insert/update/delete() methods. PyGreSQL uses a new fast built-in parser to achieve this. The automatic conversion of arrays to lists can be disabled with set_array(False).
- The pkey() method of the classic interface now returns tuples instead of frozensets, with the same order of columns as the primary key index.
- Like the DB-API 2 module, the classic module now also returns bool values from the database as Python bool objects instead of strings. You can still restore the old behavior by calling set_bool(False).
- Like the DB-API 2 module, the classic module now also returns bytea data fetched from the database as byte strings, so you don't need to call unescape_bytea() any more. This has been made configurable though, and you can restore the old behavior by calling set_bytea_escaped(True).
- A method set_jsondecode() has been added for changing or removing the function that automatically decodes JSON data coming from the database. By default, decoding JSON is now enabled and uses the decoder function in the standard library with its default parameters.
- The table name that is affixed to the name of the OID column returned by the get() method of the classic interface will not automatically be fully qualified any more. This reduces overhead from the interface, but it means you must always write the table name in the same way when you are using tables with OIDs and call methods that make use of these. Also, OIDs are now only used when access via primary key is not possible. Note that OIDs are considered deprecated anyway, and they are not created by default any more in PostgreSQL 8.1 and later.
- The internal caching and automatic quoting of class names in the classic interface has been simplified and improved, it should now perform better and use less memory. Also, overhead for quoting values in the DB wrapper methods has been reduced and security has been improved by passing the values to libpq separately as parameters instead of inline.
- It is now possible to use the registered type names instead of the more coarse-grained type names that are used by default in PyGreSQL, without breaking any of the mechanisms for quoting and typecasting, which rely on the type information. This is achieved while maintaining simplicity and backward compatibility by augmenting the type name string objects with all the necessary information under the cover. To switch registered type names on or off (this is the default), call the DB wrapper method use_regtypes().
- A new method query_formatted() has been added to the DB wrapper class that allows using the format specifications from Python. A flag "inline" can be set to specify whether parameters should be sent to the database separately or formatted into the SQL.
- A new type helper Bytea() has been added.

- **Changes in the DB-API 2 module (pgdb):**
  - The DB-API 2 module now always returns result rows as named tuples instead of simply lists as before. The documentation explains how you can restore the old behavior or use custom row objects instead.
  - Various classes used by the classic and DB-API 2 modules have been renamed to become simpler, more intuitive and in line with the names used in the DB-API 2 documentation. Since the API provides objects of these types only through constructor functions, this should not cause any incompatibilities.
  - The DB-API 2 module now supports the callproc() cursor method. Note that output parameters are currently not replaced in the return value.
  - The DB-API 2 module now supports copy operations between data streams on the client and database tables via the COPY command of PostgreSQL. The cursor method copy_from() can be used to copy data from the database to the client, and the cursor method copy_to() can be used to copy data from the client to the database.

  - The 7-tuples returned by the description attribute of a pgdb cursor are now named tuples, i.e. their elements can be also accessed by name. The column names and types can now also be requested through the colnames and coltypes attributes, which are not part of DB-API 2 though. The type_code provided by the description attribute is still equal to the PostgreSQL internal type name, but now carries some more information in additional attributes. The size, precision and scale information that is part of the description is now properly set for numeric types.
  - If you pass a Python list as one of the parameters to a DB-API 2 cursor, it is now automatically bound using an ARRAY constructor. If you pass a Python tuple, it is bound using a ROW constructor. This is useful for passing records as well as making use of the IN syntax.
  - Inversely, when a fetch method of a DB-API 2 cursor returns a PostgreSQL array, it is passed to Python as a list, and when it returns a PostgreSQL composite type, it is passed to Python as a named tuple. PyGreSQL uses a new fast built-in parser to achieve this. Anonymous composite types are also supported, but yield only an ordinary tuple containing text strings.
  - New type helpers Interval() and Uuid() have been added.
  - The connection has a new attribute "closed" that can be used to check whether the connection is closed or broken.
  - SQL commands are always handled as if they include parameters, i.e. literal percent signs must always be doubled. This consistent behavior is necessary for using pgdb with wrappers like SQLAlchemy.
  - PyGreSQL 5.0 will be supported as a database driver by SQLAlchemy 1.1.

- **Changes concerning both modules:**
  - PyGreSQL now tries to raise more specific and appropriate subclasses of DatabaseError than just ProgrammingError. Particularly, when database constraints are violated, it raises an IntegrityError now.
  - The modules now provide get_typecast() and set_typecast() methods allowing to control the typecasting on the global level. The connection objects have type caches with the same methods which give control over the typecasting on the level of the current connection. See the documentation for details about the type cache and the typecast mechanisms provided by PyGreSQL.
  - Dates, times, timestamps and time intervals are now returned as the corresponding Python objects from the datetime module of the standard library. In earlier versions of PyGreSQL they had been returned as strings. You can restore the old behavior by deactivating the respective typecast functions, e.g. set_typecast('date', str).
  - PyGreSQL now supports the "uuid" data type, converting such columns automatically to and from Python uuid.UUID objects.
  - PyGreSQL now supports the "hstore" data type, converting such columns automatically to and from Python dictionaries. If you want to insert Python objects as JSON data using DB-API 2, you should wrap them in the new HStore() type constructor as a hint to PyGreSQL.
  - PyGreSQL now supports the "json" and "jsonb" data types, converting such columns automatically to and from Python objects. If you want to insert Python objects as JSON data using DB-API 2, you should wrap them in the new Json() type constructor as a hint to PyGreSQL.
  - A new type helper Literal() for inserting parameters literally as SQL has been added. This is useful for table names, for instance.
  - Fast parsers cast_array(), cast_record() and cast_hstore for the input and output syntax for PostgreSQL arrays, composite types and the hstore type have been added to the C extension module. The array parser also allows using multi-dimensional arrays with PyGreSQL.
  - The tty parameter and attribute of database connections has been removed since it is not supported by PostgreSQL versions newer than 7.4.

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 of 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 registered 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 unnecessary 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 low-level C extension, which serves as a wrapper for the "libpq" library, the C API to PostgreSQL.

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.

Another option is using PyGreSQL indirectly as a database driver for the high-level SQLAlchemy SQL toolkit and ORM, which supports PyGreSQL starting with SQLAlchemy 1.1 and which provides a way to use PyGreSQL in a multi-threaded environment using the concept of "thread local storage". Database URLs for PyGreSQL take this form:

```
postgresql+pygresql://username:password@host:port/database
```

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

Before doing anything else, it's necessary 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 *Connection* 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 *Connection.inserttable()* method. Note that this method uses the COPY command of PostgreSQL to insert all data in one batch operation, which is much faster than sending many individual 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
```

(continues on next page)

(continued from previous page)

```
 2|banana
 3|cherimaya
 4|durian
 5|eggfruit
 6|fig
 7|grapefruit
(7 rows)
```

Instead of simply printing the *Query* 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'}]
```

You can also return the rows as named tuples:

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

In PyGreSQL 5.1 and newer, you can also use the *Query* instance directly as an iterable that yields the rows as tuples, and there are also methods that return iterables for rows as dictionaries, named tuples or scalar values. Other methods like *Query.one()* or *Query.onescalar()* return only one row or only the first field of that row. You can get the number of rows with the `len()` function.

Using the method *DB.get_as_dict()*, you can easily import the whole table into a Python dictionary mapping the primary key *id* to the *name*:

```
>>> db.get_as_dict('fruits', scalar=True)
OrderedDict([(1, 'apple'),
             (2, 'banana'),
             (3, 'cherimaya'),
             (4, 'durian'),
             (5, 'eggfruit'),
             (6, 'fig'),
             (7, 'grapefruit')])
```

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')
```

As 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 has a method that lets you execute database queries:

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

You can also use this method to insert data into the table:

```
>>> 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',))
```

To insert 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)
```

The cursor also has a *Cursor.copy_from()* method to quickly insert large amounts of data into the database, and a *Cursor.copy_to()* method to quickly dump large amounts of data from the database, using the PostgreSQL COPY command. Note however, that these methods are an extension provided by PyGreSQL, they are not part of the DB-API 2 standard.

Also 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:

```
>>> 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 the changes will be discarded.

In a similar way, you can 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()
Row(id=1, name='apple')
```

The result is a named tuple. This means you can access its elements either using an index number as for an ordinary tuple, or using the column name as for access to object attributes.

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

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

The output is a list of named tuples.

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

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

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

```
>>> db.execute("drop table fruits")
>>> cur.close()
>>> con.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 `Connection` instances, which handle the connection and all the requests to the database,
- the `LargeObject` instances, which handle all the accesses to PostgreSQL large objects,
- the `Query` instances that handle query results

and it provides a convenient wrapper class `DB` for the basic `Connection` class.

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*][, *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)
- **user** (*str or None*) – PostgreSQL user (*None* = defuser)
- **passwd** (*str or None*) – password for user (*None* = defpasswd)

Returns If successful, the *Connection* handling the connection

Return type *Connection*

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. The opt parameter can be used to pass command-line options to the server. For a precise description of the parameters, please refer to the PostgreSQL user manual.

If you want to add additional parameters not specified here, you must pass a connection string or a connection URI instead of the dbname (as in con3 and con4 in the following example).

Example:

```
import pg

con1 = pg.connect('testdb', 'myhost', 5432, None, 'bob', None)
con2 = pg.connect(dbname='testdb', host='myhost', user='bob')
con3 = pg.connect('host=myhost user=bob dbname=testdb connect_timeout=10')
con4 = pg.connect('postgresql://bob@myhost/testdb?connect_timeout=10')
```

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_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 *Connection* 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 = 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 *Connection* 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 bytes

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

Converts an escaped string representation of binary data stored as `bytea` into the raw byte string representing the binary data – this is the reverse of *escape_bytea()*. Since the *Query* results will already return unescaped byte strings, you normally don't have to use this method.

Note that there is also a *DB* method with the same name which does exactly the same.

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 activated, but you can disable this with the *set_bool()* function.

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 activated, but you can disable this by calling `set_bool(True)`.

New in version 4.2.

Changed in version 5.0: Boolean values had been returned as string by default in earlier versions.

get/set_array – whether arrays are returned as list objects

`pg.`**`get_array`**`()`
Check whether arrays are returned as list objects

Returns whether or not list objects will be returned

Return type bool

This function checks whether PyGreSQL returns PostgreSQL arrays converted to Python list objects, or simply as text in the internal special output syntax of PostgreSQL. By default, conversion to list objects is activated, but you can disable this with the *set_array()* function.

New in version 5.0.

pg.**set_array**(*on*)
Set whether arrays are returned as list objects

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

This function can be used to specify whether PyGreSQL shall return PostgreSQL arrays converted to Python list objects, or simply as text in the internal special output syntax of PostgreSQL. By default, conversion to list objects is activated, but you can disable this by calling `set_array(False)`.

New in version 5.0.

Changed in version 5.0: Arrays had been always returned as text strings in earlier versions.

get/set_bytea_escaped – whether bytea data is returned escaped

pg.**get_bytea_escaped**()
Check whether bytea values are returned as escaped strings

Returns whether or not bytea objects will be returned escaped

Return type bool

This function checks whether PyGreSQL returns PostgreSQL `bytea` values in escaped form or in unescaped from as byte strings. By default, bytea values will be returned unescaped as byte strings, but you can change this with the *set_bytea_escaped()* function.

New in version 5.0.

pg.**set_bytea_escaped**(*on*)
Set whether bytea values are returned as escaped strings

Parameters **on** – whether or not bytea objects shall be returned escaped

This function can be used to specify whether PyGreSQL shall return PostgreSQL `bytea` values in escaped form or in unescaped from as byte strings. By default, bytea values will be returned unescaped as byte strings, but you can change this by calling `set_bytea_escaped(True)`.

New in version 5.0.

Changed in version 5.0: Bytea data had been returned in escaped form by default in earlier versions.

get/set_jsondecode – decoding JSON format

pg.**get_jsondecode**()
Get the function that deserializes JSON formatted strings

This returns the function used by PyGreSQL to construct Python objects from JSON formatted strings.

pg.**set_jsondecode**(*func*)
Set a function that will deserialize JSON formatted strings

Parameters **func** – the function to be used for deserializing JSON strings

You can use this if you do not want to deserialize JSON strings coming in from the database, or if want to use a different function than the standard function `json.loads()` or if you want to use it with parameters different from the default ones. If you set this function to *None*, then the automatic deserialization of JSON strings will be deactivated.

New in version 5.0.

Changed in version 5.0: JSON data had been always returned as text strings in earlier versions.

get/set_datestyle – assume a fixed date style

pg.**get_datestyle**()
Get the assumed date style for typecasting

This returns the PostgreSQL date style that is silently assumed when typecasting dates or *None* if no fixed date style is assumed, in which case the date style is requested from the database when necessary (this is the default). Note that this method will *not* get the date style that is currently set in the session or in the database. You can get the current setting with the methods *DB.get_parameter()* and *Connection.parameter()*. You can also get the date format corresponding to the current date style by calling *Connection.date_format()*.

New in version 5.0.

pg.**set_datestyle**(*datestyle*)
Set a fixed date style that shall be assumed when typecasting

Parameters **datestyle** (*str*) – the date style that shall be assumed, or *None* if no fixed dat style shall be assumed

PyGreSQL is able to automatically pick up the right date style for typecasting date values from the database, even if you change it for the current session with a `SET DateStyle` command. This is happens very effectively without an additional database request being involved. If you still want to have PyGreSQL always assume a fixed date style instead, then you can set one with this function. Note that calling this function will *not* alter the date style of the database or the current session. You can do that by calling the method *DB.set_parameter()* instead.

New in version 5.0.

get/set_typecast – custom typecasting

PyGreSQL uses typecast functions to cast the raw data coming from the database to Python objects suitable for the particular database type. These functions take a single string argument that represents the data to be casted and must return the casted value.

PyGreSQL provides through its C extension module basic typecast functions for the common database types, but if you want to add more typecast functions, you can set these using the following functions.

pg.**get_typecast**(*typ*)
Get the global cast function for the given database type

Parameters **typ** (*str*) – PostgreSQL type name

Returns the typecast function for the specified type

Return type function or None

New in version 5.0.

pg.**set_typecast**(*typ*, *cast*)
Set a global typecast function for the given database type(s)

Parameters

- **typ** (*str or int*) – PostgreSQL type name or list of type names
- **cast** – the typecast function to be set for the specified type(s)

The typecast function must take one string object as argument and return a Python object into which the PostgreSQL type shall be casted. If the function takes another parameter named *connection*, then the current database connection will also be passed to the typecast function. This may sometimes be necessary to look up certain database settings.

New in version 5.0.

Note that database connections cache types and their cast functions using connection specific *DbTypes* objects. You can also get, set and reset typecast functions on the connection level using the methods *DbTypes.get_typecast()*, *DbTypes.set_typecast()* and *DbTypes.reset_typecast()* of the *DB.dbtypes* object. This will not affect other connections or future connections. In order to be sure a global change is picked up by a running connection, you must reopen it or call *DbTypes.reset_typecast()* on the *DB.dbtypes* object.

Also note that the typecasting for all of the basic types happens already in the C extension module. The typecast functions that can be set with the above methods are only called for the types that are not already supported by the C extension module.

cast_array/record – fast parsers for arrays and records

PostgreSQL returns arrays and records (composite types) using a special output syntax with several quirks that cannot easily and quickly be parsed in Python. Therefore the C extension module provides two fast parsers that allow quickly turning these text representations into Python objects: Arrays will be converted to Python lists, and records to Python tuples. These fast parsers are used automatically by PyGreSQL in order to return arrays and records from database queries as lists and tuples, so you normally don't need to call them directly. You may only need them for typecasting arrays of data types that are not supported by default in PostgreSQL.

pg.**cast_array**(*string*[, *cast*][, *delim*])

Cast a string representing a PostgreSQL array to a Python list

Parameters

- **string** (*str*) – the string with the text representation of the array
- **cast** (*callable or None*) – a typecast function for the elements of the array
- **delim** – delimiter character between adjacent elements

Returns a list representing the PostgreSQL array in Python

Return type list

Raises

- **TypeError** – invalid argument types
- **ValueError** – error in the syntax of the given array

This function takes a *string* containing the text representation of a PostgreSQL array (which may look like `'{{1,2}{3,4}}'` for a two-dimensional array), a typecast function *cast* that is called for every element, and an optional delimiter character *delim* (usually a comma), and returns a Python list representing the array (which may be nested like `[[1, 2], [3, 4]]` in this example). The cast function must take a single argument which will be the text representation of the element and must output the corresponding Python object that shall be put into the list. If you don't pass a cast function or set it to *None*, then unprocessed text strings will be returned as elements of the array. If you don't pass a delimiter character, then a comma will be used by default.

New in version 5.0.

pg.**cast_record**(*string*[, *cast*][, *delim*])

Cast a string representing a PostgreSQL record to a Python tuple

Parameters

- **string** (*str*) – the string with the text representation of the record
- **cast** (*callable, list or tuple of callables, or None*) – typecast function(s) for the elements of the record
- **delim** – delimiter character between adjacent elements

Returns a tuple representing the PostgreSQL record in Python

Return type tuple

Raises

- **TypeError** – invalid argument types
- **ValueError** – error in the syntax of the given array

This function takes a *string* containing the text representation of a PostgreSQL record (which may look like `'(1,a,2,b)'` for a record composed of four fields), a typecast function *cast* that is called for every element, or a list or tuple of such functions corresponding to the individual fields of the record, and an optional delimiter character *delim* (usually a comma), and returns a Python tuple representing the record (which may be inhomogeneous like `(1, 'a', 2, 'b')` in this example). The cast function(s) must take a single argument which will be the text representation of the element and must output the corresponding Python object that shall be put into the tuple. If you don't pass cast function(s) or pass *None* instead, then unprocessed text strings will be returned as elements of the tuple. If you don't pass a delimiter character, then a comma will be used by default.

New in version 5.0.

Note that besides using parentheses instead of braces, there are other subtle differences in escaping special characters and NULL values between the syntax used for arrays and the one used for composite types, which these functions take into account.

Type helpers

The module provides the following type helper functions. You can wrap parameters with these functions when passing them to *DB.query()* or *DB.query_formatted()* in order to give PyGreSQL a hint about the type of the parameters, if it cannot be derived from the context.

pg.**Bytea**(*bytes*)
A wrapper for holding a bytea value

New in version 5.0.

pg.**HStore**(*dict*)
A wrapper for holding an hstore dictionary

New in version 5.0.

pg.**Json**(*obj*)
A wrapper for holding an object serializable to JSON

New in version 5.0.

The following additional type helper is only meaningful when used with *DB.query_formatted()*. It marks a parameter as text that shall be literally included into the SQL. This is useful for passing table names for instance.

pg.**Literal**(*sql*)
A wrapper for holding a literal SQL string

New in version 5.0.

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:

pg.**version**

pg.**__version__**
constants that give the current version

pg.**INV_READ**

pg.**INV_WRITE**
large objects access modes, used by *Connection.locreate()* and *LargeObject.open()*

pg.**SEEK_SET**

pg.**SEEK_CUR**

pg.**SEEK_END**
positional flags, used by *LargeObject.seek()*

pg.**TRANS_IDLE**

pg.**TRANS_ACTIVE**

pg.**TRANS_INTRANS**

pg.**TRANS_INERROR**

pg.**TRANS_UNKNOWN**
transaction states, used by *Connection.transaction()*

Connection – The connection object

class pg.**Connection**

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

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

Parameters

- **command** (*str*) – SQL command
- **args** – optional parameter values

Returns result values

Return type *Query*, 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 as an integer. If the query is an update or delete statement, or an insert statement that did not insert exactly one row, or on a table without 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 *Query*. Otherwise, it returns `None`.

You can use the *Query* object as an iterator that yields all results as tuples, or call *Query.getresult()* to get the result as a list of tuples. Alternatively, you can call *Query.dictresult()* or *Query.dictiter()* if you want to get the rows as dictionaries, or *Query.namedresult()* or *Query.namediter()* if you want to get the rows as named tuples. You can also simply print the *Query* object to show the query results on the console.

The SQL command may optionally contain positional parameters of the form `$1`, `$2`, etc instead of literal data, in which case the values must be supplied separately 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.

If you don't pass any parameters, the command string can also include multiple SQL commands (separated by semicolons). You will only get the return value for the last command in this case.

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

query_prepared – execute a prepared statement

`Connection.`**`query_prepared`**(*name*[, *args*])

Execute a prepared statement

Parameters

- **name** (*str*) – name of the prepared statement
- **args** – optional parameter values

Returns result values

Return type *Query*, 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
- **pg.OperationalError** – prepared statement does not exist

This method works exactly like *Connection.query()* except that instead of passing the command itself, you pass the name of a prepared statement. An empty name corresponds to the unnamed statement. You must have previously created the corresponding named or unnamed statement with *Connection.prepare()*, or an pg.OperationalError will be raised.

New in version 5.1.

prepare – create a prepared statement

Connection.**prepare**(*name*, *command*)

Create a prepared statement

Parameters

- **name** (*str*) – name of the prepared statement
- **command** (*str*) – SQL command

Return type None

Raises

- **TypeError** – bad argument types, or wrong number of arguments
- **TypeError** – invalid connection
- **pg.ProgrammingError** – error in query or duplicate query

This method creates a prepared statement with the specified name for the given command for later execution with the *Connection.query_prepared()* method. The name can be empty to create an unnamed statement, in which case any pre-existing unnamed statement is automatically replaced; otherwise a pg.ProgrammingError is raised if the statement name is already defined in the current database session.

The SQL command may optionally contain positional parameters of the form $1, $2, etc instead of literal data. The corresponding values must then later be passed to the *Connection.query_prepared()* method separately as a tuple.

New in version 5.1.

describe_prepared – describe a prepared statement

Connection.**describe_prepared**(*name*)

Describe a prepared statement

Parameters **name** (*str*) – name of the prepared statement

Return type *Query*

Raises

- **TypeError** – bad argument type, or too many arguments
- **TypeError** – invalid connection
- **pg.OperationalError** – prepared statement does not exist

This method returns a *Query* object describing the prepared statement with the given name. You can also pass an empty name in order to describe the unnamed statement. Information on the fields of the corresponding query can be obtained through the *Query.listfields()*, *Query.fieldname()* and *Query.fieldnum()* methods.

New in version 5.1.

reset – reset the connection

Connection.**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

Connection.**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

Connection.**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.

transaction – get the current transaction state

Connection.**transaction**()
Get the current in-transaction status of the server

Returns the current in-transaction status

Return type int

Raises

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

The status returned by this method can be *TRANS_IDLE* (currently idle), *TRANS_ACTIVE* (a command is in progress), *TRANS_INTRANS* (idle, in a valid transaction block), or *TRANS_INERROR* (idle, in a failed transaction block). *TRANS_UNKNOWN* is reported if the connection is bad. The status *TRANS_ACTIVE* is reported only when a query has been sent to the server and not yet completed.

parameter – get a current server parameter setting

Connection.**parameter**(*name*)

Look up a current parameter setting of the server

Parameters **name** (*str*) – the name of the parameter to look up

Returns the current setting of the specified parameter

Return type str or None

Raises

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

Certain parameter values are reported by the server automatically at connection startup or whenever their values change. This method can be used to interrogate these settings. It returns the current value of a parameter if known, or *None* if the parameter is not known.

You can use this method to check the settings of important parameters such as *server_version*, *server_encoding*, *client_encoding*, *application_name*, *is_superuser*, *session_authorization*, *DateStyle*, *IntervalStyle*, *TimeZone*, *integer_datetimes*, and *standard_conforming_strings*.

Values that are not reported by this method can be requested using *DB.get_parameter()*.

New in version 4.0.

date_format – get the currently used date format

Connection.**date_format**()

Look up the date format currently being used by the database

Returns the current date format

Return type str

Raises

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

This method returns the current date format used by the server. Note that it is cheap to call this method, since there is no database query involved and the setting is also cached internally. You will need the date format when you want to manually typecast dates and timestamps coming from the database instead of using the built-in typecast functions.

The date format returned by this method can be directly used with date formatting functions such as `datetime.strptime()`. It is derived from the current setting of the database parameter `DateStyle`.

New in version 5.0.

fileno – get the socket used to connect to the database

`Connection.`**`fileno`**`()`

Get 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

`Connection.`**`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 *`Connection.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

`Connection.`**`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 looks whether or not it knows how to handle such types.

get/set_cast_hook – fallback typecast function

Connection.**get_cast_hook**()
Get the function that handles all external typecasting

Returns the current external typecast function

Return type callable, None

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

This returns the callback function used by PyGreSQL to provide plug-in Python typecast functions for the connection.

New in version 5.0.

Connection.**set_cast_hook**(*func*)
Set a function that will handle all external typecasting

Parameters **func** – the function to be used as a callback

Return type None

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

This methods allows setting a custom fallback function for providing Python typecast functions for the connection to supplement the C extension module. If you set this function to *None*, then only the typecast functions implemented in the C extension module are enabled. You normally would not want to change this. Instead, you can use *get_typecast()* and *set_typecast()* to add or change the plug-in Python typecast functions.

New in version 5.0.

get/set_notice_receiver – custom notice receiver

Connection.**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 *Connection.set_notice_receiver()*, or None if no custom notice receiver has ever been set on the connection.

New in version 4.1.

Connection.**set_notice_receiver**(*func*)
Set a custom notice receiver

Parameters **func** – 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 `Notice` object, which provides the following read-only attributes:

`Notice.`**pgcnx**
the connection

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

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

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

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

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

New in version 4.1.

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

`Connection.`**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]

`Connection.`**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]

Connection.**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]

Connection.**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 *LargeObject*

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]

Connection.**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 *LargeObject*

Raises

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

This method allows reusing a previously created large object through the *LargeObject* interface, provided the user has its OID.

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

Connection.**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 *LargeObject*

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

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

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

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

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

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

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

New in version 4.0.

Connection.**server_version**
the backend version (int, e.g. 90305 for 9.3.5)

New in version 4.0.

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

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

Connection.**socket**
the file descriptor number of the connection socket to the server (int)

New in version 5.1.

Connection.**backend_pid**
the PID of the backend process handling this connection (int)

New in version 5.1.

Connection.**ssl_in_use**
: this is True if the connection uses SSL, False if not

New in version 5.1: (needs PostgreSQL >= 9.5)

Connection.**ssl_attributes**
: SSL-related information about the connection (dict)

New in version 5.1: (needs PostgreSQL >= 9.5)

The DB wrapper class

class pg.**DB**

The *Connection* methods are wrapped in the class *DB* which also adds convenient higher level methods for working with the database. It also serves as a context manager for the connection. The preferred way to use this module is as follows:

```
import pg

with pg.DB(...) as db:  # for parameters, 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 *Connection.close()* and `Connection.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

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

This method returns the primary key of a table. Single primary keys are returned as strings unless you set the composite flag. Composite primary keys are always represented as tuples. 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 views 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 an ordered dictionary mapping attribute names to type names

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

Returns a read-only dictionary of attribute names (the names are the keys, the values are the names of the attributes' types) with the column names in the proper order if you iterate over it.

By default, only a limited number of simple types will be returned. You can get the registered types instead, if enabled by calling the *DB.use_regtypes()* method.

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.

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.

Note that you can request most of the important parameters also using *Connection.parameter()* which does not involve a database query, unlike *DB.get_parameter()* and *DB.set_parameter()*.

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.

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.

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, discarding all its changes.

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*, *row*[, *keyname*])
Get a row from a database table or view

Parameters

- **table** (*str*) – name of table or view
- **row** – 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** – table has no primary key or missing privilege
- **KeyError** – missing key value for the row

This method is the basic mechanism to get a single row. It assumes that the *keyname* specifies a unique row. It must be the name of a single column or a tuple of column names. If *keyname* is not specified, then the primary key for the table is used.

If *row* is a dictionary, then the value for the key is taken from it. Otherwise, the row must be a single value or a tuple of values corresponding to the passed *keyname* or primary key. The fetched row from the table will be returned as a new dictionary or used to replace the existing values if the row was passed as a dictionary.

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(table)` using the actual name of the table.

Note that since PyGreSQL 5.0 this will return the value of an array type column as a Python list by default.

insert – insert a row into a database table

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

Parameters

- **table** (*str*) – name of table
- **row** (*dict*) – optional dictionary of values
- **col** – optional keyword arguments for updating the dictionary

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 reloaded with the values actually inserted in order to pick up values modified by rules, triggers, etc.

Note that since PyGreSQL 5.0 it is possible to insert a value for an array type column by passing it as a Python list.

update – update a row in a database table

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

Parameters

- **table** (*str*) – name of table
- **row** (*dict*) – optional dictionary of values
- **col** – optional keyword arguments for updating the dictionary

Returns the new row in the database

Return type dict

Raises

- **pg.ProgrammingError** – table has no primary key or missing privilege
- **KeyError** – missing key value for the row

Similar to insert, but updates an existing row. The update is based on the primary key of the table or the OID value as munged by *DB.get()* or passed as keyword. The OID will take precedence if provided, so that it is possible to update the primary key itself.

The dictionary is then modified 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 specified using the `'oid'` keyword or in the dictionary, in which case the OID must be munged.

upsert – insert a row with conflict resolution

DB.**upsert**(*table*[, *row*][, *col*=*val*, ...])

Insert a row into a database table with conflict resolution

Parameters

- **table** (*str*) – name of table
- **row** (*dict*) – optional dictionary of values
- **col** – optional keyword arguments for specifying the update

Returns the new row in the database

Return type dict

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

This method inserts a row into a table, but instead of raising a ProgrammingError exception in case of violating a constraint or unique index, an update will be executed instead. This will be performed as a single atomic operation on the database, so race conditions can be avoided.

Like the insert method, the first parameter is the name of the table and the second parameter can be used to pass the values to be inserted as a dictionary.

Unlike the insert und update statement, keyword parameters are not used to modify the dictionary, but to specify which columns shall be updated in case of a conflict, and in which way:

A value of *False* or *None* means the column shall not be updated, a value of *True* means the column shall be updated with the value that has been proposed for insertion, i.e. has been passed as value in the dictionary. Columns that are not specified by keywords but appear as keys in the dictionary are also updated like in the case keywords had been passed with the value *True*.

So if in the case of a conflict you want to update every column that has been passed in the dictionary *d* , you would call `upsert(table, d)`. If you don't want to do anything in case of a conflict, i.e. leave the existing row as it is, call `upsert(table, d, **dict.fromkeys(d))`.

If you need more fine-grained control of what gets updated, you can also pass strings in the keyword parameters. These strings will be used as SQL expressions for the update columns. In these expressions you can refer to the value that already exists in the table by writing the table prefix `included.` before the column name, and you can refer to the value that has been proposed for insertion by writing `excluded.` as table prefix.

The dictionary is modified in any case to reflect the values in the database after the operation has completed.

Note: The method uses the PostgreSQL "upsert" feature which is only available since PostgreSQL 9.5. With older PostgreSQL versions, you will get a ProgrammingError if you use this method.

New in version 5.0.

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 *Query*, 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 *Connection* function with the same name, except that positional arguments can be passed either as a single list or tuple, or as individual positional arguments. These arguments will then be used as parameter values of parameterized queries.

Example:

```
name = input("Name? ")
phone = 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]
```

query_formatted – execute a formatted SQL command string

DB.**query_formatted**(*command*[, *parameters*][, *types*][, *inline*])
Execute a formatted SQL command string

Parameters

- **command** (*str*) – SQL command
- **parameters** (*tuple, list or dict*) – the values of the parameters for the SQL command
- **types** (*tuple, list or dict*) – optionally, the types of the parameters
- **inline** (*bool*) – whether the parameters should be passed in the SQL

Return type *Query*, 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 *DB.query()*, but using Python format placeholders of the form `%s` or `%(names)s` instead of PostgreSQL placeholders of the form `$1`. The parameters must be passed as a tuple, list or dict. You can also pass a corresponding tuple, list or dict of database types in order to format the parameters properly in case there is ambiguity.

If you set *inline* to True, the parameters will be sent to the database embedded in the SQL command, otherwise they will be sent separately.

If you set *inline* to True or don't pass any parameters, the command string can also include multiple SQL commands (separated by semicolons). You will only get the result for the last command in this case.

Note that the adaptation and conversion of the parameters causes a certain performance overhead. Depending on the type of values, the overhead can be smaller for *inline* queries or if you pass the types of the parameters, so that they don't need to be guessed from the values. For best performance, we recommend using a raw *DB.query()* or *DB.query_prepared()* if you are executing many of the same operations with different parameters.

Example:

```
name = input("Name? ")
phone = input("Phone? ")
rows = db.query_formatted(
    "update employees set phone=%s where name=%s",
    (phone, name)).getresult()[0][0]
# or
rows = db.query_formatted(
    "update employees set phone=%(phone)s where name=%(name)s",
    dict(name=name, phone=phone)).getresult()[0][0]
```

query_prepared – execute a prepared statement

DB.**query_prepared**(*name*[, *arg1*[, *arg2*, ...]])

Execute a prepared statement

Parameters

- **name** (*str*) – name of the prepared statement
- **arg*** – optional positional arguments

Returns result values

Return type *Query*, 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
- **pg.OperationalError** – prepared statement does not exist

This methods works like the *DB.query()* method, except that instead of passing the SQL command, you pass the name of a prepared statement created previously using the *DB.prepare()* method.

Passing an empty string or *None* as the name will execute the unnamed statement (see warning about the limited lifetime of the unnamed statement in *DB.prepare()*).

The functionality of this method is equivalent to that of the SQL `EXECUTE` command. Note that calling EXECUTE would require parameters to be sent inline, and be properly sanitized (escaped, quoted).

New in version 5.1.

prepare – create a prepared statement

`DB.`**`prepare`**(*name*, *command*)
Create a prepared statement

Parameters

- **command** (*str*) – SQL command
- **name** (*str*) – name of the prepared statement

Return type None

Raises

- **TypeError** – bad argument types, or wrong number of arguments
- **TypeError** – invalid connection
- **pg.ProgrammingError** – error in query or duplicate query

This method creates a prepared statement with the specified name for later execution of the given command with the *DB.query_prepared()* method.

If the name is empty or *None*, the unnamed prepared statement is used, in which case any pre-existing unnamed statement is replaced.

Otherwise, if a prepared statement with the specified name is already defined in the current database session, a `pg.ProgrammingError` is raised.

The SQL command may optionally contain positional parameters of the form `$1`, `$2`, etc instead of literal data. The corresponding values must then be passed to the *Connection.query_prepared()* method as positional arguments.

The functionality of this method is equivalent to that of the SQL `PREPARE` command.

Example:

```
db.prepare('change phone',
    "update employees set phone=$2 where ein=$1")
while True:
    ein = input("Employee ID? ")
    if not ein:
        break
    phone = input("Phone? ")
    db.query_prepared('change phone', ein, phone)
```

Note: We recommend always using named queries, since unnamed queries have a limited lifetime and can be automatically replaced or destroyed by various operations on the database.

New in version 5.1.

describe_prepared – describe a prepared statement

DB.**describe_prepared**([*name*])
Describe a prepared statement

Parameters **name** (*str*) – name of the prepared statement

Return type *Query*

Raises

- **TypeError** – bad argument type, or too many arguments
- **TypeError** – invalid connection
- **pg.OperationalError** – prepared statement does not exist

This method returns a *Query* object describing the prepared statement with the given name. You can also pass an empty name in order to describe the unnamed statement. Information on the fields of the corresponding query can be obtained through the *Query.listfields()*, *Query.fieldname()* and *Query.fieldnum()* methods.

New in version 5.1.

delete_prepared – delete a prepared statement

DB.**delete_prepared**([*name*])
Delete a prepared statement

Parameters **name** (*str*) – name of the prepared statement

Return type None

Raises

- **TypeError** – bad argument type, or too many arguments
- **TypeError** – invalid connection
- **pg.OperationalError** – prepared statement does not exist

This method deallocates a previously prepared SQL statement with the given name, or deallocates all prepared statements if you do not specify a name. Note that prepared statements are always deallocated automatically when the current session ends.

New in version 5.1.

clear – clear row values in memory

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

Parameters

- **table** (*str*) – name of table
- **row** (*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 *False*, and everything else is set to the empty string. If the row argument is present, it is used as the row dictionary 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*[, *row*][, *col*=*val*, ...])
Delete a row from a database table

Parameters

- **table** (*str*) – name of table
- **d** (*dict*) – optional dictionary of values
- **col** – optional keyword arguments for updating the dictionary

Return type None

Raises

- **pg.ProgrammingError** – table has no primary key, row is still referenced or missing privilege
- **KeyError** – missing key value for the row

This method deletes the row from a table. It deletes based on the primary key of the table or the OID value as munged by *DB.get()* or passed as keyword. The OID will take precedence if provided.

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).

Note that if the row cannot be deleted because e.g. it is still referenced by another table, this method will raise a ProgrammingError.

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.

get_as_list/dict – read a table as a list or dictionary

DB.**get_as_list**(*table*[, *what*][, *where*][, *order*][, *limit*][, *offset*][, *scalar*])
Get a table as a list

Parameters

- **table** (*str*) – the name of the table (the FROM clause)
- **what** (*str, list, tuple or None*) – column(s) to be returned (the SELECT clause)
- **where** (*str, list, tuple or None*) – conditions(s) to be fulfilled (the WHERE clause)
- **order** (*str, list, tuple, False or None*) – column(s) to sort by (the ORDER BY clause)
- **limit** (*int*) – maximum number of rows returned (the LIMIT clause)
- **offset** (*int*) – number of rows to be skipped (the OFFSET clause)
- **scalar** (*bool*) – whether only the first column shall be returned

Returns the content of the table as a list

Return type list

Raises **TypeError** – the table name has not been specified

This gets a convenient representation of the table as a list of named tuples in Python. You only need to pass the name of the table (or any other SQL expression returning rows). Note that by default this will return the full content of the table which can be huge and overflow your memory. However, you can control the amount of data returned using the other optional parameters.

The parameter *what* can restrict the query to only return a subset of the table columns. The parameter *where* can restrict the query to only return a subset of the table rows. The specified SQL expressions all need to be fulfilled for a row to get into the result. The parameter *order* specifies the ordering of the rows. If no ordering is specified, the result will be ordered by the primary key(s) or all columns if no primary key exists. You can set *order* to *False* if you don't care about the ordering. The parameters *limit* and *offset* specify the maximum number of rows returned and a number of rows skipped over.

If you set the *scalar* option to *True*, then instead of the named tuples you will get the first items of these tuples. This is useful if the result has only one column anyway.

New in version 5.0.

DB.**get_as_dict**(*table*[, *keyname*][, *what*][, *where*][, *order*][, *limit*][, *offset*][, *scalar*])
Get a table as a dictionary

Parameters

- **table** (*str*) – the name of the table (the FROM clause)
- **keyname** (*str, list, tuple or None*) – column(s) to be used as key(s) of the dictionary
- **what** (*str, list, tuple or None*) – column(s) to be returned (the SELECT clause)

- **where** (*str, list, tuple or None*) – conditions(s) to be fulfilled (the WHERE clause)
- **order** (*str, list, tuple, False or None*) – column(s) to sort by (the ORDER BY clause)
- **limit** (*int*) – maximum number of rows returned (the LIMIT clause)
- **offset** (*int*) – number of rows to be skipped (the OFFSET clause)
- **scalar** (*bool*) – whether only the first column shall be returned

Returns the content of the table as a list

Return type dict or OrderedDict

Raises

- **TypeError** – the table name has not been specified
- **KeyError** – keyname(s) are invalid or not part of the result
- **pg.ProgrammingError** – no keyname(s) and table has no primary key

This method is similar to *DB.get_as_list()*, but returns the table as a Python dict instead of a Python list, which can be even more convenient. The primary key column(s) of the table will be used as the keys of the dictionary, while the other column(s) will be the corresponding values. The keys will be named tuples if the table has a composite primary key. The rows will be also named tuples unless the *scalar* option has been set to *True*. With the optional parameter *keyname* you can specify a different set of columns to be used as the keys of the dictionary.

If the Python version supports it, the dictionary will be an *OrderedDict* using the order specified with the *order* parameter or the key column(s) if not specified. You can set *order* to *False* if you don't care about the ordering. In this case the returned dictionary will be an ordinary one.

New in version 5.0.

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 be misinterpreted 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_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

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).

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_bytea()* with the same name, but the behavior of this method is adjusted depending on the connection properties (in particular, whether standard-conforming strings are enabled).

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 bytes

Converts an escaped string representation of binary data stored as bytea into the raw byte string representing the binary data – this is the reverse of *DB.escape_bytea()*. Since the *Query* results will already return unescaped byte strings, you normally don't have to use this method.

encode/decode_json – encode and decode JSON data

The following methods can be used to encode end decode data in JSON format.

DB.**encode_json**(*obj*)
Encode a Python object for use within SQL as type json or jsonb

Parameters **obj** (*dict, list or None*) – Python object that shall be encoded to JSON format

Returns string representation of the Python object in JSON format

Return type str

This method serializes a Python object into a JSON formatted string that can be used within SQL. You don't need to use this method on the data stored with *DB.insert()* and similar, only if you store the data directly as part of an SQL command or parameter with *DB.query()*. This is the same as the `json.dumps()` function from the standard library.

New in version 5.0.

`DB.`**`decode_json`**(*string*)
Decode `json` or `jsonb` data that has been retrieved as text

Parameters **`string`** (*str*) – JSON formatted string shall be decoded into a Python object

Returns Python object representing the JSON formatted string

Return type dict, list or None

This method deserializes a JSON formatted string retrieved as text from the database to a Python object. You normally don't need to use this method as JSON data is automatically decoded by PyGreSQL. If you don't want the data to be decoded, then you can cast `json` or `jsonb` columns to `text` in PostgreSQL or you can set the decoding function to *None* or a different function using *pg.set_jsondecode()*. By default this is the same as the `json.loads()` function from the standard library.

New in version 5.0.

use_regtypes – choose usage of registered type names

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

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

Returns whether registered type names are used

The *DB.get_attnames()* method can return either simplified "classic" type names (the default) or more fine-grained "registered" 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 registered type names shall be used. The method can also be used to check through its return value whether registered type names are currently 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.

Attributes of the DB wrapper class

DB.**db**
The wrapped *Connection* object

You normally don't need this, since all of the members can be accessed from the *DB* wrapper class as well.

DB.**dbname**
The name of the database that the connection is using

DB.**dbtypes**
A dictionary with the various type names for the PostgreSQL types

This can be used for getting more information on the PostgreSQL database types or changing the typecast functions used for the connection. See the description of the *DbTypes* class for details.

New in version 5.0.

DB.**adapter**
A class with some helper functions for adapting parameters

This can be used for building queries with parameters. You normally will not need this, as you can use the *DB.query_formatted* method.

New in version 5.0.

Query methods

class pg.**Query**

The *Query* object returned by *Connection.query()* and *DB.query()* can be used as an iterable returning rows as tuples. You can also directly access row tuples using their index, and get the number of rows with the `len()` function. The *Query* class also provides the following methods for accessing the results of the query:

getresult – get query values as list of tuples

Query.**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 query results as a list of tuples. More information about this result may be accessed using *Query.listfields()*, *Query.fieldname()* and *Query.fieldnum()* methods.

Note that since PyGreSQL 5.0 this method will return the values of array type columns as Python lists.

Since PyGreSQL 5.1 the *Query* can be also used directly as an iterable sequence, i.e. you can iterate over the *Query* object to get the same tuples as returned by *Query.getresult()*. This is slightly more efficient than getting the full list of results, but note that the full result is always fetched from the server anyway when the query is executed.

You can also call `len()` on a query to find the number of rows in the result, and access row tuples using their index directly on the *Query* object.

dictresult/dictiter – get query values as dictionaries

Query.**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 query results as a list of dictionaries which have the field names as keys.

If the query has duplicate field names, you will get the value for the field with the highest index in the query.

Note that since PyGreSQL 5.0 this method will return the values of array type columns as Python lists.

Query.**dictiter**()
Get query values as iterable of dictionaries

Returns result values as an iterable of dictionaries

Return type iterable

Raises

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

This method returns query results as an iterable of dictionaries which have the field names as keys. This is slightly more efficient than getting the full list of results as dictionaries, but note that the full result is always fetched from the server anyway when the query is executed.

If the query has duplicate field names, you will get the value for the field with the highest index in the query.

New in version 5.1.

namedresult/namediter – get query values as named tuples

Query.**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 query results as a list of named tuples with proper field names.

Column names in the database that are not valid as field names for named tuples (particularly, names starting with an underscore) are automatically renamed to valid positional names.

Note that since PyGreSQL 5.0 this method will return the values of array type columns as Python lists.

New in version 4.1.

Query.**namediter**()
Get query values as iterable of named tuples

Returns result values as an iterable of named tuples

Return type iterable

Raises

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

This method returns query results as an iterable of named tuples with proper field names. This is slightly more efficient than getting the full list of results as named tuples, but note that the full result is always fetched from the server anyway when the query is executed.

Column names in the database that are not valid as field names for named tuples (particularly, names starting with an underscore) are automatically renamed to valid positional names.

New in version 5.1.

scalarresult/scalariter – get query values as scalars

Query.**scalarresult**()
Get first fields from query result as list of scalar values

Returns first fields from result as a list of scalar values

Return type list

Raises

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

This method returns the first fields from the query results as a list of scalar values in the order returned by the server.

New in version 5.1.

Query.**scalariter**()
Get first fields from query result as iterable of scalar values

Returns first fields from result as an iterable of scalar values

Return type list

Raises

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

This method returns the first fields from the query results as an iterable of scalar values in the order returned by the server. This is slightly more efficient than getting the full list of results as rows or scalar values, but note that the full result is always fetched from the server anyway when the query is executed.

New in version 5.1.

one/onedict/onenamed/onescalar – get one result of a query

Query.**one**()
Get one row from the result of a query as a tuple

Returns next row from the query results as a tuple of fields

Return type tuple or None

Raises

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

Returns only one row from the result as a tuple of fields.

This method can be called multiple times to return more rows. It returns None if the result does not contain one more row.

New in version 5.1.

Query.**onedict**()
Get one row from the result of a query as a dictionary

Returns next row from the query results as a dictionary

Return type dict or None

Raises

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

Returns only one row from the result as a dictionary with the field names used as the keys.

This method can be called multiple times to return more rows. It returns None if the result does not contain one more row.

New in version 5.1.

Query.**onenamed**()
Get one row from the result of a query as named tuple

Returns next row from the query results as a named tuple

Return type named tuple or None

Raises

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

Returns only one row from the result as a named tuple with proper field names.

Column names in the database that are not valid as field names for named tuples (particularly, names starting with an underscore) are automatically renamed to valid positional names.

This method can be called multiple times to return more rows. It returns None if the result does not contain one more row.

New in version 5.1.

Query.**onescalar**()
Get one row from the result of a query as scalar value

Returns next row from the query results as a scalar value

Return type type of first field or None

Raises

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

Returns the first field of the next row from the result as a scalar value.

This method can be called multiple times to return more rows as scalars. It returns None if the result does not contain one more row.

New in version 5.1.

single/singledict/singlenamed/singlescalar – get single result of a query

`Query.single()`
Get single row from the result of a query as a tuple

Returns single row from the query results as a tuple of fields

Return type tuple :raises InvalidResultError: result does not have exactly one row

Raises

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

Returns a single row from the result as a tuple of fields.

This method returns the same single row when called multiple times. It raises an `pg.InvalidResultError` if the result does not have exactly one row. More specifically, this will be of type `pg.NoResultError` if it is empty and of type `pg.MultipleResultsError` if it has multiple rows.

New in version 5.1.

`Query.singledict()`
Get single row from the result of a query as a dictionary

Returns single row from the query results as a dictionary

Return type dict :raises InvalidResultError: result does not have exactly one row

Raises

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

Returns a single row from the result as a dictionary with the field names used as the keys.

This method returns the same single row when called multiple times. It raises an `pg.InvalidResultError` if the result does not have exactly one row. More specifically, this will be of type `pg.NoResultError` if it is empty and of type `pg.MultipleResultsError` if it has multiple rows.

New in version 5.1.

`Query.singlenamed()`
Get single row from the result of a query as named tuple

Returns single row from the query results as a named tuple

Return type named tuple :raises InvalidResultError: result does not have exactly one row

Raises

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

Returns single row from the result as a named tuple with proper field names.

Column names in the database that are not valid as field names for named tuples (particularly, names starting with an underscore) are automatically renamed to valid positional names.

This method returns the same single row when called multiple times. It raises an `pg.InvalidResultError` if the result does not have exactly one row. More specifically, this will be of type `pg.NoResultError` if it is empty and of type `pg.MultipleResultsError` if it has multiple rows.

New in version 5.1.

`Query.`**`singlescalar`**`()`
Get single row from the result of a query as scalar value

Returns single row from the query results as a scalar value

Return type type of first field :raises InvalidResultError: result does not have exactly one row

Raises

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

Returns the first field of a single row from the result as a scalar value.

This method returns the same single row as scalar when called multiple times. It raises an `pg.InvalidResultError` if the result does not have exactly one row. More specifically, this will be of type `pg.NoResultError` if it is empty and of type `pg.MultipleResultsError` if it has multiple rows.

New in version 5.1.

listfields – list fields names of previous query result

`Query.`**`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 field names defined for the query result. The fields are in the same order as the result values.

fieldname, fieldnum – field name/number conversion

`Query.`**`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.

Query.**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 given 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 query result.

ntuples – return number of tuples in query object

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

Returns number of tuples in *Query*

Return type int

Raises **TypeError** – Too many arguments.

This method returns the number of tuples in the query result.

Deprecated since version 5.1: You can use the normal `len()` function instead.

LargeObject – Large Objects

class pg.**LargeObject**

Objects that are instances of the class *LargeObject* 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 *Connection* instances do, thus only keeping significant parameters in function calls. The *LargeObject* instance keeps a reference to the *Connection* object used for its creation, sending requests though with its parameters. Any modification but dereferencing the *Connection* object will thus affect the *LargeObject* instance. Dereferencing the initial *Connection* object is not a problem since Python won't deallocate it before the *LargeObject* instance 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

LargeObject.**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

LargeObject.**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

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

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

Returns the read buffer

Return type bytes

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.

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

Parameters **string** (*bytes*) – 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.

LargeObject.**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*).

LargeObject.**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.

LargeObject.**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

LargeObject.**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

LargeObject.**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

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

LargeObject.**oid**
the OID associated with the large object (int)

LargeObject.**pgcnx**
the *Connection* object associated with the large object

LargeObject.**error**
the last warning/error message of the connection (str)

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

DbTypes – The internal cache for database types

class pg.**DbTypes**

New in version 5.0.

The *DbTypes* object is essentially a dictionary mapping PostgreSQL internal type names and type OIDs to PyGreSQL "type names" (which are also returned by *DB.get_attnames()* as dictionary values).

These type names are strings which are equal to either the simple PyGreSQL names or to the more fine-grained registered PostgreSQL type names if these have been enabled with *DB.use_regtypes()*. Besides being strings, they carry additional information about the associated PostgreSQL type in the following attributes:

- *oid* – the PostgreSQL type OID
- *pgtype* – the internal PostgreSQL data type name
- *regtype* – the registered PostgreSQL data type name
- *simple* – the more coarse-grained PyGreSQL type name
- *typtype* – *b* = base type, *c* = composite type etc.
- *category* – *A* = Array, *b* =Boolean, *C* = Composite etc.

- *delim* – delimiter for array types
- *relid* – corresponding table for composite types
- *attnames* – attributes for composite types

For details, see the PostgreSQL documentation on pg_type.

In addition to the dictionary methods, the *DbTypes* class also provides the following methods:

DbTypes.**get_attnames**(*typ*)
Get the names and types of the fields of composite types

Parameters **typ** (*str or int*) – PostgreSQL type name or OID of a composite type

Returns an ordered dictionary mapping field names to type names

DbTypes.**get_typecast**(*typ*)
Get the cast function for the given database type

Parameters **typ** (*str*) – PostgreSQL type name

Returns the typecast function for the specified type

Return type function or None

DbTypes.**set_typecast**(*typ*, *cast*)
Set a typecast function for the given database type(s)

Parameters

- **typ** (*str or int*) – PostgreSQL type name or list of type names
- **cast** – the typecast function to be set for the specified type(s)

The typecast function must take one string object as argument and return a Python object into which the PostgreSQL type shall be casted. If the function takes another parameter named *connection*, then the current database connection will also be passed to the typecast function. This may sometimes be necessary to look up certain database settings.

DbTypes.**reset_typecast**([*typ*])
Reset the typecasts for the specified (or all) type(s) to their defaults

Parameters **typ** (*str, list or None*) – PostgreSQL type name or list of type names, or None to reset all typecast functions

DbTypes.**typecast**(*value*, *typ*)
Cast the given value according to the given database type

Parameters **typ** (*str*) – PostgreSQL type name or type code

Returns the casted value

Note: Note that *DbTypes* object is always bound to a database connection. You can also get and set and reset typecast functions on a global level using the functions *pg.get_typecast()* and *pg.set_typecast()*. If you do this, the current database connections will continue to use their already cached typecast functions unless you reset the typecast functions by calling the *DbTypes.reset_typecast()* method on *DB.dbtypes* objects of the running connections.

Also note that the typecasting for all of the basic types happens already in the C low-level extension module. The typecast functions that can be set with the above methods are only called for the types that are not already supported by the C extension.

Remarks on Adaptation and Typecasting

Both PostgreSQL and Python have the concept of data types, but there are of course differences between the two type systems. Therefore PyGreSQL needs to adapt Python objects to the representation required by PostgreSQL when passing values as query parameters, and it needs to typecast the representation of PostgreSQL data types returned by database queries to Python objects. Here are some explanations about how this works in detail in case you want to better understand or change the default behavior of PyGreSQL.

Supported data types

The following automatic data type conversions are supported by PyGreSQL out of the box. If you need other automatic type conversions or want to change the default conversions, you can achieve this by using the methods explained in the next two sections.

PostgreSQL	Python
char, bpchar, name, text, varchar	str
bool	bool
bytea	bytes
int2, int4, int8, oid, serial	int[1]
int2vector	list of int
float4, float8	float
numeric, money	Decimal
date	datetime.date
time, timetz	datetime.time
timestamp, timestamptz	datetime.datetime
interval	datetime.timedelta
hstore	dict
json, jsonb	list or dict
uuid	uuid.UUID
array	list[2]
record	tuple

Note: Elements of arrays and records will also be converted accordingly.

Adaptation of parameters

When you use the higher level methods of the classic `pg` module like `DB.insert()` or `DB.update()`, you don't need to care about adaptation of parameters, since all of this is happening automatically behind the scenes. You only need to consider this issue when creating SQL commands manually and sending them to the database using the `DB.query()` method.

Imagine you have created a user login form that stores the login name as *login* and the password as *passwd* and you now want to get the user data for that user. You may be tempted to execute a query like this:

[1] int8 is converted to long in Python 2

[2] The first element of the array will always be the first element of the Python list, no matter what the lower bound of the PostgreSQL array is. The information about the start index of the array (which is usually 1 in PostgreSQL, but can also be different from 1) is ignored and gets lost in the conversion to the Python list. If you need that information, you can request it separately with the *array_lower()* function provided by PostgreSQL.

```
>>> db = pg.DB(...)
>>> sql = "SELECT * FROM user_table WHERE login = '%s' AND passwd = '%s'"
>>> db.query(sql % (login, passwd)).getresult()[0]
```

This seems to work at a first glance, but you will notice an error as soon as you try to use a login name containing a single quote. Even worse, this error can be exploited through so-called "SQL injection", where an attacker inserts malicious SQL statements into the query that you never intended to be executed. For instance, with a login name something like `' OR ''='` the attacker could easily log in and see the user data of another user in the database.

One solution for this problem would be to cleanse your input of "dangerous" characters like the single quote, but this is tedious and it is likely that you overlook something or break the application e.g. for users with names like "D'Arcy". A better solution is to use the escaping functions provided by PostgreSQL which are available as methods on the *DB* object:

```
>>> login = "D'Arcy"
>>> db.escape_string(login)
"D''Arcy"
```

As you see, *DB.escape_string()* has doubled the single quote which is the right thing to do in SQL. However, there are better ways of passing parameters to the query, without having to manually escape them. If you pass the parameters as positional arguments to *DB.query()*, then PyGreSQL will send them to the database separately, without the need for quoting them inside the SQL command, and without the problems inherent with that process. In this case you must put placeholders of the form `$1`, `$2` etc. in the SQL command in place of the parameters that should go there. For instance:

```
>>> sql = "SELECT * FROM user_table WHERE login = $1 AND passwd = $2"
>>> db.query(sql, login, passwd).getresult()[0]
```

That's much better. So please always keep the following warning in mind:

Warning: Remember to **never** insert parameters directly into your queries using the `%` operator. Always pass the parameters separately.

If you like the `%` format specifications of Python better than the placeholders used by PostgreSQL, there is still a way to use them, via the *DB.query_formatted()* method:

```
>>> sql = "SELECT * FROM user_table WHERE login = %s AND passwd = %s"
>>> db.query_formatted(sql, (login, passwd)).getresult()[0]
```

Note that we need to pass the parameters not as positional arguments here, but as a single tuple. Also note again that we did not use the `%` operator of Python to format the SQL string, we just used the `%s` format specifications of Python and let PyGreSQL care about the formatting. Even better, you can also pass the parameters as a dictionary if you use the *DB.query_formatted()* method:

```
>>> sql = """SELECT * FROM user_table
...     WHERE login = %(login)s AND passwd = %(passwd)s"""
>>> parameters = dict(login=login, passwd=passwd)
>>> db.query_formatted(sql, parameters).getresult()[0]
```

Here is another example:

```
>>> sql = "SELECT 'Hello, ' || %s || '!'"
>>> db.query_formatted(sql, (login,)).getresult()[0]
```

You would think that the following even simpler example should work, too:

```
>>> sql = "SELECT %s"
>>> db.query_formatted(sql, (login,)).getresult()[0]
ProgrammingError: Could not determine data type of parameter $1
```

The issue here is that *DB.query_formatted()* by default still uses PostgreSQL parameters, transforming the Python style `%s` placeholder into a `$1` placeholder, and sending the login name separately from the query. In the query we looked at before, the concatenation with other strings made it clear that it should be interpreted as a string. This simple query however does not give PostgreSQL a clue what data type the `$1` placeholder stands for.

This is different when you are embedding the login name directly into the query instead of passing it as parameter to PostgreSQL. You can achieve this by setting the *inline* parameter of *DB.query_formatted()*, like so:

```
>>> sql = "SELECT %s"
>>> db.query_formatted(sql, (login,), inline=True).getresult()[0]
```

Another way of making this query work while still sending the parameters separately is to simply cast the parameter values:

```
>>> sql = "SELECT %s::text"
>>> db.query_formatted(sql, (login,), inline=False).getresult()[0]
```

In real world examples you will rarely have to cast your parameters like that, since in an INSERT statement or a WHERE clause comparing the parameter to a table column, the data type will be clear from the context.

When binding the parameters to a query, PyGreSQL not only adapts the basic types like `int`, `float`, `bool` and `str`, but also tries to make sense of Python lists and tuples.

Lists are adapted as PostgreSQL arrays:

```
>>> params = dict(array=[[1, 2],[3, 4]])
>>> db.query_formatted("SELECT %(array)s::int[]", params).getresult()[0][0]
[[1, 2], [3, 4]]
```

Note that again we need to cast the array parameter or use inline parameters only because this simple query does not provide enough context. Also note that the query gives the value back as Python lists again. This is achieved by the typecasting mechanism explained in the next section.

Tuples are adapted as PostgreSQL composite types. If you use inline parameters, they can also be used with the `IN` syntax.

Let's think of a more real world example again where we create a table with a composite type in PostgreSQL:

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer)
```

We assume the composite type `inventory_item` has been created like this:

```
CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric)
```

In Python we can use a named tuple as an equivalent to this PostgreSQL type:

```
>>> from collections import namedtuple
>>> inventory_item = namedtuple(
...     'inventory_item', ['name', 'supplier_id', 'price'])
```

Using the automatic adaptation of Python tuples, an item can now be inserted into the database and then read back as follows:

```
>>> db.query_formatted("INSERT INTO on_hand VALUES (%(item)s, %(count)s)",
...     dict(item=inventory_item('fuzzy dice', 42, 1.99), count=1000))
>>> db.query("SELECT * FROM on_hand").getresult()[0][0]
Row(item=inventory_item(name='fuzzy dice', supplier_id=42,
        price=Decimal('1.99')), count=1000)
```

The *DB.insert()* method provides a simpler way to achieve the same:

```
>>> row = dict(item=inventory_item('fuzzy dice', 42, 1.99), count=1000)
>>> db.insert('on_hand', row)
{'count': 1000,  'item': inventory_item(name='fuzzy dice',
        supplier_id=42, price=Decimal('1.99'))}
```

Perhaps we want to use custom Python classes instead of named tuples to hold our values:

```
>>> class InventoryItem:
...
...     def __init__(self, name, supplier_id, price):
...         self.name = name
...         self.supplier_id = supplier_id
...         self.price = price
...
...     def __str__(self):
...         return '%s (from %s, at $%s)' % (
...             self.name, self.supplier_id, self.price)
```

But when we try to insert an instance of this class in the same way, we will get an error. This is because PyGreSQL tries to pass the string representation of the object as a parameter to PostgreSQL, but this is just a human readable string and not useful for PostgreSQL to build a composite type. However, it is possible to make such custom classes adapt themselves to PostgreSQL by adding a "magic" method with the name `__pg_str__`, like so:

```
>>> class InventoryItem:
...
...     ...
...
...     def __str__(self):
...         return '%s (from %s, at $%s)' % (
...             self.name, self.supplier_id, self.price)
...
...     def __pg_str__(self, typ):
...         return (self.name, self.supplier_id, self.price)
```

Now you can insert class instances the same way as you insert named tuples. You can even make these objects adapt to different types in different ways:

```
>>> class InventoryItem:
...
...     ...
...
...     def __pg_str__(self, typ):
...         if typ == 'text':
...             return str(self)
...        return (self.name, self.supplier_id, self.price)
...
```

(continues on next page)

(continued from previous page)

```
>>> db.query("ALTER TABLE on_hand ADD COLUMN remark varchar")
>>> item=InventoryItem('fuzzy dice', 42, 1.99)
>>> row = dict(item=item, remark=item, count=1000)
>>> db.insert('on_hand', row)
{'count': 1000, 'item': inventory_item(name='fuzzy dice',
    supplier_id=42, price=Decimal('1.99')),
    'remark': 'fuzzy dice (from 42, at $1.99)'}
```

There is also another "magic" method `__pg_repr__` which does not take the *typ* parameter. That method is used instead of `__pg_str__` when passing parameters inline. You must be more careful when using `__pg_repr__`, because it must return a properly escaped string that can be put literally inside the SQL. The only exception is when you return a tuple or list, because these will be adapted and properly escaped by PyGreSQL again.

Typecasting to Python

As you noticed, PyGreSQL automatically converted the PostgreSQL data to suitable Python objects when returning values via the *DB.get()*, *Query.getresult()* and similar methods. This is done by the use of built-in typecast functions.

If you want to use different typecast functions or add your own if no built-in typecast function is available, then this is possible using the *set_typecast()* function. With the *get_typecast()* function you can check which function is currently set. If no typecast function is set, then PyGreSQL will return the raw strings from the database.

For instance, you will find that PyGreSQL uses the normal `int` function to cast PostgreSQL `int4` type values to Python:

```
>>> pg.get_typecast('int4')
int
```

In the classic PyGreSQL module, the typecasting for these basic types is always done internally by the C extension module for performance reasons. We can set a different typecast function for `int4`, but it will not become effective, the C module continues to use its internal typecasting.

However, we can add new typecast functions for the database types that are not supported by the C module. For example, we can create a typecast function that casts items of the composite PostgreSQL type used as example in the previous section to instances of the corresponding Python class.

To do this, at first we get the default typecast function that PyGreSQL has created for the current *DB* connection. This default function casts composite types to named tuples, as we have seen in the section before. We can grab it from the *DB.dbtypes* object as follows:

```
>>> cast_tuple = db.dbtypes.get_typecast('inventory_item')
```

Now we can create a new typecast function that converts the tuple to an instance of our custom class:

```
>>> cast_item = lambda value: InventoryItem(*cast_tuple(value))
```

Finally, we set this typecast function, either globally with *set_typecast()*, or locally for the current connection like this:

```
>>> db.dbtypes.set_typecast('inventory_item', cast_item)
```

Now we can get instances of our custom class directly from the database:

```
>>> item = db.query("SELECT * FROM on_hand").getresult()[0][0]
>>> str(item)
'fuzzy dice (from 42, at $1.99)'
```

Note that some of the typecast functions used by the C module are configurable with separate module level functions, such as *set_decimal()*, *set_bool()* or *set_jsondecode()*. You need to use these instead of *set_typecast()* if you want to change the behavior of the C module.

Also note that after changing global typecast functions with *set_typecast()*, you may need to run `db.dbtypes.reset_typecast()` to make these changes effective on connections that were already open.

As one last example, let us try to typecast the geometric data type `circle` of PostgreSQL into a SymPy `Circle` object. Let's assume we have created and populated a table with two circles, like so:

```
CREATE TABLE circle (
    name varchar(8) primary key, circle circle);
INSERT INTO circle VALUES ('C1', '<(2, 3), 3>');
INSERT INTO circle VALUES ('C2', '<(1, -1), 4>');
```

With PostgreSQL we can easily calculate that these two circles overlap:

```
>>> q = db.query("""SELECT c1.circle && c2.circle
...     FROM circle c1, circle c2
...     WHERE c1.name = 'C1' AND c2.name = 'C2'""")
>>> q.getresult()[0][0]
True
```

However, calculating the intersection points between the two circles using the # operator does not work (at least not as of PostgreSQL version 12). So let's resort to SymPy to find out. To ease importing circles from PostgreSQL to SymPy, we create and register the following typecast function:

```
>>> from sympy import Point, Circle
>>>
>>> def cast_circle(s):
...     p, r = s[1:-1].split(',')
...     p = p[1:-1].split(',')
...     return Circle(Point(float(p[0]), float(p[1])), float(r))
...
>>> pg.set_typecast('circle', cast_circle)
```

Now we can import the circles in the table into Python simply using:

```
>>> circle = db.get_as_dict('circle', scalar=True)
```

The result is a dictionary mapping circle names to SymPy `Circle` objects. We can verify that the circles have been imported correctly:

```
>>> circle['C1']
Circle(Point(2, 3), 3.0)
>>> circle['C2']
Circle(Point(1, -1), 4.0)
```

Finally we can find the exact intersection points with SymPy:

```
>>> circle['C1'].intersection(circle['C2'])
[Point(29/17 + 64564173230121*sqrt(17)/100000000000000,
    -80705216537651*sqrt(17)/500000000000000 + 31/17),
```

(continues on next page)

(continued from previous page)

```
Point(-64564173230121*sqrt(17)/100000000000000 + 29/17,
   80705216537651*sqrt(17)/500000000000000 + 31/17)]
```

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*][, ***kwargs*])

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
- **kwargs** (*dict*) – other connection parameters

Returns a connection object

Return type *Connection*

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

This function takes parameters specifying how to connect to a PostgreSQL database and returns a *Connection* object using these parameters. If specified, the *dsn* parameter must be a string with the format `'host:base:user:passwd:opt'`. 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. You can pass additional connection parameters using the optional *kwargs* keyword arguments.

Example:

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

Changed in version 5.0.1: Support for additional parameters passed as *kwargs*.

get/set/reset_typecast – Control the global typecast functions

PyGreSQL uses typecast functions to cast the raw data coming from the database to Python objects suitable for the particular database type. These functions take a single string argument that represents the data to be casted and must return the casted value.

PyGreSQL provides built-in typecast functions for the common database types, but if you want to change these or add more typecast functions, you can set these up using the following functions.

Note: The following functions are not part of the DB-API 2 standard.

`pgdb.`**`get_typecast`**`(`*typ*`)`
Get the global cast function for the given database type

Parameters **`typ`** (*`str`*) – PostgreSQL type name or type code

Returns the typecast function for the specified type

Return type function or None

New in version 5.0.

`pgdb.`**`set_typecast`**`(`*typ*, *cast*`)`
Set a global typecast function for the given database type(s)

Parameters

- **`typ`** (*`str or int`*) – PostgreSQL type name or type code, or list of such
- **`cast`** – the typecast function to be set for the specified type(s)

The typecast function must take one string object as argument and return a Python object into which the PostgreSQL type shall be casted. If the function takes another parameter named *connection*, then the current database connection will also be passed to the typecast function. This may sometimes be necessary to look up certain database settings.

New in version 5.0.

As of version 5.0.3 you can also use this method to change the typecasting of PostgreSQL array types. You must run `set_typecast('anyarray', cast)` in order to do this. The `cast` method must take a string value and a cast function for the base type and return the array converted to a Python object. For instance, run `set_typecast('anyarray', lambda v, c: v)` to switch off the casting of arrays completely, and always return them encoded as strings.

`pgdb.`**`reset_typecast`**`(`[*typ*]`)`
Reset the typecasts for the specified (or all) type(s) to their defaults

Parameters **`typ`** (*`str, list or None`*) – PostgreSQL type name or type code, or list of such, or None to reset all typecast functions

New in version 5.0.

Note that database connections cache types and their cast functions using connection specific *TypeCache* objects. You can also get, set and reset typecast functions on the connection level using the methods *TypeCache.get_typecast()*, *TypeCache.set_typecast()* and *TypeCache.reset_typecast()* of the *Connection.type_cache*. This will not affect other connections or future connections. In order to be sure a global change is picked up by a running connection, you must reopen it or call *TypeCache.reset_typecast()* on the *Connection.type_cache*.

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.

Connection – The connection object

class pgdb.**Connection**

These connection objects respond to the following methods.

Note that `pgdb.Connection` objects also implement the context manager protocol, i.e. you can use them in a `with` statement. When the `with` block ends, the current transaction will be automatically committed or rolled back if there was an exception, and you won't need to do this manually.

close – close the connection

Connection.**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

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

Return type None

Note that connections always use a transaction, unless you set the *Connection.autocommit* attribute described below.

rollback – roll back the connection

Connection.**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

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

Returns a connection object

Return type *Cursor*

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

Attributes that are not part of the standard

Note: The following attributes are not part of the DB-API 2 standard.

`Connection.`**`closed`**
This is *True* if the connection has been closed or has become invalid

`Connection.`**`cursor_type`**
The default cursor type used by the connection

If you want to use your own custom subclass of the *`Cursor`* class with he connection, set this attribute to your custom cursor class. You will then get your custom cursor whenever you call *`Connection.cursor()`*.

New in version 5.0.

`Connection.`**`type_cache`**
A dictionary with the various type codes for the PostgreSQL types

This can be used for getting more information on the PostgreSQL database types or changing the typecast functions used for the connection. See the description of the *`TypeCache`* class for details.

New in version 5.0.

`Connection.`**`autocommit`**
A read/write attribute to get/set the autocommit mode

Normally, all DB-API 2 SQL commands are run inside a transaction. Sometimes this behavior is not desired; there are also some SQL commands such as VACUUM which cannot be run inside a transaction.

By setting this attribute to `True` you can change this behavior so that no transactions will be started for that connection. In this case every executed SQL command has immediate effect on the database and you don't need to call *`Connection.commit()`* explicitly. In this mode, you can still use `with con:` blocks to run parts of the code using the connection `con` inside a transaction.

By default, this attribute is set to `False` which conforms to the behavior specified by the DB-API 2 standard (manual commit required).

New in version 5.1.

Cursor – The cursor object

`class` `pgdb.`**`Cursor`**

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 `Cursor` 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

`Cursor.`**`description`**
This read-only attribute is a sequence of 7-item named tuples.

Each of these named tuples contains information describing one result column:

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

The values for *precision* and *scale* are only set for numeric types. The values for *display_size* and *null_ok* are always `None`.

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

Changed in version 5.0: Before version 5.0, this attribute was an ordinary tuple.

rowcount – number of rows of the result

`Cursor.`**`rowcount`**

This read-only attribute specifies the number of rows that the last *`Cursor.execute()`* or *`Cursor.executemany()`* call produced (for DQL statements like SELECT) or affected (for DML statements like UPDATE or INSERT). It is also set by the *`Cursor.copy_from()`* and *`Cursor.copy_to()`* methods. 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

`Cursor.`**`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

`Cursor.`**`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: *Cursor.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

Cursor.**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 bound to the query using Python extended format codes, e.g. `" ... WHERE name=%(name)s"`.

callproc – Call a stored procedure

Cursor.callproc(self, procname, [parameters]):
Call a stored database procedure with the given name

Parameters

- **procname** (*str*) – the name of the database function
- **parameters** – a sequence of parameters (can be empty or omitted)

This method calls a stored procedure (function) in the PostgreSQL database.

The sequence of parameters must contain one entry for each input argument that the function expects. The result of the call is the same as this input sequence; replacement of output and input/output parameters in the return value is currently not supported.

The function may also provide a result set as output. These can be requested through the standard fetch methods of the cursor.

New in version 5.0.

fetchone – fetch next row of the query result

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

Returns the next row of the query result set

Return type named tuple or None

Fetch the next row of a query result set, returning a single named tuple, or `None` when no more data is available. The field names of the named tuple are the same as the column names of the database query as long as they are valid Python identifiers.

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

Changed in version 5.0: Before version 5.0, this method returned ordinary tuples.

fetchmany – fetch next set of rows of the query result

`Cursor.`**`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 named tuples

Fetch the next set of rows of a query result, returning a list of named tuples. An empty sequence is returned when no more rows are available. The field names of the named tuple are the same as the column names of the database query as long as they are valid Python identifiers.

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 *Cursor.execute()* or *Cursor.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 *Cursor.fetchmany()* call to the next.

Changed in version 5.0: Before version 5.0, this method returned ordinary tuples.

fetchall – fetch all rows of the query result

`Cursor.`**`fetchall`**()

Fetch all (remaining) rows of a query result

Returns the set of all rows of the query result

Return type list of named tuples

Fetch all (remaining) rows of a query result, returning them as list of named tuples. The field names of the named tuple are the same as the column names of the database query as long as they are valid as field names for named tuples, otherwise they are given positional names.

Note that the cursor's `arraysize` attribute can affect the performance of this operation.

Changed in version 5.0: Before version 5.0, this method returned ordinary tuples.

arraysize - the number of rows to fetch at a time

Cursor.**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 *Cursor.fetchmany()*. It defaults to 1, meaning to fetch a single row at a time.

Methods and attributes that are not part of the standard

Note: The following methods and attributes are not part of the DB-API 2 standard.

Cursor.**copy_from**(*stream*, *table*[, *format*][, *sep*][, *null*][, *size*][, *columns*])
Copy data from an input stream to the specified table

Parameters

- **stream** – the input stream (must be a file-like object, a string or an iterable returning strings)
- **table** (*str*) – the name of a database table
- **format** (*str*) – the format of the data in the input stream, can be 'text' (the default), 'csv', or 'binary'
- **sep** (*str*) – a single character separator (the default is '\t' for text and ',' for csv)
- **null** (*str*) – the textual representation of the NULL value, can also be an empty string (the default is '\\N')
- **size** (*int*) – the size of the buffer when reading file-like objects
- **column** (*list*) – an optional list of column names

Returns the cursor, so you can chain commands

Raises

- **TypeError** – parameters with wrong types
- **ValueError** – invalid parameters
- **IOError** – error when executing the copy operation

This method can be used to copy data from an input stream on the client side to a database table on the server side using the COPY FROM command. The input stream can be provided in form of a file-like object (which must have a read() method), a string, or an iterable returning one row or multiple rows of input data on each iteration.

The format must be text, csv or binary. The sep option sets the column separator (delimiter) used in the non binary formats. The null option sets the textual representation of NULL in the input.

The size option sets the size of the buffer used when reading data from file-like objects.

The copy operation can be restricted to a subset of columns. If no columns are specified, all of them will be copied.

New in version 5.0.

Cursor.**copy_to**(*stream*, *table*[, *format*][, *sep*][, *null*][, *decode*][, *columns*])
Copy data from the specified table to an output stream

Parameters

- **stream** – the output stream (must be a file-like object or None)
- **table** (*str*) – the name of a database table or a SELECT query
- **format** (*str*) – the format of the data in the input stream, can be 'text' (the default), 'csv', or 'binary'
- **sep** (*str*) – a single character separator (the default is '\t' for text and ',' for csv)
- **null** (*str*) – the textual representation of the NULL value, can also be an empty string (the default is '\\N')
- **decode** (*bool*) – whether decoded strings shall be returned for non-binary formats (the default is True in Python 3)
- **column** (*list*) – an optional list of column names

Returns a generator if stream is set to None, otherwise the cursor

Raises

- **TypeError** – parameters with wrong types
- **ValueError** – invalid parameters
- **IOError** – error when executing the copy operation

This method can be used to copy data from a database table on the server side to an output stream on the client side using the COPY TO command.

The output stream can be provided in form of a file-like object (which must have a write() method). Alternatively, if None is passed as the output stream, the method will return a generator yielding one row of output data on each iteration.

Output will be returned as byte strings unless you set decode to true.

Note that you can also use a SELECT query instead of the table name.

The format must be text, csv or binary. The sep option sets the column separator (delimiter) used in the non binary formats. The null option sets the textual representation of NULL in the output.

The copy operation can be restricted to a subset of columns. If no columns are specified, all of them will be copied.

New in version 5.0.

Cursor.**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

This method is used for processing result rows before returning them through one of the fetch methods. By default, rows are returned as named tuples. You can overwrite this method with a custom row factory if you want to return the rows as different kids of objects. This same row factory will then be used for all result sets. If you overwrite this method, the method *Cursor.build_row_factory()* for creating row factories dynamically will be ignored.

Note that named tuples are very efficient and can be easily converted to dicts (even OrderedDicts) by calling row._asdict(). If you still want to return rows as dicts, you can create a custom cursor class like this:

```
class DictCursor(pgdb.Cursor):

    def row_factory(self, row):
        return {key: value for key, value in zip(self.colnames, row)}

cur = DictCursor(con)  # get one DictCursor instance or
con.cursor_type = DictCursor  # always use DictCursor instances
```

New in version 4.0.

Cursor.**build_row_factory**()
Build a row factory based on the current description

Returns callable with the signature of *Cursor.row_factory()*

This method returns row factories for creating named tuples. It is called whenever a new result set is created, and *Cursor.row_factory* is then assigned the return value of this method. You can overwrite this method with a custom row factory builder if you want to use different row factories for different result sets. Otherwise, you can also simply overwrite the *Cursor.row_factory()* method. This method will then be ignored.

The default implementation that delivers rows as named tuples essentially looks like this:

```
def build_row_factory(self):
    return namedtuple('Row', self.colnames, rename=True)._make
```

New in version 5.0.

Cursor.**colnames**
The list of columns names of the current result set

The values in this list are the same values as the *name* elements in the *Cursor.description* attribute. Always use the latter if you want to remain standard compliant.

New in version 5.0.

Cursor.**coltypes**
The list of columns types of the current result set

The values in this list are the same values as the *type_code* elements in the *Cursor.description* attribute. Always use the latter if you want to remain standard compliant.

New in version 5.0.

Type – 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 *Cursor.execute()* and *Cursor.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`. Or a list could be bound as a `array` or a `json` object. 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*][, *second*][, *microsecond*][, *tzinfo*])
Construct an object holding a time value

pgdb.**Timestamp**(*year*, *month*, *day*[, *hour*][, *minute*][, *second*][, *microsecond*][, *tzinfo*])
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

Additionally, PyGreSQL provides the following constructors for PostgreSQL specific data types:

pgdb.**Interval**(*days*, *hours=0*, *minutes=0*, *seconds=0*, *microseconds=0*)
Construct an object holding a time interval value

New in version 5.0.

pgdb.**Uuid**([*hex*][, *bytes*][, *bytes_le*][, *fields*][, *int*][, *version*])
Construct an object holding a UUID value

New in version 5.0.

pgdb.**Hstore**(*dict*)
Construct a wrapper for holding an hstore dictionary

New in version 5.0.

pgdb.**Json**(*obj*[, *encode*])
Construct a wrapper for holding an object serializable to JSON

You can pass an optional serialization function as a parameter. By default, PyGreSQL uses `json.dumps()` to serialize it.

pgdb.**Literal**(*sql*)
Construct a wrapper for holding a literal SQL string

New in version 5.0.

Example for using a type constructor:

```
>>> cursor.execute("create table jsondata (data jsonb)")
>>> data = {'id': 1, 'name': 'John Doe', 'kids': ['Johnnie', 'Janie']}
>>> cursor.execute("insert into jsondata values (%s)", [Json(data)])
```

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

Type objects

class pgdb.**Type**

The *Cursor.description* attribute returns information about each of the result columns of a query. The *type_code* must compare equal to one of the *Type* 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

UUID
Used to describe uuid columns

HSTORE
Used to describe hstore columns

New in version 5.0.

JSON
Used to describe `json` and `jsonb` columns

New in version 5.0.

ARRAY
Used to describe columns containing PostgreSQL arrays

New in version 5.0.

RECORD
Used to describe columns containing PostgreSQL records

New in version 5.0.

Example for using some type objects:

```
>>> cursor = con.cursor()
>>> cursor.execute("create table jsondata (created date, data jsonb)")
>>> cursor.execute("select * from jsondata")
>>> (created, data) = (d.type_code for d in cursor.description)
>>> created == DATE
True
>>> created == DATETIME
True
>>> created == TIME
False
>>> data == JSON
True
>>> data == STRING
False
```

TypeCache – The internal cache for database types

class pgdb.**TypeCache**

New in version 5.0.

The internal *TypeCache* of PyGreSQL is not part of the DB-API 2 standard, but is documented here in case you need full control and understanding of the internal handling of database types.

The TypeCache is essentially a dictionary mapping PostgreSQL internal type names and type OIDs to DB-API 2 "type codes" (which are also returned as the *type_code* field of the *Cursor.description* attribute).

These type codes are strings which are equal to the PostgreSQL internal type name, but they are also carrying additional information about the associated PostgreSQL type in the following attributes:

- *oid* – the OID of the type
- *len* – the internal size
- *type* – `'b'` = base, `'c'` = composite, ...
- *category* – `'A'` = Array, `'B'` = Boolean, ...
- *delim* – delimiter to be used when parsing arrays
- *relid* – the table OID for composite types

For details, see the PostgreSQL documentation on pg_type.

In addition to the dictionary methods, the *TypeCache* provides the following methods:

TypeCache.**get_fields**(*typ*)
Get the names and types of the fields of composite types

Parameters **typ** (*str or int*) – PostgreSQL type name or OID of a composite type

Returns a list of pairs of field names and types

Return type list

TypeCache.**get_typecast**(*typ*)
Get the cast function for the given database type

Parameters **typ** (*str*) – PostgreSQL type name or type code

Returns the typecast function for the specified type

Return type function or None

TypeCache.**set_typecast**(*typ*, *cast*)
Set a typecast function for the given database type(s)

Parameters

- **typ** (*str or int*) – PostgreSQL type name or type code, or list of such
- **cast** – the typecast function to be set for the specified type(s)

The typecast function must take one string object as argument and return a Python object into which the PostgreSQL type shall be casted. If the function takes another parameter named *connection*, then the current database connection will also be passed to the typecast function. This may sometimes be necessary to look up certain database settings.

TypeCache.**reset_typecast**([*typ*])
Reset the typecasts for the specified (or all) type(s) to their defaults

Parameters **typ** (*str, list or None*) – PostgreSQL type name or type code, or list of such, or None to reset all typecast functions

TypeCache.**typecast**(*value*, *typ*)
Cast the given value according to the given database type

Parameters **typ** (*str*) – PostgreSQL type name or type code

Returns the casted value

Note: Note that the *TypeCache* is always bound to a database connection. You can also get, set and reset typecast functions on a global level using the functions *pgdb.get_typecast()*, *pgdb.set_typecast()* and *pgdb.reset_typecast()*. If you do this, the current database connections will continue to use their already cached typecast functions unless call the *TypeCache.reset_typecast()* method on the *Connection.type_cache* objects of the running connections.

Remarks on Adaptation and Typecasting

Both PostgreSQL and Python have the concept of data types, but there are of course differences between the two type systems. Therefore PyGreSQL needs to adapt Python objects to the representation required by PostgreSQL when passing values as query parameters, and it needs to typecast the representation of PostgreSQL data types returned by database queries to Python objects. Here are some explanations about how this works in detail in case you want to better understand or change the default behavior of PyGreSQL.

Supported data types

The following automatic data type conversions are supported by PyGreSQL out of the box. If you need other automatic type conversions or want to change the default conversions, you can achieve this by using the methods explained in the next two sections.

PostgreSQL	Python
char, bpchar, name, text, varchar	str
bool	bool
bytea	bytes
int2, int4, int8, oid, serial	int[1]
int2vector	list of int
float4, float8	float
numeric, money	Decimal
date	datetime.date
time, timetz	datetime.time
timestamp, timestamptz	datetime.datetime
interval	datetime.timedelta
hstore	dict
json, jsonb	list or dict
uuid	uuid.UUID
array	list[2]
record	tuple

Note: Elements of arrays and records will also be converted accordingly.

Adaptation of parameters

PyGreSQL knows how to adapt the common Python types to get a suitable representation of their values for PostgreSQL when you pass parameters to a query. For example:

```
>>> con = pgdb.connect(...)
>>> cur = con.cursor()
>>> parameters = (144, 3.75, 'hello', None)
>>> tuple(cur.execute('SELECT %s, %s, %s, %s', parameters).fetchone()
(144, Decimal('3.75'), 'hello', None)
```

This is the result we can expect, so obviously PyGreSQL has adapted the parameters and sent the following query to PostgreSQL:

```
SELECT 144, 3.75, 'hello', NULL
```

Note the subtle, but important detail that even though the SQL string passed to `cur.execute()` contains conversion specifications normally used in Python with the `%` operator for formatting strings, we didn't use the `%` operator to format the parameters, but passed them as the second argument to `cur.execute()`. I.e. we **didn't** write the following:

[1] int8 is converted to long in Python 2

[2] The first element of the array will always be the first element of the Python list, no matter what the lower bound of the PostgreSQL array is. The information about the start index of the array (which is usually 1 in PostgreSQL, but can also be different from 1) is ignored and gets lost in the conversion to the Python list. If you need that information, you can request it separately with the *array_lower()* function provided by PostgreSQL.

```
>>> tuple(cur.execute('SELECT %s, %s, %s, %s' % parameters).fetchone()
```

If we had done this, PostgreSQL would have complained because the parameters were not adapted. Particularly, there would be no quotes around the value `'hello'`, so PostgreSQL would have interpreted this as a database column, which would have caused a *ProgrammingError*. Also, the Python value `None` would have been included in the SQL command literally, instead of being converted to the SQL keyword `NULL`, which would have been another reason for PostgreSQL to complain about our bad query:

```
SELECT 144, 3.75, hello, None
```

Even worse, building queries with the use of the `%` operator makes us vulnerable to so called "SQL injection" exploits, where an attacker inserts malicious SQL statements into our queries that we never intended to be executed. We could avoid this by carefully quoting and escaping the parameters, but this would be tedious and if we overlook something, our code will still be vulnerable. So please don't do this. This cannot be emphasized enough, because it is such a subtle difference and using the `%` operator looks so natural:

Warning: Remember to **never** insert parameters directly into your queries using the `%` operator. Always pass the parameters separately.

The good thing is that by letting PyGreSQL do the work for you, you can treat all your parameters equally and don't need to ponder where you need to put quotes or need to escape strings. You can and should also always use the general `%s` specification instead of e.g. using `%d` for integers. Actually, to avoid mistakes and make it easier to insert parameters at more than one location, you can and should use named specifications, like this:

```
>>> params = dict(greeting='Hello', name='HAL')
>>> sql = """SELECT %(greeting)s || ', ' || %(name)s
...     || '. Do you read me, ' || %(name)s || '?'"""
>>> cur.execute(sql, params).fetchone()[0]
'Hello, HAL. Do you read me, HAL?'
```

PyGreSQL does not only adapt the basic types like `int`, `float`, `bool` and `str`, but also tries to make sense of Python lists and tuples.

Lists are adapted as PostgreSQL arrays:

```
>>> params = dict(array=[[1, 2],[3, 4]])
>>> cur.execute("SELECT %(array)s", params).fetchone()[0]
[[1, 2], [3, 4]]
```

Note that the query gives the value back as Python lists again. This is achieved by the typecasting mechanism explained in the next section. The query that was actually executed was this:

```
SELECT ARRAY[[1,2],[3,4]]
```

Again, if we had inserted the list using the `%` operator without adaptation, the `ARRAY` keyword would have been missing in the query.

Tuples are adapted as PostgreSQL composite types:

```
>>> params = dict(record=('Bond', 'James'))
>>> cur.execute("SELECT %(record)s", params).fetchone()[0]
('Bond', 'James')
```

You can also use this feature with the `IN` syntax of SQL:

```
>>> params = dict(what='needle', where=('needle', 'haystack'))
>>> cur.execute("SELECT %(what)s IN %(where)s", params).fetchone()[0]
True
```

Sometimes a Python type can be ambiguous. For instance, you might want to insert a Python list not into an array column, but into a JSON column. Or you want to interpret a string as a date and insert it into a DATE column. In this case you can give PyGreSQL a hint by using *Type constructors*:

```
>>> cur.execute("CREATE TABLE json_data (data json, created date)")
>>> params = dict(
...     data=pgdb.Json([1, 2, 3]), created=pgdb.Date(2016, 1, 29))
>>> sql = ("INSERT INTO json_data VALUES (%(data)s, %(created)s)")
>>> cur.execute(sql, params)
>>> cur.execute("SELECT * FROM json_data").fetchone()
Row(data=[1, 2, 3], created='2016-01-29')
```

Let's think of another example where we create a table with a composite type in PostgreSQL:

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer)
```

We assume the composite type `inventory_item` has been created like this:

```
CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric)
```

In Python we can use a named tuple as an equivalent to this PostgreSQL type:

```
>>> from collections import namedtuple
>>> inventory_item = namedtuple(
...     'inventory_item', ['name', 'supplier_id', 'price'])
```

Using the automatic adaptation of Python tuples, an item can now be inserted into the database and then read back as follows:

```
>>> cur.execute("INSERT INTO on_hand VALUES (%(item)s, %(count)s)",
...     dict(item=inventory_item('fuzzy dice', 42, 1.99), count=1000))
>>> cur.execute("SELECT * FROM on_hand").fetchone()
Row(item=inventory_item(name='fuzzy dice', supplier_id=42,
        price=Decimal('1.99')), count=1000)
```

However, we may not want to use named tuples, but custom Python classes to hold our values, like this one:

```
>>> class InventoryItem:
...
...     def __init__(self, name, supplier_id, price):
...         self.name = name
...         self.supplier_id = supplier_id
...         self.price = price
...
...     def __str__(self):
...         return '%s (from %s, at $%s)' % (
...             self.name, self.supplier_id, self.price)
```

But when we try to insert an instance of this class in the same way, we will get an error:

```
>>> cur.execute("INSERT INTO on_hand VALUES (%(item)s, %(count)s)",
...     dict(item=InventoryItem('fuzzy dice', 42, 1.99), count=1000))
InterfaceError: Do not know how to adapt type <class 'InventoryItem'>
```

While PyGreSQL knows how to adapt tuples, it does not know what to make out of our custom class. To simply convert the object to a string using the `str` function is not a solution, since this yields a human readable string that is not useful for PostgreSQL. However, it is possible to make such custom classes adapt themselves to PostgreSQL by adding a "magic" method with the name `__pg_repr__`, like this:

```
>>> class InventoryItem:
  ...
  ...     ...
  ...
  ...     def __str__(self):
  ...         return '%s (from %s, at $%s)' % (
  ...             self.name, self.supplier_id, self.price)
  ...
  ...     def __pg_repr__(self):
  ...         return (self.name, self.supplier_id, self.price)
```

Now you can insert class instances the same way as you insert named tuples.

Note that PyGreSQL adapts the result of `__pg_repr__` again if it is a tuple or a list. Otherwise, it must be a properly escaped string.

Typecasting to Python

As you noticed, PyGreSQL automatically converted the PostgreSQL data to suitable Python objects when returning values via one of the "fetch" methods of a cursor. This is done by the use of built-in typecast functions.

If you want to use different typecast functions or add your own if no built-in typecast function is available, then this is possible using the *`set_typecast()`* function. With the *`get_typecast()`* function you can check which function is currently set, and *`reset_typecast()`* allows you to reset the typecast function to its default. If no typecast function is set, then PyGreSQL will return the raw strings from the database.

For instance, you will find that PyGreSQL uses the normal `int` function to cast PostgreSQL `int4` type values to Python:

```
>>> pgdb.get_typecast('int4')
int
```

You can change this to return float values instead:

```
>>> pgdb.set_typecast('int4', float)
>>> con = pgdb.connect(...)
>>> cur = con.cursor()
>>> cur.execute('select 42::int4').fetchone()[0]
42.0
```

Note that the connections cache the typecast functions, so you may need to reopen the database connection, or reset the cache of the connection to make this effective, using the following command:

```
>>> con.type_cache.reset_typecast()
```

The *`TypeCache`* of the connection can also be used to change typecast functions locally for one database connection only.

As a more useful example, we can create a typecast function that casts items of the composite type used as example in the previous section to instances of the corresponding Python class:

```
>>> con.type_cache.reset_typecast()
>>> cast_tuple = con.type_cache.get_typecast('inventory_item')
>>> cast_item = lambda value: InventoryItem(*cast_tuple(value))
>>> con.type_cache.set_typecast('inventory_item', cast_item)
>>> str(cur.execute("SELECT * FROM on_hand").fetchone()[0])
'fuzzy dice (from 42, at $1.99)'
```

As you saw in the last section you, PyGreSQL also has a typecast function for JSON, which is the default JSON decoder from the standard library. Let's assume we want to use a slight variation of that decoder in which every integer in JSON is converted to a float in Python. This can be accomplished as follows:

```
>>> from json import loads
>>> cast_json = lambda v: loads(v, parse_int=float)
>>> pgdb.set_typecast('json', cast_json)
>>> cur.execute("SELECT data FROM json_data").fetchone()[0]
[1.0, 2.0, 3.0]
```

Note again that you may need to run `con.type_cache.reset_typecast()` to make this effective. Also note that the two types `json` and `jsonb` have their own typecast functions, so if you use `jsonb` instead of `json`, you need to use this type name when setting the typecast function:

```
>>> pgdb.set_typecast('jsonb', cast_json)
```

As one last example, let us try to typecast the geometric data type `circle` of PostgreSQL into a SymPy `Circle` object. Let's assume we have created and populated a table with two circles, like so:

```
CREATE TABLE circle (
    name varchar(8) primary key, circle circle);
INSERT INTO circle VALUES ('C1', '<(2, 3), 3>');
INSERT INTO circle VALUES ('C2', '<(1, -1), 4>');
```

With PostgreSQL we can easily calculate that these two circles overlap:

```
>>> con.cursor().execute("""SELECT c1.circle && c2.circle
...     FROM circle c1, circle c2
...     WHERE c1.name = 'C1' AND c2.name = 'C2'""").fetchone()[0]
True
```

However, calculating the intersection points between the two circles using the # operator does not work (at least not as of PostgreSQL version 9.5). So let' resort to SymPy to find out. To ease importing circles from PostgreSQL to SymPy, we create and register the following typecast function:

```
>>> from sympy import Point, Circle
>>>
>>> def cast_circle(s):
...     p, r = s[1:-1].rsplit(',', 1)
...     p = p[1:-1].split(',')
...     return Circle(Point(float(p[0]), float(p[1])), float(r))
...
>>> pgdb.set_typecast('circle', cast_circle)
```

Now we can import the circles in the table into Python quite easily:

```
>>> circle = {c.name: c.circle for c in con.cursor().execute(
...     "SELECT * FROM circle").fetchall()}
```

The result is a dictionary mapping circle names to SymPy `Circle` objects. We can verify that the circles have been imported correctly:

```
>>> circle
{'C1': Circle(Point(2, 3), 3.0),
 'C2': Circle(Point(1, -1), 4.0)}
```

Finally we can find the exact intersection points with SymPy:

```
>>> circle['C1'].intersection(circle['C2'])
[Point(29/17 + 64564173230121*sqrt(17)/100000000000000,
    -80705216537651*sqrt(17)/500000000000000 + 31/17),
 Point(-64564173230121*sqrt(17)/100000000000000 + 29/17,
    80705216537651*sqrt(17)/500000000000000 + 31/17)]
```

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 *Connection* 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 the columns to which the values correspond. The columns can be specified in any order. You may also omit any number of columns, such as with 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 *Query* 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 *Query.getresult()* method:

```
>>> from pprint import pprint
>>> q = db.query("SELECT * FROM weather")
>>> pprint(q.getresult())
```

(continues on next page)

(continued from previous page)

```
[('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 *Query.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, you can also retrieve the results as a list of named tuples, using the *Query.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 different 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 = db.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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
----+-----
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 extract a column out of the resulting 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'
```

(continues on next page)

(continued from previous page)

```
...     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 SQL commands to make all supported changes. 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 = db.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
```

(continues on next page)

(continued from previous page)

```
        AND i.indkey[0] = a.attnum AND a.attrelid = bc.oid
        AND NOT a.attisdropped AND a.attnum>0
    ORDER BY class_name, index_name, attname"""))
```

List user defined attributes

This query lists all user-defined attributes and their types in user-defined tables:

```
print(query("""SELECT c.relname, a.attname,
        format_type(a.atttypid, a.atttypmod)
    FROM pg_class c, pg_attribute a
    WHERE c.relkind = 'r' AND c.relnamespace!=ALL(ARRAY[
        'pg_catalog','pg_toast', 'information_schema']::regnamespace[])
        AND a.attnum > 0
        AND a.attrelid = c.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 *A PostgreSQL Primer* 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

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W