Psycopg Documentation

Release 2.7.4

Federico Di Gregorio

Jul 15, 2019

CONTENTS

Psycopg is the most popular PostgreSQL database adapter for the Python programming language. Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety (several threads can share the same connection). It was designed for heavily multi-threaded applications that create and destroy lots of cursors and make a large number of concurrent `INSERT`s or `UPDATE`s.

Psycopg 2 is mostly implemented in C as a libpq wrapper, resulting in being both efficient and secure. It features client-side and *server-side* cursors, *asynchronous communication* and *notifications*, *COPY* support. Many Python types are supported out-of-the-box and *adapted to matching PostgreSQL data types*; adaptation can be extended and customized thanks to a flexible *objects adaptation system*.

Psycopg 2 is both Unicode and Python 3 friendly.

Contents

CHAPTER

ONE

INTRODUCTION

Psycopg is a PostgreSQL adapter for the Python programming language. It is a wrapper for the libpq, the official PostgreSQL client library.

The *psycopg2* package is the current mature implementation of the adapter: it is a C extension and as such it is only compatible with CPython. If you want to use Psycopg on a different Python implementation (PyPy, Jython, IronPython) there is an experimental porting of Psycopg for Ctypes, but it is not as mature as the C implementation yet.

1.1 Prerequisites

The current `psycopg2` implementation supports:

- Python 2 versions from 2.6 to 2.7
- Python 3 versions from 3.2 to 3.6
- PostgreSQL server versions from 7.4 to 10
- PostgreSQL client library version from 9.1

1.1.1 Build prerequisites

The build prerequisites are to be met in order to install Psycopg from source code, either from a source distribution package or from PyPI.

Psycopg is a C wrapper around the libpq PostgreSQL client library. To install it from sources you will need:

- A C compiler.
- The Python header files. They are usually installed in a package such as **python-dev**. A message such as *error: Python.h: No such file or directory* is an indication that the Python headers are missing.
- The libpq header files. They are usually installed in a package such as **libpq-dev**. If you get an *error: libpq-fe.h: No such file or directory* you are missing them.
- The **`pg_config`** program: it is usually installed by the **libpq-dev** package but sometimes it is not in a `PATH` directory. Having it in the `PATH` greatly streamlines the installation, so try running `pg_config --version`: if it returns an error or an unexpected version number then locate the directory containing the **`pg_config`** shipped with the right libpq version (usually `/usr/lib/postgresql/X.Y/bin/`) and add it to the `PATH`:

```
$ export PATH=/usr/lib/postgresql/X.Y/bin/:$PATH
```

  You only need **`pg_config`** to compile `psycopg2`, not for its regular usage.

Once everything is in place it's just a matter of running the standard:

```
$ pip install psycopg2
```

or, from the directory containing the source code:

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

1.1.2 Runtime requirements

Unless you compile `psycopg2` as a static library, or you install it from a self-contained wheel package, it will need the libpq library at runtime (usually distributed in a `libpq.so` or `libpq.dll` file). `psycopg2` relies on the host OS to find the library if the library is installed in a standard location there is usually no problem; if the library is in a non-standard location you will have to tell somehow Psycopg how to find it, which is OS-dependent (for instance setting a suitable `LD_LIBRARY_PATH` on Linux).

Note: The libpq header files used to compile `psycopg2` should match the version of the library linked at runtime. If you get errors about missing or mismatching libraries when importing `psycopg2` check (e.g. using **`ldd`**) if the module `psycopg2/_psycopg.so` is linked to the right `libpq.so`.

Note: Whatever version of libpq `psycopg2` is compiled with, it will be possible to connect to PostgreSQL servers of any supported version: just install the most recent libpq version or the most practical, without trying to match it to the version of the PostgreSQL server you will have to connect to.

1.2 Binary install from PyPI

`psycopg2` is also available on PyPI in the form of wheel packages for the most common platform (Linux, OSX, Windows): this should make you able to install a binary version of the module, not requiring the above build or runtime prerequisites, simply using:

```
$ pip install psycopg2-binary
```

Make sure to use an up-to-date version of **`pip`** (you can upgrade it using something like `pip install -U pip`)

Note: The binary packages come with their own versions of a few C libraries, among which `libpq` and `libssl`, which will be used regardless of other libraries available on the client: upgrading the system libraries will not upgrade the libraries used by `psycopg2`. Please build `psycopg2` from source if you want to maintain binary upgradeability.

Warning: The `psycopg2` wheel package comes packaged, among the others, with its own `libssl` binary. This may create conflicts with other extension modules binding with `libssl` as well, for instance with the Python `ssl` module: in some cases, under concurrency, the interaction between the two libraries may result in a segfault. In case of doubts you are advised to use a package built from source.

1.2.1 Disabling wheel packages for Psycopg 2.7

In version 2.7.x, `pip install psycopg2` would have tried to install the wheel binary package of Psycopg. Because of the problems the wheel package have displayed, `psycopg2-binary` has become a separate package, and from 2.8 it has become the only way to install the binary package.

If you are using psycopg 2.7 and you want to disable the use of wheel binary packages, relying on the system system libraries available on your client, you can use the **`pip`** `--no-binary` option:

```
$ pip install --no-binary psycopg2
```

which can be specified in your `requirements.txt` files too, e.g. use:

```
psycopg2>=2.7,<2.8 --no-binary psycopg2
```

to use the last bugfix release of the `psycopg2` 2.7 package, specifying to always compile it from source. Of course in this case you will have to meet the *build prerequisites*.

1.3 Non-standard builds

If you have less standard requirements such as:

- creating a *debug build*,
- using **`pg_config`** not in the `PATH`,
- supporting `mx.DateTime`,

then take a look at the `setup.cfg` file.

Some of the options available in `setup.cfg` are also available as command line arguments of the `build_ext` sub-command. For instance you can specify an alternate **`pg_config`** location using:

```
$ python setup.py build_ext --pg-config /path/to/pg_config build
```

Use `python setup.py build_ext --help` to get a list of the options supported.

1.3.1 Creating a debug build

In case of problems, Psycopg can be configured to emit detailed debug messages, which can be very useful for diagnostics and to report a bug. In order to create a debug package:

- Download and unpack the Psycopg source package.
- Edit the `setup.cfg` file adding the `PSYCOPG_DEBUG` flag to the `define` option.
- *Compile and install* the package.
- Set the `PSYCOPG_DEBUG` environment variable:

```
$ export PSYCOPG_DEBUG=1
```

- Run your program (making sure that the `psycopg2` package imported is the one you just compiled and not e.g. the system one): you will have a copious stream of informations printed on stderr.

1.4 Running the test suite

Once `psycopg2` is installed you can run the test suite to verify it is working correctly. You can run:

```
$ python -c "from psycopg2 import tests; tests.unittest.main(defaultTest='tests.test_
↪suite')" --verbose
```

The tests run against a database called `psycopg2_test` on UNIX socket and the standard port. You can configure a different database to run the test by setting the environment variables:

- `PSYCOPG2_TESTDB`
- `PSYCOPG2_TESTDB_HOST`
- `PSYCOPG2_TESTDB_PORT`
- `PSYCOPG2_TESTDB_USER`

The database should already exist before running the tests.

1.5 If you still have problems

Try the following. *In order:*

- Read again the *Build prerequisites*.
- Read the *FAQ*.
- Google for `psycopg2` *your error message*. Especially useful the week after the release of a new OS X version.
- Write to the Mailing List.
- Complain on your blog or on Twitter that `psycopg2` is the worst package ever and about the quality time you have wasted figuring out the correct `ARCHFLAGS`. Especially useful from the Starbucks near you.

CHAPTER

TWO

BASIC MODULE USAGE

The basic Psycopg usage is common to all the database adapters implementing the DB API 2.0 protocol. Here is an interactive session showing some of the basic commands:

```
>>> import psycopg2

# Connect to an existing database
>>> conn = psycopg2.connect("dbname=test user=postgres")

# Open a cursor to perform database operations
>>> cur = conn.cursor()

# Execute a command: this creates a new table
>>> cur.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data varchar);
↪")

# Pass data to fill a query placeholders and let Psycopg perform
# the correct conversion (no more SQL injections!)
>>> cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)",
...      (100, "abc'def"))

# Query the database and obtain data as Python objects
>>> cur.execute("SELECT * FROM test;")
>>> cur.fetchone()
(1, 100, "abc'def")

# Make the changes to the database persistent
>>> conn.commit()

# Close communication with the database
>>> cur.close()
>>> conn.close()
```

The main entry points of Psycopg are:

- The function *connect()* creates a new database session and returns a new *connection* instance.
- The class *connection* encapsulates a database session. It allows to:
  - create new *cursor* instances using the *cursor()* method to execute database commands and queries,
  - terminate transactions using the methods *commit()* or *rollback()*.
- The class *cursor* allows interaction with the database:
  - send commands to the database using methods such as *execute()* and *executemany()*,

  - retrieve data from the database *by iteration* or using methods such as `fetchone()`, `fetchmany()`, `fetchall()`.

2.1 Passing parameters to SQL queries

Psycopg converts Python variables to SQL values using their types: the Python type determines the function used to convert the object into a string representation suitable for PostgreSQL. Many standard Python types are already *adapted to the correct SQL representation*.

Passing parameters to an SQL statement happens in functions such as `cursor.execute()` by using `%s` placeholders in the SQL statement, and passing a sequence of values as the second argument of the function. For example the Python function call:

```
>>> cur.execute("""
...     INSERT INTO some_table (an_int, a_date, a_string)
...     VALUES (%s, %s, %s);
...     """,
...     (10, datetime.date(2005, 11, 18), "O'Reilly"))
```

is converted into a SQL command similar to:

```
INSERT INTO some_table (an_int, a_date, a_string)
VALUES (10, '2005-11-18', 'O''Reilly');
```

Named arguments are supported too using `%(name)s` placeholders in the query and specifying the values into a mapping. Using named arguments allows to specify the values in any order and to repeat the same value in several places in the query:

```
>>> cur.execute("""
...     INSERT INTO some_table (an_int, a_date, another_date, a_string)
...     VALUES (%(int)s, %(date)s, %(date)s, %(str)s);
...     """,
...     {'int': 10, 'str': "O'Reilly", 'date': datetime.date(2005, 11, 18)})
```

Using characters `%`, `(`, `)` in the argument names is not supported.

When parameters are used, in order to include a literal `%` in the query you can use the `%%` string:

```
>>> cur.execute("SELECT (%s % 2) = 0 AS even", (10,))       # WRONG
>>> cur.execute("SELECT (%s %% 2) = 0 AS even", (10,))      # correct
```

While the mechanism resembles regular Python strings manipulation, there are a few subtle differences you should care about when passing parameters to a query.

- The Python string operator `%` *must not be used*: the `execute()` method accepts a tuple or dictionary of values as second parameter. ***Never** use `%` or + to merge values into queries*:

  ```
  >>> cur.execute("INSERT INTO numbers VALUES (%s, %s)" % (10, 20)) # WRONG
  >>> cur.execute("INSERT INTO numbers VALUES (%s, %s)", (10, 20))  # correct
  ```

- For positional variables binding, *the second argument must always be a sequence*, even if it contains a single variable (remember that Python requires a comma to create a single element tuple):

  ```
  >>> cur.execute("INSERT INTO foo VALUES (%s)", "bar")    # WRONG
  >>> cur.execute("INSERT INTO foo VALUES (%s)", ("bar"))  # WRONG
  ```

(continues on next page)

(continued from previous page)

```
>>> cur.execute("INSERT INTO foo VALUES (%s)", ("bar",)) # correct
>>> cur.execute("INSERT INTO foo VALUES (%s)", ["bar"])  # correct
```

- The placeholder *must not be quoted*. Psycopg will add quotes where needed:

```
>>> cur.execute("INSERT INTO numbers VALUES ('%s')", (10,)) # WRONG
>>> cur.execute("INSERT INTO numbers VALUES (%s)", (10,))   # correct
```

- The variables placeholder *must always be a* `%s`, even if a different placeholder (such as a `%d` for integers or `%f` for floats) may look more appropriate:

```
>>> cur.execute("INSERT INTO numbers VALUES (%d)", (10,))   # WRONG
>>> cur.execute("INSERT INTO numbers VALUES (%s)", (10,))   # correct
```

- Only query values should be bound via this method: it shouldn't be used to merge table or field names to the query (Psycopg will try quoting the table name as a string value, generating invalid SQL). If you need to generate dynamically SQL queries (for instance choosing dynamically a table name) you can use the facilities provided by the *`psycopg2.sql`* module:

```
>>> cur.execute("INSERT INTO %s VALUES (%s)", ('numbers', 10))   # WRONG
>>> cur.execute(                                                 # correct
...     SQL("INSERT INTO {} VALUES (%s)").format(Identifier('numbers')),
...     (10,))
```

2.1.1 The problem with the query parameters

The SQL representation of many data types is often different from their Python string representation. The typical example is with single quotes in strings: in SQL single quotes are used as string literal delimiters, so the ones appearing inside the string itself must be escaped, whereas in Python single quotes can be left unescaped if the string is delimited by double quotes.

Because of the difference, sometime subtle, between the data types representations, a naïve approach to query strings composition, such as using Python strings concatenation, is a recipe for *terrible* problems:

```
>>> SQL = "INSERT INTO authors (name) VALUES ('%s');" # NEVER DO THIS
>>> data = ("O'Reilly", )
>>> cur.execute(SQL % data) # THIS WILL FAIL MISERABLY
ProgrammingError: syntax error at or near "Reilly"
LINE 1: INSERT INTO authors (name) VALUES ('O'Reilly')
                                             ^
```

If the variables containing the data to send to the database come from an untrusted source (such as a form published on a web site) an attacker could easily craft a malformed string, either gaining access to unauthorized data or performing destructive operations on the database. This form of attack is called SQL injection and is known to be one of the most widespread forms of attack to database servers. Before continuing, please print this page as a memo and hang it onto your desk.

Psycopg can *automatically convert Python objects to and from SQL literals*: using this feature your code will be more robust and reliable. We must stress this point:

> **Warning:** Never, **never**, **NEVER** use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string. Not even at gunpoint.

The correct way to pass variables in a SQL command is using the second argument of the *execute()* method:

```
>>> SQL = "INSERT INTO authors (name) VALUES (%s);" # Note: no quotes
>>> data = ("O'Reilly", )
>>> cur.execute(SQL, data) # Note: no % operator
```

2.2 Adaptation of Python values to SQL types

Many standard Python types are adapted into SQL and returned as Python objects when a query is executed.

The following table shows the default mapping between Python and PostgreSQL types:

The mapping is fairly customizable: see *Adapting new Python types to SQL syntax* and *Type casting of SQL types into Python objects*. You can also find a few other specialized adapters in the *psycopg2.extras* module.

2.2.1 Constants adaptation

Python `None` and boolean values `True` and `False` are converted into the proper SQL literals:

```
>>> cur.mogrify("SELECT %s, %s, %s;", (None, True, False))
'SELECT NULL, true, false;'
```

2.2.2 Numbers adaptation

Python numeric objects `int`, `long`, `float`, `Decimal` are converted into a PostgreSQL numerical representation:

```
>>> cur.mogrify("SELECT %s, %s, %s, %s;", (10, 10L, 10.0, Decimal("10.00")))
'SELECT 10, 10, 10.0, 10.00;'
```

Reading from the database, integer types are converted into `int`, floating point types are converted into `float`, `numeric`/`decimal` are converted into `Decimal`.

Note: Sometimes you may prefer to receive `numeric` data as `float` instead, for performance reason or ease of manipulation: you can configure an adapter to *cast PostgreSQL numeric to Python float*. This of course may imply a loss of precision.

See also:

PostgreSQL numeric types

2.2.3 Strings adaptation

Python `str` and `unicode` are converted into the SQL string syntax. `unicode` objects (`str` in Python 3) are encoded in the connection *encoding* before sending to the backend: trying to send a character not supported by the encoding will result in an error. Data is usually received as `str` (*i.e.* it is *decoded* on Python 3, left *encoded* on Python 2). However it is possible to receive `unicode` on Python 2 too: see *Unicode handling*.

Unicode handling

Psycopg can exchange Unicode data with a PostgreSQL database. Python `unicode` objects are automatically *encoded* in the client encoding defined on the database connection (the PostgreSQL encoding, available in *connection.encoding*, is translated into a Python encoding using the *encodings* mapping):

```
>>> print u, type(u)
àèìòù€ <type 'unicode'>

>>> cur.execute("INSERT INTO test (num, data) VALUES (%s,%s);", (74, u))
```

When reading data from the database, in Python 2 the strings returned are usually 8 bit `str` objects encoded in the database client encoding:

```
>>> print conn.encoding
UTF8

>>> cur.execute("SELECT data FROM test WHERE num = 74")
>>> x = cur.fetchone()[0]
>>> print x, type(x), repr(x)
àèìòù€ <type 'str'> '\xc3\xa0\xc3\xa8\xc3\xac\xc3\xb2\xc3\xb9\xe2\x82\xac'

>>> conn.set_client_encoding('LATIN9')

>>> cur.execute("SELECT data FROM test WHERE num = 74")
>>> x = cur.fetchone()[0]
>>> print type(x), repr(x)
<type 'str'> '\xe0\xe8\xec\xf2\xf9\xa4'
```

In Python 3 instead the strings are automatically *decoded* in the connection *encoding*, as the `str` object can represent Unicode characters. In Python 2 you must register a *typecaster* in order to receive `unicode` objects:

```
>>> psycopg2.extensions.register_type(psycopg2.extensions.UNICODE, cur)

>>> cur.execute("SELECT data FROM test WHERE num = 74")
>>> x = cur.fetchone()[0]
>>> print x, type(x), repr(x)
àèìòù€ <type 'unicode'> u'\xe0\xe8\xec\xf2\xf9\u20ac'
```

In the above example, the *UNICODE* typecaster is registered only on the cursor. It is also possible to register typecasters on the connection or globally: see the function *register_type()* and *Type casting of SQL types into Python objects* for details.

Note: In Python 2, if you want to uniformly receive all your database input in Unicode, you can register the related typecasters globally as soon as Psycopg is imported:

```
import psycopg2
import psycopg2.extensions
psycopg2.extensions.register_type(psycopg2.extensions.UNICODE)
psycopg2.extensions.register_type(psycopg2.extensions.UNICODEARRAY)
```

and forget about this story.

2.2.4 Binary adaptation

Python types representing binary objects are converted into PostgreSQL binary string syntax, suitable for `bytea` fields. Such types are `buffer` (only available in Python 2), `memoryview` (available from Python 2.7), `bytearray` (available from Python 2.6) and `bytes` (only from Python 3: the name is available from Python 2.6 but it's only an alias for the type `str`). Any object implementing the Revised Buffer Protocol should be usable as binary type where the protocol is supported (i.e. from Python 2.6). Received data is returned as `buffer` (in Python 2) or `memoryview` (in Python 3).

Changed in version 2.4: only strings were supported before.

Changed in version 2.4.1: can parse the 'hex' format from 9.0 servers without relying on the version of the client library.

Note: In Python 2, if you have binary data in a `str` object, you can pass them to a `bytea` field using the *psycopg2.Binary* wrapper:

```
mypic = open('picture.png', 'rb').read()
curs.execute("insert into blobs (file) values (%s)",
    (psycopg2.Binary(mypic),))
```

Warning: Since version 9.0 PostgreSQL uses by default a new "hex" format to emit `bytea` fields. Starting from Psycopg 2.4.1 the format is correctly supported. If you use a previous version you will need some extra care when receiving bytea from PostgreSQL: you must have at least libpq 9.0 installed on the client or alternatively you can set the bytea_output configuration parameter to `escape`, either in the server configuration file or in the client session (using a query such as `SET bytea_output TO escape;`) before receiving binary data.

2.2.5 Date/Time objects adaptation

Python builtin `datetime`, `date`, `time`, `timedelta` are converted into PostgreSQL's `timestamp[tz]`, `date`, `time[tz]`, `interval` data types. Time zones are supported too. The Egenix mx.DateTime objects are adapted the same way:

```
>>> dt = datetime.datetime.now()
>>> dt
datetime.datetime(2010, 2, 8, 1, 40, 27, 425337)

>>> cur.mogrify("SELECT %s, %s, %s;", (dt, dt.date(), dt.time()))
"SELECT '2010-02-08T01:40:27.425337', '2010-02-08', '01:40:27.425337';"

>>> cur.mogrify("SELECT %s;", (dt - datetime.datetime(2010,1,1),))
"SELECT '38 days 6027.425337 seconds';"
```

See also:

PostgreSQL date/time types

Time zones handling

The PostgreSQL type `timestamp with time zone` (a.k.a. `timestamptz`) is converted into Python `datetime` objects with a `tzinfo` attribute set to a *FixedOffsetTimezone* instance.

```
>>> cur.execute("SET TIME ZONE 'Europe/Rome';")  # UTC + 1 hour
>>> cur.execute("SELECT '2010-01-01 10:30:45'::timestamptz;")
>>> cur.fetchone()[0].tzinfo
psycopg2.tz.FixedOffsetTimezone(offset=60, name=None)
```

Note that only time zones with an integer number of minutes are supported: this is a limitation of the Python `datetime` module. A few historical time zones had seconds in the UTC offset: these time zones will have the offset rounded to the nearest minute, with an error of up to 30 seconds.

```
>>> cur.execute("SET TIME ZONE 'Asia/Calcutta';")  # offset was +5:53:20
>>> cur.execute("SELECT '1930-01-01 10:30:45'::timestamptz;")
>>> cur.fetchone()[0].tzinfo
psycopg2.tz.FixedOffsetTimezone(offset=353, name=None)
```

Changed in version 2.2.2: timezones with seconds are supported (with rounding). Previously such timezones raised an error. In order to deal with them in previous versions use *psycopg2.extras.register_tstz_w_secs()*.

Infinite dates handling

PostgreSQL can store the representation of an "infinite" date, timestamp, or interval. Infinite dates are not available to Python, so these objects are mapped to `date.max`, `datetime.max`, `interval.max`. Unfortunately the mapping cannot be bidirectional so these dates will be stored back into the database with their values, such as `9999-12-31`.

It is possible to create an alternative adapter for dates and other objects to map `date.max` to `infinity`, for instance:

```
class InfDateAdapter:
    def __init__(self, wrapped):
        self.wrapped = wrapped
    def getquoted(self):
        if self.wrapped == datetime.date.max:
            return b"'infinity'::date"
        elif self.wrapped == datetime.date.min:
            return b"'-infinity'::date"
        else:
            return psycopg2.extensions.DateFromPy(self.wrapped).getquoted()

psycopg2.extensions.register_adapter(datetime.date, InfDateAdapter)
```

Of course it will not be possible to write the value of `date.max` in the database anymore: `infinity` will be stored instead.

2.2.6 Lists adaptation

Python lists are converted into PostgreSQL `ARRAY`s:

```
>>> cur.mogrify("SELECT %s;", ([10, 20, 30], ))
'SELECT ARRAY[10,20,30];'
```

Note: You can use a Python list as the argument of the `IN` operator using the PostgreSQL ANY operator.

```
ids = [10, 20, 30]
cur.execute("SELECT * FROM data WHERE id = ANY(%s);", (ids,))
```

Furthermore `ANY` can also work with empty lists, whereas `IN ()` is a SQL syntax error.

Note: Reading back from PostgreSQL, arrays are converted to lists of Python objects as expected, but only if the items are of a known type. Arrays of unknown types are returned as represented by the database (e.g. `{a,b,c}`). If you want to convert the items into Python objects you can easily create a typecaster for *array of unknown types*.

2.2.7 Tuples adaptation

Python tuples are converted into a syntax suitable for the SQL `IN` operator and to represent a composite type:

```
>>> cur.mogrify("SELECT %s IN %s;", (10, (10, 20, 30)))
'SELECT 10 IN (10, 20, 30);'
```

Note: SQL doesn't allow an empty list in the `IN` operator, so your code should guard against empty tuples. Alternatively you can *use a Python list*.

If you want PostgreSQL composite types to be converted into a Python tuple/namedtuple you can use the *`register_composite()`* function.

New in version 2.0.6: the tuple `IN` adaptation.

Changed in version 2.0.14: the tuple `IN` adapter is always active. In previous releases it was necessary to import the *`extensions`* module to have it registered.

Changed in version 2.3: `namedtuple` instances are adapted like regular tuples and can thus be used to represent composite types.

2.3 Transactions control

In Psycopg transactions are handled by the *`connection`* class. By default, the first time a command is sent to the database (using one of the *`cursor`*s created by the connection), a new transaction is created. The following database commands will be executed in the context of the same transaction – not only the commands issued by the first cursor, but the ones issued by all the cursors created by the same connection. Should any command fail, the transaction will be aborted and no further command will be executed until a call to the *`rollback()`* method.

The connection is responsible for terminating its transaction, calling either the *`commit()`* or *`rollback()`* method. Committed changes are immediately made persistent into the database. Closing the connection using the *`close()`* method or destroying the connection object (using `del` or letting it fall out of scope) will result in an implicit rollback.

It is possible to set the connection in *autocommit* mode: this way all the commands executed will be immediately committed and no rollback is possible. A few commands (e.g. `CREATE DATABASE`, `VACUUM`...) require to be run outside any transaction: in order to be able to run these commands from Psycopg, the connection must be in autocommit mode: you can use the *`autocommit`* property.

Warning: By default even a simple `SELECT` will start a transaction: in long-running programs, if no further action is taken, the session will remain "idle in transaction", an undesirable condition for several reasons (locks are held by the session, tables bloat...). For long lived scripts, either make sure to terminate a transaction as soon as possible or use an autocommit connection.

A few other transaction properties can be set session-wide by the `connection`: for instance it is possible to have read-only transactions or change the isolation level. See the *`set_session()`* method for all the details.

2.3.1 `with` statement

Starting from version 2.5, psycopg2's connections and cursors are *context managers* and can be used with the `with` statement:

```
with psycopg2.connect(DSN) as conn:
    with conn.cursor() as curs:
        curs.execute(SQL)
```

When a connection exits the `with` block, if no exception has been raised by the block, the transaction is committed. In case of exception the transaction is rolled back.

When a cursor exits the `with` block it is closed, releasing any resource eventually associated with it. The state of the transaction is not affected.

Note that, unlike file objects or other resources, exiting the connection's `with` block *doesn't close the connection* but only the transaction associated with it: a connection can be used in more than a `with` statement and each `with` block is effectively wrapped in a separate transaction:

```
conn = psycopg2.connect(DSN)

with conn:
    with conn.cursor() as curs:
        curs.execute(SQL1)

with conn:
    with conn.cursor() as curs:
        curs.execute(SQL2)

conn.close()
```

2.4 Server side cursors

When a database query is executed, the Psycopg *`cursor`* usually fetches all the records returned by the backend, transferring them to the client process. If the query returned an huge amount of data, a proportionally large amount of memory will be allocated by the client.

If the dataset is too large to be practically handled on the client side, it is possible to create a *server side* cursor. Using this kind of cursor it is possible to transfer to the client only a controlled amount of data, so that a large dataset can be examined without keeping it entirely in memory.

Server side cursor are created in PostgreSQL using the `DECLARE` command and subsequently handled using `MOVE`, `FETCH` and `CLOSE` commands.

Psycopg wraps the database server side cursor in *named cursors*. A named cursor is created using the *`cursor()`* method specifying the *name* parameter. Such cursor will behave mostly like a regular cursor, allowing the user to move in the dataset using the *`scroll()`* method and to read the data using *`fetchone()`* and *`fetchmany()`* methods. Normally you can only scroll forward in a cursor: if you need to scroll backwards you should declare your cursor *`scrollable`*.

Named cursors are also *iterable* like regular cursors. Note however that before Psycopg 2.4 iteration was performed fetching one record at time from the backend, resulting in a large overhead. The attribute *`itersize`* now controls

how many records are fetched at time during the iteration: the default value of 2000 allows to fetch about 100KB per roundtrip assuming records of 10-20 columns of mixed number and strings; you may decrease this value if you are dealing with huge records.

Named cursors are usually created `WITHOUT HOLD`, meaning they live only as long as the current transaction. Trying to fetch from a named cursor after a *`commit()`* or to create a named cursor when the connection is in *`autocommit`* mode will result in an exception. It is possible to create a `WITH HOLD` cursor by specifying a `True` value for the `withhold` parameter to *`cursor()`* or by setting the *`withhold`* attribute to `True` before calling *`execute()`* on the cursor. It is extremely important to always *`close()`* such cursors, otherwise they will continue to hold server-side resources until the connection will be eventually closed. Also note that while `WITH HOLD` cursors lifetime extends well after *`commit()`*, calling *`rollback()`* will automatically close the cursor.

Note: It is also possible to use a named cursor to consume a cursor created in some other way than using the `DECLARE` executed by *`execute()`*. For example, you may have a PL/pgSQL function returning a cursor:

```
CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
$$ LANGUAGE plpgsql;
```

You can read the cursor content by calling the function with a regular, non-named, Psycopg cursor:

```
cur1 = conn.cursor()
cur1.callproc('reffunc', ['curname'])
```

and then use a named cursor in the same transaction to "steal the cursor":

```
cur2 = conn.cursor('curname')
for record in cur2:     # or cur2.fetchone, fetchmany...
    # do something with record
    pass
```

2.5 Thread and process safety

The Psycopg module and the *`connection`* objects are *thread-safe*: many threads can access the same database either using separate sessions and creating a `connection` per thread or using the same connection and creating separate *`cursor`*s. In DB API 2.0 parlance, Psycopg is *level 2 thread safe*.

The difference between the above two approaches is that, using different connections, the commands will be executed in different sessions and will be served by different server processes. On the other hand, using many cursors on the same connection, all the commands will be executed in the same session (and in the same transaction if the connection is not in *autocommit* mode), but they will be serialized.

The above observations are only valid for regular threads: they don't apply to forked processes nor to green threads. `libpq` connections shouldn't be used by a forked processes, so when using a module such as `multiprocessing` or a forking web deploy method such as FastCGI make sure to create the connections *after* the fork.

Connections shouldn't be shared either by different green threads: see *Support for coroutine libraries* for further details.

2.6 Using COPY TO and COPY FROM

Psycopg *cursor* objects provide an interface to the efficient PostgreSQL *COPY* command to move data from files to tables and back.

Currently no adaptation is provided between Python and PostgreSQL types on `COPY`: the file can be any Python file-like object but its format must be in the format accepted by PostgreSQL COPY command (data format, escaped characters, etc).

The methods exposed are:

copy_from() Reads data *from* a file-like object appending them to a database table (`COPY table FROM file` syntax). The source file must provide both `read()` and `readline()` method.

copy_to() Writes the content of a table *to* a file-like object (`COPY table TO file` syntax). The target file must have a `write()` method.

copy_expert() Allows to handle more specific cases and to use all the `COPY` features available in PostgreSQL.

Please refer to the documentation of the single methods for details and examples.

2.7 Access to PostgreSQL large objects

PostgreSQL offers support for large objects, which provide stream-style access to user data that is stored in a special large-object structure. They are useful with data values too large to be manipulated conveniently as a whole.

Psycopg allows access to the large object using the *lobject* class. Objects are generated using the *connection.lobject()* factory method. Data can be retrieved either as bytes or as Unicode strings.

Psycopg large object support efficient import/export with file system files using the `lo_import()` and `lo_export()` libpq functions.

Changed in version 2.6: added support for large objects greated than 2GB. Note that the support is enabled only if all the following conditions are verified:

- the Python build is 64 bits;
- the extension was built against at least libpq 9.3;
- the server version is at least PostgreSQL 9.3 (*server_version* must be >= `90300`).

If Psycopg was built with 64 bits large objects support (i.e. the first two contidions above are verified), the `psycopg2.__version__` constant will contain the `lo64` flag. If any of the contition is not met several `lobject` methods will fail if the arguments exceed 2GB.

2.8 Two-Phase Commit protocol support

New in version 2.3.

Psycopg exposes the two-phase commit features available since PostgreSQL 8.1 implementing the *two-phase commit extensions* proposed by the DB API 2.0.

The DB API 2.0 model of two-phase commit is inspired by the XA specification, according to which transaction IDs are formed from three components:

- a format ID (non-negative 32 bit integer)
- a global transaction ID (string not longer than 64 bytes)

- a branch qualifier (string not longer than 64 bytes)

For a particular global transaction, the first two components will be the same for all the resources. Every resource will be assigned a different branch qualifier.

According to the DB API 2.0 specification, a transaction ID is created using the *connection.xid()* method. Once you have a transaction id, a distributed transaction can be started with *connection.tpc_begin()*, prepared using *tpc_prepare()* and completed using *tpc_commit()* or *tpc_rollback()*. Transaction IDs can also be retrieved from the database using *tpc_recover()* and completed using the above `tpc_commit()` and `tpc_rollback()`.

PostgreSQL doesn't follow the XA standard though, and the ID for a PostgreSQL prepared transaction can be any string up to 200 characters long. Psycopg's *Xid* objects can represent both XA-style transactions IDs (such as the ones created by the `xid()` method) and PostgreSQL transaction IDs identified by an unparsed string.

The format in which the Xids are converted into strings passed to the database is the same employed by the PostgreSQL JDBC driver: this should allow interoperation between tools written in Python and in Java. For example a recovery tool written in Python would be able to recognize the components of transactions produced by a Java program.

For further details see the documentation for the above methods.

CHAPTER

THREE

THE psycopg2 MODULE CONTENT

The module interface respects the standard defined in the DB API 2.0.

psycopg2.**connect**(*dsn=None*, *connection_factory=None*, *cursor_factory=None*, *async=False*, ***kwargs*)

Create a new database session and return a new *connection* object.

The connection parameters can be specified as a libpq connection string using the *dsn* parameter:

```
conn = psycopg2.connect("dbname=test user=postgres password=secret")
```

or using a set of keyword arguments:

```
conn = psycopg2.connect(dbname="test", user="postgres", password="secret")
```

or using a mix of both: if the same parameter name is specified in both sources, the *kwargs* value will have precedence over the *dsn* value. Note that either the *dsn* or at least one connection-related keyword argument is required.

The basic connection parameters are:

- `dbname` – the database name (`database` is a deprecated alias)
- `user` – user name used to authenticate
- `password` – password used to authenticate
- `host` – database host address (defaults to UNIX socket if not provided)
- `port` – connection port number (defaults to 5432 if not provided)

Any other connection parameter supported by the client library/server can be passed either in the connection string or as a keyword. The PostgreSQL documentation contains the complete list of the supported parameters. Also note that the same parameters can be passed to the client library using environment variables.

Using the *connection_factory* parameter a different class or connections factory can be specified. It should be a callable object taking a *dsn* string argument. See *Connection and cursor factories* for details. If a *cursor_factory* is specified, the connection's *cursor_factory* is set to it. If you only need customized cursors you can use this parameter instead of subclassing a connection.

Using *async*=`True` an asynchronous connection will be created: see *Asynchronous support* to know about advantages and limitations. *async_* is a valid alias for the Python version where `async` is a keyword.

Changed in version 2.4.3: any keyword argument is passed to the connection. Previously only the basic parameters (plus `sslmode`) were supported as keywords.

Changed in version 2.5: added the *cursor_factory* parameter.

Changed in version 2.7: both *dsn* and keyword arguments can be specified.

Changed in version 2.7: added *async_* alias.

See also:

- *parse_dsn*
- libpq connection string syntax
- libpq supported connection parameters
- libpq supported environment variables

DB API extension

The non-connection-related keyword parameters are Psycopg extensions to the DB API 2.0.

---

psycopg2.**apilevel**
String constant stating the supported DB API level. For *psycopg2* is `2.0`.

psycopg2.**threadsafety**
Integer constant stating the level of thread safety the interface supports. For *psycopg2* is `2`, i.e. threads can share the module and the connection. See *Thread and process safety* for details.

psycopg2.**paramstyle**
String constant stating the type of parameter marker formatting expected by the interface. For *psycopg2* is `pyformat`. See also *Passing parameters to SQL queries*.

psycopg2.**__libpq_version__**
Integer constant reporting the version of the `libpq` library this `psycopg2` module was compiled with (in the same format of *server_version*). If this value is greater or equal than `90100` then you may query the version of the actually loaded library using the *libpq_version()* function.

3.1 Exceptions

In compliance with the DB API 2.0, the module makes informations about errors available through the following exceptions:

exception psycopg2.**Warning**
Exception raised for important warnings like data truncations while inserting, etc. It is a subclass of the Python `StandardError`.

exception psycopg2.**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 not use this class as base. It is a subclass of the Python `StandardError`.

pgerror
String representing the error message returned by the backend, `None` if not available.

pgcode
String representing the error code returned by the backend, `None` if not available. The *errorcodes* module contains symbolic constants representing PostgreSQL error codes.

```
>>> try:
...     cur.execute("SELECT * FROM barf")
... except psycopg2.Error as e:
```

(continues on next page)

(continued from previous page)

```
...     pass

>>> e.pgcode
'42P01'
>>> print e.pgerror
ERROR:  relation "barf" does not exist
LINE 1: SELECT * FROM barf
                      ^
```

cursor
The cursor the exception was raised from; `None` if not applicable.

diag
A *Diagnostics* object containing further information about the error.

```
>>> try:
...     cur.execute("SELECT * FROM barf")
... except psycopg2.Error, e:
...     pass

>>> e.diag.severity
'ERROR'
>>> e.diag.message_primary
'relation "barf" does not exist'
```

New in version 2.5.

DB API extension

The *pgerror*, *pgcode*, *cursor*, and *diag* attributes are Psycopg extensions.

---

exception psycopg2.**InterfaceError**
Exception raised for errors that are related to the database interface rather than the database itself. It is a subclass of *Error*.

exception psycopg2.**DatabaseError**
Exception raised for errors that are related to the database. It is a subclass of *Error*.

exception psycopg2.**DataError**
Exception raised for errors that are due to problems with the processed data like division by zero, numeric value out of range, etc. It is a subclass of *DatabaseError*.

exception psycopg2.**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, a memory allocation error occurred during processing, etc. It is a subclass of *DatabaseError*.

exception psycopg2.**IntegrityError**
Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails. It is a subclass of *DatabaseError*.

exception psycopg2.**InternalError**
Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc. It is a subclass of *DatabaseError*.

exception psycopg2.**ProgrammingError**

Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc. It is a subclass of *DatabaseError*.

exception psycopg2.**NotSupportedError**
Exception raised in case a method or database API was used which is not supported by the database, e.g. requesting a `rollback()` on a connection that does not support transaction or has transactions turned off. It is a subclass of *DatabaseError*.

DB API extension

Psycopg may raise a few other, more specialized, exceptions: currently *QueryCanceledError* and *TransactionRollbackError* are defined. These exceptions are not exposed by the main `psycopg2` module but are made available by the *extensions* module. All the additional exceptions are subclasses of standard DB API 2.0 exceptions, so trapping them specifically is not required.

This is the exception inheritance layout:

```
StandardError
|__ Warning
|__ Error
    |__ InterfaceError
    |__ DatabaseError
        |__ DataError
        |__ OperationalError
        |   |__ psycopg2.extensions.QueryCanceledError
        |   |__ psycopg2.extensions.TransactionRollbackError
        |__ IntegrityError
        |__ InternalError
        |__ ProgrammingError
        |__ NotSupportedError
```

3.2 Type Objects and Constructors

Note: This section is mostly copied verbatim from the DB API 2.0 specification. While these objects are exposed in compliance to the DB API, Psycopg offers very accurate tools to convert data between Python and PostgreSQL formats. See *Adapting new Python types to SQL syntax* and *Type casting of SQL types into Python objects*

Many databases need to have the input in a particular format for binding to an operation's input parameters. For example, if an input is destined for a DATE column, then it must be bound to the database in a particular string format. Similar problems exist for "Row ID" columns or large binary items (e.g. blobs or RAW columns). This presents problems for Python since the parameters to the .execute*() method are untyped. When the database module sees a Python string object, it doesn't know if it should be bound as a simple CHAR column, as a raw BINARY item, or as a DATE.

To overcome this problem, a module must provide the constructors defined below to create objects that can hold special values. When passed to the cursor methods, the module can then detect the proper type of the input parameter and bind it accordingly.

A Cursor Object's description attribute returns information about each of the result columns of a query. The type_code must compare equal to one of Type Objects defined below. Type Objects may be equal to more than one type code (e.g. DATETIME could be equal to the type codes for date, time and timestamp columns; see the Implementation Hints below for details).

The module exports the following constructors and singletons:

psycopg2.**Date**(*year*, *month*, *day*)
This function constructs an object holding a date value.

psycopg2.**Time**(*hour*, *minute*, *second*)
This function constructs an object holding a time value.

psycopg2.**Timestamp**(*year*, *month*, *day*, *hour*, *minute*, *second*)
This function constructs an object holding a time stamp value.

psycopg2.**DateFromTicks**(*ticks*)
This function constructs an object holding a date value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

psycopg2.**TimeFromTicks**(*ticks*)
This function constructs an object holding a time value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

psycopg2.**TimestampFromTicks**(*ticks*)
This function constructs an object holding a time stamp value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

psycopg2.**Binary**(*string*)
This function constructs an object capable of holding a binary (long) string value.

Note: All the adapters returned by the module level factories (`Binary`, `Date`, `Time`, `Timestamp` and the `*FromTicks` variants) expose the wrapped object (a regular Python object such as `datetime`) in an `adapted` attribute.

---

psycopg2.**STRING**
This type object is used to describe columns in a database that are string-based (e.g. CHAR).

psycopg2.**BINARY**
This type object is used to describe (long) binary columns in a database (e.g. LONG, RAW, BLOBs).

psycopg2.**NUMBER**
This type object is used to describe numeric columns in a database.

psycopg2.**DATETIME**
This type object is used to describe date/time columns in a database.

psycopg2.**ROWID**
This type object is used to describe the "Row ID" column in a database.

CHAPTER

FOUR

THE CONNECTION CLASS

class connection

Handles the connection to a PostgreSQL database instance. It encapsulates a database session.

Connections are created using the factory function *connect()*.

Connections are thread safe and can be shared among many threads. See *Thread and process safety* for details.

cursor(*name=None*, *cursor_factory=None*, *scrollable=None*, *withhold=False*)

Return a new *cursor* object using the connection.

If *name* is specified, the returned cursor will be a *server side cursor* (also known as *named cursor*). Otherwise it will be a regular *client side* cursor. By default a named cursor is declared without SCROLL option and WITHOUT HOLD: set the argument or property *scrollable* to True/False and or *withhold* to True to change the declaration.

The name can be a string not valid as a PostgreSQL identifier: for example it may start with a digit and contain non-alphanumeric characters and quotes.

Changed in version 2.4: previously only valid PostgreSQL identifiers were accepted as cursor name.

The *cursor_factory* argument can be used to create non-standard cursors. The class returned must be a subclass of *psycopg2.extensions.cursor*. See *Connection and cursor factories* for details. A default factory for the connection can also be specified using the *cursor_factory* attribute.

Changed in version 2.4.3: added the *withhold* argument.

Changed in version 2.5: added the *scrollable* argument.

DB API extension

All the function arguments are Psycopg extensions to the DB API 2.0.

---

commit()

Commit any pending transaction to the database.

By default, Psycopg opens a transaction before executing the first command: if commit() is not called, the effect of any data manipulation will be lost.

The connection can be also set in "autocommit" mode: no transaction is automatically open, commands have immediate effect. See *Transactions control* for details.

Changed in version 2.5: if the connection is used in a with statement, the method is automatically called if no exception is raised in the with block.

rollback()

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.

Changed in version 2.5: if the connection is used in a `with` statement, the method is automatically called if an exception is raised in the `with` block.

close()
Close the connection now (rather than whenever `del` is executed). The connection will be unusable from this point forward; an *InterfaceError* 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 any pending change to be discarded as if a `ROLLBACK` was performed (unless a different isolation level has been selected: see *set_isolation_level()*).

Changed in version 2.2: previously an explicit `ROLLBACK` was issued by Psycopg on `close()`. The command could have been sent to the backend at an inappropriate time, so Psycopg currently relies on the backend to implicitly discard uncommitted changes. Some middleware are known to behave incorrectly though when the connection is closed during a transaction (when *status* is *STATUS_IN_TRANSACTION*), e.g. PgBouncer reports an `unclean server` and discards the connection. To avoid this problem you can ensure to terminate the transaction with a *commit()*/*rollback()* before closing.

Exceptions as connection class attributes

The `connection` also exposes as attributes the same exceptions available in the *psycopg2* module. See *Exceptions*.

Two-phase commit support methods

New in version 2.3.

See also:

Two-Phase Commit protocol support for an introductory explanation of these methods.

Note that PostgreSQL supports two-phase commit since release 8.1: these methods raise *NotSupportedError* if used with an older version server.

xid(*format_id*, *gtrid*, *bqual*)
Returns a *Xid* instance to be passed to the `tpc_*()` methods of this connection. The argument types and constraints are explained in *Two-Phase Commit protocol support*.

The values passed to the method will be available on the returned object as the members *format_id*, *gtrid*, *bqual*. The object also allows accessing to these members and unpacking as a 3-items tuple.

tpc_begin(*xid*)
Begins a TPC transaction with the given transaction ID *xid*.

This method should be called outside of a transaction (i.e. nothing may have executed since the last *commit()* or *rollback()* and *connection.status* is *STATUS_READY*).

Furthermore, it is an error to call `commit()` or `rollback()` within the TPC transaction: in this case a *ProgrammingError* is raised.

The *xid* may be either an object returned by the *xid()* method or a plain string: the latter allows to create a transaction using the provided string as PostgreSQL transaction id. See also *tpc_recover()*.

tpc_prepare()
Performs the first phase of a transaction started with *tpc_begin()*. A *ProgrammingError* is raised if this method is used outside of a TPC transaction.

After calling `tpc_prepare()`, no statements can be executed until *tpc_commit()* or *tpc_rollback()* will be called. The *reset()* method can be used to restore the status of the con-

nection to *STATUS_READY*: the transaction will remain prepared in the database and will be possible to finish it with `tpc_commit(xid)` and `tpc_rollback(xid)`.

See also:

the `PREPARE TRANSACTION` PostgreSQL command.

tpc_commit([*xid*])
When called with no arguments, `tpc_commit()` commits a TPC transaction previously prepared with *tpc_prepare()*.

If `tpc_commit()` is called prior to `tpc_prepare()`, a single phase commit is performed. A transaction manager may choose to do this if only a single resource is participating in the global transaction.

When called with a transaction ID *xid*, the database commits the given transaction. If an invalid transaction ID is provided, a *ProgrammingError* will be raised. This form should be called outside of a transaction, and is intended for use in recovery.

On return, the TPC transaction is ended.

See also:

the `COMMIT PREPARED` PostgreSQL command.

tpc_rollback([*xid*])
When called with no arguments, `tpc_rollback()` rolls back a TPC transaction. It may be called before or after *tpc_prepare()*.

When called with a transaction ID *xid*, it rolls back the given transaction. If an invalid transaction ID is provided, a *ProgrammingError* is raised. This form should be called outside of a transaction, and is intended for use in recovery.

On return, the TPC transaction is ended.

See also:

the `ROLLBACK PREPARED` PostgreSQL command.

tpc_recover()
Returns a list of *Xid* representing pending transactions, suitable for use with *tpc_commit()* or *tpc_rollback()*.

If a transaction was not initiated by Psycopg, the returned Xids will have attributes *format_id* and *bqual* set to `None` and the *gtrid* set to the PostgreSQL transaction ID: such Xids are still usable for recovery. Psycopg uses the same algorithm of the PostgreSQL JDBC driver to encode a XA triple in a string, so transactions initiated by a program using such driver should be unpacked correctly.

Xids returned by `tpc_recover()` also have extra attributes *prepared*, *owner*, *database* populated with the values read from the server.

See also:

the `pg_prepared_xacts` system view.

DB API extension

The above methods are the only ones defined by the DB API 2.0 protocol. The Psycopg connection objects exports the following additional methods and attributes.

---

closed
Read-only integer attribute: 0 if the connection is open, nonzero if it is closed or broken.

cancel()
Cancel the current database operation.

The method interrupts the processing of the current operation. If no query is being executed, it does nothing. You can call this function from a different thread than the one currently executing a database operation, for instance if you want to cancel a long running query if a button is pushed in the UI. Interrupting query execution will cause the cancelled method to raise a *QueryCanceledError*. Note that the termination of the query is not guaranteed to succeed: see the documentation for PQcancel().

New in version 2.3.

reset()
Reset the connection to the default.

The method rolls back an eventual pending transaction and executes the PostgreSQL RESET and SET SESSION AUTHORIZATION to revert the session to the default values. A two-phase commit transaction prepared using *tpc_prepare()* will remain in the database available for recover.

New in version 2.0.12.

dsn
Read-only string containing the connection string used by the connection.

If a password was specified in the connection string it will be obscured.

set_session(*isolation_level=None*, *readonly=None*, *deferrable=None*, *autocommit=None*)
Set one or more parameters for the next transactions or statements in the current session.

Parameters

- **isolation_level** – set the isolation level for the next transactions/statements. The value can be one of the literal values READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE or the equivalent *constant* defined in the *extensions* module.
- **readonly** – if True, set the connection to read only; read/write if False.
- **deferrable** – if True, set the connection to deferrable; non deferrable if False. Only available from PostgreSQL 9.1.
- **autocommit** – switch the connection to autocommit mode: not a PostgreSQL session setting but an alias for setting the *autocommit* attribute.

Arguments set to None (the default for all) will not be changed. The parameters *isolation_level*, *readonly* and *deferrable* also accept the string DEFAULT as a value: the effect is to reset the parameter to the server default. Defaults are defined by the server configuration: see values for default_transaction_isolation, default_transaction_read_only, default_transaction_deferrable.

The function must be invoked with no transaction in progress.

See also:

SET TRANSACTION for further details about the behaviour of the transaction parameters in the server.

New in version 2.4.2.

Changed in version 2.7: Before this version, the function would have set default_transaction_* attribute in the current session; this implementation has the problem of not playing well with external connection pooling working at transaction level and not resetting the state of the session: changing the default transaction would pollute the connections in the pool and create problems to other applications using the same pool.

Starting from 2.7, if the connection is not autocommit, the transaction characteristics are issued together with `BEGIN` and will leave the `default_transaction_*` settings untouched. For example:

```
conn.set_session(readonly=True)
```

will not change `default_transaction_read_only`, but following transaction will start with a `BEGIN READ ONLY`. Conversely, using:

```
conn.set_session(readonly=True, autocommit=True)
```

will set `default_transaction_read_only` to `on` and rely on the server to apply the read only state to whatever transaction, implicit or explicit, is executed in the connection.

autocommit

Read/write attribute: if `True`, no transaction is handled by the driver and every statement sent to the backend has immediate effect; if `False` a new transaction is started at the first command execution: the methods *commit()* or *rollback()* must be manually invoked to terminate the transaction.

The autocommit mode is useful to execute commands requiring to be run outside a transaction, such as `CREATE DATABASE` or `VACUUM`.

The default is `False` (manual commit) as per DBAPI specification.

> **Warning:** By default, any query execution, including a simple `SELECT` will start a transaction: for long-running programs, if no further action is taken, the session will remain "idle in transaction", an undesirable condition for several reasons (locks are held by the session, tables bloat...). For long lived scripts, either ensure to terminate a transaction as soon as possible or use an autocommit connection.

New in version 2.4.2.

isolation_level

Return or set the transaction isolation level for the current session. The value is one of the *Isolation level constants* defined in the *psycopg2.extensions* module. On set it is also possible to use one of the literal values `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, `SERIALIZABLE`, `DEFAULT`.

Changed in version 2.7: the property is writable.

Changed in version 2.7: the default value for `isolation_level` is *ISOLATION_LEVEL_DEFAULT*; previously the property would have queried the server and returned the real value applied. To know this value you can run a query such as `show transaction_isolation`. Usually the default value is `READ COMMITTED`, but this may be changed in the server configuration.

This value is now entirely separate from the *autocommit* property: in previous version, if `autocommit` was set to `True` this property would have returned *ISOLATION_LEVEL_AUTOCOMMIT*; it will now return the server isolation level.

readonly

Return or set the read-only status for the current session. Available values are `True` (new transactions will be in read-only mode), `False` (new transactions will be writable), `None` (use the default configured for the server by `default_transaction_read_only`).

New in version 2.7.

deferrable

Return or set the deferrable status for the current session. Available values are `True` (new transactions will be in deferrable mode), `False` (new transactions will be in non deferrable mode), `None` (use the default configured for the server by `default_transaction_deferrable`).

New in version 2.7.

set_isolation_level(*level*)

Note: This is a legacy method mixing `isolation_level` and `autocommit`. Using the respective properties is a better option.

Set the transaction isolation level for the current session. The level defines the different phenomena that can happen in the database between concurrent transactions.

The value set is an integer: symbolic constants are defined in the module *psycopg2.extensions*: see *Isolation level constants* for the available values.

The default level is *ISOLATION_LEVEL_DEFAULT*: at this level a transaction is automatically started the first time a database command is executed. If you want an *autocommit* mode, switch to *ISOLATION_LEVEL_AUTOCOMMIT* before executing any command:

```
>>> conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
```

See also *Transactions control*.

encoding

set_client_encoding(*enc*)
Read or set the client encoding for the current session. The default is the encoding defined by the database. It should be one of the characters set supported by PostgreSQL

notices
A list containing all the database messages sent to the client during the session.

```
>>> cur.execute("CREATE TABLE foo (id serial PRIMARY KEY);")
>>> pprint(conn.notices)
['NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "foo_pkey"␣
↪for table "foo"\n',
 'NOTICE:  CREATE TABLE will create implicit sequence "foo_id_seq" for serial␣
↪column "foo.id"\n']
```

Changed in version 2.7: The `notices` attribute is writable: the user may replace it with any Python object exposing an `append()` method. If appending raises an exception the notice is silently dropped.

To avoid a leak in case excessive notices are generated, only the last 50 messages are kept. This check is only in place if the `notices` attribute is a list: if any other object is used it will be up to the user to guard from leakage.

You can configure what messages to receive using PostgreSQL logging configuration parameters such as `log_statement`, `client_min_messages`, `log_min_duration_statement` etc.

notifies
List of *Notify* objects containing asynchronous notifications received by the session.

For other details see *Asynchronous notifications*.

Changed in version 2.3: Notifications are instances of the `Notify` object. Previously the list was composed by 2 items tuples (*pid*, *channel*) and the payload was not accessible. To keep backward compatibility, `Notify` objects can still be accessed as 2 items tuples.

Changed in version 2.7: The `notifies` attribute is writable: the user may replace it with any Python object exposing an `append()` method. If appending raises an exception the notification is silently dropped.

cursor_factory
The default cursor factory used by *cursor()* if the parameter is not specified.

New in version 2.5.

get_backend_pid()
Returns the process ID (PID) of the backend server process handling this connection.

Note that the PID belongs to a process executing on the database server host, not the local host!

See also:

libpq docs for PQbackendPID() for details.

New in version 2.0.8.

get_parameter_status(*parameter*)
Look up a current parameter setting of the server.

Potential values for `parameter` are: `server_version`, `server_encoding`, `client_encoding`, `is_superuser`, `session_authorization`, `DateStyle`, `TimeZone`, `integer_datetimes`, and `standard_conforming_strings`.

If server did not report requested parameter, return `None`.

See also:

libpq docs for PQparameterStatus() for details.

New in version 2.0.12.

get_dsn_parameters()
Get the effective dsn parameters for the connection as a dictionary.

The *password* parameter is removed from the result.

Example:

```
>>> conn.get_dsn_parameters()
{'dbname': 'test', 'user': 'postgres', 'port': '5432', 'sslmode': 'prefer'}
```

Requires libpq >= 9.3.

See also:

libpq docs for PQconninfo() for details.

New in version 2.7.

get_transaction_status()
Return the current session transaction status as an integer. Symbolic constants for the values are defined in the module *psycopg2.extensions*: see *Transaction status constants* for the available values.

See also:

libpq docs for PQtransactionStatus() for details.

protocol_version
A read-only integer representing frontend/backend protocol being used. Currently Psycopg supports only protocol 3, which allows connection to PostgreSQL server from version 7.4. Psycopg versions previous than 2.3 support both protocols 2 and 3.

See also:

libpq docs for PQprotocolVersion() for details.

New in version 2.0.12.

server_version
A read-only integer representing the backend version.

The number is formed by converting the major, minor, and revision numbers into two-decimal-digit numbers and appending them together. For example, version 8.1.5 will be returned as `80105`.

See also:

libpq docs for PQserverVersion() for details.

New in version 2.0.12.

status
A read-only integer representing the status of the connection. Symbolic constants for the values are defined in the module *psycopg2.extensions*: see *Connection status constants* for the available values.

The status is undefined for *closed* connections.

lobject(*[oid[, mode[, new_oid[, new_file[, lobject_factory]]]]]*)
Return a new database large object as a *lobject* instance.

See *Access to PostgreSQL large objects* for an overview.

Parameters

- **oid** – The OID of the object to read or write. 0 to create a new large object and and have its OID assigned automatically.
- **mode** – Access mode to the object, see below.
- **new_oid** – Create a new object using the specified OID. The function raises *OperationalError* if the OID is already in use. Default is 0, meaning assign a new one automatically.
- **new_file** – The name of a file to be imported in the the database (using the `lo_import()` function)
- **lobject_factory** – Subclass of *lobject* to be instantiated.

Available values for *mode* are:

mode	meaning
`r`	Open for read only
`w`	Open for write only
`rw`	Open for read/write
`n`	Don't open the file
`b`	Don't decode read data (return data as `str` in Python 2 or `bytes` in Python 3)
`t`	Decode read data according to *connection.encoding* (return data as `unicode` in Python 2 or `str` in Python 3)

`b` and `t` can be specified together with a read/write mode. If neither `b` nor `t` is specified, the default is `b` in Python 2 and `t` in Python 3.

New in version 2.0.8.

Changed in version 2.4: added `b` and `t` mode and unicode support.

Methods related to asynchronous support.

New in version 2.2.0.

See also:

Asynchronous support and *Support for coroutine libraries*.

async
async_
Read only attribute: 1 if the connection is asynchronous, 0 otherwise.

Changed in version 2.7: added the `async_` alias for Python versions where `async` is a keyword.

poll()
Used during an asynchronous connection attempt, or when a cursor is executing a query on an asynchronous connection, make communication proceed if it wouldn't block.

Return one of the constants defined in *Poll constants*. If it returns *POLL_OK* then the connection has been established or the query results are available on the client. Otherwise wait until the file descriptor returned by *fileno()* is ready to read or to write, as explained in *Asynchronous support*. *poll()* should be also used by the function installed by *set_wait_callback()* as explained in *Support for coroutine libraries*.

poll() is also used to receive asynchronous notifications from the database: see *Asynchronous notifications* from further details.

fileno()
Return the file descriptor underlying the connection: useful to read its status during asynchronous communication.

isexecuting()
Return `True` if the connection is executing an asynchronous operation.

CHAPTER

FIVE

THE CURSOR CLASS

class cursor

Allows Python code to execute PostgreSQL command in a database session. Cursors are created by the *connection.cursor()* method: they are bound to the connection for the entire lifetime and all the commands are executed in the context of the database session wrapped by the connection.

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 connections' *isolation level*. See also *rollback()* and *commit()* methods.

Cursors are *not* thread safe: a multithread application can create many cursors from the same connection and should use each cursor from a single thread. See *Thread and process safety* for details.

description

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

Each of these sequences is a named tuple (a regular tuple if `collections.namedtuple()` is not available) containing information describing one result column:

0. `name`: the name of the column returned.
1. `type_code`: the PostgreSQL OID of the column. You can use the `pg_type` system table to get more informations about the type. This is the value used by Psycopg to decide what Python type use to represent the value. See also *Type casting of SQL types into Python objects*.
2. `display_size`: the actual length of the column in bytes. Obtaining this value is computationally intensive, so it is always `None` unless the `PSYCOPG_DISPLAY_SIZE` parameter is set at compile time. See also PQgetlength.
3. `internal_size`: the size in bytes of the column associated to this column on the server. Set to a negative value for variable-size types See also PQfsize.
4. `precision`: total number of significant digits in columns of type `NUMERIC`. `None` for other types.
5. `scale`: count of decimal digits in the fractional part in columns of type `NUMERIC`. `None` for other types.
6. `null_ok`: always `None` as not easy to retrieve from the libpq.

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

Changed in version 2.4: if possible, columns descriptions are named tuple instead of regular tuples.

close()

Close the cursor now (rather than whenever `del` is executed). The cursor will be unusable from this point forward; an *InterfaceError* will be raised if any operation is attempted with the cursor.

Changed in version 2.5: if the cursor is used in a `with` statement, the method is automatically called at the end of the `with` block.

closed
Read-only boolean attribute: specifies if the cursor is closed (`True`) or not (`False`).

DB API extension

The *closed* attribute is a Psycopg extension to the DB API 2.0.

New in version 2.0.7.

connection
Read-only attribute returning a reference to the *connection* object on which the cursor was created.

name
Read-only attribute containing the name of the cursor if it was creates as named cursor by *connection.cursor()*, or `None` if it is a client side cursor. See *Server side cursors*.

DB API extension

The *name* attribute is a Psycopg extension to the DB API 2.0.

---

scrollable
Read/write attribute: specifies if a named cursor is declared `SCROLL`, hence is capable to scroll backwards (using *scroll()*). If `True`, the cursor can be scrolled backwards, if `False` it is never scrollable. If `None` (default) the cursor scroll option is not specified, usually but not always meaning no backward scroll (see the `DECLARE` notes).

Note: set the value before calling *execute()* or use the *connection.cursor()* *scrollable* parameter, otherwise the value will have no effect.

New in version 2.5.

DB API extension

The *scrollable* attribute is a Psycopg extension to the DB API 2.0.

---

withhold
Read/write attribute: specifies if a named cursor lifetime should extend outside of the current transaction, i.e., it is possible to fetch from the cursor even after a *connection.commit()* (but not after a *connection.rollback()*). See *Server side cursors*

Note: set the value before calling *execute()* or use the *connection.cursor()* *withhold* parameter, otherwise the value will have no effect.

New in version 2.4.3.

DB API extension

The *withhold* attribute is a Psycopg extension to the DB API 2.0.

Commands execution methods

execute(*query*, *vars=None*)
Execute a database operation (query or command).

Parameters may be provided as sequence or mapping and will be bound to variables in the operation. Variables are specified either with positional (`%s`) or named (`%(name)s`) placeholders. See *Passing parameters to SQL queries*.

The method returns `None`. If a query was executed, the returned values can be retrieved using *fetch*()* methods.

executemany(*query*, *vars_list*)
Execute a database operation (query or command) against all parameter tuples or mappings found in the sequence *vars_list*.

The function is mostly useful for commands that update the database: any result set returned by the query is discarded.

Parameters are bounded to the query using the same rules described in the *execute()* method.

> **Warning:** In its current implementation this method is not faster than executing *execute()* in a loop. For better performance you can use the functions described in *Fast execution helpers*.

callproc(*procname*[, *parameters*])
Call a stored database procedure with the given name. The sequence of parameters must contain one entry for each argument that the procedure expects. Overloaded procedures are supported. Named parameters can be used by supplying the parameters as a dictionary.

This function is, at present, not DBAPI-compliant. The return value is supposed to consist of the sequence of parameters with modified output and input/output parameters. In future versions, the DBAPI-compliant return value may be implemented, but for now the function returns None.

The procedure may provide a result set as output. This is then made available through the standard *fetch*()* methods.

Changed in version 2.7: added support for named arguments.

mogrify(*operation*[, *parameters*])
Return a query string after arguments binding. The string returned is exactly the one that would be sent to the database running the *execute()* method or similar.

The returned string is always a bytes string.

```
>>> cur.mogrify("INSERT INTO test (num, data) VALUES (%s, %s)", (42, 'bar'))
"INSERT INTO test (num, data) VALUES (42, E'bar')"
```

DB API extension

The *mogrify()* method is a Psycopg extension to the DB API 2.0.

setinputsizes(*sizes*)
This method is exposed in compliance with the DB API 2.0. It currently does nothing but it is safe to call it.

Results retrieval methods

The following methods are used to read data from the database after an *execute()* call.

Note: *cursor* objects are iterable, so, instead of calling explicitly *fetchone()* in a loop, the object itself can be used:

```
>>> cur.execute("SELECT * FROM test;")
>>> for record in cur:
...     print record
...
(1, 100, "abc'def")
(2, None, 'dada')
(3, 42, 'bar')
```

Changed in version 2.4: iterating over a *named cursor* fetches *itersize* records at time from the backend. Previously only one record was fetched per roundtrip, resulting in a large overhead.

---

fetchone()
Fetch the next row of a query result set, returning a single tuple, or None when no more data is available:

```
>>> cur.execute("SELECT * FROM test WHERE id = %s", (3,))
>>> cur.fetchone()
(3, 42, 'bar')
```

A *ProgrammingError* is raised if the previous call to *execute*()* did not produce any result set or no call was issued yet.

fetchmany([*size=cursor.arraysize*])
Fetch the next set of rows of a query result, returning a list of tuples. An empty list is returned when no more rows are available.

The number of rows to fetch per call is specified by the parameter. If it is not given, the cursor's *arraysize* determines the number of rows to be fetched. The method should try 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:

```
>>> cur.execute("SELECT * FROM test;")
>>> cur.fetchmany(2)
[(1, 100, "abc'def"), (2, None, 'dada')]
>>> cur.fetchmany(2)
[(3, 42, 'bar')]
>>> cur.fetchmany(2)
[]
```

A *ProgrammingError* is raised if the previous call to *execute*()* 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 *fetchmany()* call to the next.

fetchall()
Fetch all (remaining) rows of a query result, returning them as a list of tuples. An empty list is returned if there is no more record to fetch.

```
>>> cur.execute("SELECT * FROM test;")
>>> cur.fetchall()
[(1, 100, "abc'def"), (2, None, 'dada'), (3, 42, 'bar')]
```

A *ProgrammingError* is raised if the previous call to *execute*()* did not produce any result set or no call was issued yet.

scroll(*value*[, *mode='relative'*])
Scroll the cursor in the result set to a new position according to mode.

If `mode` is `relative` (default), value is taken as offset to the current position in the result set, if set to `absolute`, value states an absolute target position.

If the scroll operation would leave the result set, a *ProgrammingError* is raised and the cursor position is not changed.

Note: According to the DB API 2.0, the exception raised for a cursor out of bound should have been `IndexError`. The best option is probably to catch both exceptions in your code:

```
try:
    cur.scroll(1000 * 1000)
except (ProgrammingError, IndexError), exc:
    deal_with_it(exc)
```

The method can be used both for client-side cursors and *server-side cursors*. Server-side cursors can usually scroll backwards only if declared *scrollable*. Moving out-of-bound in a server-side cursor doesn't result in an exception, if the backend doesn't raise any (Postgres doesn't tell us in a reliable way if we went out of bound).

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

itersize
Read/write attribute specifying the number of rows to fetch from the backend at each network roundtrip during *iteration* on a *named cursor*. The default is 2000.

New in version 2.4.

DB API extension

The *itersize* attribute is a Psycopg extension to the DB API 2.0.

---

rowcount
This read-only attribute specifies the number of rows that the last *execute*()* produced (for DQL (Data Query Language) statements like `SELECT`) or affected (for DML (Data Manipulation Language) statements like `UPDATE` or `INSERT`).

The attribute is -1 in case no `execute*()` has been performed on the cursor or the row count of the last operation if it can't be determined by the interface.

Note: The DB API 2.0 interface reserves to redefine the latter case to have the object return `None` instead of -1 in future versions of the specification.

---

rownumber
This read-only attribute provides the current 0-based index of the cursor in the result set or `None` if the index cannot be determined.

The index can be seen as index of the cursor in a sequence (the result set). The next fetch operation will fetch the row indexed by *rownumber* in that sequence.

lastrowid
This read-only attribute provides the OID of the last row inserted by the cursor. If the table wasn't created with OID support or the last operation is not a single record insert, the attribute is set to `None`.

Note: PostgreSQL currently advices to not create OIDs on the tables and the default for `CREATE TABLE` is to not support them. The `INSERT ... RETURNING` syntax available from PostgreSQL 8.3 allows more flexibility.

---

query
Read-only attribute containing the body of the last query sent to the backend (including bound arguments) as bytes string. `None` if no query has been executed yet:

```
>>> cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)", (42, 'bar'))
>>> cur.query
"INSERT INTO test (num, data) VALUES (42, E'bar')"
```

DB API extension

The *query* attribute is a Psycopg extension to the DB API 2.0.

---

statusmessage
Read-only attribute containing the message returned by the last command:

```
>>> cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)", (42, 'bar'))
>>> cur.statusmessage
'INSERT 0 1'
```

DB API extension

The *statusmessage* attribute is a Psycopg extension to the DB API 2.0.

---

cast(*oid*, *s*)
Convert a value from the PostgreSQL string representation to a Python object.

Use the most specific of the typecasters registered by *register_type()*.

New in version 2.4.

DB API extension

The *cast()* method is a Psycopg extension to the DB API 2.0.

---

tzinfo_factory
The time zone factory used to handle data types such as `TIMESTAMP WITH TIME ZONE`. It should be a `tzinfo` object. A few implementations are available in the *psycopg2.tz* module.

nextset()
This method is not supported (PostgreSQL does not have multiple data sets) and will raise a *NotSupportedError* exception.

setoutputsize(*size*[, *column*])
This method is exposed in compliance with the DB API 2.0. It currently does nothing but it is safe to call it.

COPY-related methods

Efficiently copy data from file-like objects to the database and back. See *Using COPY TO and COPY FROM* for an overview.

DB API extension

The `COPY` command is a PostgreSQL extension to the SQL standard. As such, its support is a Psycopg extension to the DB API 2.0.

---

copy_from(*file*, *table*, *sep='\t'*, *null='\\N'*, *size=8192*, *columns=None*)
Read data *from* the file-like object *file* appending them to the table named *table*.

Parameters

- **file** – file-like object to read data from. It must have both `read()` and `readline()` methods.
- **table** – name of the table to copy data into.
- **sep** – columns separator expected in the file. Defaults to a tab.
- **null** – textual representation of `NULL` in the file. The default is the two characters string `\N`.
- **size** – size of the buffer used to read from the file.
- **columns** – iterable with name of the columns to import. The length and types should match the content of the file to read. If not specified, it is assumed that the entire table matches the file structure.

Example:

```
>>> f = StringIO("42\tfoo\n74\tbar\n")
>>> cur.copy_from(f, 'test', columns=('num', 'data'))
>>> cur.execute("select * from test where id > 5;")
>>> cur.fetchall()
[(6, 42, 'foo'), (7, 74, 'bar')]
```

Note: the name of the table is not quoted: if the table name contains uppercase letters or special characters it must be quoted with double quotes:

```
cur.copy_from(f, '"TABLE"')
```

Changed in version 2.0.6: added the *columns* parameter.

Changed in version 2.4: data read from files implementing the `io.TextIOBase` interface are encoded in the connection *encoding* when sent to the backend.

copy_to(*file*, *table*, *sep='\t'*, *null='\\N'*, *columns=None*)
Write the content of the table named *table to* the file-like object *file*. See *Using COPY TO and COPY FROM* for an overview.

Parameters

- **file** – file-like object to write data into. It must have a `write()` method.
- **table** – name of the table to copy data from.
- **sep** – columns separator expected in the file. Defaults to a tab.
- **null** – textual representation of `NULL` in the file. The default is the two characters string `\N`.
- **columns** – iterable with name of the columns to export. If not specified, export all the columns.

Example:

```
>>> cur.copy_to(sys.stdout, 'test', sep="|")
1|100|abc'def
2|\N|dada
...
```

Note: the name of the table is not quoted: if the table name contains uppercase letters or special characters it must be quoted with double quotes:

```
cur.copy_to(f, '"TABLE"')
```

Changed in version 2.0.6: added the *columns* parameter.

Changed in version 2.4: data sent to files implementing the `io.TextIOBase` interface are decoded in the connection *encoding* when read from the backend.

copy_expert(*sql*, *file*, *size=8192*)
Submit a user-composed `COPY` statement. The method is useful to handle all the parameters that PostgreSQL makes available (see `COPY` command documentation).

Parameters

- **sql** – the `COPY` statement to execute.
- **file** – a file-like object to read or write (according to *sql*).
- **size** – size of the read buffer to be used in `COPY FROM`.

The *sql* statement should be in the form `COPY table TO STDOUT` to export *table* to the *file* object passed as argument or `COPY table FROM STDIN` to import the content of the *file* object into *table*. If you need to compose a `COPY` statement dynamically (because table, fields, or query parameters are in Python variables) you may use the objects provided by the *psycopg2.sql* module.

file must be a readable file-like object (as required by *copy_from()*) for *sql* statement `COPY ... FROM STDIN` or a writable one (as required by *copy_to()*) for `COPY ... TO STDOUT`.

Example:

```
>>> cur.copy_expert("COPY test TO STDOUT WITH CSV HEADER", sys.stdout)
id,num,data
1,100,abc'def
```

(continues on next page)

(continued from previous page)

```
2,,dada
...
```

New in version 2.0.6.

Changed in version 2.4: files implementing the `io.TextIOBase` interface are dealt with using Unicode data instead of bytes.

CHAPTER

SIX

MORE ADVANCED TOPICS

6.1 Connection and cursor factories

Psycopg exposes two new-style classes that can be sub-classed and expanded to adapt them to the needs of the programmer: *psycopg2.extensions.cursor* and *psycopg2.extensions.connection*. The *connection* class is usually sub-classed only to provide an easy way to create customized cursors but other uses are possible. *cursor* is much more interesting, because it is the class where query building, execution and result type-casting into Python variables happens.

The *extras* module contains several examples of *connection and cursor subclasses*.

Note: If you only need a customized cursor class, since Psycopg 2.5 you can use the *cursor_factory* parameter of a regular connection instead of creating a new `connection` subclass.

An example of cursor subclass performing logging is:

```
import psycopg2
import psycopg2.extensions
import logging

class LoggingCursor(psycopg2.extensions.cursor):
    def execute(self, sql, args=None):
        logger = logging.getLogger('sql_debug')
        logger.info(self.mogrify(sql, args))

        try:
            psycopg2.extensions.cursor.execute(self, sql, args)
        except Exception, exc:
            logger.error("%s: %s" % (exc.__class__.__name__, exc))
            raise

conn = psycopg2.connect(DSN)
cur = conn.cursor(cursor_factory=LoggingCursor)
cur.execute("INSERT INTO mytable VALUES (%s, %s, %s);",
             (10, 20, 30))
```

6.2 Adapting new Python types to SQL syntax

Any Python class or type can be adapted to an SQL string. Adaptation mechanism is similar to the Object Adaptation proposed in the **PEP 246** and is exposed by the *psycopg2.extensions.adapt()* function.

The *execute()* method adapts its arguments to the *ISQLQuote* protocol. Objects that conform to this protocol expose a `getquoted()` method returning the SQL representation of the object as a string (the method must return `bytes` in Python 3). Optionally the conform object may expose a *prepare()* method.

There are two basic ways to have a Python object adapted to SQL:

- the object itself is conform, or knows how to make itself conform. Such object must expose a `__conform__()` method that will be called with the protocol object as argument. The object can check that the protocol is `ISQLQuote`, in which case it can return `self` (if the object also implements `getquoted()`) or a suitable wrapper object. This option is viable if you are the author of the object and if the object is specifically designed for the database (i.e. having Psycopg as a dependency and polluting its interface with the required methods doesn't bother you). For a simple example you can take a look at the source code for the *psycopg2.extras.Inet* object.
- If implementing the `ISQLQuote` interface directly in the object is not an option (maybe because the object to adapt comes from a third party library), you can use an *adaptation function*, taking the object to be adapted as argument and returning a conforming object. The adapter must be registered via the *register_adapter()* function. A simple example wrapper is `psycopg2.extras.UUID_adapter` used by the *register_uuid()* function.

A convenient object to write adapters is the *AsIs* wrapper, whose `getquoted()` result is simply the `str()`ing conversion of the wrapped object.

Example: mapping of a `Point` class into the `point` PostgreSQL geometric type:

```
>>> from psycopg2.extensions import adapt, register_adapter, AsIs

>>> class Point(object):
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y

>>> def adapt_point(point):
...     x = adapt(point.x).getquoted()
...     y = adapt(point.y).getquoted()
...     return AsIs("'(%s, %s)'" % (x, y))

>>> register_adapter(Point, adapt_point)

>>> cur.execute("INSERT INTO atable (apoint) VALUES (%s)",
...             (Point(1.23, 4.56),))
```

The above function call results in the SQL command:

```
INSERT INTO atable (apoint) VALUES ('(1.23, 4.56)');
```

6.3 Type casting of SQL types into Python objects

PostgreSQL objects read from the database can be adapted to Python objects through an user-defined adapting function. An adapter function takes two arguments: the object string representation as returned by PostgreSQL and the cursor currently being read, and should return a new Python object. For example, the following function parses the PostgreSQL `point` representation into the previously defined `Point` class:

```
>>> def cast_point(value, cur):
...     if value is None:
...         return None
```

(continues on next page)

(continued from previous page)

```
...
...     # Convert from (f1, f2) syntax using a regular expression.
...     m = re.match(r"\(([^)]+),([^)]+)\)", value)
...     if m:
...         return Point(float(m.group(1)), float(m.group(2)))
...     else:
...         raise InterfaceError("bad point representation: %r" % value)
```

In order to create a mapping from a PostgreSQL type (either standard or user-defined), its OID must be known. It can be retrieved either by the second column of the *cursor.description*:

```
>>> cur.execute("SELECT NULL::point")
>>> point_oid = cur.description[0][1]
>>> point_oid
600
```

or by querying the system catalog for the type name and namespace (the namespace for system objects is `pg_catalog`):

```
>>> cur.execute("""
...     SELECT pg_type.oid
...       FROM pg_type JOIN pg_namespace
...               ON typnamespace = pg_namespace.oid
...      WHERE typname = %(typename)s
...        AND nspname = %(namespace)s""",
...     {'typename': 'point', 'namespace': 'pg_catalog'})
>>> point_oid = cur.fetchone()[0]
>>> point_oid
600
```

After you know the object OID, you can create and register the new type:

```
>>> POINT = psycopg2.extensions.new_type((point_oid,), "POINT", cast_point)
>>> psycopg2.extensions.register_type(POINT)
```

The *new_type()* function binds the object OIDs (more than one can be specified) to the adapter function. *register_type()* completes the spell. Conversion is automatically performed when a column whose type is a registered OID is read:

```
>>> cur.execute("SELECT '(10.2,20.3)'::point")
>>> point = cur.fetchone()[0]
>>> print type(point), point.x, point.y
<class 'Point'> 10.2 20.3
```

A typecaster created by `new_type()` can be also used with *new_array_type()* to create a typecaster converting a PostgreSQL array into a Python list.

6.4 Asynchronous notifications

Psycopg allows asynchronous interaction with other database sessions using the facilities offered by PostgreSQL commands `LISTEN` and `NOTIFY`. Please refer to the PostgreSQL documentation for examples about how to use this form of communication.

Notifications are instances of the *Notify* object made available upon reception in the *connection.notifies* list. Notifications can be sent from Python code simply executing a `NOTIFY` command in an *execute()* call.

Because of the way sessions interact with notifications (see `NOTIFY` documentation), you should keep the connection in *`autocommit`* mode if you wish to receive or send notifications in a timely manner.

Notifications are received after every query execution. If the user is interested in receiving notifications but not in performing any query, the *`poll()`* method can be used to check for new messages without wasting resources.

A simple application could poll the connection from time to time to check if something new has arrived. A better strategy is to use some I/O completion function such as `select()` to sleep until awakened by the kernel when there is some data to read on the connection, thereby using no CPU unless there is something to read:

```
import select
import psycopg2
import psycopg2.extensions

conn = psycopg2.connect(DSN)
conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)

curs = conn.cursor()
curs.execute("LISTEN test;")

print "Waiting for notifications on channel 'test'"
while 1:
    if select.select([conn],[],[],5) == ([],[],[]):
        print "Timeout"
    else:
        conn.poll()
        while conn.notifies:
            notify = conn.notifies.pop(0)
            print "Got NOTIFY:", notify.pid, notify.channel, notify.payload
```

Running the script and executing a command such as `NOTIFY test, 'hello'` in a separate **`psql`** shell, the output may look similar to:

```
Waiting for notifications on channel 'test'
Timeout
Timeout
Got NOTIFY: 6535 test hello
Timeout
...
```

Note that the payload is only available from PostgreSQL 9.0: notifications received from a previous version server will have the *`payload`* attribute set to the empty string.

Changed in version 2.3: Added *`Notify`* object and handling notification payload.

Changed in version 2.7: The *`notifies`* attribute is writable: it is possible to replace it with any object exposing an `append()` method. An useful example would be to use a `deque` object.

6.5 Asynchronous support

New in version 2.2.0.

Psycopg can issue asynchronous queries to a PostgreSQL database. An asynchronous communication style is established passing the parameter *async*=1 to the *`connect()`* function: the returned connection will work in *asynchronous mode*.

In asynchronous mode, a Psycopg connection will rely on the caller to poll the socket file descriptor, checking if it is ready to accept data or if a query result has been transferred and is ready to be read on the client. The caller can use

the method *fileno()* to get the connection file descriptor and *poll()* to make communication proceed according to the current connection state.

The following is an example loop using methods `fileno()` and `poll()` together with the Python `select()` function in order to carry on asynchronous operations with Psycopg:

```
def wait(conn):
    while 1:
        state = conn.poll()
        if state == psycopg2.extensions.POLL_OK:
            break
        elif state == psycopg2.extensions.POLL_WRITE:
            select.select([], [conn.fileno()], [])
        elif state == psycopg2.extensions.POLL_READ:
            select.select([conn.fileno()], [], [])
        else:
            raise psycopg2.OperationalError("poll() returned %s" % state)
```

The above loop of course would block an entire application: in a real asynchronous framework, `select()` would be called on many file descriptors waiting for any of them to be ready. Nonetheless the function can be used to connect to a PostgreSQL server only using nonblocking commands and the connection obtained can be used to perform further nonblocking queries. After `poll()` has returned *POLL_OK*, and thus `wait()` has returned, the connection can be safely used:

```
>>> aconn = psycopg2.connect(database='test', async=1)
>>> wait(aconn)
>>> acurs = aconn.cursor()
```

Note that there are a few other requirements to be met in order to have a completely non-blocking connection attempt: see the libpq documentation for `PQconnectStart()`.

The same loop should be also used to perform nonblocking queries: after sending a query via *execute()* or *callproc()*, call `poll()` on the connection available from *cursor.connection* until it returns `POLL_OK`, at which point the query has been completely sent to the server and, if it produced data, the results have been transferred to the client and available using the regular cursor methods:

```
>>> acurs.execute("SELECT pg_sleep(5); SELECT 42;")
>>> wait(acurs.connection)
>>> acurs.fetchone()[0]
42
```

When an asynchronous query is being executed, *connection.isexecuting()* returns `True`. Two cursors can't execute concurrent queries on the same asynchronous connection.

There are several limitations in using asynchronous connections: the connection is always in *autocommit* mode and it is not possible to change it. So a transaction is not implicitly started at the first query and is not possible to use methods *commit()* and *rollback()*: you can manually control transactions using *execute()* to send database commands such as `BEGIN`, `COMMIT` and `ROLLBACK`. Similarly *set_session()* can't be used but it is still possible to invoke the `SET` command with the proper `default_transaction_...` parameter.

With asynchronous connections it is also not possible to use *set_client_encoding()*, *executemany()*, *large objects*, *named cursors*.

COPY commands are not supported either in asynchronous mode, but this will be probably implemented in a future release.

6.6 Support for coroutine libraries

New in version 2.2.

Psycopg can be used together with coroutine-based libraries and participate in cooperative multithreading.

Coroutine-based libraries (such as Eventlet or gevent) can usually patch the Python standard library in order to enable a coroutine switch in the presence of blocking I/O: the process is usually referred as making the system *green*, in reference to the green threads.

Because Psycopg is a C extension module, it is not possible for coroutine libraries to patch it: Psycopg instead enables cooperative multithreading by allowing the registration of a *wait callback* using the *psycopg2.extensions.set_wait_callback()* function. When a wait callback is registered, Psycopg will use libpq non-blocking calls instead of the regular blocking ones, and will delegate to the callback the responsibility to wait for the socket to become readable or writable.

Working this way, the caller does not have the complete freedom to schedule the socket check whenever they want as with an *asynchronous connection*, but has the advantage of maintaining a complete DB API 2.0 semantics: from the point of view of the end user, all Psycopg functions and objects will work transparently in the coroutine environment (blocking the calling green thread and giving other green threads the possibility to be scheduled), allowing non modified code and third party libraries (such as SQLAlchemy) to be used in coroutine-based programs.

Warning: Psycopg connections are not *green thread safe* and can't be used concurrently by different green threads. Trying to execute more than one command at time using one cursor per thread will result in an error (or a deadlock on versions before 2.4.2).

Therefore, programmers are advised to either avoid sharing connections between coroutines or to use a library-friendly lock to synchronize shared connections, e.g. for pooling.

Coroutine libraries authors should provide a callback implementation (and possibly a method to register it) to make Psycopg as green as they want. An example callback (using `select()` to block) is provided as *psycopg2.extras.wait_select()*: it boils down to something similar to:

```
def wait_select(conn):
    while 1:
        state = conn.poll()
        if state == extensions.POLL_OK:
            break
        elif state == extensions.POLL_READ:
            select.select([conn.fileno()], [], [])
        elif state == extensions.POLL_WRITE:
            select.select([], [conn.fileno()], [])
        else:
            raise OperationalError("bad state from poll: %s" % state)
```

Providing callback functions for the single coroutine libraries is out of psycopg2 scope, as the callback can be tied to the libraries' implementation details. You can check the psycogreen project for further informations and resources about the topic.

Warning: *COPY commands* are currently not supported when a wait callback is registered, but they will be probably implemented in a future release.

Large objects are not supported either: they are not compatible with asynchronous connections.

6.7 Replication protocol support

New in version 2.7.

Modern PostgreSQL servers (version 9.0 and above) support replication. The replication protocol is built on top of the client-server protocol and can be operated using `libpq`, as such it can be also operated by `psycopg2`. The replication protocol can be operated on both synchronous and *asynchronous* connections.

Server version 9.4 adds a new feature called *Logical Replication*.

See also:

- PostgreSQL Streaming Replication Protocol

6.7.1 Logical replication Quick-Start

You must be using PostgreSQL server version 9.4 or above to run this quick start.

Make sure that replication connections are permitted for user `postgres` in `pg_hba.conf` and reload the server configuration. You also need to set `wal_level=logical` and `max_wal_senders`, `max_replication_slots` to value greater than zero in `postgresql.conf` (these changes require a server restart). Create a database `psycopg2_test`.

Then run the following code to quickly try the replication support out. This is not production code – it has no error handling, it sends feedback too often, etc. – and it's only intended as a simple demo of logical replication:

```
from __future__ import print_function
import sys
import psycopg2
import psycopg2.extras

conn = psycopg2.connect('dbname=psycopg2_test user=postgres',
   connection_factory=psycopg2.extras.LogicalReplicationConnection)
cur = conn.cursor()
try:
    # test_decoding produces textual output
    cur.start_replication(slot_name='pytest', decode=True)
except psycopg2.ProgrammingError:
    cur.create_replication_slot('pytest', output_plugin='test_decoding')
    cur.start_replication(slot_name='pytest', decode=True)

class DemoConsumer(object):
    def __call__(self, msg):
        print(msg.payload)
        msg.cursor.send_feedback(flush_lsn=msg.data_start)

democonsumer = DemoConsumer()

print("Starting streaming, press Control-C to end...", file=sys.stderr)
try:
   cur.consume_stream(democonsumer)
except KeyboardInterrupt:
   cur.close()
   conn.close()
   print("The slot 'pytest' still exists. Drop it with "
      "SELECT pg_drop_replication_slot('pytest'); if no longer needed.",
      file=sys.stderr)
```

(continues on next page)

(continued from previous page)

```
    print("WARNING: Transaction logs will accumulate in pg_xlog "
       "until the slot is dropped.", file=sys.stderr)
```

You can now make changes to the `psycopg2_test` database using a normal psycopg2 session, `psql`, etc. and see the logical decoding stream printed by this demo client.

This will continue running until terminated with `Control-C`.

For the details see *Replication connection and cursor classes*.

CHAPTER

SEVEN

PSYCOPG2.EXTENSIONS – EXTENSIONS TO THE DB API

The module contains a few objects and function extending the minimum set of functionalities defined by the DB API 2.0.

7.1 Classes definitions

Instances of these classes are usually returned by factory functions or attributes. Their definitions are exposed here to allow subclassing, introspection etc.

class psycopg2.extensions.**connection**(*dsn*, *async=False*)

Is the class usually returned by the *connect()* function. It is exposed by the `extensions` module in order to allow subclassing to extend its behaviour: the subclass should be passed to the `connect()` function using the `connection_factory` parameter. See also *Connection and cursor factories*.

For a complete description of the class, see *connection*.

Changed in version 2.7: *async_* can be used as alias for *async*.

class psycopg2.extensions.**cursor**(*conn*, *name=None*)

It is the class usually returned by the *connection.cursor()* method. It is exposed by the `extensions` module in order to allow subclassing to extend its behaviour: the subclass should be passed to the `cursor()` method using the `cursor_factory` parameter. See also *Connection and cursor factories*.

For a complete description of the class, see *cursor*.

class psycopg2.extensions.**lobject**(*conn*[, *oid*[, *mode*[, *new_oid*[, *new_file*]]]])

Wrapper for a PostgreSQL large object. See *Access to PostgreSQL large objects* for an overview.

The class can be subclassed: see the *connection.lobject()* to know how to specify a `lobject` subclass.

New in version 2.0.8.

oid

Database OID of the object.

mode

The mode the database was open. See *connection.lobject()* for a description of the available modes.

read(*bytes=-1*)

Read a chunk of data from the current file position. If -1 (default) read all the remaining data.

The result is an Unicode string (decoded according to *connection.encoding*) if the file was open in `t` mode, a bytes string for `b` mode.

Changed in version 2.4: added Unicode support.

write(*str*)
Write a string to the large object. Return the number of bytes written. Unicode strings are encoded in the *connection.encoding* before writing.

Changed in version 2.4: added Unicode support.

export(*file_name*)
Export the large object content to the file system.

The method uses the efficient `lo_export()` libpq function.

seek(*offset*, *whence=0*)
Set the lobject current position.

Changed in version 2.6.0: added support for *offset* > 2GB.

tell()
Return the lobject current position.

New in version 2.2.0.

Changed in version 2.6.0: added support for return value > 2GB.

truncate(*len=0*)
Truncate the lobject to the given size.

The method will only be available if Psycopg has been built against libpq from PostgreSQL 8.3 or later and can only be used with PostgreSQL servers running these versions. It uses the `lo_truncate()` libpq function.

New in version 2.2.0.

Changed in version 2.6.0: added support for *len* > 2GB.

Warning: If Psycopg is built with `lo_truncate()` support or with the 64 bits API support (resp. from PostgreSQL versions 8.3 and 9.3) but at runtime an older version of the dynamic library is found, the `psycopg2` module will fail to import. See *the lo_truncate FAQ* about the problem.

close()
Close the object.

closed
Boolean attribute specifying if the object is closed.

unlink()
Close the object and remove it from the database.

class psycopg2.extensions.**Notify**(*pid*, *channel*, *payload=''*)
A notification received from the backend.

`Notify` instances are made available upon reception on the *notifies* member of the listening connection. The object can be also accessed as a 2 items tuple returning the members `(pid, channel)` for backward compatibility.

See *Asynchronous notifications* for details.

New in version 2.3.

channel
The name of the channel to which the notification was sent.

payload
The payload message of the notification.

Attaching a payload to a notification is only available since PostgreSQL 9.0: for notifications received from previous versions of the server this member is always the empty string.

pid
The ID of the backend process that sent the notification.

Note: if the sending session was handled by Psycopg, you can use *get_backend_pid()* to know its PID.

class psycopg2.extensions.**Xid**(*format_id*, *gtrid*, *bqual*)
A transaction identifier used for two-phase commit.

Usually returned by the connection methods *xid()* and *tpc_recover()*. Xid instances can be unpacked as a 3-item tuples containing the items (*format_id*, *gtrid*, *bqual*). The str() of the object returns the *transaction ID* used in the commands sent to the server.

See *Two-Phase Commit protocol support* for an introduction.

New in version 2.3.

static from_string(*s*)
Create a Xid object from a string representation. Static method.

If *s* is a PostgreSQL transaction ID produced by a XA transaction, the returned object will have *format_id*, *gtrid*, *bqual* set to the values of the preparing XA id. Otherwise only the gtrid is populated with the unparsed string. The operation is the inverse of the one performed by str(xid).

bqual
Branch qualifier of the transaction.

In a XA transaction every resource participating to a transaction receives a distinct branch qualifier. None if the transaction doesn't follow the XA standard.

database
Database the recovered transaction belongs to.

format_id
Format ID in a XA transaction.

A non-negative 32 bit integer. None if the transaction doesn't follow the XA standard.

gtrid
Global transaction ID in a XA transaction.

If the transaction doesn't follow the XA standard, it is the plain *transaction ID* used in the server commands.

owner
Name of the user who prepared a recovered transaction.

prepared
Timestamp (with timezone) in which a recovered transaction was prepared.

class psycopg2.extensions.**Diagnostics**(*exception*)
Details from a database error report.

The object is returned by the *diag* attribute of the Error object. All the information available from the PQresultErrorField() function are exposed as attributes by the object, e.g. the severity attribute returns the PG_DIAG_SEVERITY code. Please refer to the PostgreSQL documentation for the meaning of all the attributes.

New in version 2.5.

The attributes currently available are:

column_name
constraint_name
context
datatype_name
internal_position
internal_query
message_detail
message_hint
message_primary
schema_name
severity
source_file
source_function
source_line
sqlstate
statement_position
table_name
A string with the error field if available; `None` if not available. The attribute value is available only if the error sent by the server: not all the fields are available for all the errors and for all the server versions.

7.2 SQL adaptation protocol objects

Psycopg provides a flexible system to adapt Python objects to the SQL syntax (inspired to the **PEP 246**), allowing serialization in PostgreSQL. See *Adapting new Python types to SQL syntax* for a detailed description. The following objects deal with Python objects adaptation:

psycopg2.extensions.**adapt**(*obj*)
Return the SQL representation of *obj* as an *ISQLQuote*. Raise a *ProgrammingError* if how to adapt the object is unknown. In order to allow new objects to be adapted, register a new adapter for it using the *register_adapter()* function.

The function is the entry point of the adaptation mechanism: it can be used to write adapters for complex objects by recursively calling `adapt()` on its components.

psycopg2.extensions.**register_adapter**(*class*, *adapter*)
Register a new adapter for the objects of class *class*.

adapter should be a function taking a single argument (the object to adapt) and returning an object conforming to the *ISQLQuote* protocol (e.g. exposing a `getquoted()` method). The *AsIs* is often useful for this task.

Once an object is registered, it can be safely used in SQL queries and by the *adapt()* function.

class psycopg2.extensions.**ISQLQuote**(*wrapped_object*)
Represents the SQL adaptation protocol. Objects conforming this protocol should implement a *getquoted()* and optionally a *prepare()* method.

Adapters may subclass `ISQLQuote`, but is not necessary: it is enough to expose a `getquoted()` method to be conforming.

_wrapped
The wrapped object passes to the constructor

getquoted()
Subclasses or other conforming objects should return a valid SQL string representing the wrapped object. In Python 3 the SQL must be returned in a `bytes` object. The `ISQLQuote` implementation does nothing.

prepare(*conn*)
Prepare the adapter for a connection. The method is optional: if implemented, it will be invoked before `getquoted()` with the connection to adapt for as argument.

A conform object can implement this method if the SQL representation depends on any server parameter, such as the server version or the `standard_conforming_string` setting. Container objects may store the connection and use it to recursively prepare contained objects: see the implementation for *psycopg2.extensions.SQL_IN* for a simple example.

class psycopg2.extensions.**AsIs**(*object*)
Adapter conform to the *ISQLQuote* protocol useful for objects whose string representation is already valid as SQL representation.

getquoted()
Return the `str()` conversion of the wrapped object.

```
>>> AsIs(42).getquoted()
'42'
```

class psycopg2.extensions.**QuotedString**(*str*)
Adapter conform to the *ISQLQuote* protocol for string-like objects.

getquoted()
Return the string enclosed in single quotes. Any single quote appearing in the the string is escaped by doubling it according to SQL string constants syntax. Backslashes are escaped too.

```
>>> QuotedString(r"O'Reilly").getquoted()
"'O''Reilly'"
```

class psycopg2.extensions.**Binary**(*str*)
Adapter conform to the *ISQLQuote* protocol for binary objects.

getquoted()
Return the string enclosed in single quotes. It performs the same escaping of the *QuotedString* adapter, plus it knows how to escape non-printable chars.

```
>>> Binary("\x00\x08\x0F").getquoted()
"'\\\\000\\\\010\\\\017'"
```

Changed in version 2.0.14: previously the adapter was not exposed by the `extensions` module. In older versions it can be imported from the implementation module `psycopg2._psycopg`.

class psycopg2.extensions.**Boolean**
class psycopg2.extensions.**Float**
class psycopg2.extensions.**SQL_IN**
Specialized adapters for builtin objects.

class psycopg2.extensions.**DateFromPy**
class psycopg2.extensions.**TimeFromPy**
class psycopg2.extensions.**TimestampFromPy**
class psycopg2.extensions.**IntervalFromPy**
Specialized adapters for Python datetime objects.

class psycopg2.extensions.**DateFromMx**
class psycopg2.extensions.**TimeFromMx**
class psycopg2.extensions.**TimestampFromMx**

class psycopg2.extensions.**IntervalFromMx**
Specialized adapters for mx.DateTime objects.

psycopg2.extensions.**adapters**
Dictionary of the currently registered object adapters. Use *register_adapter()* to add an adapter for a new type.

7.3 Database types casting functions

These functions are used to manipulate type casters to convert from PostgreSQL types to Python objects. See *Type casting of SQL types into Python objects* for details.

psycopg2.extensions.**new_type**(*oids*, *name*, *adapter*)
Create a new type caster to convert from a PostgreSQL type to a Python object. The object created must be registered using *register_type()* to be used.

Parameters

- **oids** – tuple of OIDs of the PostgreSQL type to convert.
- **name** – the name of the new type adapter.
- **adapter** – the adaptation function.

The object OID can be read from the *cursor.description* attribute or by querying from the PostgreSQL catalog.

adapter should have signature `fun(value, cur)` where *value* is the string representation returned by PostgreSQL and *cur* is the cursor from which data are read. In case of `NULL`, *value* will be `None`. The adapter should return the converted object.

See *Type casting of SQL types into Python objects* for an usage example.

psycopg2.extensions.**new_array_type**(*oids*, *name*, *base_caster*)
Create a new type caster to convert from a PostgreSQL array type to a list of Python object. The object created must be registered using *register_type()* to be used.

Parameters

- **oids** – tuple of OIDs of the PostgreSQL type to convert. It should probably contain the oid of the array type (e.g. the `typarray` field in the `pg_type` table).
- **name** – the name of the new type adapter.
- **base_caster** – a Psycopg typecaster, e.g. created using the *new_type()* function. The caster should be able to parse a single item of the desired type.

New in version 2.4.3.

Note: The function can be used to create a generic array typecaster, returning a list of strings: just use *psycopg2.STRING* as base typecaster. For instance, if you want to receive an array of `macaddr` from the database, each address represented by string, you can use:

```
# select typarray from pg_type where typname = 'macaddr' -> 1040
psycopg2.extensions.register_type(
    psycopg2.extensions.new_array_type(
        (1040,), 'MACADDR[]', psycopg2.STRING))
```

psycopg2.extensions.**register_type**(*obj*[, *scope*])
Register a type caster created using *new_type()*.

If *scope* is specified, it should be a *connection* or a *cursor*: the type caster will be effective only limited to the specified object. Otherwise it will be globally registered.

psycopg2.extensions.**string_types**
The global register of type casters.

psycopg2.extensions.**encodings**
Mapping from PostgreSQL encoding to Python encoding names. Used by Psycopg when adapting or casting unicode strings. See *Unicode handling*.

7.4 Additional exceptions

The module exports a few exceptions in addition to the *standard ones* defined by the DB API 2.0.

exception psycopg2.extensions.**QueryCanceledError**
(subclasses *OperationalError*)

Error related to SQL query cancellation. It can be trapped specifically to detect a timeout.

New in version 2.0.7.

exception psycopg2.extensions.**TransactionRollbackError**
(subclasses *OperationalError*)

Error causing transaction rollback (deadlocks, serialization failures, etc). It can be trapped specifically to detect a deadlock.

New in version 2.0.7.

7.5 Coroutines support functions

These functions are used to set and retrieve the callback function for *cooperation with coroutine libraries*.

New in version 2.2.0.

psycopg2.extensions.**set_wait_callback**(*f*)
Register a callback function to block waiting for data.

The callback should have signature `fun(conn)` and is called to wait for data available whenever a blocking function from the libpq is called. Use `set_wait_callback(None)` to revert to the original behaviour (i.e. using blocking libpq functions).

The function is an hook to allow coroutine-based libraries (such as Eventlet or gevent) to switch when Psycopg is blocked, allowing other coroutines to run concurrently.

See *wait_select()* for an example of a wait callback implementation.

psycopg2.extensions.**get_wait_callback**()
Return the currently registered wait callback.

Return `None` if no callback is currently registered.

7.6 Other functions

psycopg2.extensions.**libpq_version**()
Return the version number of the libpq dynamic library loaded as an integer, in the same format of *server_version*.

Raise *NotSupportedError* if the psycopg2 module was compiled with a libpq version lesser than 9.1 (which can be detected by the *__libpq_version__* constant).

New in version 2.7.

See also:

libpq docs for PQlibVersion().

psycopg2.extensions.**make_dsn**(*dsn=None*, ***kwargs*)
Create a valid connection string from arguments.

Put together the arguments in *kwargs* into a connection string. If *dsn* is specified too, merge the arguments coming from both the sources. If the same argument name is specified in both the sources, the *kwargs* value overrides the *dsn* value.

The input arguments are validated: the output should always be a valid connection string (as far as *parse_dsn()* is concerned). If not raise *ProgrammingError*.

Example:

```
>>> from psycopg2.extensions import make_dsn
>>> make_dsn('dbname=foo host=example.com', password="s3cr3t")
'host=example.com password=s3cr3t dbname=foo'
```

New in version 2.7.

psycopg2.extensions.**parse_dsn**(*dsn*)
Parse connection string into a dictionary of keywords and values.

Parsing is delegated to the libpq: different versions of the client library may support different formats or parameters (for example, connection URIs are only supported from libpq 9.2). Raise *ProgrammingError* if the *dsn* is not valid.

Example:

```
>>> from psycopg2.extensions import parse_dsn
>>> parse_dsn('dbname=test user=postgres password=secret')
{'password': 'secret', 'user': 'postgres', 'dbname': 'test'}
>>> parse_dsn("postgresql://someone@example.com/somedb?connect_timeout=10")
{'host': 'example.com', 'user': 'someone', 'dbname': 'somedb', 'connect_timeout':
↪'10'}
```

New in version 2.7.

See also:

libpq docs for PQconninfoParse().

psycopg2.extensions.**quote_ident**(*str*, *scope*)
Return quoted identifier according to PostgreSQL quoting rules.

The *scope* must be a *connection* or a *cursor*, the underlying connection encoding is used for any necessary character conversion.

New in version 2.7.

See also:

libpq docs for PQescapeIdentifier()

7.7 Isolation level constants

Psycopg2 *connection* objects hold informations about the PostgreSQL transaction isolation level. By default Psycopg doesn't change the default configuration of the server (*ISOLATION_LEVEL_DEFAULT*); the default for PostgreSQL servers is typically READ COMMITTED, but this may be changed in the server configuration files. A different isolation level can be set through the *set_isolation_level()* or *set_session()* methods. The level can be set to one of the following constants:

psycopg2.extensions.**ISOLATION_LEVEL_AUTOCOMMIT**
No transaction is started when commands are executed and no *commit()* or *rollback()* is required. Some PostgreSQL command such as CREATE DATABASE or VACUUM can't run into a transaction: to run such command use:

```
>>> conn.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT)
```

See also *Transactions control*.

psycopg2.extensions.**ISOLATION_LEVEL_READ_UNCOMMITTED**
The READ UNCOMMITTED isolation level is defined in the SQL standard but not available in the MVCC (Multiversion concurrency control) model of PostgreSQL: it is replaced by the stricter READ COMMITTED.

psycopg2.extensions.**ISOLATION_LEVEL_READ_COMMITTED**
This is usually the the default PostgreSQL value, but a different default may be set in the database configuration.

A new transaction is started at the first *execute()* command on a cursor and at each new execute() after a *commit()* or a *rollback()*. The transaction runs in the PostgreSQL READ COMMITTED isolation level: a SELECT query sees only data committed before the query began; it never sees either uncommitted data or changes committed during query execution by concurrent transactions.

See also:

Read Committed Isolation Level in PostgreSQL documentation.

psycopg2.extensions.**ISOLATION_LEVEL_REPEATABLE_READ**
As in ISOLATION_LEVEL_READ_COMMITTED, a new transaction is started at the first *execute()* command. Transactions run at a REPEATABLE READ isolation level: all the queries in a transaction see a snapshot as of the start of the transaction, not as of the start of the current query within the transaction. However applications using this level must be prepared to retry transactions due to serialization failures.

While this level provides a guarantee that each transaction sees a completely stable view of the database, this view will not necessarily always be consistent with some serial (one at a time) execution of concurrent transactions of the same level.

Changed in version 2.4.2: The value was an alias for ISOLATION_LEVEL_SERIALIZABLE before. The two levels are distinct since PostgreSQL 9.1

See also:

Repeatable Read Isolation Level in PostgreSQL documentation.

psycopg2.extensions.**ISOLATION_LEVEL_SERIALIZABLE**
As in ISOLATION_LEVEL_READ_COMMITTED, a new transaction is started at the first *execute()* command. Transactions run at a SERIALIZABLE isolation level. This is the strictest transactions isolation level, equivalent to having the transactions executed serially rather than concurrently. However applications using this level must be prepared to retry transactions due to serialization failures.

Starting from PostgreSQL 9.1, this mode monitors for conditions which could make execution of a concurrent set of serializable transactions behave in a manner inconsistent with all possible serial (one at a time) executions of those transaction. In previous version the behaviour was the same of the `REPEATABLE READ` isolation level.

See also:

Serializable Isolation Level in PostgreSQL documentation.

`psycopg2.extensions.`**`ISOLATION_LEVEL_DEFAULT`**
A new transaction is started at the first *`execute()`* command, but the isolation level is not explicitly selected by Psycopg: the server will use whatever level is defined in its configuration or by statements executed within the session outside Pyscopg control. If you want to know what the value is you can use a query such as `show transaction_isolation`.

New in version 2.7.

7.8 Transaction status constants

These values represent the possible status of a transaction: the current value can be read using the *`connection.get_transaction_status()`* method.

`psycopg2.extensions.`**`TRANSACTION_STATUS_IDLE`**
The session is idle and there is no current transaction.

`psycopg2.extensions.`**`TRANSACTION_STATUS_ACTIVE`**
A command is currently in progress.

`psycopg2.extensions.`**`TRANSACTION_STATUS_INTRANS`**
The session is idle in a valid transaction block.

`psycopg2.extensions.`**`TRANSACTION_STATUS_INERROR`**
The session is idle in a failed transaction block.

`psycopg2.extensions.`**`TRANSACTION_STATUS_UNKNOWN`**
Reported if the connection with the server is bad.

7.9 Connection status constants

These values represent the possible status of a connection: the current value can be read from the *`status`* attribute.

It is possible to find the connection in other status than the one shown below. Those are the only states in which a working connection is expected to be found during the execution of regular Python client code: other states are for internal usage and Python code should not rely on them.

`psycopg2.extensions.`**`STATUS_READY`**
Connection established. No transaction in progress.

`psycopg2.extensions.`**`STATUS_BEGIN`**
Connection established. A transaction is currently in progress.

`psycopg2.extensions.`**`STATUS_IN_TRANSACTION`**
An alias for *`STATUS_BEGIN`*

`psycopg2.extensions.`**`STATUS_PREPARED`**
The connection has been prepared for the second phase in a *two-phase commit* transaction. The connection can't be used to send commands to the database until the transaction is finished with *`tpc_commit()`* or *`tpc_rollback()`*.

New in version 2.3.

7.10 Poll constants

New in version 2.2.0.

These values can be returned by *connection.poll()* during asynchronous connection and communication. They match the values in the libpq enum `PostgresPollingStatusType`. See *Asynchronous support* and *Support for coroutine libraries*.

psycopg2.extensions.**POLL_OK**
The data being read is available, or the file descriptor is ready for writing: reading or writing will not block.

psycopg2.extensions.**POLL_READ**
Some data is being read from the backend, but it is not available yet on the client and reading would block. Upon receiving this value, the client should wait for the connection file descriptor to be ready *for reading*. For example:

```
select.select([conn.fileno()], [], [])
```

psycopg2.extensions.**POLL_WRITE**
Some data is being sent to the backend but the connection file descriptor can't currently accept new data. Upon receiving this value, the client should wait for the connection file descriptor to be ready *for writing*. For example:

```
select.select([], [conn.fileno()], [])
```

psycopg2.extensions.**POLL_ERROR**
There was a problem during connection polling. This value should actually never be returned: in case of poll error usually an exception containing the relevant details is raised.

7.11 Additional database types

The `extensions` module includes typecasters for many standard PostgreSQL types. These objects allow the conversion of returned data into Python objects. All the typecasters are automatically registered, except *UNICODE* and *UNICODEARRAY*: you can register them using *register_type()* in order to receive Unicode objects instead of strings from the database. See *Unicode handling* for details.

psycopg2.extensions.**BOOLEAN**
psycopg2.extensions.**DATE**
psycopg2.extensions.**DECIMAL**
psycopg2.extensions.**FLOAT**
psycopg2.extensions.**INTEGER**
psycopg2.extensions.**INTERVAL**
psycopg2.extensions.**LONGINTEGER**
psycopg2.extensions.**TIME**
psycopg2.extensions.**UNICODE**
Typecasters for basic types. Note that a few other ones (*BINARY*, *DATETIME*, *NUMBER*, *ROWID*, *STRING*) are exposed by the *psycopg2* module for DB API 2.0 compliance.

psycopg2.extensions.**BINARYARRAY**
psycopg2.extensions.**BOOLEANARRAY**
psycopg2.extensions.**DATEARRAY**
psycopg2.extensions.**DATETIMEARRAY**
psycopg2.extensions.**DECIMALARRAY**

psycopg2.extensions.**FLOATARRAY**
psycopg2.extensions.**INTEGERARRAY**
psycopg2.extensions.**INTERVALARRAY**
psycopg2.extensions.**LONGINTEGERARRAY**
psycopg2.extensions.**ROWIDARRAY**
psycopg2.extensions.**STRINGARRAY**
psycopg2.extensions.**TIMEARRAY**
psycopg2.extensions.**UNICODEARRAY**

Typecasters to convert arrays of sql types into Python lists.

psycopg2.extensions.**PYDATE**
psycopg2.extensions.**PYDATETIME**
psycopg2.extensions.**PYDATETIMETZ**
psycopg2.extensions.**PYINTERVAL**
psycopg2.extensions.**PYTIME**
psycopg2.extensions.**PYDATEARRAY**
psycopg2.extensions.**PYDATETIMEARRAY**
psycopg2.extensions.**PYDATETIMETZARRAY**
psycopg2.extensions.**PYINTERVALARRAY**
psycopg2.extensions.**PYTIMEARRAY**

Typecasters to convert time-related data types to Python `datetime` objects.

psycopg2.extensions.**MXDATE**
psycopg2.extensions.**MXDATETIME**
psycopg2.extensions.**MXDATETIMETZ**
psycopg2.extensions.**MXINTERVAL**
psycopg2.extensions.**MXTIME**
psycopg2.extensions.**MXDATEARRAY**
psycopg2.extensions.**MXDATETIMEARRAY**
psycopg2.extensions.**MXDATETIMETZARRAY**
psycopg2.extensions.**MXINTERVALARRAY**
psycopg2.extensions.**MXTIMEARRAY**

Typecasters to convert time-related data types to mx.DateTime objects. Only available if Psycopg was compiled with `mx` support.

Changed in version 2.2.0: previously the *`DECIMAL`* typecaster and the specific time-related typecasters (`PY*` and `MX*`) were not exposed by the `extensions` module. In older versions they can be imported from the implementation module `psycopg2._psycopg`.

Changed in version 2.7.2: added `*DATETIMETZ*` objects.

CHAPTER

EIGHT

PSYCOPG2.EXTRAS – MISCELLANEOUS GOODIES FOR PSYCOPG 2

This module is a generic place used to hold little helper functions and classes until a better place in the distribution is found.

8.1 Connection and cursor subclasses

A few objects that change the way the results are returned by the cursor or modify the object behavior in some other way. Typically `cursor` subclasses are passed as *cursor_factory* argument to *`connect()`* so that the connection's *`cursor()`* method will generate objects of this class. Alternatively a `cursor` subclass can be used one-off by passing it as the *cursor_factory* argument to the `cursor()` method.

If you want to use a `connection` subclass you can pass it as the *connection_factory* argument of the `connect()` function.

8.1.1 Dictionary-like cursor

The dict cursors allow to access to the retrieved records using an interface similar to the Python dictionaries instead of the tuples.

```
>>> dict_cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
>>> dict_cur.execute("INSERT INTO test (num, data) VALUES(%s, %s)",
...                  (100, "abc'def"))
>>> dict_cur.execute("SELECT * FROM test")
>>> rec = dict_cur.fetchone()
>>> rec['id']
1
>>> rec['num']
100
>>> rec['data']
"abc'def"
```

The records still support indexing as the original tuple:

```
>>> rec[2]
"abc'def"
```

class psycopg2.extras.**DictCursor**(**args*, ***kwargs*)
A cursor that keeps a list of column name -> index mappings.

class psycopg2.extras.**DictConnection**
A connection that uses *`DictCursor`* automatically.

Note: Not very useful since Psycopg 2.5: you can use *psycopg2.connect*(dsn, cursor_factory=DictCursor) instead of DictConnection.

class psycopg2.extras.**DictRow**(*cursor*)
A row object that allow by-column-name access to data.

8.1.2 Real dictionary cursor

class psycopg2.extras.**RealDictCursor**(**args*, ***kwargs*)
A cursor that uses a real dict as the base type for rows.

Note that this cursor is extremely specialized and does not allow the normal access (using integer indices) to fetched data. If you need to access database rows both as a dictionary and a list, then use the generic *DictCursor* instead of RealDictCursor.

class psycopg2.extras.**RealDictConnection**
A connection that uses *RealDictCursor* automatically.

Note: Not very useful since Psycopg 2.5: you can use *psycopg2.connect*(dsn, cursor_factory=RealDictCursor) instead of RealDictConnection.

class psycopg2.extras.**RealDictRow**(*cursor*)
A dict subclass representing a data record.

8.1.3 namedtuple cursor

New in version 2.3.

class psycopg2.extras.**NamedTupleCursor**
A cursor that generates results as namedtuple.

fetch*() methods will return named tuples instead of regular tuples, so their elements can be accessed both as regular numeric items as well as attributes.

```
>>> nt_cur = conn.cursor(cursor_factory=psycopg2.extras.NamedTupleCursor)
>>> rec = nt_cur.fetchone()
>>> rec
Record(id=1, num=100, data="abc'def")
>>> rec[1]
100
>>> rec.data
"abc'def"
```

class psycopg2.extras.**NamedTupleConnection**
A connection that uses *NamedTupleCursor* automatically.

Note: Not very useful since Psycopg 2.5: you can use *psycopg2.connect*(dsn, cursor_factory=NamedTupleCursor) instead of NamedTupleConnection.

8.1.4 Logging cursor

class psycopg2.extras.**LoggingConnection**
A connection that logs all queries to a file or logger object.

filter(*msg*, *curs*)
Filter the query before logging it.

This is the method to overwrite to filter unwanted queries out of the log or to add some extra data to the output. The default implementation just does nothing.

initialize(*logobj*)
Initialize the connection to log to `logobj`.

The `logobj` parameter can be an open file object or a Logger instance from the standard logging module.

class psycopg2.extras.**LoggingCursor**
A cursor that logs queries using its connection logging facilities.

Note: Queries that are executed with *cursor.executemany()* are not logged.

---

class psycopg2.extras.**MinTimeLoggingConnection**
A connection that logs queries based on execution time.

This is just an example of how to sub-class *LoggingConnection* to provide some extra filtering for the logged queries. Both the *initialize()* and *filter()* methods are overwritten to make sure that only queries executing for more than `mintime` ms are logged.

Note that this connection uses the specialized cursor *MinTimeLoggingCursor*.

filter(*msg*, *curs*)
Filter the query before logging it.

This is the method to overwrite to filter unwanted queries out of the log or to add some extra data to the output. The default implementation just does nothing.

initialize(*logobj*, *mintime=0*)
Initialize the connection to log to `logobj`.

The `logobj` parameter can be an open file object or a Logger instance from the standard logging module.

class psycopg2.extras.**MinTimeLoggingCursor**
The cursor sub-class companion to *MinTimeLoggingConnection*.

8.1.5 Replication connection and cursor classes

See *Replication protocol support* for an introduction to the topic.

The following replication types are defined:

psycopg2.extras.**REPLICATION_LOGICAL**

psycopg2.extras.**REPLICATION_PHYSICAL**

class psycopg2.extras.**LogicalReplicationConnection**(**args*, ***kwargs*)
This connection factory class can be used to open a special type of connection that is used for logical replication.

Example:

```
from psycopg2.extras import LogicalReplicationConnection
log_conn = psycopg2.connect(dsn, connection_factory=LogicalReplicationConnection)
log_cur = log_conn.cursor()
```

class psycopg2.extras.**PhysicalReplicationConnection**(**args*, ***kwargs*)
This connection factory class can be used to open a special type of connection that is used for physical replication.

Example:

```
from psycopg2.extras import PhysicalReplicationConnection
phys_conn = psycopg2.connect(dsn, connection_
↪factory=PhysicalReplicationConnection)
phys_cur = phys_conn.cursor()
```

Both *LogicalReplicationConnection* and *PhysicalReplicationConnection* use *ReplicationCursor* for actual communication with the server.

The individual messages in the replication stream are represented by *ReplicationMessage* objects (both logical and physical type):

class psycopg2.extras.**ReplicationMessage**
A replication protocol message.

payload
The actual data received from the server.

An instance of either `bytes()` or `unicode()`, depending on the value of `decode` option passed to *start_replication()* on the connection. See *read_message()* for details.

data_size
The raw size of the message payload (before possible unicode conversion).

data_start
LSN position of the start of the message.

wal_end
LSN position of the current end of WAL on the server.

send_time
A `datetime` object representing the server timestamp at the moment when the message was sent.

cursor
A reference to the corresponding *ReplicationCursor* object.

class psycopg2.extras.**ReplicationCursor**
A cursor used for communication on replication connections.

create_replication_slot(*slot_name*, *slot_type=None*, *output_plugin=None*)
Create streaming replication slot.

Parameters

- **slot_name** – name of the replication slot to be created
- **slot_type** – type of replication: should be either *REPLICATION_LOGICAL* or *REPLICATION_PHYSICAL*
- **output_plugin** – name of the logical decoding output plugin to be used by the slot; required for logical replication connections, disallowed for physical

Example:

```
log_cur.create_replication_slot("logical1", "test_decoding")
phys_cur.create_replication_slot("physical1")

# either logical or physical replication connection
cur.create_replication_slot("slot1", slot_type=REPLICATION_LOGICAL)
```

When creating a slot on a logical replication connection, a logical replication slot is created by default. Logical replication requires name of the logical decoding output plugin to be specified.

When creating a slot on a physical replication connection, a physical replication slot is created by default. No output plugin parameter is required or allowed when creating a physical replication slot.

In either case the type of slot being created can be specified explicitly using *slot_type* parameter.

Replication slots are a feature of PostgreSQL server starting with version 9.4.

drop_replication_slot(*slot_name*)

Drop streaming replication slot.

Parameters **slot_name** – name of the replication slot to drop

Example:

```
# either logical or physical replication connection
cur.drop_replication_slot("slot1")
```

Replication slots are a feature of PostgreSQL server starting with version 9.4.

start_replication(*slot_name=None*, *slot_type=None*, *start_lsn=0*, *timeline=0*, *options=None*, *decode=False*)

Start replication on the connection.

Parameters

- **slot_name** – name of the replication slot to use; required for logical replication, physical replication can work with or without a slot
- **slot_type** – type of replication: should be either *REPLICATION_LOGICAL* or *REPLICATION_PHYSICAL*
- **start_lsn** – the optional LSN position to start replicating from, can be an integer or a string of hexadecimal digits in the form `XXX/XXX`
- **timeline** – WAL history timeline to start streaming from (optional, can only be used with physical replication)
- **options** – a dictionary of options to pass to logical replication slot (not allowed with physical replication)
- **decode** – a flag indicating that unicode conversion should be performed on messages received from the server

If a *slot_name* is specified, the slot must exist on the server and its type must match the replication type used.

If not specified using *slot_type* parameter, the type of replication is defined by the type of replication connection. Logical replication is only allowed on logical replication connection, but physical replication can be used with both types of connection.

On the other hand, physical replication doesn't require a named replication slot to be used, only logical replication does. In any case logical replication and replication slots are a feature of PostgreSQL server starting with version 9.4. Physical replication can be used starting with 9.0.

If *start_lsn* is specified, the requested stream will start from that LSN. The default is `None` which passes the LSN `0/0` causing replay to begin at the last point for which the server got flush confirmation from the client, or the oldest available point for a new slot.

The server might produce an error if a WAL file for the given LSN has already been recycled or it may silently start streaming from a later position: the client can verify the actual position using information provided by the *ReplicationMessage* attributes. The exact server behavior depends on the type of replication and use of slots.

The *timeline* parameter can only be specified with physical replication and only starting with server version 9.3.

A dictionary of *options* may be passed to the logical decoding plugin on a logical replication slot. The set of supported options depends on the output plugin that was used to create the slot. Must be `None` for physical replication.

If *decode* is set to `True` the messages received from the server would be converted according to the connection *encoding*. *This parameter should not be set with physical replication or with logical replication plugins that produce binary output.*

This function constructs a `START_REPLICATION` command and calls *start_replication_expert()* internally.

After starting the replication, to actually consume the incoming server messages use *consume_stream()* or implement a loop around *read_message()* in case of *asynchronous connection*.

start_replication_expert(*command*, *decode=False*)

Start replication on the connection using provided `START_REPLICATION` command.

Parameters

- **command** – The full replication command. It can be a string or a *Composable* instance for dynamic generation.
- **decode** – a flag indicating that unicode conversion should be performed on messages received from the server.

consume_stream(*consume*, *keepalive_interval=10*)

Parameters

- **consume** – a callable object with signature `consume(msg)`
- **keepalive_interval** – interval (in seconds) to send keepalive messages to the server

This method can only be used with synchronous connection. For asynchronous connections see *read_message()*.

Before using this method to consume the stream call *start_replication()* first.

This method enters an endless loop reading messages from the server and passing them to `consume()` one at a time, then waiting for more messages from the server. In order to make this method break out of the loop and return, `consume()` can throw a *StopReplication* exception. Any unhandled exception will make it break out of the loop as well.

The *msg* object passed to `consume()` is an instance of *ReplicationMessage* class. See *read_message()* for details about message decoding.

This method also sends keepalive messages to the server in case there were no new data from the server for the duration of *keepalive_interval* (in seconds). The value of this parameter must be set to at least 1 second, but it can have a fractional part.

After processing certain amount of messages the client should send a confirmation message to the server. This should be done by calling *send_feedback()* method on the corresponding replication cursor. A reference to the cursor is provided in the *ReplicationMessage* as an attribute.

The following example is a sketch implementation of `consume()` callable for logical replication:

```
class LogicalStreamConsumer(object):

    # ...

    def __call__(self, msg):
        self.process_message(msg.payload)

        if self.should_send_feedback(msg):
            msg.cursor.send_feedback(flush_lsn=msg.data_start)

consumer = LogicalStreamConsumer()
cur.consume_stream(consumer)
```

Warning: When using replication with slots, failure to constantly consume *and* report success to the server appropriately can eventually lead to "disk full" condition on the server, because the server retains all the WAL segments that might be needed to stream the changes via all of the currently open replication slots.

On the other hand, it is not recommended to send confirmation after *every* processed message, since that will put an unnecessary load on network and the server. A possible strategy is to confirm after every COMMIT message.

send_feedback(*write_lsn=0*, *flush_lsn=0*, *apply_lsn=0*, *reply=False*)

Parameters

- **write_lsn** – a LSN position up to which the client has written the data locally
- **flush_lsn** – a LSN position up to which the client has processed the data reliably (the server is allowed to discard all and every data that predates this LSN)
- **apply_lsn** – a LSN position up to which the warm standby server has applied the changes (physical replication master-slave protocol only)
- **reply** – request the server to send back a keepalive message immediately

Use this method to report to the server that all messages up to a certain LSN position have been processed on the client and may be discarded on the server.

This method can also be called with all default parameters' values to just send a keepalive message to the server.

Low-level replication cursor methods for *asynchronous connection* operation.

With the synchronous connection a call to *consume_stream()* handles all the complexity of handling the incoming messages and sending keepalive replies, but at times it might be beneficial to use low-level interface for better control, in particular to `select` on multiple sockets. The following methods are provided for asynchronous operation:

read_message()

Try to read the next message from the server without blocking and return an instance of *ReplicationMessage* or `None`, in case there are no more data messages from the server at the moment.

This method should be used in a loop with asynchronous connections (after calling *start_replication()* once). For synchronous connections see *consume_stream()*.

The returned message's *payload* is an instance of `unicode` decoded according to connection *encoding* *iff* *decode* was set to `True` in the initial call to *start_replication()* on this connection, otherwise it is an instance of `bytes` with no decoding.

It is expected that the calling code will call this method repeatedly in order to consume all of the messages that might have been buffered until `None` is returned. After receiving `None` from this method the caller should use `select()` or `poll()` on the corresponding connection to block the process until there is more data from the server.

The server can send keepalive messages to the client periodically. Such messages are silently consumed by this method and are never reported to the caller.

fileno()
Call the corresponding connection's *fileno()* method and return the result.

This is a convenience method which allows replication cursor to be used directly in `select()` or `poll()` calls.

io_timestamp
A `datetime` object representing the timestamp at the moment of last communication with the server (a data or keepalive message in either direction).

An actual example of asynchronous operation might look like this:

```
from select import select
from datetime import datetime

def consume(msg):
    # ...

keepalive_interval = 10.0
while True:
    msg = cur.read_message()
    if msg:
        consume(msg)
    else:
        now = datetime.now()
        timeout = keepalive_interval - (now - cur.io_timestamp).total_seconds()
        try:
            sel = select([cur], [], [], max(0, timeout))
            if not any(sel):
                cur.send_feedback()  # timed out, send keepalive message
        except InterruptedError:
            pass  # recalculate timeout and continue
```

class psycopg2.extras.**StopReplication**
Exception used to break out of the endless loop in *consume_stream()*.

Subclass of `Exception`. Intentionally *not* inherited from *Error* as occurrence of this exception does not indicate an error.

8.2 Additional data types

8.2.1 JSON adaptation

New in version 2.5.

Changed in version 2.5.4: added `jsonb` support. In previous versions `jsonb` values are returned as strings. See *the FAQ* for a workaround.

Psycopg can adapt Python objects to and from the PostgreSQL `json` and `jsonb` types. With PostgreSQL 9.2 and following versions adaptation is available out-of-the-box. To use JSON data with previous database versions (either with the 9.1 json extension, but even if you want to convert text fields to JSON) you can use the *`register_json()`* function.

The Python library used by default to convert Python objects to JSON and to parse data from the database depends on the language version: with Python 2.6 and following the `json` module from the standard library is used; with previous versions the simplejson module is used if available. Note that the last `simplejson` version supporting Python 2.4 is the 2.0.9.

In order to pass a Python object to the database as query argument you can use the *`Json`* adapter:

```
curs.execute("insert into mytable (jsondata) values (%s)",
    [Json({'a': 100})])
```

Reading from the database, `json` and `jsonb` values will be automatically converted to Python objects.

Note: If you are using the PostgreSQL `json` data type but you want to read it as string in Python instead of having it parsed, your can either cast the column to `text` in the query (it is an efficient operation, that doesn't involve a copy):

```
cur.execute("select jsondata::text from mytable")
```

or you can register a no-op `loads()` function with *`register_default_json()`*:

```
psycopg2.extras.register_default_json(loads=lambda x: x)
```

Note: You can use *`register_adapter()`* to adapt any Python dictionary to JSON, either registering *`Json`* or any subclass or factory creating a compatible adapter:

```
psycopg2.extensions.register_adapter(dict, psycopg2.extras.Json)
```

This setting is global though, so it is not compatible with similar adapters such as the one registered by *`register_hstore()`*. Any other object supported by JSON can be registered the same way, but this will clobber the default adaptation rule, so be careful to unwanted side effects.

If you want to customize the adaptation from Python to PostgreSQL you can either provide a custom `dumps()` function to *`Json`*:

```
curs.execute("insert into mytable (jsondata) values (%s)",
    [Json({'a': 100}, dumps=simplejson.dumps)])
```

or you can subclass it overriding the *`dumps()`* method:

```
class MyJson(Json):
    def dumps(self, obj):
        return simplejson.dumps(obj)

curs.execute("insert into mytable (jsondata) values (%s)",
    [MyJson({'a': 100})])
```

Customizing the conversion from PostgreSQL to Python can be done passing a custom `loads()` function to *`register_json()`*. For the builtin data types (`json` from PostgreSQL 9.2, `jsonb` from PostgreSQL 9.4) use *`register_default_json()`* and *`register_default_jsonb()`*. For example, if you want to convert the float values from `json` into `Decimal` you can use:

```
loads = lambda x: json.loads(x, parse_float=Decimal)
psycopg2.extras.register_json(conn, loads=loads)
```

class psycopg2.extras.**Json**(*adapted*, *dumps=None*)

An *`ISQLQuote`* wrapper to adapt a Python object to `json` data type.

`Json` can be used to wrap any object supported by the provided *dumps* function. If none is provided, the standard `json.dumps()` is used (`simplejson` for Python < 2.6; *`getquoted()`* will raise `ImportError` if the module is not available).

dumps(*obj*)

Serialize *obj* in JSON format.

The default is to call `json.dumps()` or the *dumps* function provided in the constructor. You can override this method to create a customized JSON wrapper.

psycopg2.extras.**register_json**(*conn_or_curs=None*, *globally=False*, *loads=None*, *oid=None*, *array_oid=None*, *name='json'*)

Create and register typecasters converting `json` type to Python objects.

Parameters

- **conn_or_curs** – a connection or cursor used to find the `json` and `json[]` oids; the typecasters are registered in a scope limited to this object, unless *globally* is set to `True`. It can be `None` if the oids are provided
- **globally** – if `False` register the typecasters only on *conn_or_curs*, otherwise register them globally
- **loads** – the function used to parse the data into a Python object. If `None` use `json.loads()`, where `json` is the module chosen according to the Python version (see above)
- **oid** – the OID of the `json` type if known; If not, it will be queried on *conn_or_curs*
- **array_oid** – the OID of the `json[]` array type if known; if not, it will be queried on *conn_or_curs*
- **name** – the name of the data type to look for in *conn_or_curs*

The connection or cursor passed to the function will be used to query the database and look for the OID of the `json` type (or an alternative type if *name* if provided). No query is performed if *oid* and *array_oid* are provided. Raise *`ProgrammingError`* if the type is not found.

Changed in version 2.5.4: added the *name* parameter to enable `jsonb` support.

psycopg2.extras.**register_default_json**(*conn_or_curs=None*, *globally=False*, *loads=None*)

Create and register `json` typecasters for PostgreSQL 9.2 and following.

Since PostgreSQL 9.2 `json` is a builtin type, hence its oid is known and fixed. This function allows specifying a customized *loads* function for the default `json` type without querying the database. All the parameters have the same meaning of *register_json()*.

psycopg2.extras.**register_default_jsonb**(*conn_or_curs=None*, *globally=False*, *loads=None*)
Create and register `jsonb` typecasters for PostgreSQL 9.4 and following.

As in *register_default_json()*, the function allows to register a customized *loads* function for the `jsonb` type at its known oid for PostgreSQL 9.4 and following versions. All the parameters have the same meaning of *register_json()*.

New in version 2.5.4.

8.2.2 Hstore data type

New in version 2.3.

The `hstore` data type is a key-value store embedded in PostgreSQL. It has been available for several server versions but with the release 9.0 it has been greatly improved in capacity and usefulness with the addition of many functions. It supports GiST or GIN indexes allowing search by keys or key/value pairs as well as regular BTree indexes for equality, uniqueness etc.

Psycopg can convert Python `dict` objects to and from `hstore` structures. Only dictionaries with string/unicode keys and values are supported. `None` is also allowed as value but not as a key. Psycopg uses a more efficient `hstore` representation when dealing with PostgreSQL 9.0 but previous server versions are supported as well. By default the adapter/typecaster are disabled: they can be enabled using the *register_hstore()* function.

psycopg2.extras.**register_hstore**(*conn_or_curs*, *globally=False*, *unicode=False*, *oid=None*, *array_oid=None*)
Register adapter and typecaster for `dict`-`hstore` conversions.

Parameters

- **conn_or_curs** – a connection or cursor: the typecaster will be registered only on this object unless *globally* is set to `True`
- **globally** – register the adapter globally, not only on *conn_or_curs*
- **unicode** – if `True`, keys and values returned from the database will be `unicode` instead of `str`. The option is not available on Python 3
- **oid** – the OID of the `hstore` type if known. If not, it will be queried on *conn_or_curs*.
- **array_oid** – the OID of the `hstore` array type if known. If not, it will be queried on *conn_or_curs*.

The connection or cursor passed to the function will be used to query the database and look for the OID of the `hstore` type (which may be different across databases). If querying is not desirable (e.g. with *asynchronous connections*) you may specify it in the *oid* parameter, which can be found using a query such as `SELECT 'hstore'::regtype::oid`. Analogously you can obtain a value for *array_oid* using a query such as `SELECT 'hstore[]'::regtype::oid`.

Note that, when passing a dictionary from Python to the database, both strings and unicode keys and values are supported. Dictionaries returned from the database have keys/values according to the *unicode* parameter.

The `hstore` contrib module must be already installed in the database (executing the `hstore.sql` script in your `contrib` directory). Raise *ProgrammingError* if the type is not found.

Changed in version 2.4: added the *oid* parameter. If not specified, the typecaster is installed also if `hstore` is not installed in the `public` schema.

Changed in version 2.4.3: added support for `hstore` array.

8.2.3 Composite types casting

New in version 2.4.

Using *register_composite()* it is possible to cast a PostgreSQL composite type (either created with the CREATE TYPE command or implicitly defined after a table row type) into a Python named tuple, or into a regular tuple if collections.namedtuple() is not found.

```
>>> cur.execute("CREATE TYPE card AS (value int, suit text);")
>>> psycopg2.extras.register_composite('card', cur)
<psycopg2.extras.CompositeCaster object at 0x...>

>>> cur.execute("select (8, 'hearts')::card")
>>> cur.fetchone()[0]
card(value=8, suit='hearts')
```

Nested composite types are handled as expected, provided that the type of the composite components are registered as well.

```
>>> cur.execute("CREATE TYPE card_back AS (face card, back text);")
>>> psycopg2.extras.register_composite('card_back', cur)
<psycopg2.extras.CompositeCaster object at 0x...>

>>> cur.execute("select ((8, 'hearts'), 'blue')::card_back")
>>> cur.fetchone()[0]
card_back(face=card(value=8, suit='hearts'), back='blue')
```

Adaptation from Python tuples to composite types is automatic instead and requires no adapter registration.

Note: If you want to convert PostgreSQL composite types into something different than a namedtuple you can subclass the *CompositeCaster* overriding *make()*. For example, if you want to convert your type into a Python dictionary you can use:

```
>>> class DictComposite(psycopg2.extras.CompositeCaster):
...     def make(self, values):
...         return dict(zip(self.attnames, values))

>>> psycopg2.extras.register_composite('card', cur,
...     factory=DictComposite)

>>> cur.execute("select (8, 'hearts')::card")
>>> cur.fetchone()[0]
{'suit': 'hearts', 'value': 8}
```

---

psycopg2.extras.**register_composite**(*name*, *conn_or_curs*, *globally=False*, *factory=None*)

Register a typecaster to convert a composite type into a tuple.

Parameters

- **name** – the name of a PostgreSQL composite type, e.g. created using the CREATE TYPE command
- **conn_or_curs** – a connection or cursor used to find the type oid and components; the typecaster is registered in a scope limited to this object, unless *globally* is set to True
- **globally** – if False (default) register the typecaster only on *conn_or_curs*, otherwise register it globally

- **factory** – if specified it should be a `CompositeCaster` subclass: use it to *customize how to cast composite types*

Returns the registered `CompositeCaster` or *factory* instance responsible for the conversion

Changed in version 2.4.3: added support for array of composite types

Changed in version 2.5: added the *factory* parameter

class psycopg2.extras.**CompositeCaster**(*name*, *oid*, *attrs*, *array_oid=None*, *schema=None*)
Helps conversion of a PostgreSQL composite type into a Python object.

The class is usually created by the `register_composite()` function. You may want to create and register manually instances of the class if querying the database at registration time is not desirable (such as when using an *asynchronous connections*).

make(*values*)
Return a new Python object representing the data being casted.

values is the list of attributes, already casted into their Python representation.

You can subclass this method to *customize the composite cast*.

New in version 2.5.

Object attributes:

name
The name of the PostgreSQL type.

schema
The schema where the type is defined.

New in version 2.5.

oid
The oid of the PostgreSQL type.

array_oid
The oid of the PostgreSQL array type, if available.

type
The type of the Python objects returned. If `collections.namedtuple()` is available, it is a named tuple with attributes equal to the type components. Otherwise it is just the `tuple` object.

attnames
List of component names of the type to be casted.

atttypes
List of component type oids of the type to be casted.

8.2.4 Range data types

New in version 2.5.

Psycopg offers a `Range` Python type and supports adaptation between them and PostgreSQL `range` types. Builtin `range` types are supported out-of-the-box; user-defined `range` types can be adapted using `register_range()`.

class psycopg2.extras.**Range**(*lower=None*, *upper=None*, *bounds='[)'*, *empty=False*)
Python representation for a PostgreSQL `range` type.

Parameters

- **lower** – lower bound for the range. `None` means unbound

- **upper** – upper bound for the range. `None` means unbound
- **bounds** – one of the literal strings `()`, `[)`, `(]`, `[]`, representing whether the lower or upper bounds are included
- **empty** – if `True`, the range is empty

This Python type is only used to pass and retrieve range values to and from PostgreSQL and doesn't attempt to replicate the PostgreSQL range features: it doesn't perform normalization and doesn't implement all the operators supported by the database.

`Range` objects are immutable, hashable, and support the `in` operator (checking if an element is within the range). They can be tested for equivalence. Empty ranges evaluate to `False` in boolean context, nonempty evaluate to `True`.

Changed in version 2.5.3: `Range` objects can be sorted although, as on the server-side, this ordering is not particularly meangingful. It is only meant to be used by programs assuming objects using `Range` as primary key can be sorted on them. In previous versions comparing `Ranges` raises `TypeError`.

Although it is possible to instantiate `Range` objects, the class doesn't have an adapter registered, so you cannot normally pass these instances as query arguments. To use range objects as query arguments you can either use one of the provided subclasses, such as *NumericRange* or create a custom subclass using *register_range()*.

Object attributes:

isempty
`True` if the range is empty.

lower
The lower bound of the range. `None` if empty or unbound.

upper
The upper bound of the range. `None` if empty or unbound.

lower_inc
`True` if the lower bound is included in the range.

upper_inc
`True` if the upper bound is included in the range.

lower_inf
`True` if the range doesn't have a lower bound.

upper_inf
`True` if the range doesn't have an upper bound.

The following *Range* subclasses map builtin PostgreSQL `range` types to Python objects: they have an adapter registered so their instances can be passed as query arguments. `range` values read from database queries are automatically casted into instances of these classes.

class psycopg2.extras.**NumericRange**(*lower=None*, *upper=None*, *bounds='[)'*, *empty=False*)
A *Range* suitable to pass Python numeric types to a PostgreSQL range.

PostgreSQL types `int4range`, `int8range`, `numrange` are casted into `NumericRange` instances.

class psycopg2.extras.**DateRange**(*lower=None*, *upper=None*, *bounds='[)'*, *empty=False*)
Represents `daterange` values.

class psycopg2.extras.**DateTimeRange**(*lower=None*, *upper=None*, *bounds='[)'*, *empty=False*)
Represents `tsrange` values.

class psycopg2.extras.**DateTimeTZRange**(*lower=None*, *upper=None*, *bounds='[)'*, *empty=False*)

Represents `tstzrange` values.

Note: Python lacks a representation for `infinity` date so Psycopg converts the value to `date.max` and such. When written into the database these dates will assume their literal value (e.g. `9999-12-31` instead of `infinity`). Check *Infinite dates handling* for an example of an alternative adapter to map `date.max` to `infinity`. An alternative dates adapter will be used automatically by the *DateRange* adapter and so on.

Custom `range` types (created with `CREATE TYPE ... AS RANGE`) can be adapted to a custom *Range* subclass:

psycopg2.extras.**register_range**(*pgrange*, *pyrange*, *conn_or_curs*, *globally=False*)

Create and register an adapter and the typecasters to convert between a PostgreSQL `range` type and a PostgreSQL *Range* subclass.

Parameters

- **pgrange** – the name of the PostgreSQL `range` type. Can be schema-qualified
- **pyrange** – a *Range* strict subclass, or just a name to give to a new class
- **conn_or_curs** – a connection or cursor used to find the oid of the range and its subtype; the typecaster is registered in a scope limited to this object, unless *globally* is set to `True`
- **globally** – if `False` (default) register the typecaster only on *conn_or_curs*, otherwise register it globally

Returns *RangeCaster* instance responsible for the conversion

If a string is passed to *pyrange*, a new *Range* subclass is created with such name and will be available as the *range* attribute of the returned *RangeCaster* object.

The function queries the database on *conn_or_curs* to inspect the *pgrange* type and raises *ProgrammingError* if the type is not found. If querying the database is not advisable, use directly the *RangeCaster* class and register the adapter and typecasters using the provided functions.

class psycopg2.extras.**RangeCaster**(*pgrange*, *pyrange*, *oid*, *subtype_oid*, *array_oid=None*)

Helper class to convert between *Range* and PostgreSQL range types.

Objects of this class are usually created by *register_range()*. Manual creation could be useful if querying the database is not advisable: in this case the oids must be provided.

Object attributes:

range
The `Range` subclass adapted.

adapter
The *ISQLQuote* responsible to adapt `range`.

typecaster
The object responsible for casting.

array_typecaster
The object responsible to cast arrays, if available, else `None`.

8.2.5 UUID data type

New in version 2.0.9.

Changed in version 2.0.13: added UUID array support.

```
>>> psycopg2.extras.register_uuid()
<psycopg2._psycopg.type object at 0x...>

>>> # Python UUID can be used in SQL queries
>>> import uuid
>>> my_uuid = uuid.UUID('{12345678-1234-5678-1234-567812345678}')
>>> psycopg2.extensions.adapt(my_uuid).getquoted()
"'12345678-1234-5678-1234-567812345678'::uuid"

>>> # PostgreSQL UUID are transformed into Python UUID objects.
>>> cur.execute("SELECT 'a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11'::uuid")
>>> cur.fetchone()[0]
UUID('a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11')
```

psycopg2.extras.**register_uuid**(*oids=None*, *conn_or_curs=None*)
Create the UUID type and an uuid.UUID adapter.

Parameters

- **oids** – oid for the PostgreSQL `uuid` type, or 2-items sequence with oids of the type and the array. If not specified, use PostgreSQL standard oids.
- **conn_or_curs** – where to register the typecaster. If not specified, register it globally.

class psycopg2.extras.**UUID_adapter**(*uuid*)
Adapt Python's uuid.UUID type to PostgreSQL's uuid.

8.2.6 Networking data types

By default Psycopg casts the PostgreSQL networking data types (`inet`, `cidr`, `macaddr`) into ordinary strings; array of such types are converted into lists of strings.

Changed in version 2.7: in previous version array of networking types were not treated as arrays.

psycopg2.extras.**register_ipaddress**(*conn_or_curs=None*)
Register conversion support between `ipaddress` objects and network types.

Parameters **conn_or_curs** – the scope where to register the type casters. If `None` register them globally.

After the function is called, PostgreSQL `inet` values will be converted into `IPv4Interface` or `IPv6Interface` objects, `cidr` values into into `IPv4Network` or `IPv6Network`.

psycopg2.extras.**register_inet**(*oid=None*, *conn_or_curs=None*)
Create the INET type and an Inet adapter.

Parameters

- **oid** – oid for the PostgreSQL `inet` type, or 2-items sequence with oids of the type and the array. If not specified, use PostgreSQL standard oids.
- **conn_or_curs** – where to register the typecaster. If not specified, register it globally.

Deprecated since version 2.7: this function will not receive further development and may disappear in future versions.

```
>>> psycopg2.extras.register_inet()
<psycopg2._psycopg.type object at 0x...>
```

(continues on next page)

(continued from previous page)

```
>>> cur.mogrify("SELECT %s", (Inet('127.0.0.1/32'),))
"SELECT E'127.0.0.1/32'::inet"

>>> cur.execute("SELECT '192.168.0.1/24'::inet")
>>> cur.fetchone()[0].addr
'192.168.0.1/24'
```

class psycopg2.extras.**Inet**(*addr*)

Wrap a string to allow for correct SQL-quoting of inet values.

Note that this adapter does NOT check the passed value to make sure it really is an inet-compatible address but DOES call adapt() on it to make sure it is impossible to execute an SQL-injection by passing an evil value to the initializer.

Deprecated since version 2.7: this object will not receive further development and may disappear in future versions.

8.3 Fast execution helpers

The current implementation of *executemany()* is (using an extremely charitable understatement) not particularly performing. These functions can be used to speed up the repeated execution of a statement againts a set of parameters. By reducing the number of server roundtrips the performance can be orders of magnitude better than using executemany().

psycopg2.extras.**execute_batch**(*cur*, *sql*, *argslist*, *page_size=100*)

Execute groups of statements in fewer server roundtrips.

Execute *sql* several times, against all parameters set (sequences or mappings) found in *argslist*.

The function is semantically similar to
cur.executemany(*sql*, *argslist*)
but has a different implementation: Psycopg will join the statements into fewer multi-statement commands, each one containing at most *page_size* statements, resulting in a reduced number of server roundtrips.

After the execution of the function the *cursor.rowcount* property will **not** contain a total result.

New in version 2.7.

Note: execute_batch() can be also used in conjunction with PostgreSQL prepared statements using PREPARE, EXECUTE, DEALLOCATE. Instead of executing:

```
execute_batch(cur,
    "big and complex SQL with %s %s params",
    params_list)
```

it is possible to execute something like:

```
cur.execute("PREPARE stmt AS big and complex SQL with $1 $2 params")
execute_batch(cur, "EXECUTE stmt (%s, %s)", params_list)
cur.execute("DEALLOCATE stmt")
```

which may bring further performance benefits: if the operation to perform is complex, every single execution will be faster as the query plan is already cached; furthermore the amount of data to send on the server will be lesser (one EXECUTE per param set instead of the whole, likely longer, statement).

---

psycopg2.extras.**execute_values**(*cur*, *sql*, *argslist*, *template=None*, *page_size=100*)
Execute a statement using VALUES with a sequence of parameters.

Parameters

- **cur** – the cursor to use to execute the query.
- **sql** – the query to execute. It must contain a single %s placeholder, which will be replaced by a VALUES list. Example: "INSERT INTO mytable (id, f1, f2) VALUES %s".
- **argslist** – sequence of sequences or dictionaries with the arguments to send to the query. The type and content must be consistent with *template*.
- **template** – the snippet to merge to every item in *argslist* to compose the query.
  - If the *argslist* items are sequences it should contain positional placeholders (e.g. "(%s, %s, %s)", or "(%s, %s, 42)" if there are constants value...).
  - If the *argslist* items are mappings it should contain named placeholders (e.g. "(%(id)s, %(f1)s, 42)").

  If not specified, assume the arguments are sequence and use a simple positional template (i.e. (%s, %s, ...)), with the number of placeholders sniffed by the first element in *argslist*.
- **page_size** – maximum number of *argslist* items to include in every statement. If there are more items the function will execute more than one statement.

After the execution of the function the *cursor.rowcount* property will **not** contain a total result.

While INSERT is an obvious candidate for this function it is possible to use it with other statements, for example:

```
>>> cur.execute(
... "create table test (id int primary key, v1 int, v2 int)")

>>> execute_values(cur,
... "INSERT INTO test (id, v1, v2) VALUES %s",
... [(1, 2, 3), (4, 5, 6), (7, 8, 9)])

>>> execute_values(cur,
... """UPDATE test SET v1 = data.v1 FROM (VALUES %s) AS data (id, v1)
... WHERE test.id = data.id""",
... [(1, 20), (4, 50)])

>>> cur.execute("select * from test order by id")
>>> cur.fetchall()
[(1, 20, 3), (4, 50, 6), (7, 8, 9)])
```

New in version 2.7.

8.4 Fractional time zones

psycopg2.extras.**register_tstz_w_secs**(*oids=None*, *conn_or_curs=None*)
The function used to register an alternate type caster for TIMESTAMP WITH TIME ZONE to deal with historical time zones with seconds in the UTC offset.

These are now correctly handled by the default type caster, so currently the function doesn't do anything.

New in version 2.0.9.

Changed in version 2.2.2: function is no-op: see *Time zones handling*.

8.5 Coroutine support

psycopg2.extras.**wait_select**(*conn*)
Wait until a connection or cursor has data available.

The function is an example of a wait callback to be registered with `set_wait_callback()`. This function uses `select()` to wait for data available.

Changed in version 2.6.2: allow to cancel a query using `Ctrl-C`, see *the FAQ* for an example.

CHAPTER

NINE

PSYCOPG2.SQL – SQL STRING COMPOSITION

New in version 2.7.

The module contains objects and functions useful to generate SQL dynamically, in a convenient and safe way. SQL identifiers (e.g. names of tables and fields) cannot be passed to the *execute()* method like query arguments:

```
# This will not work
table_name = 'my_table'
cur.execute("insert into %s values (%s, %s)", [table_name, 10, 20])
```

The SQL query should be composed before the arguments are merged, for instance:

```
# This works, but it is not optimal
table_name = 'my_table'
cur.execute(
    "insert into %s values (%%s, %%s)" % table_name,
    [10, 20])
```

This sort of works, but it is an accident waiting to happen: the table name may be an invalid SQL literal and need quoting; even more serious is the security problem in case the table name comes from an untrusted source. The name should be escaped using *quote_ident()*:

```
# This works, but it is not optimal
table_name = 'my_table'
cur.execute(
    "insert into %s values (%%s, %%s)" % ext.quote_ident(table_name),
    [10, 20])
```

This is now safe, but it somewhat ad-hoc. In case, for some reason, it is necessary to include a value in the query string (as opposite as in a value) the merging rule is still different (*adapt()* should be used...). It is also still relatively dangerous: if `quote_ident()` is forgotten somewhere, the program will usually work, but will eventually crash in the presence of a table or field name with containing characters to escape, or will present a potentially exploitable weakness.

The objects exposed by the `psycopg2.sql` module allow generating SQL statements on the fly, separating clearly the variable parts of the statement from the query parameters:

```
from psycopg2 import sql

cur.execute(
    sql.SQL("insert into {} values (%s, %s)")
        .format(sql.Identifier('my_table')),
    [10, 20])
```

The objects exposed by the `sql` module can be used to compose a query as a Python string (using the *as_string()* method) or passed directly to cursor methods such as *execute()*, *executemany()*, *copy_expert()*.

class psycopg2.sql.**Composable**(*wrapped*)
Abstract base class for objects that can be used to compose an SQL string.

`Composable` objects can be passed directly to *execute()*, *executemany()*, *copy_expert()* in place of the query string.

`Composable` objects can be joined using the + operator: the result will be a *Composed* instance containing the objects joined. The operator * is also supported with an integer argument: the result is a `Composed` instance containing the left argument repeated as many times as requested.

as_string(*context*)
Return the string value of the object.

Parameters **context** (*connection* or *cursor*) – the context to evaluate the string into.

The method is automatically invoked by *execute()*, *executemany()*, *copy_expert()* if a `Composable` is passed instead of the query string.

class psycopg2.sql.**SQL**(*string*)
A *Composable* representing a snippet of SQL statement.

`SQL` exposes *join()* and *format()* methods useful to create a template where to merge variable parts of a query (for instance field or table names).

The *string* doesn't undergo any form of escaping, so it is not suitable to represent variable identifiers or values: you should only use it to pass constant strings representing templates or snippets of SQL statements; use other objects such as *Identifier* or *Literal* to represent variable parts.

Example:

```
>>> query = sql.SQL("select {0} from {1}").format(
...     sql.SQL(', ').join([sql.Identifier('foo'), sql.Identifier('bar')]),
...     sql.Identifier('table'))
>>> print(query.as_string(conn))
select "foo", "bar" from "table"
```

string
The string wrapped by the `SQL` object.

format(**args*, ***kwargs*)
Merge *Composable* objects into a template.

Parameters

- **args** (`Composable`) – parameters to replace to numbered (`{0}`, `{1}`) or auto-numbered (`{}`) placeholders
- **kwargs** (`Composable`) – parameters to replace to named (`{name}`) placeholders

Returns the union of the `SQL` string with placeholders replaced

Return type *Composed*

The method is similar to the Python `str.format()` method: the string template supports auto-numbered (`{}`, only available from Python 2.7), numbered (`{0}`, `{1}...`), and named placeholders (`{name}`), with positional arguments replacing the numbered placeholders and keywords replacing the named ones. However placeholder modifiers (`{0!r}`, `{0:<10}`) are not supported. Only `Composable` objects can be passed to the template.

Example:

```
>>> print(sql.SQL("select * from {} where {} = %s")
...     .format(sql.Identifier('people'), sql.Identifier('id'))
...     .as_string(conn))
select * from "people" where "id" = %s

>>> print(sql.SQL("select * from {tbl} where {pkey} = %s")
...     .format(tbl=sql.Identifier('people'), pkey=sql.Identifier('id'))
...     .as_string(conn))
select * from "people" where "id" = %s
```

join(*seq*)
Join a sequence of *Composable*.

Parameters **seq** (iterable of `Composable`) – the elements to join.

Use the `SQL` object's *string* to separate the elements in *seq*. Note that *Composed* objects are iterable too, so they can be used as argument for this method.

Example:

```
>>> snip = sql.SQL(', ').join(
...     sql.Identifier(n) for n in ['foo', 'bar', 'baz'])
>>> print(snip.as_string(conn))
"foo", "bar", "baz"
```

class psycopg2.sql.**Identifier**(*string*)
A *Composable* representing an SQL identifer.

Identifiers usually represent names of database objects, such as tables or fields. PostgreSQL identifiers follow different rules than SQL string literals for escaping (e.g. they use double quotes instead of single).

Example:

```
>>> t1 = sql.Identifier("foo")
>>> t2 = sql.Identifier("ba'r")
>>> t3 = sql.Identifier('ba"z')
>>> print(sql.SQL(', ').join([t1, t2, t3]).as_string(conn))
"foo", "ba'r", "ba""z"
```

string
The string wrapped by the *Identifier*.

class psycopg2.sql.**Literal**(*wrapped*)
A *Composable* representing an SQL value to include in a query.

Usually you will want to include placeholders in the query and pass values as *execute()* arguments. If however you really really need to include a literal value in the query you can use this object.

The string returned by `as_string()` follows the normal *adaptation rules* for Python objects.

Example:

```
>>> s1 = sql.Literal("foo")
>>> s2 = sql.Literal("ba'r")
>>> s3 = sql.Literal(42)
>>> print(sql.SQL(', ').join([s1, s2, s3]).as_string(conn))
'foo', 'ba''r', 42
```

wrapped
The object wrapped by the `Literal`.

class psycopg2.sql.**Placeholder**(*name=None*)

A *Composable* representing a placeholder for query parameters.

If the name is specified, generate a named placeholder (e.g. `%(name)s`), otherwise generate a positional placeholder (e.g. `%s`).

The object is useful to generate SQL queries with a variable number of arguments.

Examples:

```
>>> names = ['foo', 'bar', 'baz']

>>> q1 = sql.SQL("insert into table ({}) values ({})").format(
...     sql.SQL(', ').join(map(sql.Identifier, names)),
...     sql.SQL(', ').join(sql.Placeholder() * len(names)))
>>> print(q1.as_string(conn))
insert into table ("foo", "bar", "baz") values (%s, %s, %s)

>>> q2 = sql.SQL("insert into table ({}) values ({})").format(
...     sql.SQL(', ').join(map(sql.Identifier, names)),
...     sql.SQL(', ').join(map(sql.Placeholder, names)))
>>> print(q2.as_string(conn))
insert into table ("foo", "bar", "baz") values (%(foo)s, %(bar)s, %(baz)s)
```

name

The name of the `Placeholder`.

class psycopg2.sql.**Composed**(*seq*)

A *Composable* object made of a sequence of `Composable`.

The object is usually created using `Composable` operators and methods. However it is possible to create a `Composed` directly specifying a sequence of `Composable` as arguments.

Example:

```
>>> comp = sql.Composed(
...     [sql.SQL("insert into "), sql.Identifier("table")])
>>> print(comp.as_string(conn))
insert into "table"
```

`Composed` objects are iterable (so they can be used in *SQL.join* for instance).

seq

The list of the content of the `Composed`.

join(*joiner*)

Return a new `Composed` interposing the *joiner* with the `Composed` items.

The *joiner* must be a *SQL* or a string which will be interpreted as an *SQL*.

Example:

```
>>> fields = sql.Identifier('foo') + sql.Identifier('bar')  # a Composed
>>> print(fields.join(', ').as_string(conn))
"foo", "bar"
```

CHAPTER

TEN

PSYCOPG2.TZ — TZINFO IMPLEMENTATIONS FOR PSYCOPG 2

This module holds two different tzinfo implementations that can be used as the `tzinfo` argument to `datetime` constructors, directly passed to Psycopg functions or used to set the *`cursor.tzinfo_factory`* attribute in cursors.

class psycopg2.tz.**FixedOffsetTimezone**(*offset=None*, *name=None*)

Fixed offset in minutes east from UTC.

This is exactly the implementation found in Python 2.3.x documentation, with a small change to the `__init__()` method to allow for pickling and a default name in the form `sHH:MM` (`s` is the sign.).

The implementation also caches instances. During creation, if a FixedOffsetTimezone instance has previously been created with the same offset and name that instance will be returned. This saves memory and improves comparability.

class psycopg2.tz.**LocalTimezone**

Platform idea of local timezone.

This is the exact implementation from the Python 2.3 documentation.

CHAPTER

ELEVEN

PSYCOPG2.POOL – CONNECTIONS POOLING

Creating new PostgreSQL connections can be an expensive operation. This module offers a few pure Python classes implementing simple connection pooling directly in the client application.

class psycopg2.pool.**AbstractConnectionPool**(*minconn*, *maxconn*, **args*, ***kwargs*)
Base class implementing generic key-based pooling code.

New *minconn* connections are created automatically. The pool will support a maximum of about *maxconn* connections. **args* and ***kwargs* are passed to the *connect()* function.

The following methods are expected to be implemented by subclasses:

getconn(*key=None*)
Get a free connection from the pool.

The *key* parameter is optional: if used, the connection will be associated to the key and calling `getconn()` with the same key again will return the same connection.

putconn(*conn*, *key=None*, *close=False*)
Put away a connection.

If *close* is `True`, discard the connection from the pool. *key* should be used consistently with *getconn()*.

closeall()
Close all the connections handled by the pool.

Note that all the connections are closed, including ones eventually in use by the application.

The following classes are *AbstractConnectionPool* subclasses ready to be used.

class psycopg2.pool.**SimpleConnectionPool**(*minconn*, *maxconn*, **args*, ***kwargs*)
A connection pool that can't be shared across different threads.

Note: This pool class is useful only for single-threaded applications.

class psycopg2.pool.**ThreadedConnectionPool**(*minconn*, *maxconn*, **args*, ***kwargs*)
A connection pool that works with the threading module.

Note: This pool class can be safely used in multi-threaded applications.

class psycopg2.pool.**PersistentConnectionPool**(*minconn*, *maxconn*, **args*, ***kwargs*)
A pool that assigns persistent connections to different threads.

Note that this connection pool generates by itself the required keys using the current thread id. This means that until a thread puts away a connection it will always get the same connection object by successive `getconn()` calls. This also means that a thread can't use more than one single connection from the pool.

Note: This pool class is mostly designed to interact with Zope and probably not useful in generic applications.

CHAPTER

TWELVE

PSYCOPG2.ERRORCODES – ERROR CODES DEFINED BY POSTGRESQL

New in version 2.0.6.

This module contains symbolic names for all PostgreSQL error codes and error classes codes. Subclasses of *Error* make the PostgreSQL error code available in the *pgcode* attribute.

From PostgreSQL documentation:

> All messages emitted by the PostgreSQL server are assigned five-character error codes that follow the SQL standard's conventions for SQLSTATE codes. Applications that need to know which error condition has occurred should usually test the error code, rather than looking at the textual error message. The error codes are less likely to change across PostgreSQL releases, and also are not subject to change due to localization of error messages. Note that some, but not all, of the error codes produced by PostgreSQL are defined by the SQL standard; some additional error codes for conditions not defined by the standard have been invented or borrowed from other databases.
>
> According to the standard, the first two characters of an error code denote a class of errors, while the last three characters indicate a specific condition within that class. Thus, an application that does not recognize the specific error code can still be able to infer what to do from the error class.

See also:

PostgreSQL Error Codes table

An example of the available constants defined in the module:

```
>>> errorcodes.CLASS_SYNTAX_ERROR_OR_ACCESS_RULE_VIOLATION
'42'
>>> errorcodes.UNDEFINED_TABLE
'42P01'
```

Constants representing all the error values defined by PostgreSQL versions between 8.1 and 10 are included in the module.

psycopg2.errorcodes.**lookup**(*code*)

Lookup an error code or class code and return its symbolic name.

Raise KeyError if the code is not found.

```
>>> try:
...     cur.execute("SELECT ouch FROM aargh;")
... except Exception, e:
...     pass
...
>>> errorcodes.lookup(e.pgcode[:2])
```

(continues on next page)

(continued from previous page)

```
'CLASS_SYNTAX_ERROR_OR_ACCESS_RULE_VIOLATION'
>>> errorcodes.lookup(e.pgcode)
'UNDEFINED_TABLE'
```

New in version 2.0.14.

CHAPTER

THIRTEEN

FREQUENTLY ASKED QUESTIONS

Here are a few gotchas you may encounter using *psycopg2*. Feel free to suggest new entries!

13.1 Problems with transactions handling

Why does psycopg2 leave database sessions "idle in transaction"? Psycopg normally starts a new transaction the first time a query is executed, e.g. calling *cursor.execute()*, even if the command is a SELECT. The transaction is not closed until an explicit *commit()* or *rollback()*.

If you are writing a long-living program, you should probably make sure to call one of the transaction closing methods before leaving the connection unused for a long time (which may also be a few seconds, depending on the concurrency level in your database). Alternatively you can use a connection in *autocommit* mode to avoid a new transaction to be started at the first command.

I receive the error *current transaction is aborted, commands ignored until end of transaction block* and can't do anything else! There was a problem *in the previous* command to the database, which resulted in an error. The database will not recover automatically from this condition: you must run a *rollback()* before sending new commands to the session (if this seems too harsh, remember that PostgreSQL supports nested transactions using the SAVEPOINT command).

Why do I get the error *current transaction is aborted, commands ignored until end of transaction block* when I use multiproces Psycopg's connections can't be shared across processes (but are thread safe). If you are forking the Python process make sure to create a new connection in each forked child. See *Thread and process safety* for further informations.

13.2 Problems with type conversions

Why does cursor.execute() raise the exception *can't adapt*? Psycopg converts Python objects in a SQL string representation by looking at the object class. The exception is raised when you are trying to pass as query parameter an object for which there is no adapter registered for its class. See *Adapting new Python types to SQL syntax* for informations.

I can't pass an integer or a float parameter to my query: it says *a number is required*, but *it is* a number! In your query string, you always have to use %s placeholders, even when passing a number. All Python objects are converted by Psycopg in their SQL representation, so they get passed to the query as strings. See *Passing parameters to SQL queries*.

```
>>> cur.execute("INSERT INTO numbers VALUES (%d)", (42,)) # WRONG
>>> cur.execute("INSERT INTO numbers VALUES (%s)", (42,)) # correct
```

I try to execute a query but it fails with the error *not all arguments converted during string formatting* (or *object does not support*
Psycopg always require positional arguments to be passed as a sequence, even when the query takes a single parameter. And remember that to make a single item tuple in Python you need a comma! See *Passing parameters to SQL queries*.

```
>>> cur.execute("INSERT INTO foo VALUES (%s)", "bar")    # WRONG
>>> cur.execute("INSERT INTO foo VALUES (%s)", ("bar"))  # WRONG
>>> cur.execute("INSERT INTO foo VALUES (%s)", ("bar",)) # correct
>>> cur.execute("INSERT INTO foo VALUES (%s)", ["bar"])  # correct
```

My database is Unicode, but I receive all the strings as UTF-8 `str`. Can I receive `unicode` objects instead?
The following magic formula will do the trick:

```
psycopg2.extensions.register_type(psycopg2.extensions.UNICODE)
psycopg2.extensions.register_type(psycopg2.extensions.UNICODEARRAY)
```

See *Unicode handling* for the gory details.

Psycopg converts `decimal/numeric` database types into Python `Decimal` objects. Can I have `float` instead?
You can register a customized adapter for PostgreSQL decimal type:

```
DEC2FLOAT = psycopg2.extensions.new_type(
    psycopg2.extensions.DECIMAL.values,
    'DEC2FLOAT',
    lambda value, curs: float(value) if value is not None else None)
psycopg2.extensions.register_type(DEC2FLOAT)
```

See *Type casting of SQL types into Python objects* to read the relevant documentation. If you find `psycopg2.extensions.DECIMAL` not available, use `psycopg2._psycopg.DECIMAL` instead.

Psycopg automatically converts PostgreSQL `json` data into Python objects. How can I receive strings instead?
The easiest way to avoid JSON parsing is to register a no-op function with *`register_default_json()`*:

```
psycopg2.extras.register_default_json(loads=lambda x: x)
```

See *JSON adaptation* for further details.

Psycopg converts `json` values into Python objects but `jsonb` values are returned as strings. Can `jsonb` be converted automat
Automatic conversion of `jsonb` values is supported from Psycopg release 2.5.4. For previous versions you can register the `json` typecaster on the `jsonb` oids (which are known and not suppsed to change in future PostgreSQL versions):

```
psycopg2.extras.register_json(oid=3802, array_oid=3807, globally=True)
```

See *JSON adaptation* for further details.

How can I pass field/table names to a query? The arguments in the *`execute()`* methods can only represent data to pass to the query: they cannot represent a table or field name:

```
# This doesn't work
cur.execute("insert into %s values (%s)", ["my_table", 42])
```

If you want to build a query dynamically you can use the objects exposed by the *`psycopg2.sql`* module:

```
cur.execute(
    sql.SQL("insert into %s values (%%s)") % [sql.Identifier("my_table")],
    [42])
```

Transferring binary data from PostgreSQL 9.0 doesn't work. PostgreSQL 9.0 uses by default the "hex" format to transfer `bytea` data: the format can't be parsed by the libpq 8.4 and earlier. The problem is solved in Psycopg 2.4.1, that uses its own parser for the `bytea` format. For previous Psycopg releases, three options to solve the problem are:

- set the bytea_output parameter to `escape` in the server;
- execute the database command `SET bytea_output TO escape;` in the session before reading binary data;
- upgrade the libpq library on the client to at least 9.0.

Arrays of *TYPE* are not casted to list. Arrays are only casted to list when their oid is known, and an array typecaster is registered for them. If there is no typecaster, the array is returned unparsed from PostgreSQL (e.g. `{a,b,c}`). It is easy to create a generic arrays typecaster, returning a list of array: an example is provided in the *`new_array_type()`* documentation.

13.3 Best practices

When should I save and re-use a cursor as opposed to creating a new one as needed? Cursors are lightweight objects and creating lots of them should not pose any kind of problem. But note that cursors used to fetch result sets will cache the data and use memory in proportion to the result set size. Our suggestion is to almost always create a new cursor and dispose old ones as soon as the data is not required anymore (call *`close()`* on them.) The only exception are tight loops where one usually use the same cursor for a whole bunch of `INSERT`s or `UPDATE`s.

When should I save and re-use a connection as opposed to creating a new one as needed? Creating a connection can be slow (think of SSL over TCP) so the best practice is to create a single connection and keep it open as long as required. It is also good practice to rollback or commit frequently (even after a single `SELECT` statement) to make sure the backend is never left "idle in transaction". See also *`psycopg2.pool`* for lightweight connection pooling.

What are the advantages or disadvantages of using named cursors? The only disadvantages is that they use up resources on the server and that there is a little overhead because a at least two queries (one to create the cursor and one to fetch the initial result set) are issued to the backend. The advantage is that data is fetched one chunk at a time: using small *`fetchmany()`* values it is possible to use very little memory on the client and to skip or discard parts of the result set.

How do I interrupt a long-running query in an interactive shell? Normally the interactive shell becomes unresponsive to `Ctrl-C` when running a query. Using a connection in green mode allows Python to receive and handle the interrupt, although it may leave the connection broken, if the async callback doesn't handle the `KeyboardInterrupt` correctly.

Starting from psycopg 2.6.2, the *`wait_select`* callback can handle a `Ctrl-C` correctly. For previous versions, you can use this implementation.

```
>>> psycopg2.extensions.set_wait_callback(psycopg2.extras.wait_select)
>>> cnn = psycopg2.connect('')
>>> cur = cnn.cursor()
>>> cur.execute("select pg_sleep(10)")
^C
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  QueryCanceledError: canceling statement due to user request

>>> cnn.rollback()
>>> # You can use the connection and cursor again from here
```

13.4 Problems compiling and deploying psycopg2

I can't compile `psycopg2`: the compiler says *error: Python.h: No such file or directory*. What am I missing?
You need to install a Python development package: it is usually called `python-dev`.

I can't compile `psycopg2`: the compiler says *error: libpq-fe.h: No such file or directory*. What am I missing?
You need to install the development version of the libpq: the package is usually called `libpq-dev`.

`psycopg2` raises `ImportError` with message *_psycopg.so: undefined symbol: lo_truncate* when imported.
This means that Psycopg was compiled with `lo_truncate()` support (*i.e.* the libpq used at compile time was version >= 8.3) but at runtime an older libpq dynamic library is found.

Fast-forward several years, if the message reports *undefined symbol: lo_truncate64* it means that Psycopg was built with large objects 64 bits API support (*i.e.* the libpq used at compile time was at least 9.3) but at runtime an older libpq dynamic library is found.

You can use:

```
$ ldd /path/to/packages/psycopg2/_psycopg.so | grep libpq
```

to find what is the libpq dynamic library used at runtime.

You can avoid the problem by using the same version of the **`pg_config`** at install time and the libpq at runtime.

Psycopg raises *ImportError: cannot import name tz* on import in mod_wsgi / ASP, but it works fine otherwise.
If `psycopg2` is installed in an egg (e.g. because installed by **`easy_install`**), the user running the program may be unable to write in the eggs cache. Set the env variable `PYTHON_EGG_CACHE` to a writable directory. With modwsgi you can use the WSGIPythonEggs directive.

CHAPTER

FOURTEEN

RELEASE NOTES

14.1 Current release

14.1.1 What's new in psycopg 2.7.4

- Moving away from installing the wheel package by default. Packages installed from wheel raise a warning on import. Added package `psycopg2-binary` to install from wheel instead (ticket #543).
- Convert fields names into valid Python identifiers in *`NamedTupleCursor`* (ticket #211).
- Fixed Solaris 10 support (ticket #532).
- *`cursor.mogrify()`* can be called on closed cursors (ticket #579).
- Fixed setting session characteristics in corner cases on autocommit connections (ticket #580).
- Fixed *`MinTimeLoggingCursor`* on Python 3 (ticket #609).
- Fixed parsing of array of points as floats (ticket #613).
- Fixed *`__libpq_version__`* building with libpq >= 10.1 (ticket 632).
- Fixed *`rowcount`* after *`executemany()`* with `RETURNING` statements (ticket 633).
- Fixed compatibility problem with pypy3 (ticket #649).
- Wheel packages compiled against PostgreSQL 10.1 libpq and OpenSSL 1.0.2n.
- Wheel packages for Python 2.6 no more available (support dropped from wheel building infrastructure).

14.1.2 What's new in psycopg 2.7.3.2

- Wheel package compiled against PostgreSQL 10.0 libpq and OpenSSL 1.0.2l (tickets #601, #602).

14.1.3 What's new in psycopg 2.7.3.1

- Dropped libresolv from wheel package to avoid incompatibility with glibc 2.26 (wheels ticket #2).

14.1.4 What's new in psycopg 2.7.3

- Restored default `timestamptz[]` typecasting to Python `datetime`. Regression introduced in Psycopg 2.7.2 (ticket #578).

14.1.5 What's new in psycopg 2.7.2

- Fixed inconsistent state in externally closed connections (tickets #263, #311, #443). Was fixed in 2.6.2 but not included in 2.7 by mistake.
- Fixed Python exceptions propagation in green callback (ticket #410).
- Don't display the password in *connection.dsn* when the connection string is specified as an URI (ticket #528).
- Return objects with timezone parsing "infinity" `timestamptz` (ticket #536).
- Dropped dependency on VC9 runtime on Windows binary packages (ticket #541).
- Fixed segfault in *lobject()* when *mode*=`None` (ticket #544).
- Fixed *lobject()* keyword argument *lobject_factory* (ticket #545).
- Fixed *consume_stream()* *keepalive_interval* argument (ticket #547).
- Maybe fixed random import error on Python 3.6 in multiprocess environment (ticket #550).
- Fixed random `SystemError` upon receiving abort signal (ticket #551).
- Accept *Composable* objects in *start_replication_expert()* (ticket 554).
- Parse intervals returned as microseconds from Redshift (ticket #558).
- Added *Json* `prepare()` method to consider connection params when adapting (ticket #562).
- *errorcodes* map updated to PostgreSQL 10 beta 1.

14.1.6 What's new in psycopg 2.7.1

- Ignore `None` arguments passed to *connect()* and *make_dsn()* (ticket #517).
- OpenSSL upgraded from major version 0.9.8 to 1.0.2 in the Linux wheel packages (ticket #518).
- Fixed build with libpq versions < 9.3 (ticket #520).

14.2 What's new in psycopg 2.7

New features:

- Added *sql* module to generate SQL dynamically (ticket #308).
- Added *Replication protocol support* (ticket #322). Main authors are Oleksandr Shulgin and Craig Ringer, who deserve a huge thank you.
- Added *parse_dsn()* and *make_dsn()* functions (tickets #321, #363). *connect()* now can take both *dsn* and keyword arguments, merging them together.
- Added *__libpq_version__* and *libpq_version()* to inspect the version of the `libpq` library the module was compiled/loaded with (tickets #35, #323).
- The attributes *notices* and *notifies* can be customized replacing them with any object exposing an `append()` method (ticket #326).
- Adapt network types to `ipaddress` objects when available. When not enabled, convert arrays of network types to lists by default. The old `Inet` adapter is deprecated (tickets #317, #343, #387).
- Added *quote_ident()* function (ticket #359).

- Added *get_dsn_parameters()* connection method (ticket #364).
- *callproc()* now accepts a dictionary of parameters (ticket #381).
- Give precedence to `__conform__()` over superclasses to choose an object adapter (ticket #456).
- Using Python C API decoding functions and codecs caching for faster unicode encoding/decoding (ticket #473).
- *executemany()* slowness addressed by *execute_batch()* and *execute_values()* (ticket #491).
- Added `async_` as an alias for `async` to support Python 3.7 where `async` will become a keyword (ticket #495).
- Unless in autocommit, do not use `default_transaction_*` settings to control the session characteristics as it may create problems with external connection pools such as pgbouncer; use `BEGIN` options instead (ticket #503).
- *isolation_level* is now writable and entirely separated from *autocommit*; added *readonly*, *deferrable* writable attributes.

Bug fixes:

- Throw an exception trying to pass `NULL` chars as parameters (ticket #420).
- Fixed error caused by missing decoding *LoggingConnection* (ticket #483).
- Fixed integer overflow in `interval` seconds (ticket #512).
- Make *Range* objects picklable (ticket #462).
- Fixed version parsing and building with PostgreSQL 10 (ticket #489).

Other changes:

- Dropped support for Python 2.5 and 3.1.
- Dropped support for client library older than PostgreSQL 9.1 (but older server versions are still supported).
- *isolation_level* doesn't read from the database but will return *ISOLATION_LEVEL_DEFAULT* if no value was set on the connection.
- Empty arrays no more converted into lists if they don't have a type attached (ticket #506)

14.2.1 What's new in psycopg 2.6.2

- Fixed inconsistent state in externally closed connections (tickets #263, #311, #443).
- Report the server response status on errors (such as ticket #281).
- Raise `NotSupportedError` on unhandled server response status (ticket #352).
- Allow overriding string adapter encoding with no connection (ticket #331).
- The *wait_select* callback allows interrupting a long-running query in an interactive shell using `Ctrl-C` (ticket #333).
- Fixed `PersistentConnectionPool` on Python 3 (ticket #348).
- Fixed segfault on `repr()` of an unitialized connection (ticket #361).
- Allow adapting bytes using *QuotedString* on Python 3 (ticket #365).
- Added support for setuptools/wheel (ticket #370).
- Fix build on Windows with Python 3.5, VS 2015 (ticket #380).
- Fixed `errorcodes.lookup` initialization thread-safety (ticket #382).

- Fixed `read()` exception propagation in copy_from (ticket #412).
- Fixed possible NULL TZ decref (ticket #424).
- *`errorcodes`* map updated to PostgreSQL 9.5.

14.2.2 What's new in psycopg 2.6.1

- Lists consisting of only `None` are escaped correctly (ticket #285).
- Fixed deadlock in multithread programs using OpenSSL (ticket #290).
- Correctly unlock the connection after error in flush (ticket #294).
- Fixed `MinTimeLoggingCursor.callproc()` (ticket #309).
- Added support for MSVC 2015 compiler (ticket #350).

14.3 What's new in psycopg 2.6

New features:

- Added support for large objects larger than 2GB. Many thanks to Blake Rouse and the MAAS Team for the feature development.
- Python `time` objects with a tzinfo specified and PostgreSQL `timetz` data are converted into each other (ticket #272).

Bug fixes:

- Json adapter's `str()` returns the adapted content instead of the `repr()` (ticket #191).

14.3.1 What's new in psycopg 2.5.5

- Named cursors used as context manager don't swallow the exception on exit (ticket #262).
- *`cursor.description`* can be pickled (ticket #265).
- Propagate read error messages in COPY FROM (ticket #270).
- PostgreSQL time 24:00 is converted to Python 00:00 (ticket #278).

14.3.2 What's new in psycopg 2.5.4

- Added `jsonb` support for PostgreSQL 9.4 (ticket #226).
- Fixed segfault if COPY statements are passed to *`execute()`* instead of using the proper methods (ticket #219).
- Force conversion of pool arguments to integer to avoid potentially unbounded pools (ticket #220).
- Cursors `WITH HOLD` don't begin a new transaction upon move/fetch/close (ticket #228).
- Cursors `WITH HOLD` can be used in autocommit (ticket #229).
- *`callproc()`* doesn't silently ignore an argument without a length.
- Fixed memory leak with large objects (ticket #256).
- Make sure the internal `_psycopg.so` module can be imported stand-alone (to allow modules juggling such as the one described in ticket #201).

14.3.3 What's new in psycopg 2.5.3

- Work around pip issue #1630 making installation via `pip -e git+url` impossible (ticket #18).
- Copy operations correctly set the *`cursor.rowcount`* attribute (ticket #180).
- It is now possible to call `get_transaction_status()` on closed connections.
- Fixed unsafe access to object names causing assertion failures in Python 3 debug builds (ticket #188).
- Mark the connection closed if found broken on `poll()` (from ticket #192 discussion)
- Fixed handling of dsn and closed attributes in connection subclasses failing to connect (from ticket #192 discussion).
- Added arbitrary but stable order to `Range` objects, thanks to Chris Withers (ticket #193).
- Avoid blocking async connections on connect (ticket #194). Thanks to Adam Petrovich for the bug report and diagnosis.
- Don't segfault using poorly defined cursor subclasses which forgot to call the superclass init (ticket #195).
- Mark the connection closed when a Socket connection is broken, as it happens for TCP connections instead (ticket #196).
- Fixed overflow opening a lobject with an oid not fitting in a signed int (ticket #203).
- Fixed handling of explicit default `cursor_factory=None` in *`connection.cursor()`* (ticket #210).
- Fixed possible segfault in named cursors creation.
- Fixed debug build on Windows, thanks to James Emerton.

14.3.4 What's new in psycopg 2.5.2

- Fixed segfault pickling the exception raised on connection error (ticket #170).
- Meaningful connection errors report a meaningful message, thanks to Alexey Borzenkov (ticket #173).
- Manually creating `lobject` with the wrong parameter doesn't segfault (ticket #187).

14.3.5 What's new in psycopg 2.5.1

- Fixed build on Solaris 10 and 11 where the round() function is already declared (ticket #146).
- Fixed comparison of `Range` with non-range objects (ticket #164). Thanks to Chris Withers for the patch.
- Fixed double-free on connection dealloc (ticket #166). Thanks to Gangadharan S.A. for the report and fix suggestion.

14.4 What's new in psycopg 2.5

New features:

- Added *JSON adaptation*.
- Added *support for PostgreSQL 9.2 range types*.
- *`connection`* and *`cursor`* objects can be used in `with` statements as context managers as specified by recent DB API 2.0 extension.

- Added *Diagnostics* object to get extended info from a database error. Many thanks to Matthew Woodcraft for the implementation (ticket #149).
- Added *connection.cursor_factory* attribute to customize the default object returned by *cursor()*.
- Added support for backward scrollable cursors. Thanks to Jon Nelson for the initial patch (ticket #108).
- Added a simple way to *customize casting of composite types* into Python objects other than namedtuples. Many thanks to Ronan Dunklau and Tobias Oberstein for the feature development.
- *connection.reset()* implemented using `DISCARD ALL` on server versions supporting it.

Bug fixes:

- Properly cleanup memory of broken connections (ticket #148).
- Fixed bad interaction of `setup.py` with other dependencies in Distribute projects on Python 3 (ticket #153).

Other changes:

- Added support for Python 3.3.
- Dropped support for Python 2.4. Please use Psycopg 2.4.x if you need it.
- *errorcodes* map updated to PostgreSQL 9.2.
- Dropped Zope adapter from source repository. ZPsycopgDA now has its own project at <http://github.com/psycopg/ZPsycopgDA>.

14.4.1 What's new in psycopg 2.4.6

- Fixed 'cursor()' arguments propagation in connection subclasses and overriding of the 'cursor_factory' argument. Thanks to Corry Haines for the report and the initial patch (ticket #105).
- Dropped GIL release during string adaptation around a function call invoking a Python API function, which could cause interpreter crash. Thanks to Manu Cupcic for the report (ticket #110).
- Close a green connection if there is an error in the callback. Maybe a harsh solution but it leaves the program responsive (ticket #113).
- 'register_hstore()', 'register_composite()', 'tpc_recover()' work with RealDictConnection and Cursor (ticket #114).
- Fixed broken pool for Zope and connections re-init across ZSQL methods in the same request (tickets #123, #125, #142).
- connect() raises an exception instead of swallowing keyword arguments when a connection string is specified as well (ticket #131).
- Discard any result produced by 'executemany()' (ticket #133).
- Fixed pickling of FixedOffsetTimezone objects (ticket #135).
- Release the GIL around PQgetResult calls after COPY (ticket #140).
- Fixed empty strings handling in composite caster (ticket #141).
- Fixed pickling of DictRow and RealDictRow objects.

14.4.2 What's new in psycopg 2.4.5

- The close() methods on connections and cursors don't raise exceptions if called on already closed objects.
- Fixed fetchmany() with no argument in cursor subclasses (ticket #84).

- Use lo_creat() instead of lo_create() when possible for better interaction with pgpool-II (ticket #88).
- Error and its subclasses are picklable, useful for multiprocessing interaction (ticket #90).
- Better efficiency and formatting of timezone offset objects thanks to Menno Smits (tickets #94, #95).
- Fixed 'rownumber' during iteration on cursor subclasses. Regression introduced in 2.4.4 (ticket #100).
- Added support for 'inet' arrays.
- Fixed 'commit()' concurrency problem (ticket #103).
- Codebase cleaned up using the GCC Python plugin's static analysis tool, which has revealed several unchecked return values, possible NULL dereferences, reference counting problems. Many thanks to David Malcolm for the useful tool and the assistance provided using it.

14.4.3 What's new in psycopg 2.4.4

- 'register_composite()' also works with the types implicitly defined after a table row, not only with the ones created by 'CREATE TYPE'.
- Values for the isolation level symbolic constants restored to what they were before release 2.4.2 to avoid breaking apps using the values instead of the constants.
- Named DictCursor/RealDictCursor honour itersize (ticket #80).
- Fixed rollback on error on Zope (ticket #73).
- Raise 'DatabaseError' instead of 'Error' with empty libpq errors, consistently with other disconnection-related errors: regression introduced in release 2.4.1 (ticket #82).

14.4.4 What's new in psycopg 2.4.3

- connect() supports all the keyword arguments supported by the database
- Added 'new_array_type()' function for easy creation of array typecasters.
- Added support for arrays of hstores and composite types (ticket #66).
- Fixed segfault in case of transaction started with connection lost (and possibly other events).
- Fixed adaptation of Decimal type in sub-interpreters, such as in certain mod_wsgi configurations (ticket #52).
- Rollback connections in transaction or in error before putting them back into a pool. Also discard broken connections (ticket #62).
- Lazy import of the slow uuid module, thanks to Marko Kreen.
- Fixed NamedTupleCursor.executemany() (ticket #65).
- Fixed –static-libpq setup option (ticket #64).
- Fixed interaction between RealDictCursor and named cursors (ticket #67).
- Dropped limit on the columns length in COPY operations (ticket #68).
- Fixed reference leak with arguments referenced more than once in queries (ticket #81).
- Fixed typecasting of arrays containing consecutive backslashes.
- 'errorcodes' map updated to PostgreSQL 9.1.

14.4.5 What's new in psycopg 2.4.2

- Added 'set_session()' method and 'autocommit' property to the connection. Added support for read-only sessions and, for PostgreSQL 9.1, for the "repeatable read" isolation level and the "deferrable" transaction property.
- Psycopg doesn't execute queries at connection time to find the default isolation level.
- Fixed bug with multithread code potentially causing loss of sync with the server communication or lock of the client (ticket #55).
- Don't fail import if mx.DateTime module can't be found, even if its support was built (ticket #53).
- Fixed escape for negative numbers prefixed by minus operator (ticket #57).
- Fixed refcount issue during copy. Reported and fixed by Dave Malcolm (ticket #58, Red Hat Bug 711095).
- Trying to execute concurrent operations on the same connection through concurrent green thread results in an error instead of a deadlock.
- Fixed detection of pg_config on Window. Report and fix, plus some long needed setup.py cleanup by Steve Lacy: thanks!

14.4.6 What's new in psycopg 2.4.1

- Use own parser for bytea output, not requiring anymore the libpq 9.0 to parse the hex format.
- Don't fail connection if the client encoding is a non-normalized variant. Issue reported by Peter Eisentraut.
- Correctly detect an empty query sent to the backend (ticket #46).
- Fixed a SystemError clobbering libpq errors raised without SQLSTATE. Bug vivisectioned by Eric Snow.
- Fixed interaction between NamedTuple and server-side cursors.
- Allow to specify –static-libpq on setup.py command line instead of just in 'setup.cfg'. Patch provided by Matthew Ryan (ticket #48).

14.5 What's new in psycopg 2.4

New features and changes:

- Added support for Python 3.1 and 3.2. The conversion has also brought several improvements:
  - Added 'b' and 't' mode to large objects: write can deal with both bytes strings and unicode; read can return either bytes strings or decoded unicode.
  - COPY sends Unicode data to files implementing 'io.TextIOBase'.
  - Improved PostgreSQL-Python encodings mapping.
  - Added a few missing encodings: EUC_CN, EUC_JIS_2004, ISO885910, ISO885916, LATIN10, SHIFT_JIS_2004.
  - Dropped repeated dictionary lookups with unicode query/parameters.
- Improvements to the named cursors:
  - More efficient iteration on named cursors, fetching 'itersize' records at time from the backend.
  - The named cursors name can be an invalid identifier.
- Improvements in data handling:

  - Added 'register_composite()' function to cast PostgreSQL composite types into Python tuples/namedtuples.
  - Adapt types 'bytearray' (from Python 2.6), 'memoryview' (from Python 2.7) and other objects implementing the "Revised Buffer Protocol" to 'bytea' data type.
  - The 'hstore' adapter can work even when the data type is not installed in the 'public' namespace.
  - Raise a clean exception instead of returning bad data when receiving bytea in 'hex' format and the client libpq can't parse them.
  - Empty lists correctly roundtrip Python -> PostgreSQL -> Python.
- Other changes:
  - 'cursor.description' is provided as named tuples if available.
  - The build script refuses to guess values if 'pg_config' is not found.
  - Connections and cursors are weakly referenceable.

Bug fixes:

- Fixed adaptation of None in composite types (ticket #26). Bug report by Karsten Hilbert.
- Fixed several reference leaks in less common code paths.
- Fixed segfault when a large object is closed and its connection no more available.
- Added missing icon to ZPsycopgDA package, not available in Zope 2.12.9 (ticket #30). Bug report and patch by Pumukel.
- Fixed conversion of negative infinity (ticket #40). Bug report and patch by Marti Raudsepp.

14.5.1 What's new in psycopg 2.3.2

- Fixed segfault with middleware not passing DateStyle to the client (ticket #24). Bug report and patch by Marti Raudsepp.

14.5.2 What's new in psycopg 2.3.1

- Fixed build problem on CentOS 5.5 x86_64 (ticket #23).

14.6 What's new in psycopg 2.3

psycopg 2.3 aims to expose some new features introduced in PostgreSQL 9.0.

Main new features:

- `dict` to `hstore` adapter and `hstore` to `dict` typecaster, using both 9.0 and pre-9.0 syntax.
- Two-phase commit protocol support as per DBAPI specification.
- Support for payload in notifications received from the backend.
- `namedtuple`-returning cursor.
- Query execution cancel.

Other features and changes:

- Dropped support for protocol 2: Psycopg 2.3 can only connect to PostgreSQL servers with version at least 7.4.

- Don't issue a query at every connection to detect the client encoding and to set the datestyle to ISO if it is already compatible with what expected.
- `mogrify()` now supports unicode queries.
- Subclasses of a type that can be adapted are adapted as the superclass.
- `errorcodes` knows a couple of new codes introduced in PostgreSQL 9.0.
- Dropped deprecated Psycopg "own quoting".
- Never issue a ROLLBACK on close/GC. This behaviour was introduced as a bug in release 2.2, but trying to send a command while being destroyed has been considered not safe.

Bug fixes:

- Fixed use of `PQfreemem` instead of `free` in binary typecaster.
- Fixed access to freed memory in `conn_get_isolation_level()`.
- Fixed crash during Decimal adaptation with a few 2.5.x Python versions (ticket #7).
- Fixed notices order (ticket #9).

14.6.1 What's new in psycopg 2.2.2

Bux fixes:

- the call to logging.basicConfig() in pool.py has been dropped: it was messing with some projects using logging (and a library should not initialize the logging system anyway.)
- psycopg now correctly handles time zones with seconds in the UTC offset. The old register_tstz_w_secs() function is deprecated and will raise a warning if called.
- Exceptions raised by the column iterator are propagated.
- Exceptions raised by executemany() iterators are propagated.

14.6.2 What's new in psycopg 2.2.1

Bux fixes:

- psycopg now builds again on MS Windows.

14.7 What's new in psycopg 2.2

This is the first release of the new 2.2 series, supporting not just one but two different ways of executing asynchronous queries, thanks to Jan and Daniele (with a little help from me and others, but they did 99% of the work so they deserve their names here in the news.)

psycopg now supports both classic select() loops and "green" coroutine libraries. It is all in the documentation, so just point your browser to doc/html/advanced.html.

Other new features:

- truncate() method for lobjects.
- COPY functions are now a little bit faster.
- All builtin PostgreSQL to Python typecasters are now available from the psycopg2.extensions module.

- Notifications from the backend are now available right after the execute() call (before client code needed to call isbusy() to ensure NOTIFY reception.)
- Better timezone support.
- Lots of documentation updates.

Bug fixes:

- Fixed some gc/refcounting problems.
- Fixed reference leak in NOTIFY reception.
- Fixed problem with PostgreSQL not casting string literals to the correct types in some situations: psycopg now add an explicit cast to dates, times and bytea representations.
- Fixed TimestampFromTicks() and TimeFromTicks() for seconds >= 59.5.
- Fixed spurious exception raised when calling C typecasters from Python ones.

14.7.1 What's new in psycopg 2.0.14

New features:

- Support for adapting tuples to PostgreSQL arrays is now enabled by default and does not require importing psycopg2.extensions anymore.
- "can't adapt" error message now includes full type information.
- Thank to Daniele Varrazzo (piro) psycopg2's source package now includes full documentation in HTML and plain text format.

Bug fixes:

- No loss of precision when using floats anymore.
- decimal.Decimal "nan" and "infinity" correctly converted to PostgreSQL numeric NaN values (note that PostgreSQL numeric type does not support infinity but just NaNs.)
- psycopg2.extensions now includes Binary.

It seems we're good citizens of the free software ecosystem and that big big big companies and people ranting on the pgsql-hackers mailing list we'll now not dislike us. *g* (See LICENSE file for the details.)

14.7.2 What's new in psycopg 2.0.13

New features:

- Support for UUID arrays.
- It is now possible to build psycopg linking to a static libpq library.

Bug fixes:

- Fixed a deadlock related to using the same connection with multiple cursors from different threads.
- Builds again with MSVC.

14.7.3 What's new in psycopg 2.0.12

New features:

- The connection object now has a reset() method that can be used to reset the connection to its default state.

Bug fixes:

- copy_to() and copy_from() now accept a much larger number of columns.
- Fixed PostgreSQL version detection.
- Fixed ZPsycopgDA version check.
- Fixed regression in ZPsycopgDA that made it behave wrongly when receiving serialization errors: now the query is re-issued as it should be by propagating the correct exception to Zope.
- Writing "large" large objects should now work.

14.7.4 What's new in psycopg 2.0.11

New features:

- DictRow and RealDictRow now use less memory. If you inherit on them remember to set __slots__ for your new attributes or be prepare to go back to old memory usage.

Bug fixes:

- Fixed exception in setup.py.
- More robust detection of PostgreSQL development versions.
- Fixed exception in RealDictCursor, introduced in 2.0.10.

14.7.5 What's new in psycopg 2.0.10

New features:

- A specialized type-caster that can parse time zones with seconds is now available. Note that after enabling it (see extras.py) "wrong" time zones will be parsed without raising an exception but the result will be rounded.
- DictCursor can be used as a named cursor.
- DictRow now implements more dict methods.
- The connection object now expose PostgreSQL server version as the .server_version attribute and the protocol version used as .protocol_version.
- The connection object has a .get_parameter_status() methods that can be used to obtain useful information from the server.

Bug fixes:

- None is now correctly always adapted to NULL.
- Two double memory free errors provoked by multithreading and garbage collection are now fixed.
- Fixed usage of internal Python code in the notice processor; this should fix segfaults when receiving a lot of notices in multithreaded programs.
- Should build again on MSVC and Solaris.
- Should build with development versions of PostgreSQL (ones with -devel version string.)

- Fixed some tests that failed even when psycopg was right.

14.7.6 What's new in psycopg 2.0.9

New features:

- "import psycopg2.extras" to get some support for handling times and timestamps with seconds in the time zone offset.
- DictCursors can now be used as named cursors.

Bug fixes:

- register_type() now accept an explicit None as its second parameter.
- psycopg2 should build again on MSVC and Solaris.

14.7.7 What's new in psycopg 2.0.9

New features:

- COPY TO/COPY FROM queries now can be of any size and psycopg will correctly quote separators.
- float values Inf and NaN are now correctly handled and can round-trip to the database.
- executemany() now return the numer of total INSERTed or UPDATEd rows. Note that, as it has always been, executemany() should not be used to execute multiple SELECT statements and while it will execute the statements without any problem, it will return the wrong value.
- copy_from() and copy_to() can now use quoted separators.
- "import psycopg2.extras" to get UUID support.

Bug fixes:

- register_type() now works on connection and cursor subclasses.
- fixed a memory leak when using lobjects.

14.7.8 What's new in psycopg 2.0.8

New features:

- The connection object now has a get_backend_pid() method that returns the current PostgreSQL connection backend process PID.
- The PostgreSQL large object API has been exposed through the Cursor.lobject() method.

Bug fixes:

- Some fixes to ZPsycopgDA have been merged from the Debian package.
- A memory leak was fixed in Cursor.executemany().
- A double free was fixed in pq_complete_error(), that caused crashes under some error conditions.

14.7.9 What's new in psycopg 2.0.7

Improved error handling:

- All instances of psycopg2.Error subclasses now have pgerror, pgcode and cursor attributes. They will be set to None if no value is available.
- Exception classes are now chosen based on the SQLSTATE value from the result. (#184)
- The commit() and rollback() methods now set the pgerror and pgcode attributes on exceptions. (#152)
- errors from commit() and rollback() are no longer considered fatal. (#194)
- If a disconnect is detected during execute(), an exception will be raised at that point rather than resulting in "ProgrammingError: no results to fetch" later on. (#186)

Better PostgreSQL compatibility:

- If the server uses standard_conforming_strings, perform appropriate quoting.
- BC dates are now handled if psycopg is compiled with mxDateTime support. If using datetime, an appropriate ValueError is raised. (#203)

Other bug fixes:

- If multiple sub-interpreters are in use, do not share the Decimal type between them. (#192)
- Buffer objects obtained from psycopg are now accepted by psycopg too, without segfaulting. (#209)
- A few small changes were made to improve DB-API compatibility. All the dbapi20 tests now pass.

Miscellaneous:

- The PSYCOPG_DISPLAY_SIZE option is now off by default. This means that display size will always be set to "None" in cursor.description. Calculating the display size was expensive, and infrequently used so this should improve performance.
- New QueryCanceledError and TransactionRollbackError exceptions have been added to the psycopg2.extensions module. They can be used to detect statement timeouts and deadlocks respectively.
- Cursor objects now have a "closed" attribute. (#164)
- If psycopg has been built with debug support, it is now necessary to set the PSYCOPG_DEBUG environment variable to turn on debug spew.

14.7.10 What's new in psycopg 2.0.6

Better support for PostgreSQL, Python and win32:

- full support for PostgreSQL 8.2, including NULLs in arrays
- support for almost all existing PostgreSQL encodings
- full list of PostgreSQL error codes available by importing the psycopg2.errorcodes module
- full support for Python 2.5 and 64 bit architectures
- better build support on win32 platform

Support for per-connection type-casters (used by ZPsycopgDA too, this fixes a long standing bug that made different connections use a random set of date/time type-casters instead of the configured one.)

Better management of times and dates both from Python and in Zope.

copy_to and copy_from now take an extra "columns" parameter.

Python tuples are now adapted to SQL sequences that can be used with the "IN" operator by default if the psycopg2.extensions module is imported (i.e., the SQL_IN adapter was moved from extras to extensions.)

Fixed some small buglets and build glitches:

- removed double mutex destroy
- removed all non-constant initializers
- fixed PyObject_HEAD declarations to avoid memory corruption on 64 bit architectures
- fixed several Python API calls to work on 64 bit architectures
- applied compatibility macros from PEP 353
- now using more than one argument format raise an error instead of a segfault

14.7.11 What's new in psycopg 2.0.5.1

- Now it really, really builds on MSVC and older gcc versions.

14.7.12 What's new in psycopg 2.0.5

- Fixed various buglets such as:
  - segfault when passing an empty string to Binary()
  - segfault on null queries
  - segfault and bad keyword naming in .executemany()
  - OperationalError in connection objects was always None
- Various changes to ZPsycopgDA to make it more zope2.9-ish.
- connect() now accept both integers and strings as port parameter

14.7.13 What's new in psycopg 2.0.4

- Fixed float conversion bug introduced in 2.0.3.

14.7.14 What's new in psycopg 2.0.3

- Fixed various buglets and a memory leak (see ChangeLog for details)

14.7.15 What's new in psycopg 2.0.2

- Fixed a bug in array typecasting that sometimes made psycopg forget about the last element in the array.
- Fixed some minor buglets in string memory allocations.
- Builds again with compilers different from gcc (#warning about PostgreSQL version is issued only if __GCC__ is defined.)

14.7.16 What's new in psycopg 2.0.1

- ZPsycopgDA now actually loads.

14.8 What's new in psycopg 2.0

- Fixed handle leak on win32.
- If available the new "safe" encoding functions of libpq are used.
- django and tinyerp people, please switch to psycopg 2 _without_ using a psycopg 1 compatibility layer (this release was anticipated so that you all stop grumbling about psycopg 2 is still in beta.. :)

14.8.1 What's new in psycopg 2.0 beta 7

- Ironed out last problems with times and date (should be quite solid now.)
- Fixed problems with some arrays.
- Slightly better ZPsycopgDA (no more double connection objects in the menu and other minor fixes.)
- ProgrammingError exceptions now have three extra attributes: .cursor (it is possible to access the query that caused the exception using error.cursor.query), .pgerror and .pgcode (PostgreSQL original error text and code.)
- The build system uses pg_config when available.
- Documentation in the doc/ directory! (With many kudos to piro.)

14.8.2 What's new in psycopg 2.0 beta 6

- Support for named cursors (see examples/fetch.py).
- Safer parsing of time intervals.
- Better parsing of times and dates, no more locale problems.
- Should now play well with py2exe and similar tools.
- The "decimal" module is now used if available under Python 2.3.

14.8.3 What's new in psycopg 2.0 beta 5

- Fixed all known bugs.
- The initial isolation level is now read from the server and .set_isolation_level() now takes values defined in psycopg2.extensions.
- .callproc() implemented as a SELECT of the given procedure.
- Better docstrings for a few functions/methods.
- Some time-related functions like psycopg2.TimeFromTicks() now take the local timezone into account. Also a tzinfo object (as per datetime module specifications) can be passed to the psycopg2.Time and psycopg2.Datetime constructors.
- All classes have been renamed to exist in the psycopg2._psycopg module, to fix problems with automatic documentation generators like epydoc.

- NOTIFY is correctly trapped (see examples/notify.py for example code.)

14.8.4 What's new in psycopg 2.0 beta 4

- psycopg module is now named psycopg2.
- No more segfaults when a UNICODE query can't be converted to the backend encoding.
- No more segfaults on empty queries.
- psycopg2.connect() now takes an integer for the port keyword parameter.
- "python setup.py bdist_rpm" now works.
- Fixed lots of small bugs, see ChangeLog for details.

14.8.5 What's new in psycopg 2.0 beta 3

- ZPsycopgDA now works (except table browsing.)
- psycopg build again on Python 2.2.

14.8.6 What's new in psycopg 2.0 beta 2

- Fixed ZPsycopgDA version check (ZPsycopgDA can now be imported in Zope.)
- psycopg.extras.DictRow works even after a new query on the generating cursor.
- Better setup.py for win32 (should build with MSCV or mingw.)
- Generic fixed and memory leaks plugs.

14.8.7 What's new in psycopg 2.0 beta 1

- Officially in beta (i.e., no new features will be added.)
- Array support: list objects can be passed as bound variables and are correctly returned for array columns.
- Added the psycopg.psycopg1 compatibility module (if you want instant psycopg 1 compatibility just "from psycopg import psycopg1 as psycopg".)
- Complete support for BYTEA columns and buffer objects.
- Added error codes to error messages.
- The AsIs adapter is now exported by default (also Decimal objects are adapted using the AsIs adapter (when str() is called on them they already format themselves using the right precision and scale.)
- The connect() function now takes "connection_factory" instead of "factory" as keyword argument.
- New setup.py code to build on win32 using mingw and better error messages on missing datetime headers,
- Internal changes that allow much better user-defined type casters.
- A lot of bugfixes (binary, datetime, 64 bit arches, GIL, .executemany())

14.8.8 What's new in psycopg 1.99.13

- Added missing .executemany() method.
- Optimized type cast from PostgreSQL to Python (psycopg should be even faster than before.)

14.8.9 What's new in psycopg 1.99.12

- .rowcount should be ok and in sync with psycopg 1.
- Implemented the new COPY FROM/COPY TO code when connection to the backend using libpq protocol 3 (this also removes all asprintf calls: build on win32 works again.) A protocol 3-enabled psycopg *can* connect to an old protocol 2 database and will detect it and use the right code.
- getquoted() called for real by the mogrification code.

14.8.10 What's new in psycopg 1.99.11

- 'cursor' argument in .cursor() connection method renamed to 'cursor_factory'.
- changed 'tuple_factory' cursor attribute name to 'row_factory'.
- the .cursor attribute is gone and connections and cursors are properly gc-managed.
- fixes to the async core.

14.8.11 What's new in psycopg 1.99.10

- The adapt() function now fully supports the adaptation protocol described in PEP 246. Note that the adapters registry now is indexed by (type, protocol) and not by type alone. Change your adapters accordingly.
- More configuration options moved from setup.py to setup.cfg.
- Fixed two memory leaks: one in cursor deallocation and one in row fetching (.fetchXXX() methods.)

14.8.12 What's new in psycopg 1.99.9

- Added simple pooling code (psycopg.pool module); see the reworked examples/threads.py for example code.
- Added DECIMAL typecaster to convert postgresql DECIMAL and NUMERIC types (i.e, all types with an OID of NUMERICOID.) Note that the DECIMAL typecaster does not set scale and precision on the created objects but uses Python defaults.
- ZPsycopgDA back in and working using the new pooling code.
- Isn't that enough? :)

14.8.13 What's new in psycopg 1.99.8

- added support for UNICODE queries.
- added UNICODE typecaster; to activate it just do:

```
psycopg.extensions.register_type(psycopg.extensions.UNICODE)
```

  Note that the UNICODE typecaster override the STRING one, so it is not activated by default.

- cursors now really support the iterator protocol.
- solved the rounding errors in time conversions.
- now cursors support .fileno() and .isready() methods, to be used in select() calls.
- .copy_from() and .copy_in() methods are back in (still using the old protocol, will be updated to use new one in next release.)
- fixed memory corruption bug reported on win32 platform.

14.8.14 What's new in psycopg 1.99.7

- added support for tuple factories in cursor objects (removed factory argument in favor of a .tuple_factory attribute on the cursor object); see the new module psycopg.extras for a cursor (DictCursor) that return rows as objects that support indexing both by position and column name.
- added support for tzinfo objects in datetime.timestamp objects: the PostgreSQL type "timestamp with time zone" is converted to datetime.timestamp with a FixedOffsetTimezone initialized as necessary.

14.8.15 What's new in psycopg 1.99.6

- sslmode parameter from 1.1.x
- various datetime conversion improvements.
- now psycopg should compile without mx or without native datetime (not both, obviously.)
- included various win32/MSVC fixes (pthread.h changes, winsock2 library, include path in setup.py, etc.)
- ported interval fixes from 1.1.14/1.1.15.
- the last query executed by a cursor is now available in the .query attribute.
- conversion of unicode strings to backend encoding now uses a table (that still need to be filled.)
- cursors now have a .mogrify() method that return the query string instead of executing it.
- connection objects now have a .dsn read-only attribute that holds the connection string.
- moved psycopg C module to _psycopg and made psycopg a python module: this allows for a neat separation of DBAPI-2.0 functionality and psycopg extensions; the psycopg namespace will be also used to provide python-only extensions (like the pooling code, some ZPsycopgDA support functions and the like.)

14.8.16 What's new in psycopg 1.99.3

- added support for python 2.3 datetime types (both ways) and made datetime the default set of typecasters when available.
- added example: dt.py.

14.8.17 What's new in psycopg 1.99.3

- initial working support for unicode bound variables: UTF-8 and latin-1 backend encodings are natively supported (and the encoding.py example even works!)
- added .set_client_encoding() method on the connection object.
- added examples: encoding.py, binary.py, lastrowid.py.

14.8.18 What's new in psycopg 1.99.2

- better typecasting:
  - DateTimeDelta used for postgresql TIME (merge from 1.1)
  - BYTEA now is converted to a real buffer object, not to a string
- buffer objects are now adapted into Binary objects automatically.
- ported scroll method from 1.1 (DBAPI-2.0 extension for cursors)
- initial support for some DBAPI-2.0 extensions:
  - .rownumber attribute for cursors
  - .connection attribute for cursors
  - .next() and .__iter__() methods to have cursors support the iterator protocol
  - all exception objects are exported to the connection object

14.8.19 What's new in psycopg 1.99.1

- implemented microprotocols to adapt arbitrary types to the interface used by psycopg to bind variables in execute;
- moved qstring, pboolean and mxdatetime to the new adapter layout (binary is still missing; python 2.3 datetime needs to be written).

14.8.20 What's new in psycopg 1.99.0

- reorganized the whole source tree;
- async core is in place;
- splitted QuotedString objects from mx stuff;
- dropped autotools and moved to pythonic setup.py (needs work.)

Indices and tables

- genindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X