psycopg

Release 3.1.4

Daniele Varrazzo

Nov 02, 2022

CONTENTS

Psycopg 3 is a newly designed PostgreSQL database adapter for the Python programming language.

Psycopg 3 presents a familiar interface for everyone who has used Psycopg 2 or any other DB-API 2.0 database adapter, but allows to use more modern PostgreSQL and Python features, such as:

- *Asynchronous support*
- *COPY support from Python objects*
- *A redesigned connection pool*
- *Support for static typing*
- *Server-side parameters binding*
- *Prepared statements*
- *Statements pipeline*
- *Binary communication*
- *Direct access to the libpq functionalities*

CHAPTER

ONE

DOCUMENTATION

1.1 Getting started with Psycopg 3

This section of the documentation will explain *how to install Psycopg* and how to perform normal activities such as *querying the database* or *loading data using COPY*.

Important: If you are familiar with psycopg2 please take a look at *Differences from psycopg2* to see what is changed.

1.1.1 Installation

In short, if you use a *supported system*:

```
pip install --upgrade pip           # upgrade pip to at least 20.3
pip install "psycopg[binary]"
```

and you should be *ready to start*. Read further for alternative ways to install.

Supported systems

The Psycopg version documented here has *official and tested* support for:

- Python: from version 3.7 to 3.11
  - Python 3.6 supported before Psycopg 3.1
- PostgreSQL: from version 10 to 15
- OS: Linux, macOS, Windows

The tests to verify the supported systems run in Github workflows: anything that is not tested there is not officially supported. This includes:

- Unofficial Python distributions such as Conda;
- Alternative PostgreSQL implementation;
- macOS hardware and releases not available on Github workflows.

If you use an unsupported system, things might work (because, for instance, the database may use the same wire protocol as PostgreSQL) but we cannot guarantee the correct working or a smooth ride.

Binary installation

The quickest way to start developing with Psycopg 3 is to install the binary packages by running:

```
pip install "psycopg[binary]"
```

This will install a self-contained package with all the libraries needed. **You will need pip 20.3 at least**: please run `pip install --upgrade pip` to update it beforehand.

The above package should work in most situations. It **will not work** in some cases though.

If your platform is not supported you should proceed to a *local installation* or a *pure Python installation*.

See also:

Did Psycopg 3 install ok? Great! You can now move on to the *basic module usage* to learn how it works.

Keep on reading if the above method didn't work and you need a different way to install Psycopg 3.

For further information about the differences between the packages see *pq module implementations*.

Local installation

A "Local installation" results in a performing and maintainable library. The library will include the speed-up C module and will be linked to the system libraries (`libpq`, `libssl`...) so that system upgrade of libraries will upgrade the libraries used by Psycopg 3 too. This is the preferred way to install Psycopg for a production site.

In order to perform a local installation you need some prerequisites:

- a C compiler,
- Python development headers (e.g. the `python3-dev` package).
- PostgreSQL client development headers (e.g. the `libpq-dev` package).
- The `pg_config` program available in the `PATH`.

You **must be able** to troubleshoot an extension build, for instance you must be able to read your compiler's error message. If you are not, please don't try this and follow the *binary installation* instead.

If your build prerequisites are in place you can run:

```
pip install "psycopg[c]"
```

Pure Python installation

If you simply install:

```
pip install psycopg
```

without `[c]` or `[binary]` extras you will obtain a pure Python implementation. This is particularly handy to debug and hack, but it still requires the system libpq to operate (which will be imported dynamically via `ctypes`).

In order to use the pure Python installation you will need the `libpq` installed in the system: for instance on Debian system you will probably need:

```
sudo apt install libpq5
```

Note: The `libpq` is the client library used by **`psql`**, the PostgreSQL command line client, to connect to the database. On most systems, installing **`psql`** will install the `libpq` too as a dependency.

If you are not able to fulfill this requirement please follow the *binary installation*.

Installing the connection pool

The *Psycopg connection pools* are distributed in a separate package from the `psycopg` package itself, in order to allow a different release cycle.

In order to use the pool you must install the `pool` extra, using `pip install "psycopg[pool]"`, or install the *`psycopg_pool`* package separately, which would allow to specify the release to install more precisely.

Handling dependencies

If you need to specify your project dependencies (for instance in a `requirements.txt` file, `setup.py`, `pyproject.toml` dependencies...) you should probably specify one of the following:

- If your project is a library, add a dependency on `psycopg`. This will make sure that your library will have the `psycopg` package with the right interface and leaves the possibility of choosing a specific implementation to the end user of your library.
- If your project is a final application (e.g. a service running on a server) you can require a specific implementation, for instance `psycopg[c]`, after you have made sure that the prerequisites are met (e.g. the depending libraries and tools are installed in the host machine).

In both cases you can specify which version of Psycopg to use using requirement specifiers.

If you want to make sure that a specific implementation is used you can specify the `PSYCOPG_IMPL` environment variable: importing the library will fail if the implementation specified is not available. See *pq module implementations*.

1.1.2 Basic module usage

The basic Psycopg usage is common to all the database adapters implementing the DB-API protocol. Other database adapters, such as the builtin `sqlite3` or `psycopg2`, have roughly the same pattern of interaction.

Main objects in Psycopg 3

Here is an interactive session showing some of the basic commands:

```
# Note: the module name is psycopg, not psycopg3
import psycopg

# Connect to an existing database
with psycopg.connect("dbname=test user=postgres") as conn:

    # Open a cursor to perform database operations
    with conn.cursor() as cur:

        # Execute a command: this creates a new table
        cur.execute("""
```

(continues on next page)

(continued from previous page)

```
            CREATE TABLE test (
                id serial PRIMARY KEY,
                num integer,
                data text)
            """)

        # Pass data to fill a query placeholders and let Psycopg perform
        # the correct conversion (no 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()
        # will return (1, 100, "abc'def")

        # You can use `cur.fetchmany()`, `cur.fetchall()` to return a list
        # of several records, or even iterate on the cursor
        for record in cur:
            print(record)

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

In the example you can see some of the main objects and methods and how they relate to each other:

- The function *connect()* creates a new database session and returns a new *Connection* instance. *AsyncConnection.connect()* creates an `asyncio` connection instead.
- The *Connection* class 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, iterating on the cursor or using methods such as *fetchone()*, *fetchmany()*, *fetchall()*.
- Using these objects as context managers (i.e. using `with`) will make sure to close them and free their resources at the end of the block (notice that *this is different from psycopg2*).

See also:

A few important topics you will have to deal with are:

- *Passing parameters to SQL queries*.
- *Adapting basic Python types*.
- *Transactions management*.

Shortcuts

The pattern above is familiar to `psycopg2` users. However, Psycopg 3 also exposes a few simple extensions which make the above pattern leaner:

- the *Connection* objects exposes an *execute()* method, equivalent to creating a cursor, calling its *execute()* method, and returning it.

```
# In Psycopg 2
cur = conn.cursor()
cur.execute(...)

# In Psycopg 3
cur = conn.execute(...)
```

- The *Cursor.execute()* method returns `self`. This means that you can chain a fetch operation, such as *fetchone()*, to the `execute()` call:

```
# In Psycopg 2
cur.execute(...)
record = cur.fetchone()

cur.execute(...)
for record in cur:
    ...

# In Psycopg 3
record = cur.execute(...).fetchone()

for record in cur.execute(...):
    ...
```

Using them together, in simple cases, you can go from creating a connection to using a result in a single expression:

```
print(psycopg.connect(DSN).execute("SELECT now()").fetchone()[0])
# 2042-07-12 18:15:10.706497+01:00
```

Connection context

Psycopg 3 *Connection* can be used as a context manager:

```
with psycopg.connect() as conn:
    ... # use the connection

# the connection is now closed
```

When the block is exited, if there is a transaction open, it will be committed. If an exception is raised within the block the transaction is rolled back. In both cases the connection is closed. It is roughly the equivalent of:

```
conn = psycopg.connect()
try:
    ... # use the connection
except BaseException:
```

(continues on next page)

(continued from previous page)

```
    conn.rollback()
else:
    conn.commit()
finally:
    conn.close()
```

Note: This behaviour is not what `psycopg2` does: in `psycopg2` there is no final close() and the connection can be used in several `with` statements to manage different transactions. This behaviour has been considered non-standard and surprising so it has been replaced by the more explicit *`transaction()`* block.

Note that, while the above pattern is what most people would use, *`connect()`* doesn't enter a block itself, but returns an "un-entered" connection, so that it is still possible to use a connection regardless of the code scope and the developer is free to use (and responsible for calling) *`commit()`*, *`rollback()`*, *`close()`* as and where needed.

Warning: If a connection is just left to go out of scope, the way it will behave with or without the use of a `with` block is different:

- if the connection is used without a `with` block, the server will find a connection closed INTRANS and roll back the current transaction;
- if the connection is used with a `with` block, there will be an explicit COMMIT and the operations will be finalised.

You should use a `with` block when your intention is just to execute a set of operations and then committing the result, which is the most usual thing to do with a connection. If your connection life cycle and transaction pattern is different, and want more control on it, the use without `with` might be more convenient.

See *Transactions management* for more information.

`AsyncConnection` can be also used as context manager, using `async with`, but be careful about its quirkiness: see *with async connections* for details.

Adapting pyscopg to your program

The above *pattern of use* only shows the default behaviour of the adapter. Psycopg can be customised in several ways, to allow the smoothest integration between your Python program and your PostgreSQL database:

- If your program is concurrent and based on `asyncio` instead of on threads/processes, you can use *async connections and cursors*.
- If you want to customise the objects that the cursor returns, instead of receiving tuples, you can specify your *row factories*.
- If you want to customise how Python values and PostgreSQL types are mapped into each other, beside the *basic type mapping*, you can *configure your types*.

1.1.3 Passing parameters to SQL queries

Most of the times, writing a program you will have to mix bits of SQL statements with values provided by the rest of the program:

```
SELECT some, fields FROM some_table WHERE id = ...
```

`id` equals what? Probably you will have a Python value you are looking for.

`execute()` arguments

Passing parameters to a 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 (id, created_at, last_name)
    VALUES (%s, %s, %s);
    """,
    (10, datetime.date(2020, 11, 18), "O'Reilly"))
```

is *roughly* equivalent to the SQL command:

```
INSERT INTO some_table (id, created_at, last_name)
VALUES (10, '2020-11-18', 'O''Reilly');
```

Note that the parameters will not be really merged to the query: query and the parameters are sent to the server separately: see *Server-side binding* for details.

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 (id, created_at, updated_at, last_name)
    VALUES (%(id)s, %(created)s, %(created)s, %(name)s);
    """,
    {'id': 10, 'name': "O'Reilly", 'created': datetime.date(2020, 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
cur.execute("INSERT INTO foo VALUES (%s)", ("bar",)) # correct
cur.execute("INSERT INTO foo VALUES (%s)", ["bar"])  # correct
```

- The placeholder *must not be quoted*:

```
cur.execute("INSERT INTO numbers VALUES ('%s')", ("Hello",)) # WRONG
cur.execute("INSERT INTO numbers VALUES (%s)", ("Hello",))   # 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 for the type. You may find other placeholders used in Psycopg queries (`%b` and `%t`) but they are not related to the type of the argument: see *Binary parameters and results* if you want to read more:

```
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. If you need to generate SQL queries dynamically (for instance choosing a table name at runtime) you can use the functionalities provided in the *psycopg.sql* module:

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

Danger: SQL injection

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, sometimes 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
# SyntaxError: syntax error at or near "Reilly"
```

If the variables containing the data to send to the database come from an untrusted source (such as data coming from a form 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 on database systems. Before continuing, please print this page as a memo and hang it onto your desk.

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

Warning:

- Don't manually merge values to a query: hackers from a foreign country will break into your computer and steal not only your disks, but also your cds, leaving you only with the three most embarrassing records you ever bought. On cassette tapes.
- If you use the % operator to merge values to a query, con artists will seduce your cat, who will run away taking your credit card and your sunglasses with them.
- If you use + to merge a textual value to a string, bad guys in balaclava will find their way to your fridge, drink all your beer, and leave your toilet seat up and your toilet paper in the wrong orientation.
- You don't want to manually merge values to a query: *use the provided methods* instead.

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

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

Note: Python static code checkers are not quite there yet, but, in the future, it will be possible to check your code for improper use of string expressions in queries. See *Checking literal strings in queries* for details.

See also:

Now that you know how to pass parameters to queries, you can take a look at *how Psycopg converts data types*.

Binary parameters and results

PostgreSQL has two different ways to transmit data between client and server: *TEXT*, always available, and *BINARY*, available most of the times but not always. Usually the binary format is more efficient to use.

Psycopg can support both formats for each data type. Whenever a value is passed to a query using the normal `%s` placeholder, the best format available is chosen (often, but not always, the binary format is picked as the best choice).

If you have a reason to select explicitly the binary format or the text format for a value you can use respectively a `%b` placeholder or a `%t` placeholder instead of the normal `%s`. *execute()* will fail if a *Dumper* for the right data type and format is not available.

The same two formats, text or binary, are used by PostgreSQL to return data from a query to the client. Unlike with parameters, where you can choose the format value-by-value, all the columns returned by a query will have the same format. Every type returned by the query should have a *Loader* configured, otherwise the data will be returned as unparsed `str` (for text results) or buffer (for binary results).

Note: The pg_type table defines which format is supported for each PostgreSQL data type. Text input/output is managed by the functions declared in the `typinput` and `typoutput` fields (always present), binary input/output is managed by the `typsend` and `typreceive` (which are optional).

Because not every PostgreSQL type supports binary output, by default, the data will be returned in text format. In order to return data in binary format you can create the cursor using *Connection.cursor*`(binary=True)` or execute the query using *Cursor.execute*`(binary=True)`. A case in which requesting binary results is a clear winner is when you have large binary data in the database, such as images:

```
cur.execute(
    "SELECT image_data FROM images WHERE id = %s", [image_id], binary=True)
data = cur.fetchone()[0]
```

1.1.4 Adapting basic Python types

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

Converting the following data types between Python and PostgreSQL works out-of-the-box and doesn't require any configuration. In case you need to customise the conversion you should take a look at *Data adaptation configuration*.

Booleans adaptation

Python `bool` values `True` and `False` are converted to the equivalent PostgreSQL boolean type:

```
>>> cur.execute("SELECT %s, %s", (True, False))
# equivalent to "SELECT true, false"
```

Numbers adaptation

See also:

- PostgreSQL numeric types
- Python `int` values can be converted to PostgreSQL `smallint`, `integer`, `bigint`, or `numeric`, according to their numeric value. Psycopg will choose the smallest data type available, because PostgreSQL can automatically cast a type up (e.g. passing a `smallint` where PostgreSQL expect an `integer` is gladly accepted) but will not cast down automatically (e.g. if a function has an `integer` argument, passing it a `bigint` value will fail, even if the value is 1).
- Python `float` values are converted to PostgreSQL `float8`.
- Python `Decimal` values are converted to PostgreSQL `numeric`.

On the way back, smaller types (`int2`, `int4`, `float4`) are promoted to the larger Python counterpart.

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.

Strings adaptation

See also:

- PostgreSQL character types

Python `str` are converted to PostgreSQL string syntax, and PostgreSQL types such as `text` and `varchar` are converted back to Python `str`:

```
conn = psycopg.connect()
conn.execute(
    "INSERT INTO menu (id, entry) VALUES (%s, %s)",
    (1, "Crème Brûlée at 4.99€"))
conn.execute("SELECT entry FROM menu WHERE id = 1").fetchone()[0]
'Crème Brûlée at 4.99€'
```

PostgreSQL databases have an encoding, and the session has an encoding too, exposed in the `Connection.info.encoding` attribute. If your database and connection are in UTF-8 encoding you will likely have no problem, otherwise you will have to make sure that your application only deals with the non-ASCII chars that the database can handle; failing to do so may result in encoding/decoding errors:

```
# The encoding is set at connection time according to the db configuration
conn.info.encoding
'utf-8'

# The Latin-9 encoding can manage some European accented letters
# and the Euro symbol
conn.execute("SET client_encoding TO LATIN9")
conn.execute("SELECT entry FROM menu WHERE id = 1").fetchone()[0]
'Crème Brûlée at 4.99€'

# The Latin-1 encoding doesn't have a representation for the Euro symbol
conn.execute("SET client_encoding TO LATIN1")
conn.execute("SELECT entry FROM menu WHERE id = 1").fetchone()[0]
# Traceback (most recent call last)
# ...
# UntranslatableCharacter: character with byte sequence 0xe2 0x82 0xac
# in encoding "UTF8" has no equivalent in encoding "LATIN1"
```

In rare cases you may have strings with unexpected encodings in the database. Using the `SQL_ASCII` client encoding will disable decoding of the data coming from the database, which will be returned as `bytes`:

```
conn.execute("SET client_encoding TO SQL_ASCII")
conn.execute("SELECT entry FROM menu WHERE id = 1").fetchone()[0]
b'Cr\xc3\xa8me Br\xc3\xbbl\xc3\xa9e at 4.99\xe2\x82\xac'
```

Alternatively you can cast the unknown encoding data to `bytea` to retrieve it as bytes, leaving other strings unaltered: see *Binary adaptation*

Note that PostgreSQL text cannot contain the `0x00` byte. If you need to store Python strings that may contain binary zeros you should use a `bytea` field.

Binary adaptation

Python types representing binary objects (`bytes`, `bytearray`, `memoryview`) are converted by default to `bytea` fields. By default data received is returned as `bytes`.

If you are storing large binary data in bytea fields (such as binary documents or images) you should probably use the binary format to pass and return values, otherwise binary data will undergo ASCII escaping, taking some CPU time and more bandwidth. See *Binary parameters and results* for details.

Date/time types adaptation

See also:

- PostgreSQL date/time types
- Python `date` objects are converted to PostgreSQL `date`.
- Python `datetime` objects are converted to PostgreSQL `timestamp` (if they don't have a `tzinfo` set) or `timestamptz` (if they do).
- Python `time` objects are converted to PostgreSQL `time` (if they don't have a `tzinfo` set) or `timetz` (if they do).
- Python `timedelta` objects are converted to PostgreSQL `interval`.

PostgreSQL `timestamptz` values are returned with a timezone set to the connection TimeZone setting, which is available as a Python `ZoneInfo` object in the `Connection.info.timezone` attribute:

```
>>> conn.info.timezone
zoneinfo.ZoneInfo(key='Europe/London')

>>> conn.execute("select '2048-07-08 12:00'::timestamptz").fetchone()[0]
datetime.datetime(2048, 7, 8, 12, 0, tzinfo=zoneinfo.ZoneInfo(key='Europe/London'))
```

Note: PostgreSQL `timestamptz` doesn't store "a timestamp with a timezone attached": it stores a timestamp always in UTC, which is converted, on output, to the connection TimeZone setting:

```
>>> conn.execute("SET TIMEZONE to 'Europe/Rome'")  # UTC+2 in summer
```

```
>>> conn.execute("SELECT '2042-07-01 12:00Z'::timestamptz").fetchone()[0]  # UTC input
datetime.datetime(2042, 7, 1, 14, 0, tzinfo=zoneinfo.ZoneInfo(key='Europe/Rome'))
```

Check out the PostgreSQL documentation about timezones for all the details.

JSON adaptation

Psycopg can map between Python objects and PostgreSQL json/jsonb types, allowing to customise the load and dump function used.

Because several Python objects could be considered JSON (dicts, lists, scalars, even date/time if using a dumps function customised to use them), Psycopg requires you to wrap the object to dump as JSON into a wrapper: either *psycopg.types.json.Json* or *Jsonb*.

```
from psycopg.types.json import Jsonb

thing = {"foo": ["bar", 42]}
conn.execute("INSERT INTO mytable VALUES (%s)", [Jsonb(thing)])
```

By default Psycopg uses the standard library `json.dumps` and `json.loads` functions to serialize and de-serialize Python objects to JSON. If you want to customise how serialization happens, for instance changing serialization parameters or using a different JSON library, you can specify your own functions using the *psycopg.types.json.set_json_dumps()* and *set_json_loads()* functions, to apply either globally or to a specific context (connection or cursor).

```
from functools import partial
from psycopg.types.json import Jsonb, set_json_dumps, set_json_loads
import ujson

# Use a faster dump function
set_json_dumps(ujson.dumps)

# Return floating point values as Decimal, just in one connection
set_json_loads(partial(json.loads, parse_float=Decimal), conn)

conn.execute("SELECT %s", [Jsonb({"value": 123.45})]).fetchone()[0]
# {'value': Decimal('123.45')}
```

If you need an even more specific dump customisation only for certain objects (including different configurations in the same query) you can specify a `dumps` parameter in the *Json*/*Jsonb* wrapper, which will take precedence over what is specified by `set_json_dumps()`.

```
from uuid import UUID, uuid4

class UUIDEncoder(json.JSONEncoder):
    """A JSON encoder which can dump UUID."""
    def default(self, obj):
        if isinstance(obj, UUID):
            return str(obj)
        return json.JSONEncoder.default(self, obj)

uuid_dumps = partial(json.dumps, cls=UUIDEncoder)
obj = {"uuid": uuid4()}
cnn.execute("INSERT INTO objs VALUES %s", [Json(obj, dumps=uuid_dumps)])
# will insert: {'uuid': '0a40799d-3980-4c65-8315-2956b18ab0e1'}
```

Lists adaptation

Python `list` objects are adapted to PostgreSQL arrays and back. Only lists containing objects of the same type can be dumped to PostgreSQL (but the list may contain `None` elements).

Note: If you have a list of values which you want to use with the `IN` operator... don't. It won't work (neither with a list nor with a tuple):

```
>>> conn.execute("SELECT * FROM mytable WHERE id IN %s", [[10,20,30]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
psycopg.errors.SyntaxError: syntax error at or near "$1"
LINE 1: SELECT * FROM mytable WHERE id IN $1
                                         ^
```

What you want to do instead is to use the '= ANY()' expression and pass the values as a list (not a tuple).

```
>>> conn.execute("SELECT * FROM mytable WHERE id = ANY(%s)", [[10,20,30]])
```

This has also the advantage of working with an empty list, whereas `IN ()` is not valid SQL.

UUID adaptation

Python uuid.UUID objects are adapted to PostgreSQL UUID type and back:

```
>>> conn.execute("select gen_random_uuid()").fetchone()[0]
UUID('97f0dd62-3bd2-459e-89b8-a5e36ea3c16c')

>>> from uuid import uuid4
>>> conn.execute("select gen_random_uuid() = %s", [uuid4()]).fetchone()[0]
False  # long shot
```

Network data types adaptation

Objects from the ipaddress module are converted to PostgreSQL network address types:

- IPv4Address, IPv4Interface objects are converted to the PostgreSQL inet type. On the way back, inet values indicating a single address are converted to IPv4Address, otherwise they are converted to IPv4Interface
- IPv4Network objects are converted to the cidr type and back.
- IPv6Address, IPv6Interface, IPv6Network objects follow the same rules, with IPv6 inet and cidr values.

```
>>> conn.execute("select '192.168.0.1'::inet, '192.168.0.1/24'::inet").fetchone()
(IPv4Address('192.168.0.1'), IPv4Interface('192.168.0.1/24'))

>>> conn.execute("select '::ffff:1.2.3.0/120'::cidr").fetchone()[0]
IPv6Network('::ffff:102:300/120')
```

Enum adaptation

New in version 3.1.

Psycopg can adapt Python Enum subclasses into PostgreSQL enum types (created with the CREATE TYPE ... AS ENUM (...) command).

In order to set up a bidirectional enum mapping, you should get information about the PostgreSQL enum using the *EnumInfo* class and register it using *register_enum()*. The behaviour of unregistered and registered enums is different.

- If the enum is not registered with register_enum():
  - Pure Enum classes are dumped as normal strings, using their member names as value. The unknown oid is used, so PostgreSQL should be able to use this string in most contexts (such as an enum or a text field).

    Changed in version 3.1: In previous version dumping pure enums is not supported and raise a "cannot adapt" error.
  - Mix-in enums are dumped according to their mix-in type (because a class MyIntEnum(int, Enum) is more specifically an int than an Enum, so it's dumped by default according to int rules).
  - PostgreSQL enums are loaded as Python strings. If you want to load arrays of such enums you will have to find their OIDs using *types.TypeInfo.fetch()* and register them using *register()*.
- If the enum is registered (using *EnumInfo*.fetch() and *register_enum()*):
  - Enums classes, both pure and mixed-in, are dumped by name.
  - The registered PostgreSQL enum is loaded back as the registered Python enum members.

class psycopg.types.enum.**EnumInfo**(*name*, *oid*, *array_oid*, *labels*)

Manage information about an enum type.

EnumInfo is a subclass of *TypeInfo*: refer to the latter's documentation for generic usage, especially the *fetch()* method.

labels

After *fetch()*, it contains the labels defined in the PostgreSQL enum type.

enum

After *register_enum()* is called, it will contain the Python type mapping to the registered enum.

psycopg.types.enum.**register_enum**(*info*, *context=None*, *enum=None*, ***, *mapping=None*)

Register the adapters to load and dump a enum type.

Parameters

- **info** (*EnumInfo*) – The object with the information about the enum to register.
- **context** (Optional[*AdaptContext*]) – The context where to register the adapters. If None, register it globally.
- **enum** (Optional[Type[TypeVar(E, bound= Enum)]]) – Python enum type matching to the PostgreSQL one. If None, a new enum will be generated and exposed as *EnumInfo.enum*.
- **mapping** (Union[Mapping[TypeVar(E, bound= Enum), str], Sequence[Tuple[TypeVar(E, bound= Enum), str]], None]) – Override the mapping between enum members and info labels.

After registering, fetching data of the registered enum will cast PostgreSQL enum labels into corresponding Python enum members.

If no enum is specified, a new Enum is created based on PostgreSQL enum labels.

Example:

```
>>> from enum import Enum, auto
>>> from psycopg.types.enum import EnumInfo, register_enum

>>> class UserRole(Enum):
...     ADMIN = auto()
...     EDITOR = auto()
...     GUEST = auto()

>>> conn.execute("CREATE TYPE user_role AS ENUM ('ADMIN', 'EDITOR', 'GUEST')")

>>> info = EnumInfo.fetch(conn, "user_role")
>>> register_enum(info, conn, UserRole)

>>> some_editor = info.enum.EDITOR
>>> some_editor
<UserRole.EDITOR: 2>

>>> conn.execute(
...     "SELECT pg_typeof(%(editor)s), %(editor)s",
...     {"editor": some_editor}
... ).fetchone()
('user_role', <UserRole.EDITOR: 2>)
```

(continues on next page)

(continued from previous page)

```
>>> conn.execute(
...     "SELECT ARRAY[%s, %s]",
...     [UserRole.ADMIN, UserRole.GUEST]
... ).fetchone()
[<UserRole.ADMIN: 1>, <UserRole.GUEST: 3>]
```

If the Python and the PostgreSQL enum don't match 1:1 (for instance if members have a different name, or if more than one Python enum should map to the same PostgreSQL enum, or vice versa), you can specify the exceptions using the `mapping` parameter.

`mapping` should be a dictionary with Python enum members as keys and the matching PostgreSQL enum labels as values, or a list of `(member, label)` pairs with the same meaning (useful when some members are repeated). Order matters: if an element on either side is specified more than once, the last pair in the sequence will take precedence:

```
# Legacy roles, defined in medieval times.
>>> conn.execute(
...     "CREATE TYPE abbey_role AS ENUM ('ABBOT', 'SCRIBE', 'MONK', 'GUEST')")

>>> info = EnumInfo.fetch(conn, "abbey_role")
>>> register_enum(info, conn, UserRole, mapping=[
...     (UserRole.ADMIN, "ABBOT"),
...     (UserRole.EDITOR, "SCRIBE"),
...     (UserRole.EDITOR, "MONK")])

>>> conn.execute("SELECT '{ABBOT,SCRIBE,MONK,GUEST}'::abbey_role[]").fetchone()[0]
[<UserRole.ADMIN: 1>,
 <UserRole.EDITOR: 2>,
 <UserRole.EDITOR: 2>,
 <UserRole.GUEST: 3>]

>>> conn.execute("SELECT %s::text[]", [list(UserRole)]).fetchone()[0]
['ABBOT', 'MONK', 'GUEST']
```

A particularly useful case is when the PostgreSQL labels match the *values* of a `str`-based Enum. In this case it is possible to use something like `{m: m.value for m in enum}` as mapping:

```
>>> class LowercaseRole(str, Enum):
...     ADMIN = "admin"
...     EDITOR = "editor"
...     GUEST = "guest"

>>> conn.execute(
...     "CREATE TYPE lowercase_role AS ENUM ('admin', 'editor', 'guest')")

>>> info = EnumInfo.fetch(conn, "lowercase_role")
>>> register_enum(
...     info, conn, LowercaseRole, mapping={m: m.value for m in LowercaseRole})

>>> conn.execute("SELECT 'editor'::lowercase_role").fetchone()[0]
<LowercaseRole.EDITOR: 'editor'>
```

1.1.5 Adapting other PostgreSQL types

PostgreSQL offers other data types which don't map to native Python types. Psycopg offers wrappers and conversion functions to allow their use.

Composite types casting

Psycopg can adapt PostgreSQL composite types (either created with the `CREATE TYPE` command or implicitly defined after a table row type) to and from Python tuples, `namedtuple`, or any other suitable object configured.

Before using a composite type it is necessary to get information about it using the *CompositeInfo* class and to register it using *register_composite()*.

class psycopg.types.composite.**CompositeInfo**(*name*, *oid*, *array_oid*, ***, *regtype=''*, *field_names*, *field_types*)

Manage information about a composite type.

`CompositeInfo` is a *TypeInfo* subclass: check its documentation for the generic usage, especially the *fetch()* method.

python_type

After *register_composite()* is called, it will contain the python type mapping to the registered composite.

psycopg.types.composite.**register_composite**(*info*, *context=None*, *factory=None*)

Register the adapters to load and dump a composite type.

Parameters

- **info** (*CompositeInfo*) – The object with the information about the composite to register.
- **context** (`Optional`[*AdaptContext*]) – The context where to register the adapters. If `None`, register it globally.
- **factory** (`Optional`[`Callable`[`...`, `Any`]]) – Callable to convert the sequence of attributes read from the composite into a Python object.

Note: Registering the adapters doesn't affect objects already created, even if they are children of the registered context. For instance, registering the adapter globally doesn't affect already existing connections.

After registering, fetching data of the registered composite will invoke `factory` to create corresponding Python objects.

If no factory is specified, a `namedtuple` is created and used to return data.

If the `factory` is a type (and not a generic callable), then dumpers for that type are created and registered too, so that passing objects of that type to a query will adapt them to the registered type.

Example:

```
>>> from psycopg.types.composite import CompositeInfo, register_composite

>>> conn.execute("CREATE TYPE card AS (value int, suit text)")

>>> info = CompositeInfo.fetch(conn, "card")
>>> register_composite(info, conn)
```

(continues on next page)

(continued from previous page)

```
>>> my_card = info.python_type(8, "hearts")
>>> my_card
card(value=8, suit='hearts')

>>> conn.execute(
...     "SELECT pg_typeof(%(card)s), (%(card)s).suit", {"card": my_card}
...     ).fetchone()
('card', 'hearts')

>>> conn.execute("SELECT (%s, %s)::card", [1, "spades"]).fetchone()[0]
card(value=1, suit='spades')
```

Nested composite types are handled as expected, provided that the type of the composite components are registered as well:

```
>>> conn.execute("CREATE TYPE card_back AS (face card, back text)")

>>> info2 = CompositeInfo.fetch(conn, "card_back")
>>> register_composite(info2, conn)

>>> conn.execute("SELECT ((8, 'hearts'), 'blue')::card_back").fetchone()[0]
card_back(face=card(value=8, suit='hearts'), back='blue')
```

Range adaptation

PostgreSQL range types are a family of data types representing a range of values between two elements. The type of the element is called the range *subtype*. PostgreSQL offers a few built-in range types and allows the definition of custom ones.

All the PostgreSQL range types are loaded as the *Range* Python type, which is a `Generic` type and can hold bounds of different types.

class psycopg.types.range.**Range**(*lower=None*, *upper=None*, *bounds='[)'*, *empty=False*)

Python representation for a PostgreSQL range type.

Parameters

- **lower** (`Optional`[`TypeVar`(`T`)]) – lower bound for the range. `None` means unbound
- **upper** (`Optional`[`TypeVar`(`T`)]) – upper bound for the range. `None` means unbound
- **bounds** (`str`) – one of the literal strings `()`, `[)`, `(]`, `[]`, representing whether the lower or upper bounds are included
- **empty** (`bool`) – 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.

PostgreSQL will perform normalisation on `Range` objects used as query parameters, so, when they are fetched back, they will be found in the normal form (for instance ranges on integers will have `[)` bounds).

`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 a boolean context, nonempty ones evaluate to `True`.

Range objects have the following attributes:

isempty: bool

True if the range is empty.

lower: Optional[TypeVar(``T``)]

The lower bound of the range. None if empty or unbound.

upper: Optional[TypeVar(``T``)]

The upper bound of the range. None if empty or unbound.

lower_inc: bool

True if the lower bound is included in the range.

upper_inc: bool

True if the upper bound is included in the range.

lower_inf: bool

True if the range doesn't have a lower bound.

upper_inf: bool

True if the range doesn't have an upper bound.

The built-in range objects are adapted automatically: if a Range objects contains date bounds, it is dumped using the daterange OID, and of course daterange values are loaded back as Range[date].

If you create your own range type you can use *RangeInfo* and *register_range()* to associate the range type with its subtype and make it work like the builtin ones.

class psycopg.types.range.**RangeInfo**(*name*, *oid*, *array_oid*, ***, *regtype=''*, *subtype_oid*)

Manage information about a range type.

RangeInfo is a *TypeInfo* subclass: check its documentation for generic details, especially the *fetch()* method.

psycopg.types.range.**register_range**(*info*, *context=None*)

Register the adapters to load and dump a range type.

Parameters

- **info** (*RangeInfo*) – The object with the information about the range to register.
- **context** (Optional[*AdaptContext*]) – The context where to register the adapters. If None, register it globally.

Register loaders so that loading data of this type will result in a *Range* with bounds parsed as the right subtype.

Note: Registering the adapters doesn't affect objects already created, even if they are children of the registered context. For instance, registering the adapter globally doesn't affect already existing connections.

Example:

```
>>> from psycopg.types.range import Range, RangeInfo, register_range

>>> conn.execute("CREATE TYPE strrange AS RANGE (SUBTYPE = text)")
>>> info = RangeInfo.fetch(conn, "strrange")
>>> register_range(info, conn)

>>> conn.execute("SELECT pg_typeof(%s)", [Range("a", "z")]).fetchone()[0]
```

(continues on next page)

(continued from previous page)

```
'strrange'

>>> conn.execute("SELECT '[a,z]'::strrange").fetchone()[0]
Range('a', 'z', '[]')
```

Multirange adaptation

Since PostgreSQL 14, every range type is associated with a multirange, a type representing a disjoint set of ranges. A multirange is automatically available for every range, built-in and user-defined.

All the PostgreSQL range types are loaded as the *Multirange* Python type, which is a mutable sequence of *Range* elements.

class psycopg.types.multirange.**Multirange**(*items=()*)

Python representation for a PostgreSQL multirange type.

Parameters
items (Iterable[*Range*[TypeVar(T)]]) – Sequence of ranges to initialise the object.

This Python type is only used to pass and retrieve multirange values to and from PostgreSQL and doesn't attempt to replicate the PostgreSQL multirange features: overlapping items are not merged, empty ranges are not discarded, the items are not ordered, the behaviour of multirange operators is not replicated in Python.

PostgreSQL will perform normalisation on `Multirange` objects used as query parameters, so, when they are fetched back, they will be found ordered, with overlapping ranges merged, etc.

`Multirange` objects are a `MutableSequence` and are totally ordered: they behave pretty much like a list of `Range`. Like Range, they are `Generic` on the subtype of their range, so you can declare a variable to be `Multirange[date]` and mypy will complain if you try to add it a `Range[Decimal]`.

Like for *Range*, built-in multirange objects are adapted automatically: if a `Multirange` object contains `Range` with `date` bounds, it is dumped using the `datemultirange` OID, and `datemultirange` values are loaded back as `Multirange[date]`.

If you have created your own range type you can use *MultirangeInfo* and *register_multirange()* to associate the resulting multirange type with its subtype and make it work like the builtin ones.

class psycopg.types.multirange.**MultirangeInfo**(*name*, *oid*, *array_oid*, ***, *regtype=''*, *range_oid*, *subtype_oid*)

Manage information about a multirange type.

`MultirangeInfo` is a *TypeInfo* subclass: check its documentation for generic details, especially the *fetch()* method.

psycopg.types.multirange.**register_multirange**(*info*, *context=None*)

Register the adapters to load and dump a multirange type.

Parameters

- **info** (*MultirangeInfo*) – The object with the information about the range to register.
- **context** (Optional[*AdaptContext*]) – The context where to register the adapters. If `None`, register it globally.

Register loaders so that loading data of this type will result in a `Range` with bounds parsed as the right subtype.

Note: Registering the adapters doesn't affect objects already created, even if they are children of the registered context. For instance, registering the adapter globally doesn't affect already existing connections.

Example:

```
>>> from psycopg.types.multirange import \
...     Multirange, MultirangeInfo, register_multirange
>>> from psycopg.types.range import Range

>>> conn.execute("CREATE TYPE strrange AS RANGE (SUBTYPE = text)")
>>> info = MultirangeInfo.fetch(conn, "strmultirange")
>>> register_multirange(info, conn)

>>> rec = conn.execute(
...     "SELECT pg_typeof(%(mr)s), %(mr)s",
...     {"mr": Multirange([Range("a", "q"), Range("l", "z")])}).fetchone()

>>> rec[0]
'strmultirange'
>>> rec[1]
Multirange([Range('a', 'z', '[)')])
```

Hstore adaptation

The `hstore` data type is a key-value store embedded in PostgreSQL. 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 keys and values are supported. `None` is also allowed as value but not as a key.

In order to use the `hstore` data type it is necessary to load it in a database using:

```
=# CREATE EXTENSION hstore;
```

Because `hstore` is distributed as a contrib module, its oid is not well known, so it is necessary to use `TypeInfo.fetch()` to query the database and get its oid. The resulting object can be passed to `register_hstore()` to configure dumping `dict` to `hstore` and parsing `hstore` back to `dict`, in the context where the adapter is registered.

psycopg.types.hstore.**register_hstore**(*info*, *context=None*)

Register the adapters to load and dump hstore.

Parameters

- **info** (*TypeInfo*) – The object with the information about the hstore type.
- **context** (Optional[*AdaptContext*]) – The context where to register the adapters. If `None`, register it globally.

Note: Registering the adapters doesn't affect objects already created, even if they are children of the registered context. For instance, registering the adapter globally doesn't affect already existing connections.

Example:

```
>>> from psycopg.types import TypeInfo
>>> from psycopg.types.hstore import register_hstore

>>> info = TypeInfo.fetch(conn, "hstore")
>>> register_hstore(info, conn)

>>> conn.execute("SELECT pg_typeof(%s)", [{"a": "b"}]).fetchone()[0]
'hstore'

>>> conn.execute("SELECT 'foo => bar'::hstore").fetchone()[0]
{'foo': 'bar'}
```

Geometry adaptation using Shapely

When using the PostGIS extension, it can be useful to retrieve geometry values and have them automatically converted to Shapely instances. Likewise, you may want to store such instances in the database and have the conversion happen automatically.

Warning: Psycopg doesn't have a dependency on the `shapely` package: you should install the library as an additional dependency of your project.

Warning: This module is experimental and might be changed in the future according to users' feedback.

Since PostgGIS is an extension, the `geometry` type oid is not well known, so it is necessary to use `TypeInfo.fetch()` to query the database and find it. The resulting object can be passed to `register_shapely()` to configure dumping shape instances to `geometry` columns and parsing `geometry` data back to `shape` instances, in the context where the adapters are registered.

psycopg.types.shapely.**register_shapely**()

Register Shapely dumper and loaders.

After invoking this function on an adapter, the queries retrieving PostGIS geometry objects will return Shapely's shape object instances both in text and binary mode.

Similarly, shape objects can be sent to the database.

This requires the Shapely library to be installed.

Parameters

- **info** – The object with the information about the geometry type.
- **context** – The context where to register the adapters. If `None`, register it globally.

Note: Registering the adapters doesn't affect objects already created, even if they are children of the registered context. For instance, registering the adapter globally doesn't affect already existing connections.

Example:

```
>>> from psycopg.types import TypeInfo
>>> from psycopg.types.shapely import register_shapely
>>> from shapely.geometry import Point

>>> info = TypeInfo.fetch(conn, "geometry")
>>> register_shapely(info, conn)

>>> conn.execute("SELECT pg_typeof(%s)", [Point(1.2, 3.4)]).fetchone()[0]
'geometry'

>>> conn.execute("""
... SELECT ST_GeomFromGeoJSON('{
...     "type":"Point",
...     "coordinates":[-48.23456,20.12345]}')
... """).fetchone()[0]
<shapely.geometry.multipolygon.MultiPolygon object at 0x7fb131f3cd90>
```

Notice that, if the geometry adapters are registered on a specific object (a connection or cursor), other connections and cursors will be unaffected:

```
>>> conn2 = psycopg.connect(CONN_STR)
>>> conn2.execute("""
... SELECT ST_GeomFromGeoJSON('{
...     "type":"Point",
...     "coordinates":[-48.23456,20.12345]}')
... """).fetchone()[0]
'0101000020E61000009279E40F061E48C0F2B0506B9A1F3440'
```

1.1.6 Transactions management

Psycopg has a behaviour that may seem surprising compared to **psql**: by default, any database operation will start a new transaction. As a consequence, changes made by any cursor of the connection will not be visible until *Connection.commit()* is called, and will be discarded by *Connection.rollback()*. The following operation on the same connection will start a new transaction.

If a database operation fails, the server will refuse further commands, until a `rollback()` is called.

If the cursor is closed with a transaction open, no COMMIT command is sent to the server, which will then discard the connection. Certain middleware (such as PgBouncer) will also discard a connection left in transaction state, so, if possible you will want to commit or rollback a connection before finishing working with it.

An example of what will happen, the first time you will use Psycopg (and to be disappointed by it), is likely:

```
conn = psycopg.connect()

# Creating a cursor doesn't start a transaction or affect the connection
# in any way.
cur = conn.cursor()

cur.execute("SELECT count(*) FROM my_table")
# This function call executes:
# - BEGIN
# - SELECT count(*) FROM my_table
```

(continues on next page)

(continued from previous page)

```
# So now a transaction has started.

# If your program spends a long time in this state, the server will keep
# a connection "idle in transaction", which is likely something undesired

cur.execute("INSERT INTO data VALUES (%s)", ("Hello",))
# This statement is executed inside the transaction

conn.close()
# No COMMIT was sent: the INSERT was discarded.
```

There are a few things going wrong here, let's see how they can be improved.

One obvious problem after the run above is that, firing up **psql**, you will see no new record in the table `data`. One way to fix the problem is to call `conn.commit()` before closing the connection. Thankfully, if you use the *connection context*, Psycopg will commit the connection at the end of the block (or roll it back if the block is exited with an exception):

The code modified using a connection context will result in the following sequence of database statements:

```
with psycopg.connect() as conn:

    cur = conn.cursor()

    cur.execute("SELECT count(*) FROM my_table")
    # This function call executes:
    # - BEGIN
    # - SELECT count(*) FROM my_table
    # So now a transaction has started.

    cur.execute("INSERT INTO data VALUES (%s)", ("Hello",))
    # This statement is executed inside the transaction

# No exception at the end of the block:
# COMMIT is executed.
```

This way we don't have to remember to call neither `close()` nor `commit()` and the database operations actually have a persistent effect. The code might still do something you don't expect: keep a transaction from the first operation to the connection closure. You can have a finer control over the transactions using an *autocommit transaction* and/or *transaction contexts*.

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.

Hint: If a database operation fails with an error message such as *InFailedSqlTransaction: current transaction is aborted, commands ignored until end of transaction block*, it means that **a previous operation failed** and the database session is in a state of error. You need to call `rollback()` if you want to keep on using the same connection.

Autocommit transactions

The manual commit requirement can be suspended using *autocommit*, either as connection attribute or as *connect()* parameter. This may be required to run operations that cannot be executed inside a transaction, such as `CREATE DATABASE`, `VACUUM`, `CALL` on stored procedures using transaction control.

With an autocommit transaction, the above sequence of operation results in:

```
with psycopg.connect(autocommit=True) as conn:

    cur = conn.cursor()

    cur.execute("SELECT count(*) FROM my_table")
    # This function call now only executes:
    # - SELECT count(*) FROM my_table
    # and no transaction starts.

    cur.execute("INSERT INTO data VALUES (%s)", ("Hello",))
    # The result of this statement is persisted immediately by the database

# The connection is closed at the end of the block but, because it is not
# in a transaction state, no COMMIT is executed.
```

An autocommit transaction behaves more as someone coming from **psql** would expect. This has a beneficial performance effect, because less queries are sent and less operations are performed by the database. The statements, however, are not executed in an atomic transaction; if you need to execute certain operations inside a transaction, you can achieve that with an autocommit connection too, using an explicit *transaction block*.

Transaction contexts

A more transparent way to make sure that transactions are finalised at the right time is to use `with` *Connection.transaction()* to create a transaction context. When the context is entered, a transaction is started; when leaving the context the transaction is committed, or it is rolled back if an exception is raised inside the block.

Continuing the example above, if you want to use an autocommit connection but still wrap selected groups of commands inside an atomic transaction, you can use a `transaction()` context:

```
with psycopg.connect(autocommit=True) as conn:

    cur = conn.cursor()

    cur.execute("SELECT count(*) FROM my_table")
    # The connection is autocommit, so no BEGIN executed.

    with conn.transaction():
        # BEGIN is executed, a transaction started

        cur.execute("INSERT INTO data VALUES (%s)", ("Hello",))
        cur.execute("INSERT INTO times VALUES (now())")
        # These two operation run atomically in the same transaction

    # COMMIT is executed at the end of the block.
    # The connection is in idle state again.
```

(continues on next page)

(continued from previous page)

```
# The connection is closed at the end of the block.
```

Note that connection blocks can also be used with non-autocommit connections: in this case you still need to pay attention to eventual transactions started automatically. If an operation starts an implicit transaction, a `transaction()` block will only manage *a savepoint sub-transaction*, leaving the caller to deal with the main transaction, as explained in *Transactions management*:

```
conn = psycopg.connect()

cur = conn.cursor()

cur.execute("SELECT count(*) FROM my_table")
# This function call executes:
# - BEGIN
# - SELECT count(*) FROM my_table
# So now a transaction has started.

with conn.transaction():
    # The block starts with a transaction already open, so it will execute
    # - SAVEPOINT

    cur.execute("INSERT INTO data VALUES (%s)", ("Hello",))

# The block was executing a sub-transaction so on exit it will only run:
# - RELEASE SAVEPOINT
# The transaction is still on.

conn.close()
# No COMMIT was sent: the INSERT was discarded.
```

If a `transaction()` block starts when no transaction is active then it will manage a proper transaction. In essence, a transaction context tries to leave a connection in the state it found it, and leaves you to deal with the wider context.

Hint: The interaction between non-autocommit transactions and transaction contexts is probably surprising. Although the non-autocommit default is what's demanded by the DBAPI, the personal preference of several experienced developers is to:

- use a connection block: `with psycopg.connect(...) as conn`;
- use an autocommit connection, either passing `autocommit=True` as `connect()` parameter or setting the attribute `conn.autocommit = True`;
- use `with conn.transaction()` blocks to manage transactions only where needed.

Nested transactions

Transaction blocks can be also nested (internal transaction blocks are implemented using SAVEPOINT): an exception raised inside an inner block has a chance of being handled and not completely fail outer operations. The following is an example where a series of operations interact with the database: operations are allowed to fail; at the end we also want to store the number of operations successfully processed.

```
with conn.transaction() as tx1:
    num_ok = 0
    for operation in operations:
        try:
            with conn.transaction() as tx2:
                unreliable_operation(conn, operation)
        except Exception:
            logger.exception(f"{operation} failed")
        else:
            num_ok += 1

    save_number_of_successes(conn, num_ok)
```

If `unreliable_operation()` causes an error, including an operation causing a database error, all its changes will be reverted. The exception bubbles up outside the block: in the example it is intercepted by the `try` so that the loop can complete. The outermost block is unaffected (unless other errors happen there).

You can also write code to explicitly roll back any currently active transaction block, by raising the *`Rollback`* exception. The exception "jumps" to the end of a transaction block, rolling back its transaction but allowing the program execution to continue from there. By default the exception rolls back the innermost transaction block, but any current block can be specified as the target. In the following example, a hypothetical `CancelCommand` may stop the processing and cancel any operation previously performed, but not entirely committed yet.

```
from psycopg import Rollback

with conn.transaction() as outer_tx:
    for command in commands():
        with conn.transaction() as inner_tx:
            if isinstance(command, CancelCommand):
                raise Rollback(outer_tx)
            process_command(command)

# If `Rollback` is raised, it would propagate only up to this block,
# and the program would continue from here with no exception.
```

Transaction characteristics

You can set transaction parameters for the transactions that Psycopg handles. They affect the transactions started implicitly by non-autocommit transactions and the ones started explicitly by *`Connection.transaction()`* for both autocommit and non-autocommit transactions. Leaving these parameters as `None` will use the server's default behaviour (which is controlled by server settings such as default_transaction_isolation).

In order to set these parameters you can use the connection attributes *`isolation_level`*, *`read_only`*, *`deferrable`*. For async connections you must use the equivalent *`set_isolation_level()`* method and similar. The parameters can only be changed if there isn't a transaction already active on the connection.

Warning: Applications running at *REPEATABLE_READ* or *SERIALIZABLE* isolation level are exposed to serialization failures. In certain concurrent update cases, PostgreSQL will raise an exception looking like:

```
psycopg2.errors.SerializationFailure: could not serialize access
due to concurrent update
```

In this case the application must be prepared to repeat the operation that caused the exception.

Two-Phase Commit protocol support

New in version 3.1.

Psycopg exposes the two-phase commit features available in PostgreSQL implementing the two-phase commit extensions proposed by the DBAPI.

The DBAPI 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 DBAPI 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 *Two-Phase Commit support methods*.

1.1.7 Using COPY TO and COPY FROM

Psycopg allows to operate with PostgreSQL COPY protocol. `COPY` is one of the most efficient ways to load data into the database (and to modify it, with some SQL creativity).

Copy is supported using the *Cursor.copy()* method, passing it a query of the form `COPY ... FROM STDIN` or `COPY ... TO STDOUT`, and managing the resulting *Copy* object in a `with` block:

```
with cursor.copy("COPY table_name (col1, col2) FROM STDIN") as copy:
    # pass data to the 'copy' object using write()/write_row()
```

You can compose a COPY statement dynamically by using objects from the *psycopg.sql* module:

```
with cursor.copy(
    sql.SQL("COPY {} TO STDOUT").format(sql.Identifier("table_name"))
) as copy:
    # read data from the 'copy' object using read()/read_row()
```

Changed in version 3.1: You can also pass parameters to `copy()`, like in *execute()*:

```
with cur.copy("COPY (SELECT * FROM table_name LIMIT %s) TO STDOUT", (3,)) as copy:
    # expect no more than three records
```

The connection is subject to the usual transaction behaviour, so, unless the connection is in autocommit, at the end of the COPY operation you will still have to commit the pending changes and you can still roll them back. See *Transactions management* for details.

Writing data row-by-row

Using a copy operation you can load data into the database from any Python iterable (a list of tuples, or any iterable of sequences): the Python values are adapted as they would be in normal querying. To perform such operation use a `COPY ... FROM STDIN` with *Cursor.copy()* and use *write_row()* on the resulting object in a `with` block. On exiting the block the operation will be concluded:

```
records = [(10, 20, "hello"), (40, None, "world")]

with cursor.copy("COPY sample (col1, col2, col3) FROM STDIN") as copy:
    for record in records:
        copy.write_row(record)
```

If an exception is raised inside the block, the operation is interrupted and the records inserted so far are discarded.

In order to read or write from `Copy` row-by-row you must not specify `COPY` options such as `FORMAT CSV`, `DELIMITER`, `NULL`: please leave these details alone, thank you :)

Reading data row-by-row

You can also do the opposite, reading rows out of a `COPY ... TO STDOUT` operation, by iterating on *rows()*. However this is not something you may want to do normally: usually the normal query process will be easier to use.

PostgreSQL, currently, doesn't give complete type information on `COPY TO`, so the rows returned will have unparsed data, as strings or bytes, according to the format.

```
with cur.copy("COPY (VALUES (10::int, current_date)) TO STDOUT") as copy:
    for row in copy.rows():
        print(row)  # return unparsed data: ('10', '2046-12-24')
```

You can improve the results by using *set_types()* before reading, but you have to specify them yourself.

```
with cur.copy("COPY (VALUES (10::int, current_date)) TO STDOUT") as copy:
    copy.set_types(["int4", "date"])
    for row in copy.rows():
        print(row)  # (10, datetime.date(2046, 12, 24))
```

Copying block-by-block

If data is already formatted in a way suitable for copy (for instance because it is coming from a file resulting from a previous `COPY TO` operation) it can be loaded into the database using *Copy.write()* instead.

```
with open("data", "r") as f:
    with cursor.copy("COPY data FROM STDIN") as copy:
        while data := f.read(BLOCK_SIZE):
            copy.write(data)
```

In this case you can use any `COPY` option and format, as long as the input data is compatible with what the operation in `copy()` expects. Data can be passed as `str`, if the copy is in `FORMAT TEXT`, or as `bytes`, which works with both `FORMAT TEXT` and `FORMAT BINARY`.

In order to produce data in `COPY` format you can use a `COPY ... TO STDOUT` statement and iterate over the resulting *Copy* object, which will produce a stream of `bytes` objects:

```
with open("data.out", "wb") as f:
    with cursor.copy("COPY table_name TO STDOUT") as copy:
        for data in copy:
            f.write(data)
```

Binary copy

Binary copy is supported by specifying `FORMAT BINARY` in the `COPY` statement. In order to import binary data using *write_row()*, all the types passed to the database must have a binary dumper registered; this is not necessary if the data is copied *block-by-block* using *write()*.

Warning: PostgreSQL is particularly finicky when loading data in binary mode and will apply **no cast rules**. This means, for example, that passing the value 100 to an `integer` column **will fail**, because Psycopg will pass it as a `smallint` value, and the server will reject it because its size doesn't match what expected.

You can work around the problem using the *set_types()* method of the `Copy` object and specifying carefully the types to load.

See also:

See *Binary parameters and results* for further info about binary querying.

Asynchronous copy support

Asynchronous operations are supported using the same patterns as above, using the objects obtained by an *AsyncConnection*. For instance, if `f` is an object supporting an asynchronous `read()` method returning `COPY` data, a fully-async copy operation could be:

```
async with cursor.copy("COPY data FROM STDIN") as copy:
    while data := await f.read():
        await copy.write(data)
```

The *AsyncCopy* object documentation describes the signature of the asynchronous methods and the differences from its sync *Copy* counterpart.

See also:

See *Asynchronous operations* for further info about using async objects.

Example: copying a table across servers

In order to copy a table, or a portion of a table, across servers, you can use two COPY operations on two different connections, reading from the first and writing to the second.

```
with psycopg.connect(dsn_src) as conn1, psycopg.connect(dsn_tgt) as conn2:
    with conn1.cursor().copy("COPY src TO STDOUT (FORMAT BINARY)") as copy1:
        with conn2.cursor().copy("COPY tgt FROM STDIN (FORMAT BINARY)") as copy2:
            for data in copy1:
                copy2.write(data)
```

Using `FORMAT BINARY` usually gives a performance boost, but it only works if the source and target schema are *perfectly identical*. If the tables are only *compatible* (for example, if you are copying an `integer` field into a `bigint` destination field) you should omit the `BINARY` option and perform a text-based copy. See *Binary copy* for details.

The same pattern can be adapted to use *async objects* in order to perform an *async copy*.

1.1.8 Differences from psycopg2

Psycopg 3 uses the common DBAPI structure of many other database adapters and tries to behave as close as possible to `psycopg2`. There are however a few differences to be aware of.

Note: Most of the times, the workarounds suggested here will work with both Psycopg 2 and 3, which could be useful if you are porting a program or writing a program that should work with both Psycopg 2 and 3.

Server-side binding

Psycopg 3 sends the query and the parameters to the server separately, instead of merging them on the client side. Server-side binding works for normal `SELECT` and data manipulation statements (`INSERT`, `UPDATE`, `DELETE`), but it doesn't work with many other statements. For instance, it doesn't work with `SET` or with `NOTIFY`:

```
>>> conn.execute("SET TimeZone TO %s", ["UTC"])
Traceback (most recent call last):
...
psycopg.errors.SyntaxError: syntax error at or near "$1"
LINE 1: SET TimeZone TO $1
                        ^

>>> conn.execute("NOTIFY %s, %s", ["chan", 42])
Traceback (most recent call last):
...
psycopg.errors.SyntaxError: syntax error at or near "$1"
LINE 1: NOTIFY $1, $2
               ^
```

and with any data definition statement:

```
>>> conn.execute("CREATE TABLE foo (id int DEFAULT %s)", [42])
Traceback (most recent call last):
...
psycopg.errors.UndefinedParameter: there is no parameter $1
LINE 1: CREATE TABLE foo (id int DEFAULT $1)
                                         ^
```

Sometimes, PostgreSQL offers an alternative: for instance the set_config() function can be used instead of the `SET` statement, the pg_notify() function can be used instead of `NOTIFY`:

```
>>> conn.execute("SELECT set_config('TimeZone', %s, false)", ["UTC"])

>>> conn.execute("SELECT pg_notify(%s, %s)", ["chan", "42"])
```

If this is not possible, you must merge the query and the parameter on the client side. You can do so using the *psycopg.sql* objects:

```
>>> from psycopg import sql

>>> cur.execute(sql.SQL("CREATE TABLE foo (id int DEFAULT {})").format(42))
```

or creating a *client-side binding cursor* such as *ClientCursor*:

```
>>> cur = ClientCursor(conn)
>>> cur.execute("CREATE TABLE foo (id int DEFAULT %s)", [42])
```

if you need `ClientCursor` often, you can set the *Connection.cursor_factory* to have them created by default by *Connection.cursor()*. This way, Psycopg 3 will behave largely the same way of Psycopg 2.

Note that, using `ClientCursor` parameters, you can only specify query values (aka *the strings that go in single quotes*). If you need to parametrize different parts of a statement, you must use the `psycopg.sql` module:

```
>>> from psycopg import sql

# This will quote the user and the password using the right quotes
>>> conn.execute(
...     sql.SQL("ALTER USER {} SET PASSWORD {}")
...     .format(sql.Identifier(username), password))
```

Multiple statements in the same query

As a consequence of using *server-side bindings*, when parameters are used, it is not possible to execute several statements in the same `execute()` call, separating them with a semicolon:

```
>>> conn.execute(
...     "INSERT INTO foo VALUES (%s); INSERT INTO foo VALUES (%s)",
...     (10, 20))
Traceback (most recent call last):
...
psycopg.errors.SyntaxError: cannot insert multiple commands into a prepared statement
```

One obvious way to work around the problem is to use several `execute()` calls.

There is no such limitation if no parameters are used. As a consequence, you can compose a multiple query on the client side and run them all in the same `execute()` call, using the *psycopg.sql* objects:

```
>>> from psycopg import sql
>>> conn.execute(
...     sql.SQL("INSERT INTO foo VALUES ({}); INSERT INTO foo values ({})"
...     .format(10, 20))
```

or a *client-side binding cursor*:

```
>>> cur = psycopg.ClientCursor(conn)
>>> cur.execute(
...     "INSERT INTO foo VALUES (%s); INSERT INTO foo VALUES (%s)",
...     (10, 20))
```

Note that statements that must be run outside a transaction (such as `CREATE DATABASE`) can never be executed in batch with other statements, even if the connection is in autocommit mode:

```
>>> conn.autocommit = True
>>> conn.execute("CREATE DATABASE foo; SELECT 1")
Traceback (most recent call last):
...
psycopg.errors.ActiveSqlTransaction: CREATE DATABASE cannot run inside a transaction␣
↪block
```

This happens because PostgreSQL will wrap multiple statements in a transaction itself and is different from how **psql** behaves (**psql** will split the queries on semicolons and send them separately). This is not new in Psycopg 3: the same limitation is present in `psycopg2` too.

Different cast rules

In rare cases, especially around variadic functions, PostgreSQL might fail to find a function candidate for the given data types:

```
>>> conn.execute("SELECT json_build_array(%s, %s)", ["foo", "bar"])
Traceback (most recent call last):
...
psycopg.errors.IndeterminateDatatype: could not determine data type of parameter $1
```

This can be worked around specifying the argument types explicitly via a cast:

```
>>> conn.execute("SELECT json_build_array(%s::text, %s::text)", ["foo", "bar"])
```

You cannot use `IN %s` with a tuple

`IN` cannot be used with a tuple as single parameter, as was possible with `psycopg2`:

```
>>> conn.execute("SELECT * FROM foo WHERE id IN %s", [(10,20,30)])
Traceback (most recent call last):
...
psycopg.errors.SyntaxError: syntax error at or near "$1"
LINE 1: SELECT * FROM foo WHERE id IN $1
                                      ^
```

What you can do is to use the `= ANY()` construct and pass the candidate values as a list instead of a tuple, which will be adapted to a PostgreSQL array:

```
>>> conn.execute("SELECT * FROM foo WHERE id = ANY(%s)", [[10,20,30]])
```

Note that `ANY()` can be used with `psycopg2` too, and has the advantage of accepting an empty list of values too as argument, which is not supported by the `IN` operator instead.

Different adaptation system

The adaptation system has been completely rewritten, in order to address server-side parameters adaptation, but also to consider performance, flexibility, ease of customization.

The default behaviour with builtin data should be *what you would expect*. If you have customised the way to adapt data, or if you are managing your own extension types, you should look at the *new adaptation system*.

See also:

- *Adapting basic Python types* for the basic behaviour.
- *Data adaptation configuration* for more advanced use.

Copy is no longer file-based

`psycopg2` exposes a few copy methods to interact with PostgreSQL `COPY`. Their file-based interface doesn't make it easy to load dynamically-generated data into a database.

There is now a single *`copy()`* method, which is similar to `psycopg2` `copy_expert()` in accepting a free-form `COPY` command and returns an object to read/write data, block-wise or record-wise. The different usage pattern also enables `COPY` to be used in async interactions.

See also:

See *Using COPY TO and COPY FROM* for the details.

`with` connection

In `psycopg2`, using the syntax with connection, only the transaction is closed, not the connection. This behaviour is surprising for people used to several other Python classes wrapping resources, such as files.

In Psycopg 3, using *with connection* will close the connection at the end of the `with` block, making handling the connection resources more familiar.

In order to manage transactions as blocks you can use the *`Connection.transaction()`* method, which allows for finer control, for instance to use nested transactions.

See also:

See *Transaction contexts* for details.

`callproc()` is gone

`cursor.callproc()` is not implemented. The method has a simplistic semantic which doesn't account for PostgreSQL positional parameters, procedures, set-returning functions… Use a normal *`execute()`* with `SELECT function_name(...)` or `CALL procedure_name(...)` instead.

`client_encoding` is gone

Psycopg automatically uses the database client encoding to decode data to Unicode strings. Use *`ConnectionInfo.encoding`* if you need to read the encoding. You can select an encoding at connection time using the `client_encoding` connection parameter and you can change the encoding of a connection by running a `SET client_encoding` statement… But why would you?

No default infinity dates handling

PostgreSQL can represent a much wider range of dates and timestamps than Python. While Python dates are limited to the years between 1 and 9999 (represented by constants such as `datetime.date.min` and `max`), PostgreSQL dates extend to BC dates and past the year 10K. Furthermore PostgreSQL can also represent symbolic dates "infinity", in both directions.

In psycopg2, by default, infinity dates and timestamps map to 'date.max' and similar constants. This has the problem of creating a non-bijective mapping (two Postgres dates, infinity and 9999-12-31, both map to the same Python date). There is also the perversity that valid Postgres dates, greater than Python `date.max` but arguably lesser than infinity, will still overflow.

In Psycopg 3, every date greater than year 9999 will overflow, including infinity. If you would like to customize this mapping (for instance flattening every date past Y10K on `date.max`) you can subclass and adapt the appropriate loaders: take a look at *this example* to see how.

What's new in Psycopg 3

- *Asynchronous support*
- *Server-side parameters binding*
- *Prepared statements*
- *Binary communication*
- *Python-based COPY support*
- *Support for static typing*
- *A redesigned connection pool*
- *Direct access to the libpq functionalities*

1.2 More advanced topics

Once you have familiarised yourself with the *Psycopg basic operations*, you can take a look at the chapter of this section for more advanced usages.

1.2.1 Asynchronous operations

Psycopg *Connection* and *Cursor* have counterparts *AsyncConnection* and *AsyncCursor* supporting an asyncio interface.

The design of the asynchronous objects is pretty much the same of the sync ones: in order to use them you will only have to scatter the `await` keyword here and there.

```
async with await psycopg.AsyncConnection.connect(
        "dbname=test user=postgres") as aconn:
    async with aconn.cursor() as acur:
        await acur.execute(
            "INSERT INTO test (num, data) VALUES (%s, %s)",
            (100, "abc'def"))
        await acur.execute("SELECT * FROM test")
        await acur.fetchone()
        # will return (1, 100, "abc'def")
        async for record in acur:
            print(record)
```

Changed in version 3.1: *AsyncConnection.connect()* performs DNS name resolution in a non-blocking way.

Warning: Before version 3.1, *AsyncConnection.connect()* may still block on DNS name resolution. To avoid that you should set the hostaddr connection parameter, or use the *resolve_hostaddr_async()* to do it automatically.

Warning: On Windows, Psycopg is not compatible with the default `ProactorEventLoop`. Please use a different loop, for instance the `SelectorEventLoop`.

For instance, you can use, early in your program:

```
asyncio.set_event_loop_policy(
    asyncio.WindowsSelectorEventLoopPolicy()
)
```

`with` async connections

As seen in *the basic usage*, connections and cursors can act as context managers, so you can run:

```
with psycopg.connect("dbname=test user=postgres") as conn:
    with conn.cursor() as cur:
        cur.execute(...)
    # the cursor is closed upon leaving the context
# the transaction is committed, the connection closed
```

For asynchronous connections it's *almost* what you'd expect, but not quite. Please note that *connect()* and *cursor() don't return a context*: they are both factory methods which return *an object which can be used as a context*. That's because there are several use cases where it's useful to handle the objects manually and only `close()` them when required.

As a consequence you cannot use `async with connect()`: you have to do it in two steps instead, as in

```
aconn = await psycopg.AsyncConnection.connect()
async with aconn:
    async with aconn.cursor() as cur:
        await cur.execute(...)
```

which can be condensed into `async with await`:

```
async with await psycopg.AsyncConnection.connect() as aconn:
    async with aconn.cursor() as cur:
        await cur.execute(...)
```

... but no less than that: you still need to do the double async thing.

Note that the *AsyncConnection.cursor()* function is not an `async` function (it never performs I/O), so you don't need an `await` on it; as a consequence you can use the normal `async with` context manager.

Interrupting async operations using Ctrl-C

If a long running operation is interrupted by a Ctrl-C on a normal connection running in the main thread, the operation will be cancelled and the connection will be put in error state, from which can be recovered with a normal *rollback()*.

If the query is running in an async connection, a Ctrl-C will be likely intercepted by the async loop and interrupt the whole program. In order to emulate what normally happens with blocking connections, you can use asyncio's add_signal_handler(), to call *Connection.cancel()*:

```
import asyncio
import signal

async with await psycopg.AsyncConnection.connect() as conn:
    loop.add_signal_handler(signal.SIGINT, conn.cancel)
    ...
```

Server messages

PostgreSQL can send, together with the query results, informative messages about the operation just performed, such as warnings or debug information. Notices may be raised even if the operations are successful and don't indicate an error. You are probably familiar with some of them, because they are reported by **psql**:

```
$ psql
=# ROLLBACK;
WARNING:  there is no transaction in progress
ROLLBACK
```

Messages can be also sent by the PL/pgSQL 'RAISE' statement (at a level lower than EXCEPTION, otherwise the appropriate *DatabaseError* will be raised). The level of the messages received can be controlled using the client_min_messages setting.

By default, the messages received are ignored. If you want to process them on the client you can use the *Connection.add_notice_handler()* function to register a function that will be invoked whenever a message is received. The message is passed to the callback as a *Diagnostic* instance, containing all the information passed by the server, such as the message text and the severity. The object is the same found on the *diag* attribute of the errors raised by the server:

```
>>> import psycopg

>>> def log_notice(diag):
...     print(f"The server says: {diag.severity} - {diag.message_primary}")

>>> conn = psycopg.connect(autocommit=True)
>>> conn.add_notice_handler(log_notice)

>>> cur = conn.execute("ROLLBACK")
The server says: WARNING - there is no transaction in progress
>>> print(cur.statusmessage)
ROLLBACK
```

Warning: The `Diagnostic` object received by the callback should not be used after the callback function terminates, because its data is deallocated after the callbacks have been processed. If you need to use the information later please extract the attributes requested and forward them instead of forwarding the whole `Diagnostic` object.

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.

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 as instances of *Notify*. If you are reserving a connection only to receive notifications, the simplest way is to consume the *Connection.notifies* generator. The generator can be stopped using `close()`.

Note: You don't need an *AsyncConnection* to handle notifications: a normal blocking *Connection* is perfectly valid.

The following example will print notifications and stop when one containing the `stop` message is received.

```
import psycopg
conn = psycopg.connect("", autocommit=True)
conn.execute("LISTEN mychan")
gen = conn.notifies()
for notify in gen:
    print(notify)
    if notify.payload == "stop":
        gen.close()
print("there, I stopped")
```

If you run some `NOTIFY` in a **psql** session:

```
=# NOTIFY mychan, 'hello';
NOTIFY
=# NOTIFY mychan, 'hey';
NOTIFY
=# NOTIFY mychan, 'stop';
NOTIFY
```

You may get output from the Python process such as:

```
Notify(channel='mychan', payload='hello', pid=961823)
Notify(channel='mychan', payload='hey', pid=961823)
Notify(channel='mychan', payload='stop', pid=961823)
there, I stopped
```

Alternatively, you can use *add_notify_handler()* to register a callback function, which will be invoked whenever a notification is received, during the normal query processing; you will be then able to use the connection normally. Please note that in this case notifications will not be received immediately, but only during a connection operation, such as a query.

```
conn.add_notify_handler(lambda n: print(f"got this: {n}"))

# meanwhile in psql...
# =# NOTIFY mychan, 'hey';
# NOTIFY

print(conn.execute("SELECT 1").fetchone())
# got this: Notify(channel='mychan', payload='hey', pid=961823)
# (1,)
```

Detecting disconnections

Sometimes it is useful to detect immediately when the connection with the database is lost. One brutal way to do so is to poll a connection in a loop running an endless stream of `SELECT 1`... *Don't* do so: polling is *so* out of fashion. Besides, it is inefficient (unless what you really want is a client-server generator of ones), it generates useless traffic and will only detect a disconnection with an average delay of half the polling time.

A more efficient and timely way to detect a server disconnection is to create an additional connection and wait for a notification from the OS that this connection has something to say: only then you can run some checks. You can dedicate a thread (or an asyncio task) to wait on this connection: such thread will perform no activity until awaken by the OS.

In a normal (non asyncio) program you can use the `selectors` module. Because the `Connection` implements a *fileno()* method you can just register it as a file-like object. You can run such code in a dedicated thread (and using a dedicated connection) if the rest of the program happens to have something else to do too.

```
import selectors

sel = selectors.DefaultSelector()
sel.register(conn, selectors.EVENT_READ)
while True:
    if not sel.select(timeout=60.0):
        continue  # No FD activity detected in one minute
```

(continues on next page)

(continued from previous page)

```
    # Activity detected. Is the connection still ok?
    try:
        conn.execute("SELECT 1")
    except psycopg.OperationalError:
        # You were disconnected: do something useful such as panicking
        logger.error("we lost our database!")
        sys.exit(1)
```

In an `asyncio` program you can dedicate a `Task` instead and do something similar using `add_reader`:

```
import asyncio

ev = asyncio.Event()
loop = asyncio.get_event_loop()
loop.add_reader(conn.fileno(), ev.set)

while True:
    try:
        await asyncio.wait_for(ev.wait(), 60.0)
    except asyncio.TimeoutError:
        continue  # No FD activity detected in one minute

    # Activity detected. Is the connection still ok?
    try:
        await conn.execute("SELECT 1")
    except psycopg.OperationalError:
        # Guess what happened
        ...
```

1.2.2 Static Typing

Psycopg source code is annotated according to **PEP 0484** type hints and is checked using the current version of Mypy in `--strict` mode.

If your application is checked using Mypy too you can make use of Psycopg types to validate the correct use of Psycopg objects and of the data returned by the database.

Generic types

Psycopg *`Connection`* and *`Cursor`* objects are `Generic` objects and support a `Row` parameter which is the type of the records returned.

By default methods such as *`Cursor.fetchall()`* return normal tuples of unknown size and content. As such, the *`connect()`* function returns an object of type `psycopg.Connection[Tuple[Any, ...]]` and *`Connection.cursor()`* returns an object of type `psycopg.Cursor[Tuple[Any, ...]]`. If you are writing generic plumbing code it might be practical to use annotations such as `Connection[Any]` and `Cursor[Any]`.

```
conn = psycopg.connect() # type is psycopg.Connection[Tuple[Any, ...]]

cur = conn.cursor()      # type is psycopg.Cursor[Tuple[Any, ...]]
```

(continues on next page)

(continued from previous page)

```
rec = cur.fetchone()     # type is Optional[Tuple[Any, ...]]

recs = cur.fetchall()    # type is List[Tuple[Any, ...]]
```

Type of rows returned

If you want to use connections and cursors returning your data as different types, for instance as dictionaries, you can use the `row_factory` argument of the *connect()* and the *cursor()* method, which will control what type of record is returned by the fetch methods of the cursors and annotate the returned objects accordingly. See *Row factories* for more details.

```
dconn = psycopg.connect(row_factory=dict_row)
# dconn type is psycopg.Connection[Dict[str, Any]]

dcur = conn.cursor(row_factory=dict_row)
dcur = dconn.cursor()
# dcur type is psycopg.Cursor[Dict[str, Any]] in both cases

drec = dcur.fetchone()
# drec type is Optional[Dict[str, Any]]
```

Example: returning records as Pydantic models

Using Pydantic it is possible to enforce static typing at runtime. Using a Pydantic model factory the code can be checked statically using Mypy and querying the database will raise an exception if the rows returned is not compatible with the model.

The following example can be checked with `mypy --strict` without reporting any issue. Pydantic will also raise a runtime error in case the `Person` is used with a query that returns incompatible data.

```
from datetime import date
from typing import Optional

import psycopg
from psycopg.rows import class_row
from pydantic import BaseModel

class Person(BaseModel):
    id: int
    first_name: str
    last_name: str
    dob: Optional[date]

def fetch_person(id: int) -> Person:
    with psycopg.connect() as conn:
        with conn.cursor(row_factory=class_row(Person)) as cur:
            cur.execute(
                """
                SELECT id, first_name, last_name, dob
                FROM (VALUES
```

(continues on next page)

(continued from previous page)

```
                    (1, 'John', 'Doe', '2000-01-01'::date),
                    (2, 'Jane', 'White', NULL)
                ) AS data (id, first_name, last_name, dob)
                WHERE id = %(id)s;
                """,
                {"id": id},
            )
            obj = cur.fetchone()

            # reveal_type(obj) would return 'Optional[Person]' here

            if not obj:
                raise KeyError(f"person {id} not found")

            # reveal_type(obj) would return 'Person' here

            return obj

for id in [1, 2]:
    p = fetch_person(id)
    if p.dob:
        print(f"{p.first_name} was born in {p.dob.year}")
    else:
        print(f"Who knows when {p.first_name} was born")
```

Checking literal strings in queries

The *execute()* method and similar should only receive a literal string as input, according to **PEP 675**. This means that the query should come from a literal string in your code, not from an arbitrary string expression.

For instance, passing an argument to the query should be done via the second argument to `execute()`, not by string composition:

```
def get_record(conn: psycopg.Connection[Any], id: int) -> Any:
    cur = conn.execute("SELECT * FROM my_table WHERE id = %s" % id)  # BAD!
    return cur.fetchone()

# the function should be implemented as:

def get_record(conn: psycopg.Connection[Any], id: int) -> Any:
    cur = conn.execute("select * FROM my_table WHERE id = %s", (id,))
    return cur.fetchone()
```

If you are composing a query dynamically you should use the *sql.SQL* object and similar to escape safely table and field names. The parameter of the `SQL()` object should be a literal string:

```
def count_records(conn: psycopg.Connection[Any], table: str) -> int:
    query = "SELECT count(*) FROM %s" % table  # BAD!
    return conn.execute(query).fetchone()[0]

# the function should be implemented as:
```

(continues on next page)

(continued from previous page)

```
def count_records(conn: psycopg.Connection[Any], table: str) -> int:
    query = sql.SQL("SELECT count(*) FROM {}").format(sql.Identifier(table))
    return conn.execute(query).fetchone()[0]
```

At the time of writing, no Python static analyzer implements this check (mypy doesn't implement it, Pyre does, but doesn't work with psycopg yet). Once the type checkers support will be complete, the above bad statements should be reported as errors.

1.2.3 Row factories

Cursor's `fetch*` methods, by default, return the records received from the database as tuples. This can be changed to better suit the needs of the programmer by using custom *row factories*.

The module *psycopg.rows* exposes several row factories ready to be used. For instance, if you want to return your records as dictionaries, you can use *dict_row*:

```
>>> from psycopg.rows import dict_row

>>> conn = psycopg.connect(DSN, row_factory=dict_row)

>>> conn.execute("select 'John Doe' as name, 33 as age").fetchone()
{'name': 'John Doe', 'age': 33}
```

The `row_factory` parameter is supported by the *connect()* method and the *cursor()* method. Later usage of `row_factory` overrides a previous one. It is also possible to change the *Connection.row_factory* or *Cursor.row_factory* attributes to change what they return:

```
>>> cur = conn.cursor(row_factory=dict_row)
>>> cur.execute("select 'John Doe' as name, 33 as age").fetchone()
{'name': 'John Doe', 'age': 33}

>>> from psycopg.rows import namedtuple_row
>>> cur.row_factory = namedtuple_row
>>> cur.execute("select 'John Doe' as name, 33 as age").fetchone()
Row(name='John Doe', age=33)
```

If you want to return objects of your choice you can use a row factory *generator*, for instance *class_row* or *args_row*, or you can *write your own row factory*:

```
>>> from dataclasses import dataclass

>>> @dataclass
... class Person:
...     name: str
...     age: int
...     weight: Optional[int] = None

>>> from psycopg.rows import class_row
>>> cur = conn.cursor(row_factory=class_row(Person))
>>> cur.execute("select 'John Doe' as name, 33 as age").fetchone()
Person(name='John Doe', age=33, weight=None)
```

Creating new row factories

A *row factory* is a callable that accepts a *Cursor* object and returns another callable, a *row maker*, which takes raw data (as a sequence of values) and returns the desired object.

The role of the row factory is to inspect a query result (it is called after a query is executed and properties such as *description* and *pgresult* are available on the cursor) and to prepare a callable which is efficient to call repeatedly (because, for instance, the names of the columns are extracted, sanitised, and stored in local variables).

Formally, these objects are represented by the *RowFactory* and *RowMaker* protocols.

`RowFactory` objects can be implemented as a class, for instance:

```
from typing import Any, Sequence
from psycopg import Cursor

class DictRowFactory:
    def __init__(self, cursor: Cursor[Any]):
        self.fields = [c.name for c in cursor.description]

    def __call__(self, values: Sequence[Any]) -> dict[str, Any]:
        return dict(zip(self.fields, values))
```

or as a plain function:

```
def dict_row_factory(cursor: Cursor[Any]) -> RowMaker[dict[str, Any]]:
    fields = [c.name for c in cursor.description]

    def make_row(values: Sequence[Any]) -> dict[str, Any]:
        return dict(zip(fields, values))

    return make_row
```

These can then be used by specifying a `row_factory` argument in *Connection.connect()*, *Connection.cursor()*, or by setting the *Connection.row_factory* attribute.

```
conn = psycopg.connect(row_factory=DictRowFactory)
cur = conn.execute("SELECT first_name, last_name, age FROM persons")
person = cur.fetchone()
print(f"{person['first_name']} {person['last_name']}")
```

1.2.4 Connection pools

A connection pool is an object managing a set of connections and allowing their use in functions needing one. Because the time to establish a new connection can be relatively long, keeping connections open can reduce latency.

This page explains a few basic concepts of Psycopg connection pool's behaviour. Please refer to the `ConnectionPool` object API for details about the pool operations.

Note: The connection pool objects are distributed in a package separate from the main *psycopg* package: use `pip install "psycopg[pool]"` or `pip install psycopg_pool` to make the *psycopg_pool* package available. See *Installing the connection pool*.

Pool life cycle

A simple way to use the pool is to create a single instance of it, as a global object, and to use this object in the rest of the program, allowing other functions, modules, threads to use it:

```
# module db.py in your program
from psycopg_pool import ConnectionPool

pool = ConnectionPool(conninfo, **kwargs)
# the pool starts connecting immediately.

# in another module
from .db import pool

def my_function():
    with pool.connection() as conn:
        conn.execute(...)
```

Ideally you may want to call `close()` when the use of the pool is finished. Failing to call `close()` at the end of the program is not terribly bad: probably it will just result in some warnings printed on stderr. However, if you think that it's sloppy, you could use the `atexit` module to have `close()` called at the end of the program.

If you want to avoid starting to connect to the database at import time, and want to wait for the application to be ready, you can create the pool using `open=False`, and call the `open()` and `close()` methods when the conditions are right. Certain frameworks provide callbacks triggered when the program is started and stopped (for instance FastAPI startup/shutdown events): they are perfect to initiate and terminate the pool operations:

```
pool = ConnectionPool(conninfo, open=False, **kwargs)

@app.on_event("startup")
def open_pool():
    pool.open()

@app.on_event("shutdown")
def close_pool():
    pool.close()
```

Creating a single pool as a global variable is not the mandatory use: your program can create more than one pool, which might be useful to connect to more than one database, or to provide different types of connections, for instance to provide separate read/write and read-only connections. The pool also acts as a context manager and is open and closed, if necessary, on entering and exiting the context block:

```
from psycopg_pool import ConnectionPool

with ConnectionPool(conninfo, **kwargs) as pool:
    run_app(pool)

# the pool is now closed
```

When the pool is open, the pool's background workers start creating the requested `min_size` connections, while the constructor (or the `open()` method) returns immediately. This allows the program some leeway to start before the target database is up and running. However, if your application is misconfigured, or the network is down, it means that the program will be able to start, but the threads requesting a connection will fail with a `PoolTimeout` only after the timeout on `connection()` is expired. If this behaviour is not desirable (and you prefer your program to crash hard and fast, if the surrounding conditions are not right, because something else will respawn it) you should call the `wait()`

method after creating the pool, or call `open(wait=True)`: these methods will block until the pool is full, or will raise a `PoolTimeout` exception if the pool isn't ready within the allocated time.

Connections life cycle

The pool background workers create connections according to the parameters `conninfo`, `kwargs`, and `connection_class` passed to `ConnectionPool` constructor, invoking something like *connection_class*(*conninfo*, ***kwargs*). Once a connection is created it is also passed to the `configure()` callback, if provided, after which it is put in the pool (or passed to a client requesting it, if someone is already knocking at the door).

If a connection expires (it passes `max_lifetime`), or is returned to the pool in broken state, or is found closed by `check()`), then the pool will dispose of it and will start a new connection attempt in the background.

Using connections from the pool

The pool can be used to request connections from multiple threads or concurrent tasks - it is hardly useful otherwise! If more connections than the ones available in the pool are requested, the requesting threads are queued and are served a connection as soon as one is available, either because another client has finished using it or because the pool is allowed to grow (when `max_size` > `min_size`) and a new connection is ready.

The main way to use the pool is to obtain a connection using the `connection()` context, which returns a *Connection* or subclass:

```
with my_pool.connection() as conn:
    conn.execute("what you want")
```

The `connection()` context behaves like the *Connection* object context: at the end of the block, if there is a transaction open, it will be committed, or rolled back if the context is exited with as exception.

At the end of the block the connection is returned to the pool and shouldn't be used anymore by the code which obtained it. If a `reset()` function is specified in the pool constructor, it is called on the connection before returning it to the pool. Note that the `reset()` function is called in a worker thread, so that the thread which used the connection can keep its execution without being slowed down by it.

Pool connection and sizing

A pool can have a fixed size (specifying no `max_size` or `max_size` = `min_size`) or a dynamic size (when `max_size` > `min_size`). In both cases, as soon as the pool is created, it will try to acquire `min_size` connections in the background.

If an attempt to create a connection fails, a new attempt will be made soon after, using an exponential backoff to increase the time between attempts, until a maximum of `reconnect_timeout` is reached. When that happens, the pool will call the `reconnect_failed()` function, if provided to the pool, and just start a new connection attempt. You can use this function either to send alerts or to interrupt the program and allow the rest of your infrastructure to restart it.

If more than `min_size` connections are requested concurrently, new ones are created, up to `max_size`. Note that the connections are always created by the background workers, not by the thread asking for the connection: if a client requests a new connection, and a previous client terminates its job before the new connection is ready, the waiting client will be served the existing connection. This is especially useful in scenarios where the time to establish a connection dominates the time for which the connection is used (see this analysis, for instance).

If a pool grows above `min_size`, but its usage decreases afterwards, a number of connections are eventually closed: one every time a connection is unused after the `max_idle` time specified in the pool constructor.

What's the right size for the pool?

Big question. Who knows. However, probably not as large as you imagine. Please take a look at this analysis for some ideas.

Something useful you can do is probably to use the `get_stats()` method and monitor the behaviour of your program to tune the configuration parameters. The size of the pool can also be changed at runtime using the `resize()` method.

Null connection pools

New in version 3.1.

Sometimes you may want leave the choice of using or not using a connection pool as a configuration parameter of your application. For instance, you might want to use a pool if you are deploying a "large instance" of your application and can dedicate it a handful of connections; conversely you might not want to use it if you deploy the application in several instances, behind a load balancer, and/or using an external connection pool process such as PgBouncer.

Switching between using or not using a pool requires some code change, because the `ConnectionPool` API is different from the normal *`connect()`* function and because the pool can perform additional connection configuration (in the `configure` parameter) that, if the pool is removed, should be performed in some different code path of your application.

The `psycopg_pool` 3.1 package introduces the `NullConnectionPool` class. This class has the same interface, and largely the same behaviour, of the `ConnectionPool`, but doesn't create any connection beforehand. When a connection is returned, unless there are other clients already waiting, it is closed immediately and not kept in the pool state.

A null pool is not only a configuration convenience, but can also be used to regulate the access to the server by a client program. If `max_size` is set to a value greater than 0, the pool will make sure that no more than `max_size` connections are created at any given time. If more clients ask for further connections, they will be queued and served a connection as soon as a previous client has finished using it, like for the basic pool. Other mechanisms to throttle client requests (such as `timeout` or `max_waiting`) are respected too.

Note: Queued clients will be handed an already established connection, as soon as a previous client has finished using it (and after the pool has returned it to idle state and called `reset()` on it, if necessary).

Because normally (i.e. unless queued) every client will be served a new connection, the time to obtain the connection is paid by the waiting client; background workers are not normally involved in obtaining new connections.

Connection quality

The state of the connection is verified when a connection is returned to the pool: if a connection is broken during its usage it will be discarded on return and a new connection will be created.

Warning: The health of the connection is not checked when the pool gives it to a client.

Why not? Because doing so would require an extra network roundtrip: we want to save you from its latency. Before getting too angry about it, just think that the connection can be lost any moment while your program is using it. As your program should already be able to cope with a loss of a connection during its process, it should be able to tolerate to be served a broken connection: unpleasant but not the end of the world.

Warning: The health of the connection is not checked when the connection is in the pool.

Does the pool keep a watchful eye on the quality of the connections inside it? No, it doesn't. Why not? Because you will do it for us! Your program is only a big ruse to make sure the connections are still alive…

Not (entirely) trolling: if you are using a connection pool, we assume that you are using and returning connections at a good pace. If the pool had to check for the quality of a broken connection before your program notices it, it should be polling each connection even faster than your program uses them. Your database server wouldn't be amused…

Can you do something better than that? Of course you can, there is always a better way than polling. You can use the same recipe of *Detecting disconnections*, reserving a connection and using a thread to monitor for any activity happening on it. If any activity is detected, you can call the pool `check()` method, which will run a quick check on each connection in the pool, removing the ones found in broken state, and using the background workers to replace them with fresh ones.

If you set up a similar check in your program, in case the database connection is temporarily lost, we cannot do anything for the threads which had taken already a connection from the pool, but no other thread should be served a broken connection, because `check()` would empty the pool and refill it with working connections, as soon as they are available.

Faster than you can say poll. Or pool.

Pool stats

The pool can return information about its usage using the methods `get_stats()` or `pop_stats()`. Both methods return the same values, but the latter reset the counters after its use. The values can be sent to a monitoring system such as Graphite or Prometheus.

The following values should be provided, but please don't consider them as a rigid interface: it is possible that they might change in the future. Keys whose value is 0 may not be returned.

Metric	Meaning
`pool_min`	Current value for `min_size`
`pool_max`	Current value for `max_size`
`pool_size`	Number of connections currently managed by the pool (in the pool, given to clients, being prepared)
`pool_available`	Number of connections currently idle in the pool
`requests_waiting`	Number of requests currently waiting in a queue to receive a connection
`usage_ms`	Total usage time of the connections outside the pool
`requests_num`	Number of connections requested to the pool
`requests_queued`	Number of requests queued because a connection wasn't immediately available in the pool
`requests_wait_ms`	Total time in the queue for the clients waiting
`requests_errors`	Number of connection requests resulting in an error (timeouts, queue full…)
`returns_bad`	Number of connections returned to the pool in a bad state
`connections_num`	Number of connection attempts made by the pool to the server
`connections_ms`	Total time spent to establish connections with the server
`connections_errors`	Number of failed connection attempts
`connections_lost`	Number of connections lost identified by `check()`

1.2.5 Cursor types

Psycopg can manage kinds of "cursors" which differ in where the state of a query being processed is stored: *Client-side cursors* and *Server-side cursors*.

Client-side cursors

Client-side cursors are what Psycopg uses in its normal querying process. They are implemented by the `Cursor` and `AsyncCursor` classes. In such querying pattern, after a cursor sends a query to the server (usually calling `execute()`), the server replies transferring to the client the whole set of results requested, which is stored in the state of the same cursor and from where it can be read from Python code (using methods such as `fetchone()` and siblings).

This querying process is very scalable because, after a query result has been transmitted to the client, the server doesn't keep any state. Because the results are already in the client memory, iterating its rows is very quick.

The downside of this querying method is that the entire result has to be transmitted completely to the client (with a time proportional to its size) and the client needs enough memory to hold it, so it is only suitable for reasonably small result sets.

Client-side-binding cursors

New in version 3.1.

The previously described *client-side cursors* send the query and the parameters separately to the server. This is the most efficient way to process parametrised queries and allows to build several features and optimizations. However, not all types of queries can be bound server-side; in particular no Data Definition Language query can. See *Server-side binding* for the description of these problems.

The `ClientCursor` (and its `AsyncClientCursor` async counterpart) merge the query on the client and send the query and the parameters merged together to the server. This allows to parametrize any type of PostgreSQL statement, not only queries (`SELECT`) and Data Manipulation statements (`INSERT`, `UPDATE`, `DELETE`).

Using `ClientCursor`, Psycopg 3 behaviour will be more similar to `psycopg2` (which only implements client-side binding) and could be useful to port Psycopg 2 programs more easily to Psycopg 3. The objects in the `sql` module allow for greater flexibility (for instance to parametrize a table name too, not only values); however, for simple cases, a `ClientCursor` could be the right object.

In order to obtain `ClientCursor` from a connection, you can set its `cursor_factory` (at init time or changing its attribute afterwards):

```
from psycopg import connect, ClientCursor

conn = psycopg.connect(DSN, cursor_factory=ClientCursor)
cur = conn.cursor()
# <psycopg.ClientCursor [no result] [IDLE] (database=piro) at 0x7fd977ae2880>
```

If you need to create a one-off client-side-binding cursor out of a normal connection, you can just use the `ClientCursor` class passing the connection as argument.

```
conn = psycopg.connect(DSN)
cur = psycopg.ClientCursor(conn)
```

Warning: Client-side cursors don't support *binary parameters and return values* and don't support *prepared statements*.

Tip: The best use for client-side binding cursors is probably to port large Psycopg 2 code to Psycopg 3, especially for programs making wide use of Data Definition Language statements.

The *psycopg.sql* module allows for more generic client-side query composition, to mix client- and server-side parameters binding, and allows to parametrize tables and fields names too, or entirely generic SQL snippets.

Server-side cursors

PostgreSQL has its own concept of *cursor* too (sometimes also called *portal*). When a database cursor is created, the query is not necessarily completely processed: the server might be able to produce results only as they are needed. Only the results requested are transmitted to the client: if the query result is very large but the client only needs the first few records it is possible to transmit only them.

The downside is that the server needs to keep track of the partially processed results, so it uses more memory and resources on the server.

Psycopg allows the use of server-side cursors using the classes *ServerCursor* and *AsyncServerCursor*. They are usually created by passing the *name* parameter to the *cursor()* method (reason for which, in `psycopg2`, they are usually called *named cursors*). The use of these classes is similar to their client-side counterparts: their interface is the same, but behind the scene they send commands to control the state of the cursor on the server (for instance when fetching new records or when moving using *scroll()*).

Using a server-side cursor it is possible to process datasets larger than what would fit in the client's memory. However for small queries they are less efficient because it takes more commands to receive their result, so you should use them only if you need to process huge results or if only a partial result is needed.

See also:

Server-side cursors are created and managed by *ServerCursor* using SQL commands such as DECLARE, FETCH, MOVE. The PostgreSQL documentation gives a good idea of what is possible to do with them.

"Stealing" an existing cursor

A Psycopg *ServerCursor* can be also used to consume a cursor which was created in other ways than the `DECLARE` that *ServerCursor.execute()* runs behind the scene.

For instance if you 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 run a one-off command in the same connection to call it (e.g. using *Connection.execute()*) in order to create the cursor on the server:

```
conn.execute("SELECT reffunc('curname')")
```

after which you can create a server-side cursor declared by the same name, and directly call the fetch methods, skipping the *execute()* call:

```
cur = conn.cursor('curname')
# no cur.execute()
for record in cur:  # or cur.fetchone(), cur.fetchmany()...
    # do something with record
```

1.2.6 Data adaptation configuration

The adaptation system is at the core of Psycopg and allows to customise the way Python objects are converted to PostgreSQL when a query is performed and how PostgreSQL values are converted to Python objects when query results are returned.

Note: For a high-level view of the conversion of types between Python and PostgreSQL please look at *Passing parameters to SQL queries*. Using the objects described in this page is useful if you intend to *customise* the adaptation rules.

- Adaptation configuration is performed by changing the *adapters* object of objects implementing the *AdaptContext* protocol, for instance *Connection* or *Cursor*.
- Every context object derived from another context inherits its adapters mapping: cursors created from a connection inherit the connection's configuration.

  By default, connections obtain an adapters map from the global map exposed as *psycopg.adapters*: changing the content of this object will affect every connection created afterwards. You may specify a different template adapters map using the `context` parameter on *connect()*.

  align
  center
- The `adapters` attributes are *AdaptersMap* instances, and contain the mapping from Python types and *Dumper* classes, and from PostgreSQL OIDs to *Loader* classes. Changing this mapping (e.g. writing and registering your own adapters, or using a different configuration of builtin adapters) affects how types are converted between Python and PostgreSQL.
  - Dumpers (objects implementing the *Dumper* protocol) are the objects used to perform the conversion from a Python object to a bytes sequence in a format understood by PostgreSQL. The string returned *shouldn't be quoted*: the value will be passed to the database using functions such as `PQexecParams()` so quoting and quotes escaping is not necessary. The dumper usually also suggests to the server what type to use, via its *oid* attribute.
  - Loaders (objects implementing the *Loader* protocol) are the objects used to perform the opposite operation: reading a bytes sequence from PostgreSQL and creating a Python object out of it.
  - Dumpers and loaders are instantiated on demand by a *Transformer* object when a query is executed.

Note: Changing adapters in a context only affects that context and its children objects created *afterwards*; the objects already created are not affected. For instance, changing the global context will only change newly created connections, not the ones already existing.

Writing a custom adapter: XML

Psycopg doesn't provide adapters for the XML data type, because there are just too many ways of handling XML in Python. Creating a loader to parse the PostgreSQL xml type to `ElementTree` is very simple, using the *psycopg.adapt.Loader* base class and implementing the *load()* method:

```
>>> import xml.etree.ElementTree as ET
>>> from psycopg.adapt import Loader

>>> # Create a class implementing the `load()` method.
>>> class XmlLoader(Loader):
...     def load(self, data):
...         return ET.fromstring(data)

>>> # Register the loader on the adapters of a context.
>>> conn.adapters.register_loader("xml", XmlLoader)

>>> # Now just query the database returning XML data.
>>> cur = conn.execute(
...     """select XMLPARSE (DOCUMENT '<?xml version="1.0"?>
...             <book><title>Manual</title><chapter>...</chapter></book>')
...     """)

>>> elem = cur.fetchone()[0]
>>> elem
<Element 'book' at 0x7ffb55142ef0>
```

The opposite operation, converting Python objects to PostgreSQL, is performed by dumpers. The *psycopg.adapt.Dumper* base class makes it easy to implement one: you only need to implement the *dump()* method:

```
>>> from psycopg.adapt import Dumper

>>> class XmlDumper(Dumper):
...     # Setting an OID is not necessary but can be helpful
...     oid = psycopg.adapters.types["xml"].oid
...
...     def dump(self, elem):
...         return ET.tostring(elem)

>>> # Register the dumper on the adapters of a context
>>> conn.adapters.register_dumper(ET.Element, XmlDumper)

>>> # Now, in that context, it is possible to use ET.Element objects as parameters
>>> conn.execute("SELECT xpath('//title/text()', %s)", [elem]).fetchone()[0]
['Manual']
```

Note that it is possible to use a *TypesRegistry*, exposed by any *AdaptContext*, to obtain information on builtin types, or extension types if they have been registered on that context using the *TypeInfo.register()* method.

Example: PostgreSQL numeric to Python float

Normally PostgreSQL `numeric` values are converted to Python `Decimal` instances, because both the types allow fixed-precision arithmetic and are not subject to rounding.

Sometimes, however, you may want to perform floating-point math on `numeric` values, and `Decimal` may get in the way (maybe because it is slower, or maybe because mixing `float` and `Decimal` values causes Python errors).

If you are fine with the potential loss of precision and you simply want to receive `numeric` values as Python `float`, you can register on `numeric` the same *`Loader`* class used to load `float4`/`float8` values. Because the PostgreSQL textual representation of both floats and decimal is the same, the two loaders are compatible.

```
conn = psycopg.connect()

conn.execute("SELECT 123.45").fetchone()[0]
# Decimal('123.45')

conn.adapters.register_loader("numeric", psycopg.types.numeric.FloatLoader)

conn.execute("SELECT 123.45").fetchone()[0]
# 123.45
```

In this example the customised adaptation takes effect only on the connection `conn` and on any cursor created from it, not on other connections.

Example: handling infinity date

Suppose you want to work with the "infinity" date which is available in PostgreSQL but not handled by Python:

```
>>> conn.execute("SELECT 'infinity'::date").fetchone()
Traceback (most recent call last):
  ...
DataError: date too large (after year 10K): 'infinity'
```

One possibility would be to store Python's `datetime.date.max` as PostgreSQL infinity. For this, let's create a subclass for the dumper and the loader and register them in the working scope (globally or just on a connection or cursor):

```
from datetime import date

# Subclass existing adapters so that the base case is handled normally.
from psycopg.types.datetime import DateLoader, DateDumper

class InfDateDumper(DateDumper):
    def dump(self, obj):
        if obj == date.max:
            return b"infinity"
        elif obj == date.min:
            return b"-infinity"
        else:
            return super().dump(obj)

class InfDateLoader(DateLoader):
    def load(self, data):
        if data == b"infinity":
```

(continues on next page)

(continued from previous page)

```
            return date.max
        elif data == b"-infinity":
            return date.min
        else:
            return super().load(data)

# The new classes can be registered globally, on a connection, on a cursor
cur.adapters.register_dumper(date, InfDateDumper)
cur.adapters.register_loader("date", InfDateLoader)

cur.execute("SELECT %s::text, %s::text", [date(2020, 12, 31), date.max]).fetchone()
# ('2020-12-31', 'infinity')
cur.execute("SELECT '2020-12-31'::date, 'infinity'::date").fetchone()
# (datetime.date(2020, 12, 31), datetime.date(9999, 12, 31))
```

Dumpers and loaders life cycle

Registering dumpers and loaders will instruct Psycopg to use them in the queries to follow, in the context where they have been registered.

When a query is performed on a *Cursor*, a *Transformer* object is created as a local context to manage adaptation during the query, instantiating the required dumpers and loaders and dispatching the values to perform the wanted conversions from Python to Postgres and back.

- The `Transformer` copies the adapters configuration from the `Cursor`, thus inheriting all the changes made to the global *psycopg.adapters* configuration, the current `Connection`, the `Cursor`.
- For every Python type passed as query argument, the `Transformer` will instantiate a `Dumper`. Usually all the objects of the same type will be converted by the same dumper instance.
  - According to the placeholder used (`%s`, `%b`, `%t`), Psycopg may pick a binary or a text dumper. When using the `%s` "*AUTO*" format, if the same type has both a text and a binary dumper registered, the last one registered by *register_dumper()* will be used.
  - Sometimes, just looking at the Python type is not enough to decide the best PostgreSQL type to use (for instance the PostgreSQL type of a Python list depends on the objects it contains, whether to use an `integer` or `bigint` depends on the number size...) In these cases the mechanism provided by *get_key()* and *upgrade()* is used to create more specific dumpers.
- The query is executed. Upon successful request, the result is received as a *PGresult*.
- For every OID returned by the query, the `Transformer` will instantiate a `Loader`. All the values with the same OID will be converted by the same loader instance.
- Recursive types (e.g. Python lists, PostgreSQL arrays and composite types) will use the same adaptation rules.

As a consequence it is possible to perform certain choices only once per query (e.g. looking up the connection encoding) and then call a fast-path operation for each value to convert.

Querying will fail if a Python object for which there isn't a `Dumper` registered (for the right *Format*) is used as query parameter. If the query returns a data type whose OID doesn't have a `Loader`, the value will be returned as a string (or bytes string for binary types).

1.2.7 Prepared statements

Psycopg uses an automatic system to manage *prepared statements*. When a query is prepared, its parsing and planning is stored in the server session, so that further executions of the same query on the same connection (even with different parameters) are optimised.

A query is prepared automatically after it is executed more than *prepare_threshold* times on a connection. psycopg will make sure that no more than *prepared_max* statements are planned: if further queries are executed, the least recently used ones are deallocated and the associated resources freed.

Statement preparation can be controlled in several ways:

- You can decide to prepare a query immediately by passing `prepare=True` to *Connection.execute()* or *Cursor.execute()*. The query is prepared, if it wasn't already, and executed as prepared from its first use.
- Conversely, passing `prepare=False` to `execute()` will avoid to prepare the query, regardless of the number of times it is executed. The default for the parameter is `None`, meaning that the query is prepared if the conditions described above are met.
- You can disable the use of prepared statements on a connection by setting its *prepare_threshold* attribute to `None`.

Changed in version 3.1: You can set `prepare_threshold` as a *connect()* keyword parameter too.

See also:

The PREPARE PostgreSQL documentation contains plenty of details about prepared statements in PostgreSQL.

Note however that Psycopg doesn't use SQL statements such as `PREPARE` and `EXECUTE`, but protocol level commands such as the ones exposed by `PQsendPrepare`, `PQsendQueryPrepared`.

Warning: Using external connection poolers, such as PgBouncer, is not compatible with prepared statements, because the same client connection may change the server session it refers to. If such middleware is used you should disable prepared statements, by setting the *Connection.prepare_threshold* attribute to `None`.

1.2.8 Pipeline mode support

New in version 3.1.

The *pipeline mode* allows PostgreSQL client applications to send a query without having to read the result of the previously sent query. Taking advantage of the pipeline mode, a client will wait less for the server, since multiple queries/results can be sent/received in a single network roundtrip. Pipeline mode can provide a significant performance boost to the application.

Pipeline mode is most useful when the server is distant, i.e., network latency ("ping time") is high, and also when many small operations are being performed in rapid succession. There is usually less benefit in using pipelined commands when each query takes many multiples of the client/server round-trip time to execute. A 100-statement operation run on a server 300 ms round-trip-time away would take 30 seconds in network latency alone without pipelining; with pipelining it may spend as little as 0.3 s waiting for results from the server.

The server executes statements, and returns results, in the order the client sends them. The server will begin executing the commands in the pipeline immediately, not waiting for the end of the pipeline. Note that results are buffered on the server side; the server flushes that buffer when a *synchronization point* is established.

See also:

The PostgreSQL documentation about:

- pipeline mode

- extended query message flow

contains many details around when it is most useful to use the pipeline mode and about errors management and interaction with transactions.

Client-server messages flow

In order to understand better how the pipeline mode works, we should take a closer look at the PostgreSQL client-server message flow.

During normal querying, each statement is transmitted by the client to the server as a stream of request messages, terminating with a *Sync* message to tell it that it should process the messages sent so far. The server will execute the statement and describe the results back as a stream of messages, terminating with a *ReadyForQuery*, telling the client that it may now send a new query.

For example, the statement (returning no result):

```
conn.execute("INSERT INTO mytable (data) VALUES (%s)", ["hello"])
```

results in the following two groups of messages:

Direction	Message
Python PostgreSQL	• Parse `INSERT INTO ... (VALUE $1)` (skipped if *the statement is prepared*) • Bind `'hello'` • Describe • Execute • Sync
PostgreSQL Python	• ParseComplete • BindComplete • NoData • CommandComplete `INSERT 0 1` • ReadyForQuery

and the query:

```
conn.execute("SELECT data FROM mytable WHERE id = %s", [1])
```

results in the two groups of messages:

Direction	Message
Python PostgreSQL	• Parse `SELECT data FROM mytable WHERE id = $1` • Bind 1 • Describe • Execute • Sync
PostgreSQL Python	• ParseComplete • BindComplete • RowDescription `data` • DataRow `hello` • CommandComplete `SELECT 1` • ReadyForQuery

The two statements, sent consecutively, pay the communication overhead four times, once per leg.

The pipeline mode allows the client to combine several operations in longer streams of messages to the server, then to receive more than one response in a single batch. If we execute the two operations above in a pipeline:

```
with conn.pipeline():
    conn.execute("INSERT INTO mytable (data) VALUES (%s)", ["hello"])
    conn.execute("SELECT data FROM mytable WHERE id = %s", [1])
```

they will result in a single roundtrip between the client and the server:

Direction	Message
Python PostgreSQL	• Parse `INSERT INTO ... (VALUE $1)` • Bind `'hello'` • Describe • Execute • Parse `SELECT data FROM mytable WHERE id = $1` • Bind 1 • Describe • Execute • Sync (sent only once)
PostgreSQL Python	• ParseComplete • BindComplete • NoData • CommandComplete `INSERT 0 1` • ParseComplete • BindComplete • RowDescription `data` • DataRow `hello` • CommandComplete `SELECT 1` • ReadyForQuery (sent only once)

Pipeline mode usage

Psycopg supports the pipeline mode via the *Connection.pipeline()* method. The method is a context manager: entering the `with` block yields a *Pipeline* object. At the end of block, the connection resumes the normal operation mode.

Within the pipeline block, you can use normally one or more cursors to execute several operations, using *Connection.execute()*, *Cursor.execute()* and *executemany()*.

```
>>> with conn.pipeline():
...     conn.execute("INSERT INTO mytable VALUES (%s)", ["hello"])
...     with conn.cursor() as cur:
...         cur.execute("INSERT INTO othertable VALUES (%s)", ["world"])
...         cur.executemany(
...             "INSERT INTO elsewhere VALUES (%s)",
...             [("one",), ("two",), ("four",)])
```

Unlike in normal mode, Psycopg will not wait for the server to receive the result of each query; the client will receive results in batches when the server flushes it output buffer.

When a flush (or a sync) is performed, all pending results are sent back to the cursors which executed them. If a cursor had run more than one query, it will receive more than one result; results after the first will be available, in their execution order, using *nextset()*:

```
>>> with conn.pipeline():
...     with conn.cursor() as cur:
...         cur.execute("INSERT INTO mytable (data) VALUES (%s) RETURNING *", ["hello"])
...         cur.execute("INSERT INTO mytable (data) VALUES (%s) RETURNING *", ["world"])
...         while True:
...             print(cur.fetchall())
...             if not cur.nextset():
...                 break

[(1, 'hello')]
[(2, 'world')]
```

If any statement encounters an error, the server aborts the current transaction and will not execute any subsequent command in the queue until the next *synchronization point*; a *PipelineAborted* exception is raised for each such command. Query processing resumes after the synchronization point.

Warning: Certain features are not available in pipeline mode, including:

- COPY is not supported in pipeline mode by PostgreSQL.
- *Cursor.stream()* doesn't make sense in pipeline mode (its job is the opposite of batching!)
- *ServerCursor* are currently not implemented in pipeline mode.

Note: Starting from Psycopg 3.1, *executemany()* makes use internally of the pipeline mode; as a consequence there is no need to handle a pipeline block just to call `executemany()` once.

Synchronization points

Flushing query results to the client can happen either when a synchronization point is established by Psycopg:

- using the *Pipeline.sync()* method;
- on *Connection.commit()* or *rollback()*;
- at the end of a `Pipeline` block;
- possibly when opening a nested `Pipeline` block;
- using a fetch method such as *Cursor.fetchone()* (which only flushes the query but doesn't issue a Sync and doesn't reset a pipeline state error).

The server might perform a flush on its own initiative, for instance when the output buffer is full.

Note that, even in *autocommit*, the server wraps the statements sent in pipeline mode in an implicit transaction, which will be only committed when the Sync is received. As such, a failure in a group of statements will probably invalidate the effect of statements executed after the previous Sync, and will propagate to the following Sync.

For example, in the following block:

```
>>> with psycopg.connect(autocommit=True) as conn:
...     with conn.pipeline() as p, conn.cursor() as cur:
...         try:
...             cur.execute("INSERT INTO mytable (data) VALUES (%s)", ["one"])
...             cur.execute("INSERT INTO no_such_table (data) VALUES (%s)", ["two"])
...             conn.execute("INSERT INTO mytable (data) VALUES (%s)", ["three"])
...             p.sync()
...         except psycopg.errors.UndefinedTable:
...             pass
...         cur.execute("INSERT INTO mytable (data) VALUES (%s)", ["four"])
```

there will be an error in the block, `relation "no_such_table" does not exist` caused by the insert `two`, but probably raised by the `sync()` call. At at the end of the block, the table will contain:

```
=# SELECT * FROM mytable;
+----+------+
| id | data |
+----+------+
|  2 | four |
+----+------+
(1 row)
```

because:

- the value 1 of the sequence is consumed by the statement `one`, but the record discarded because of the error in the same implicit transaction;
- the statement `three` is not executed because the pipeline is aborted (so it doesn't consume a sequence item);
- the statement `four` is executed with success after the Sync has terminated the failed transaction.

Warning: The exact Python statement where an exception caused by a server error is raised is somewhat arbitrary: it depends on when the server flushes its buffered result.

If you want to make sure that a group of statements is applied atomically by the server, do make use of transaction methods such as *commit()* or *transaction()*: these methods will also sync the pipeline and raise an exception if there was any error in the commands executed so far.

The fine prints

Warning: The Pipeline mode is an experimental feature.

Its behaviour, especially around error conditions and concurrency, hasn't been explored as much as the normal request-response messages pattern, and its async nature makes it inherently more complex.

As we gain more experience and feedback (which is welcome), we might find bugs and shortcomings forcing us to change the current interface or behaviour.

The pipeline mode is available on any currently supported PostgreSQL version, but, in order to make use of it, the client must use a libpq from PostgreSQL 14 or higher. You can use *Pipeline.is_supported()* to make sure your client has the right library.

1.3 Psycopg 3 API

This sections is a reference for all the public objects exposed by the *psycopg* module. For a more conceptual description you can take a look at *Getting started with Psycopg 3* and *More advanced topics*.

1.3.1 The psycopg module

Psycopg implements the Python Database DB API 2.0 specification. As such it also exposes the module-level objects required by the specifications.

psycopg.**connect**(*conninfo=''*, ***, *autocommit=False*, *prepare_threshold=5*, *row_factory=None*, *cursor_factory=None*, *context=None*, ***kwargs*)

Connect to a database server and return a new *Connection* instance.

Return type
Connection[Any]

This is an alias of the class method *Connection.connect*: see its documentation for details.

If you need an asynchronous connection use *AsyncConnection.connect* instead.

Exceptions

The standard DBAPI exceptions are exposed both by the psycopg module and by the *psycopg.errors* module. The latter also exposes more specific exceptions, mapping to the database error states (see *SQLSTATE exceptions*).

```
Exception
|__ Warning
|__ Error
    |__ InterfaceError
    |__ DatabaseError
        |__ DataError
```

```
|__ OperationalError
|__ IntegrityError
|__ InternalError
|__ ProgrammingError
|__ NotSupportedError
```

psycopg.**adapters**

The default adapters map establishing how Python and PostgreSQL types are converted into each other.

This map is used as a template when new connections are created, using *psycopg.connect()*. Its *types* attribute is a *TypesRegistry* containing information about every PostgreSQL builtin type, useful for adaptation customisation (see *Data adaptation configuration*):

```
>>> psycopg.adapters.types["int4"]
<TypeInfo: int4 (oid: 23, array oid: 1007)>
```

Type
AdaptersMap

1.3.2 Connection classes

The *Connection* and *AsyncConnection* classes are the main wrappers for a PostgreSQL database session. You can imagine them similar to a **psql** session.

One of the differences compared to **psql** is that a *Connection* usually handles a transaction automatically: other sessions will not be able to see the changes until you have committed them, more or less explicitly. Take a look to *Transactions management* for the details.

The Connection class

class psycopg.**Connection**(*pgconn*, *row_factory=<function tuple_row>*)

Wrapper for a connection to the database.

This class implements a DBAPI-compliant interface. It is what you want to use if you write a "classic", blocking program (eventually using threads or Eventlet/gevent for concurrency). If your program uses `asyncio` you might want to use *AsyncConnection* instead.

Connections behave as context managers: on block exit, the current transaction will be committed (or rolled back, in case of exception) and the connection will be closed.

classmethod connect(*conninfo: str = ''*, ***, *autocommit: bool = False*, *row_factory: RowFactory[Row]*, *prepare_threshold: Optional[int] = 5*, *cursor_factory: Optional[Type[Cursor[Row]]] = None*, *context: Optional[AdaptContext] = None*, ***kwargs: Union[None, int, str]*) → *Connection*[Row]

classmethod connect(*conninfo: str = ''*, ***, *autocommit: bool = False*, *prepare_threshold: Optional[int] = 5*, *cursor_factory: Optional[Type[Cursor[Any]]] = None*, *context: Optional[AdaptContext] = None*, ***kwargs: Union[None, int, str]*) → *Connection*[Tuple[Any, ...]]

Connect to a database server and return a new *Connection* instance.

Return type
Connection[Any]

Parameters

- **conninfo** – The connection string (a `postgresql://` url or a list of `key=value` pairs) to specify where and how to connect.
- **kwargs** – Further parameters specifying the connection string. They override the ones specified in `conninfo`.
- **autocommit** – If `True` don't start transactions automatically. See *Transactions management* for details.
- **row_factory** – The row factory specifying what type of records to create fetching data (default: *tuple_row()*). See *Row factories* for details.
- **cursor_factory** – Initial value for the *cursor_factory* attribute of the connection (new in Psycopg 3.1).
- **prepare_threshold** – Initial value for the *prepare_threshold* attribute of the connection (new in Psycopg 3.1).

More specialized use:

Parameters
context – A context to copy the initial adapters configuration from. It might be an *AdaptersMap* with customized loaders and dumpers, used as a template to create several connections. See *Data adaptation configuration* for further details.

This method is also aliased as *psycopg.connect()*.

See also:

- the list of the accepted connection parameters
- the environment variables affecting connection

Changed in version 3.1: added `prepare_threshold` and `cursor_factory` parameters.

close()

Close the database connection.

Note: You can use:

```
with psycopg.connect() as conn:
    ...
```

to close the connection automatically when the block is exited. See *Connection context*.

---

closed: bool

`True` if the connection is closed.

broken: bool

`True` if the connection was interrupted.

A broken connection is always *closed*, but wasn't closed in a clean way, such as using *close()* or a `with` block.

cursor(*, *binary: bool = False*, *row_factory: Optional[RowFactory] = None*) → *Cursor*

cursor(*name: str*, *, *binary: bool = False*, *row_factory: Optional[RowFactory] = None*, *scrollable: Optional[bool] = None*, *withhold: bool = False*) → *ServerCursor*

Return a new cursor to send commands and queries to the connection.

Parameters

- **name** – If not specified create a client-side cursor, if specified create a server-side cursor. See *Cursor types* for details.
- **binary** – If `True` return binary values from the database. All the types returned by the query must have a binary loader. See *Binary parameters and results* for details.
- **row_factory** – If specified override the *row_factory* set on the connection. See *Row factories* for details.
- **scrollable** – Specify the *scrollable* property of the server-side cursor created.
- **withhold** – Specify the *withhold* property of the server-side cursor created.

Returns

A cursor of the class specified by *cursor_factory* (or *server_cursor_factory* if `name` is specified).

Note: You can use:

```
with conn.cursor() as cur:
    ...
```

to close the cursor automatically when the block is exited.

---

cursor_factory: `Type[Cursor[Row]]`

The type, or factory function, returned by *cursor()* and *execute()*.

Default is *psycopg.Cursor*.

server_cursor_factory: `Type[ServerCursor[Row]]`

The type, or factory function, returned by *cursor()* when a name is specified.

Default is *psycopg.ServerCursor*.

row_factory: *RowFactory*`[Row]`

The row factory defining the type of rows returned by *fetchone()* and the other cursor fetch methods.

The default is *tuple_row*, which means that the fetch methods will return simple tuples.

See also:

See *Row factories* for details about defining the objects returned by cursors.

execute(*query*, *params=None*, ***, *prepare=None*, *binary=False*)

Execute a query and return a cursor to read its results.

Return type

Cursor[TypeVar(Row, covariant=True)]

Parameters

- **query** (`str`, `bytes`, *sql.SQL*, or *sql.Composed*) – The query to execute.
- **params** (*Sequence or Mapping*) – The parameters to pass to the query, if any.
- **prepare** – Force (`True`) or disallow (`False`) preparation of the query. By default (`None`) prepare automatically. See *Prepared statements*.

- **binary** – If `True` the cursor will return binary values from the database. All the types returned by the query must have a binary loader. See *Binary parameters and results* for details.

The method simply creates a `Cursor` instance, `execute()` the query requested, and returns it.

See *Passing parameters to SQL queries* for all the details about executing queries.

pipeline()

Switch the connection into pipeline mode.

Return type
`Iterator`[*Pipeline*]

The method is a context manager: you should call it using:

```
with conn.pipeline() as p:
    ...
```

At the end of the block, a synchronization point is established and the connection returns in normal mode.

You can call the method recursively from within a pipeline block. Innermost blocks will establish a synchronization point on exit, but pipeline mode will be kept until the outermost block exits.

See *Pipeline mode support* for details.

New in version 3.1.

Transaction management methods

For details see *Transactions management*.

commit()

Commit any pending transaction to the database.

rollback()

Roll back to the start of any pending transaction.

transaction(*savepoint_name=None*, *force_rollback=False*)

Start a context block with a new transaction or nested transaction.

Parameters

- **savepoint_name** (`Optional`[`str`]) – Name of the savepoint used to manage a nested transaction. If `None`, one will be chosen automatically.
- **force_rollback** (`bool`) – Roll back the transaction at the end of the block even if there were no error (e.g. to try a no-op process).

Return type
Transaction

Note: The method must be called with a syntax such as:

```
with conn.transaction():
    ...

with conn.transaction() as tx:
    ...
```

The latter is useful if you need to interact with the *Transaction* object. See *Transaction contexts* for details.

Inside a transaction block it will not be possible to call *commit()* or *rollback()*.

autocommit: bool

The autocommit state of the connection.

The property is writable for sync connections, read-only for async ones: you should call `await` *set_autocommit* (*value*) instead.

The following three properties control the characteristics of new transactions. See *Transaction characteristics* for details.

isolation_level: Optional[*IsolationLevel*]

The isolation level of the new transactions started on the connection.

`None` means use the default set in the default_transaction_isolation configuration parameter of the server.

read_only: Optional[bool]

The read-only state of the new transactions started on the connection.

`None` means use the default set in the default_transaction_read_only configuration parameter of the server.

deferrable: Optional[bool]

The deferrable state of the new transactions started on the connection.

`None` means use the default set in the default_transaction_deferrable configuration parameter of the server.

Checking and configuring the connection state

pgconn: *psycopg.pq.PGconn*

The *PGconn* libpq connection wrapper underlying the `Connection`.

It can be used to send low level commands to PostgreSQL and access features not currently wrapped by Psycopg.

info: *ConnectionInfo*

A *ConnectionInfo* attribute to inspect connection properties.

prepare_threshold: Optional[int]

Number of times a query is executed before it is prepared.

- If it is set to 0, every query is prepared the first time it is executed.
- If it is set to `None`, prepared statements are disabled on the connection.

Default value: 5

See *Prepared statements* for details.

prepared_max: int

Maximum number of prepared statements on the connection.

Default value: 100

If more queries need to be prepared, old ones are deallocated.

Methods you can use to do something cool

cancel()

Cancel the current operation on the connection.

notifies()

Yield *Notify* objects as soon as they are received from the database.

Return type
Generator[*Notify*, None, None]

Notifies are received after using `LISTEN` in a connection, when any sessions in the database generates a `NOTIFY` on one of the listened channels.

add_notify_handler(*callback*)

Register a callable to be invoked whenever a notification is received.

Parameters
callback (*Callable[[Notify], None]*) – the callback to call upon notification received.

See *Asynchronous notifications* for details.

remove_notify_handler(*callback*)

Unregister a notification callable previously registered.

Parameters
callback (*Callable[[Notify], None]*) – the callback to remove.

add_notice_handler(*callback*)

Register a callable to be invoked when a notice message is received.

Parameters
callback (*Callable[[Diagnostic], None]*) – the callback to call upon message received.

See *Server messages* for details.

remove_notice_handler(*callback*)

Unregister a notice message callable previously registered.

Parameters
callback (*Callable[[Diagnostic], None]*) – the callback to remove.

fileno()

Return the file descriptor of the connection.

This function allows to use the connection as file-like object in functions waiting for readiness, such as the ones defined in the `selectors` module.

Return type
int

Two-Phase Commit support methods

New in version 3.1.

See also:

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

xid(*format_id*, *gtrid*, *bqual*)

Returns a *Xid* to pass 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*.

Return type
Xid

tpc_begin(*xid*)

Begin a TPC transaction with the given transaction ID *xid*.

Parameters
xid (*Xid or str*) – The id of the transaction

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

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()

Perform 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.

See also:

The `PREPARE TRANSACTION` PostgreSQL command.

tpc_commit(*xid=None*)

Commit a prepared two-phase transaction.

Parameters
xid (*Xid or str*) – The id of the transaction

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=None*)

Roll back a prepared two-phase transaction.

Parameters
xid (*Xid or str*) – The id of the transaction

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()

Return type
List[*Xid*]

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.

The AsyncConnection class

class psycopg.**AsyncConnection**(*pgconn*, *row_factory=<function tuple_row>*)

Asynchronous wrapper for a connection to the database.

This class implements a DBAPI-inspired interface, with all the blocking methods implemented as coroutines. Unless specified otherwise, non-blocking methods are shared with the *Connection* class.

The following methods have the same behaviour of the matching `Connection` methods, but should be called using the `await` keyword.

async classmethod connect(*conninfo: str = ''*, ***, *autocommit: bool = False*, *prepare_threshold: Optional[int] = 5*, *row_factory: AsyncRowFactory[Row]*, *cursor_factory: Optional[Type[AsyncCursor[Row]]] = None*, *context: Optional[AdaptContext] = None*, ***kwargs: Union[None, int, str]*) → *AsyncConnection*[Row]

async classmethod connect(*conninfo: str = ''*, ***, *autocommit: bool = False*, *prepare_threshold: Optional[int] = 5*, *cursor_factory: Optional[Type[AsyncCursor[Any]]] = None*, *context: Optional[AdaptContext] = None*, ***kwargs: Union[None, int, str]*) → *AsyncConnection*[Tuple[Any, ...]]

Return type
AsyncConnection[Any]

Changed in version 3.1: Automatically resolve domain names asynchronously. In previous versions, name resolution blocks, unless the `hostaddr` parameter is specified, or the *resolve_hostaddr_async()* function is used.

async close()

Note: You can use `async with` to close the connection automatically when the block is exited, but be careful about the async quirkness: see *with async connections* for details.

cursor(***, *binary: bool = False*, *row_factory: Optional[RowFactory] = None*) → *AsyncCursor*

cursor(*name: str*, ***, *binary: bool = False*, *row_factory: Optional[RowFactory] = None*, *scrollable: Optional[bool] = None*, *withhold: bool = False*) → *AsyncServerCursor*

Note: You can use:

```
async with conn.cursor() as cur:
    ...
```

to close the cursor automatically when the block is exited.

cursor_factory: `Type[AsyncCursor[Row]]`

Default is *psycopg.AsyncCursor*.

server_cursor_factory: `Type[AsyncServerCursor[Row]]`

Default is *psycopg.AsyncServerCursor*.

row_factory: *AsyncRowFactory*`[Row]`

async execute(*query*, *params=None*, ***, *prepare=None*, *binary=False*)

Return type
AsyncCursor[TypeVar(Row, covariant=True)]

pipeline()

Context manager to switch the connection into pipeline mode.

Return type
AsyncIterator[*AsyncPipeline*]

Note: It must be called as:

```
async with conn.pipeline() as p:
    ...
```

async commit()

async rollback()

transaction(*savepoint_name=None*, *force_rollback=False*)

Start a context block with a new transaction or nested transaction.

Return type
AsyncTransaction

Note: It must be called as:

```
async with conn.transaction() as tx:
    ...
```

---

async notifies()

Return type
AsyncGenerator[*Notify*, None]

async set_autocommit(*value*)

Async version of the *autocommit* setter.

async set_isolation_level(*value*)

Async version of the *isolation_level* setter.

async set_read_only(*value*)

Async version of the *read_only* setter.

async set_deferrable(*value*)

Async version of the *deferrable* setter.

async tpc_prepare()

async tpc_commit(*xid=None*)

async tpc_rollback(*xid=None*)

async tpc_recover()

Return type
List[*Xid*]

1.3.3 Cursor classes

The *Cursor* and *AsyncCursor* classes are the main objects to send commands to a PostgreSQL database session. They are normally created by the connection's *cursor()* method.

Using the `name` parameter on `cursor()` will create a *ServerCursor* or *AsyncServerCursor*, which can be used to retrieve partial results from a database.

A *Connection* can create several cursors, but only one at time can perform operations, so they are not the best way to achieve parallelism (you may want to operate with several connections instead). All the cursors on the same connection have a view of the same session, so they can see each other's uncommitted data.

The `Cursor` class

class psycopg.**Cursor**(*connection*, ***, *row_factory=None*)

This class implements a DBAPI-compliant interface. It is what the classic *`Connection.cursor()`* method returns. *`AsyncConnection.cursor()`* will create instead *`AsyncCursor`* objects, which have the same set of method but expose an `asyncio` interface and require `async` and `await` keywords to operate.

Cursors behave as context managers: on block exit they are closed and further operation will not be possible. Closing a cursor will not terminate a transaction or a session though.

connection: *`Connection`*

The connection this cursor is using.

close()

Close the current cursor and free associated resources.

Note: You can use:

```
with conn.cursor() as cur:
    ...
```

to close the cursor automatically when the block is exited. See *Main objects in Psycopg 3*.

---

closed: `bool`

`True` if the cursor is closed.

Methods to send commands

execute(*query*, *params=None*, ***, *prepare=None*, *binary=None*)

Execute a query or command to the database.

Return type
`TypeVar`(_Self, bound= Cursor[Any])

Parameters

- **query** (`str`, `bytes`, *`sql.SQL`*, or *`sql.Composed`*) – The query to execute.
- **params** (*`Sequence or Mapping`*) – The parameters to pass to the query, if any.
- **prepare** – Force (`True`) or disallow (`False`) preparation of the query. By default (`None`) prepare automatically. See *Prepared statements*.
- **binary** – Specify whether the server should return data in binary format (`True`) or in text format (`False`). By default (`None`) return data as requested by the cursor's *`format`*.

Return the cursor itself, so that it will be possible to chain a fetch operation after the call.

See *Passing parameters to SQL queries* for all the details about executing queries.

Changed in version 3.1: The `query` argument must be a `StringLiteral`. If you need to compose a query dynamically, please use *`sql.SQL`* and related objects.

See **PEP 675** for details.

executemany(*query*, *params_seq*, ***, *returning=False*)

Execute the same command with a sequence of input data.

Parameters

- **query** (str, bytes, *sql.SQL*, or *sql.Composed*) – The query to execute
- **params_seq** (*Sequence of Sequences or Mappings*) – The parameters to pass to the query
- **returning** (bool) – If True, fetch the results of the queries executed

This is more efficient than performing separate queries, but in case of several INSERT (and with some SQL creativity for massive UPDATE too) you may consider using *copy()*.

If the queries return data you want to read (e.g. when executing an INSERT ... RETURNING or a SELECT with a side-effect), you can specify returning=True; the results will be available in the cursor's state and can be read using *fetchone()* and similar methods. Each input parameter will produce a separate result set: use *nextset()* to read the results of the queries after the first one.

See *Passing parameters to SQL queries* for all the details about executing queries.

Changed in version 3.1:

- Added returning parameter to receive query results.
- Performance optimised by making use of the pipeline mode, when using libpq 14 or newer.

copy(*statement*, *params=None*, *, *writer=None*)

Initiate a COPY operation and return an object to manage it.

Return type
Copy

Parameters

- **statement** (str, bytes, *sql.SQL*, or *sql.Composed*) – The copy operation to execute
- **params** (*Sequence or Mapping*) – The parameters to pass to the statement, if any.

Note: The method must be called with:

```
with cursor.copy() as copy:
    ...
```

See *Using COPY TO and COPY FROM* for information about COPY.

Changed in version 3.1: Added parameters support.

stream(*query*, *params=None*, *, *binary=None*)

Iterate row-by-row on a result from the database.

Return type
Iterator[TypeVar(Row, covariant=True)]

This command is similar to execute + iter; however it supports endless data streams. The feature is not available in PostgreSQL, but some implementations exist: Materialize TAIL and CockroachDB CHANGEFEED for instance.

The feature, and the API supporting it, are still experimental. Beware...

The parameters are the same of *execute()*.

Warning: Failing to consume the iterator entirely will result in a connection left in *transaction_status* *ACTIVE* state: this connection will refuse to receive further commands (with a message such as *another command is already in progress*).

If there is a chance that the generator is not consumed entirely, in order to restore the connection to a working state you can call `close` on the generator object returned by `stream()`. The `contextlib.closing` function might be particularly useful to make sure that `close()` is called:

```
with closing(cur.stream("select generate_series(1, 10000)")) as gen:
    for rec in gen:
        something(rec)  # might fail
```

Without calling `close()`, in case of error, the connection will be `ACTIVE` and unusable. If `close()` is called, the connection might be `INTRANS` or `INERROR`, depending on whether the server managed to send the entire resultset to the client. An autocommit connection will be `IDLE` instead.

format

The format of the data returned by the queries. It can be selected initially e.g. specifying *Connection.cursor*`(binary=True)` and changed during the cursor's lifetime. It is also possible to override the value for single queries, e.g. specifying *execute*`(binary=True)`.

Type
pq.Format

Default
TEXT

See also:

Binary parameters and results

Methods to retrieve results

Fetch methods are only available if the last operation produced results, e.g. a `SELECT` or a command with `RETURNING`. They will raise an exception if used with operations that don't return result, such as an `INSERT` with no `RETURNING` or an `ALTER TABLE`.

Note: Cursors are iterable objects, so just using the:

```
for record in cursor:
    ...
```

syntax will iterate on the records in the current recordset.

---

row_factory: rows.RowFactory[TypeVar(``Row``, covariant=True)]

Writable attribute to control how result rows are formed.

The property affects the objects returned by the *fetchone()*, *fetchmany()*, *fetchall()* methods. The default (*tuple_row*) returns a tuple for each record fetched.

See *Row factories* for details.

fetchone()

Return the next record from the current recordset.

Return `None` the recordset is finished.

Return type
Optional[Row], with Row defined by *row_factory*

fetchmany(*size=0*)

Return the next *size* records from the current recordset.

size default to `self.arraysize` if not specified.

Return type
Sequence[Row], with Row defined by *row_factory*

fetchall()

Return all the remaining records from the current recordset.

Return type
Sequence[Row], with Row defined by *row_factory*

nextset()

Move to the result set of the next query executed through *executemany()* or to the next result set if *execute()* returned more than one.

Return `True` if a new result is available, which will be the one methods `fetch*()` will operate on.

Return type
`Optional[bool]`

scroll(*value*, *mode='relative'*)

Move 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.

Raise `IndexError` in case a scroll operation would leave the result set. In this case the position will not change.

pgresult: Optional[*psycopg.pq.PGresult*]

The result returned by the last query and currently exposed by the cursor, if available, else `None`.

It can be used to obtain low level info about the last query result and to access to features not currently wrapped by Psycopg.

Information about the data

description: Optional[List[*Column*]]

A list of *Column* objects describing the current resultset.

`None` if the current resultset didn't return tuples.

statusmessage: Optional[str]

The command status tag from the last SQL command executed.

`None` if the cursor doesn't have a result available.

This is the status tag you typically see in **psql** after a successful command, such as `CREATE TABLE` or `UPDATE 42`.

rowcount: int

Number of records affected by the precedent operation.

rownumber: Optional[int]

Index of the next row to fetch in the current result.

`None` if there is no result to fetch.

_query

An helper object used to convert queries and parameters before sending them to PostgreSQL.

Note: This attribute is exposed because it might be helpful to debug problems when the communication between Python and PostgreSQL doesn't work as expected. For this reason, the attribute is available when a query fails too.

Warning: You shouldn't consider it part of the public interface of the object: it might change without warnings.

Except this warning, I guess.

If you would like to build reliable features using this object, please get in touch so we can try and design an useful interface for it.

Among the properties currently exposed by this object:

- `query` (`bytes`): the query effectively sent to PostgreSQL. It will have Python placeholders (`%s`-style) replaced with PostgreSQL ones (`$1`, `$2`-style).
- `params` (sequence of `bytes`): the parameters passed to PostgreSQL, adapted to the database format.
- `types` (sequence of `int`): the OID of the parameters passed to PostgreSQL.
- `formats` (sequence of *pq.Format*): whether the parameter format is text or binary.

The `ClientCursor` class

See also:

See *Client-side-binding cursors* for details.

class psycopg.**ClientCursor**(*connection*, *, *row_factory=None*)

This *Cursor* subclass has exactly the same interface of its parent class, but, instead of sending query and parameters separately to the server, it merges them on the client and sends them as a non-parametric query on the server. This allows, for instance, to execute parametrized data definition statements and other *problematic queries*.

New in version 3.1.

mogrify(*query*, *params=None*)

Return the query and parameters merged.

Parameters are adapted and merged to the query the same way that `execute()` would do.

Return type
str

Parameters

- **query** (`str`, `bytes`, *sql.SQL*, or *sql.Composed*) – The query to execute.
- **params** (*Sequence or Mapping*) – The parameters to pass to the query, if any.

The ServerCursor class

See also:

See *Server-side cursors* for details.

class psycopg.**ServerCursor**(*connection*, *name*, *, *row_factory=None*, *scrollable=None*, *withhold=False*)

This class also implements a DBAPI-compliant interface. It is created by *Connection.cursor()* specifying the `name` parameter. Using this object results in the creation of an equivalent PostgreSQL cursor in the server. DBAPI-extension methods (such as *copy()* or *stream()*) are not implemented on this object: use a normal *Cursor* instead.

Most attribute and methods behave exactly like in *Cursor*, here are documented the differences:

name: str

The name of the cursor.

scrollable: Optional[bool]

Whether the cursor is scrollable or not.

If `None` leave the choice to the server. Use `True` if you want to use *scroll()* on the cursor.

See also:

The PostgreSQL DECLARE statement documentation for the description of `[NO] SCROLL`.

withhold: bool

If the cursor can be used after the creating transaction has committed.

See also:

The PostgreSQL DECLARE statement documentation for the description of `{WITH|WITHOUT} HOLD`.

close()

Close the current cursor and free associated resources.

> **Warning:** Closing a server-side cursor is more important than closing a client-side one because it also releases the resources on the server, which otherwise might remain allocated until the end of the session (memory, locks). Using the pattern:
>
> ```
> with conn.cursor():
>     ...
> ```
>
> is especially useful so that the cursor is closed at the end of the block.

execute(*query*, *params=None*, *, *binary=None*, ***kwargs*)

Open a cursor to execute a query to the database.

Return type
TypeVar(_Self, bound= ServerCursor[Any])

Parameters

- **query** (str, bytes, *sql.SQL*, or *sql.Composed*) – The query to execute.
- **params** (*Sequence or Mapping*) – The parameters to pass to the query, if any.
- **binary** – Specify whether the server should return data in binary format (`True`) or in text format (`False`). By default (`None`) return data as requested by the cursor's *format*.

Create a server cursor with given *name* and the `query` in argument.

If using `DECLARE` is not appropriate (for instance because the cursor is returned by calling a stored procedure) you can avoid to use `execute()`, crete the cursor in other ways, and use directly the `fetch*()` methods instead. See *"Stealing" an existing cursor* for an example.

Using `execute()` more than once will close the previous cursor and open a new one with the same name.

executemany(*query*, *params_seq*, *, *returning=True*)

Method not implemented for server-side cursors.

fetchone()

Return the next record from the current recordset.

Return `None` the recordset is finished.

Return type
Optional[Row], with Row defined by `row_factory`

fetchmany(*size=0*)

Return the next *size* records from the current recordset.

size default to `self.arraysize` if not specified.

Return type
Sequence[Row], with Row defined by `row_factory`

fetchall()

Return all the remaining records from the current recordset.

Return type
Sequence[Row], with Row defined by `row_factory`

These methods use the FETCH SQL statement to retrieve some of the records from the cursor's current position.

Note: You can also iterate on the cursor to read its result one at time with:

```
for record in cur:
    ...
```

In this case, the records are not fetched one at time from the server but they are retrieved in batches of *itersize* to reduce the number of server roundtrips.

---

itersize: `int`

Number of records to fetch at time when iterating on the cursor. The default is 100.

scroll(*value*, *mode='relative'*)

Move 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.

Raise `IndexError` in case a scroll operation would leave the result set. In this case the position will not change.

This method uses the MOVE SQL statement to move the current position in the server-side cursor, which will affect following `fetch*()` operations. If you need to scroll backwards you should probably call *cursor()* using `scrollable=True`.

Note that PostgreSQL doesn't provide a reliable way to report when a cursor moves out of bound, so the method might not raise `IndexError` when it happens, but it might rather stop at the cursor boundary.

The `AsyncCursor` class

class psycopg.**AsyncCursor**(*connection*, ***, *row_factory=None*)

This class implements a DBAPI-inspired interface, with all the blocking methods implemented as coroutines. Unless specified otherwise, non-blocking methods are shared with the *Cursor* class.

The following methods have the same behaviour of the matching `Cursor` methods, but should be called using the `await` keyword.

connection: *AsyncConnection*

async close()

Note: You can use:

```
async with conn.cursor():
    ...
```

to close the cursor automatically when the block is exited.

---

async execute(*query*, *params=None*, ***, *prepare=None*, *binary=None*)

Return type
TypeVar(_Self, bound= AsyncCursor[Any])

async executemany(*query*, *params_seq*, ***, *returning=False*)

copy(*statement*, *params=None*, ***, *writer=None*)

Return type
AsyncCopy

Note: The method must be called with:

```
async with cursor.copy() as copy:
    ...
```

---

async stream(*query*, *params=None*, ***, *binary=None*)

Return type
AsyncIterator[TypeVar(Row, covariant=True)]

Note: The method must be called with:

```
async for record in cursor.stream(query):
    ...
```

---

async fetchone()

Return type
Optional[TypeVar(Row, covariant=True)]

async fetchmany(*size=0*)

Return type
List[TypeVar(Row, covariant=True)]

async fetchall()

Return type
List[TypeVar(Row, covariant=True)]

async scroll(*value*, *mode='relative'*)

Note: You can also use:

```
async for record in cursor:
    ...
```

to iterate on the async cursor results.

The AsyncClientCursor class

class psycopg.**AsyncClientCursor**(*connection*, ***, *row_factory=None*)

This class is the `async` equivalent of the *ClientCursor*. The difference are the same shown in *AsyncCursor*.

New in version 3.1.

The AsyncServerCursor class

class psycopg.**AsyncServerCursor**(*connection*, *name*, ***, *row_factory=None*, *scrollable=None*, *withhold=False*)

This class implements a DBAPI-inspired interface as the *AsyncCursor* does, but wraps a server-side cursor like the *ServerCursor* class. It is created by *AsyncConnection.cursor()* specifying the `name` parameter.

The following are the methods exposing a different (async) interface from the *ServerCursor* counterpart, but sharing the same semantics.

async close()

Note: You can close the cursor automatically using:

```
async with conn.cursor("name") as cursor:
    ...
```

---

async execute(*query*, *params=None*, ***, *binary=None*, ***kwargs*)

Return type
TypeVar(_Self, bound= AsyncServerCursor[Any])

async executemany(*query*, *params_seq*, *, *returning=True*)

async fetchone()

Return type
Optional[TypeVar(Row, covariant=True)]

async fetchmany(*size=0*)

Return type
List[TypeVar(Row, covariant=True)]

async fetchall()

Return type
List[TypeVar(Row, covariant=True)]

Note: You can also iterate on the cursor using:

```
async for record in cur:
    ...
```

async scroll(*value*, *mode='relative'*)

1.3.4 COPY-related objects

The main objects (*Copy*, *AsyncCopy*) present the main interface to exchange data during a COPY operations. These objects are normally obtained by the methods *Cursor.copy()* and *AsyncCursor.copy()*; however, they can be also created directly, for instance to write to a destination which is not a database (e.g. using a *FileWriter*).

See *Using COPY TO and COPY FROM* for details.

Main Copy objects

class psycopg.**Copy**(*cursor*, *, *binary=None*, *writer=None*)

Manage a COPY operation.

Parameters

- **cursor** (*Cursor*[Any]) – the cursor where the operation is performed.
- **binary** (Optional[bool]) – if True, write binary format.
- **writer** (Optional[*Writer*]) – the object to write to destination. If not specified, write to the cursor connection.

Choosing binary is not necessary if the cursor has executed a COPY operation, because the operation result describes the format too. The parameter is useful when a Copy object is created manually and no operation is performed on the cursor, such as when using writer=*FileWriter*.

The object is normally returned by with *Cursor.copy()*.

write_row(*row*)

Write a record to a table after a COPY FROM operation.

The data in the tuple will be converted as configured on the cursor; see *Data adaptation configuration* for details.

write(*buffer*)

Write a block of data to a table after a `COPY FROM` operation.

If the `COPY` is in binary format *buffer* must be `bytes`. In text mode it can be either `bytes` or `str`.

read()

Read an unparsed row after a `COPY TO` operation.

Return an empty string when the data is finished.

Return type
`memoryview`

Instead of using `read()` you can iterate on the `Copy` object to read its data row by row, using `for row in copy: ...`.

rows()

Iterate on the result of a `COPY TO` operation record by record.

Note that the records returned will be tuples of unparsed strings or bytes, unless data types are specified using *set_types()*.

Return type
`Iterator[Tuple[Any, ...]]`

Equivalent of iterating on *read_row()* until it returns `None`

read_row()

Read a parsed row of data from a table after a `COPY TO` operation.

Return `None` when the data is finished.

Note that the records returned will be tuples of unparsed strings or bytes, unless data types are specified using *set_types()*.

Return type
`Optional[Tuple[Any, ...]]`

set_types(*types*)

Set the types expected in a COPY operation.

The types must be specified as a sequence of oid or PostgreSQL type names (e.g. `int4`, `timestamptz[]`).

This operation overcomes the lack of metadata returned by PostgreSQL when a COPY operation begins:

- On `COPY TO`, `set_types()` allows to specify what types the operation returns. If `set_types()` is not used, the data will be returned as unparsed strings or bytes instead of Python objects.
- On `COPY FROM`, `set_types()` allows to choose what type the database expects. This is especially useful in binary copy, because PostgreSQL will apply no cast rule.

class psycopg.**AsyncCopy**(*cursor*, *, *binary=None*, *writer=None*)

Manage an asynchronous `COPY` operation.

The object is normally returned by `async with` *AsyncCursor.copy()*. Its methods are similar to the ones of the *Copy* object but offering an `asyncio` interface (`await`, `async for`, `async with`).

async write_row(*row*)

async write(*buffer*)

async read()

Return type
memoryview

Instead of using `read()` you can iterate on the `AsyncCopy` object to read its data row by row, using `async for row in copy: ...`.

async rows()

Return type
AsyncIterator[Tuple[Any, ...]]

Use it as `async for record in copy.rows(): ...`

async read_row()

Return type
Optional[Tuple[Any, ...]]

Writer objects

New in version 3.1.

Copy writers are helper objects to specify where to write COPY-formatted data. By default, data is written to the database (using the *LibpqWriter*). It is possible to write copy-data for offline use by using a *FileWriter*, or to customize further writing by implementing your own *Writer* or *AsyncWriter* subclass.

Writers instances can be used passing them to the cursor *copy()* method or to the *Copy* constructor, as the `writer` argument.

class psycopg.copy.**Writer**

A class to write copy data somewhere.

This is an abstract base class: subclasses are required to implement their *write()* method.

abstract write(*data*)

Write some data to destination.

finish(*exc=None*)

Called when write operations are finished.

If operations finished with an error, it will be passed to `exc`.

class psycopg.copy.**LibpqWriter**(*cursor*)

A *Writer* to write copy data to a Postgres database.

This is the writer used by default if none is specified.

class psycopg.copy.**FileWriter**(*file*)

A *Writer* to write copy data to a file-like object.

Parameters
file (IO[bytes]) – the file where to write copy data. It must be open for writing in binary mode.

This writer should be used without executing a `COPY` operation on the database. For example, if `records` is a list of tuples containing data to save in COPY format to a file (e.g. for later import), it can be used as:

```
with open("target-file.pgcopy", "wb") as f:
    with Copy(cur, writer=FileWriter(f)) as copy:
        for record in records
            copy.write_row(record)
```

class psycopg.copy.**AsyncWriter**

A class to write copy data somewhere (for async connections).

This class methods have the same semantics of the ones of *Writer*, but offer an async interface.

abstract async write(*data*)

async finish(*exc=None*)

class psycopg.copy.**AsyncLibpqWriter**(*cursor*)

An *AsyncWriter* to write copy data to a Postgres database.

1.3.5 Other top-level objects

Connection information

class psycopg.**ConnectionInfo**(*pgconn*)

Allow access to information about the connection.

The object is usually returned by *Connection.info*.

dsn: str

Return the connection string to connect to the database.

The string contains all the parameters set to a non-default value, which might come either from the connection string and parameters passed to *connect()* or from environment variables. The password is never returned (you can read it using the *password* attribute).

Note: The *get_parameters()* method returns the same information as a dict.

---

status: *pq.ConnStatus*

The status of the connection. See PQstatus().

The status can be one of a number of values. However, only two of these are seen outside of an asynchronous connection procedure: *OK* and *BAD*. A good connection to the database has the status OK. Ordinarily, an OK status will remain so until *Connection.close()*, but a communications failure might result in the status changing to BAD prematurely.

transaction_status: *pq.TransactionStatus*

The current in-transaction status of the session. See PQtransactionStatus().

The status can be *IDLE* (currently idle), *ACTIVE* (a command is in progress), *INTRANS* (idle, in a valid transaction block), or *INERROR* (idle, in a failed transaction block). *UNKNOWN* is reported if the connection is bad. ACTIVE is reported only when a query has been sent to the server and not yet completed.

pipeline_status: *pq.PipelineStatus*

The current pipeline status of the client. See PQpipelineStatus().

backend_pid: int

The process ID (PID) of the backend process handling this connection. See PQbackendPID().

vendor: *str*

A string representing the database vendor connected to.

Normally it is `PostgreSQL`; it may be different if connected to a different database.

New in version 3.1.

server_version: *int*

An integer representing the server version. See `PQserverVersion()`.

The number is formed by converting the major, minor, and revision numbers into two-decimal-digit numbers and appending them together. Starting from PostgreSQL 10 the minor version was dropped, so the second group of digits is always 00. For example, version 9.3.5 is returned as 90305, version 10.2 as 100002.

error_message: *str*

The error message most recently generated by an operation on the connection. See `PQerrorMessage()`.

get_parameters()

Return the connection parameters values.

Return all the parameters set to a non-default value, which might come either from the connection string and parameters passed to *connect()* or from environment variables. The password is never returned (you can read it using the *password* attribute).

Return type
`Dict[str, str]`

Note: The *dsn* attribute returns the same information in the form as a string.

---

timezone: *datetime.tzinfo*

The Python timezone info of the connection's timezone.

```
>>> conn.info.timezone
zoneinfo.ZoneInfo(key='Europe/Rome')
```

host: *str*

The server host name of the active connection. See `PQhost()`.

This can be a host name, an IP address, or a directory path if the connection is via Unix socket. (The path case can be distinguished because it will always be an absolute path, beginning with `/`.)

hostaddr: *str*

The server IP address of the connection. See `PQhostaddr()`.

Only available if the libpq used is at least from PostgreSQL 12. Raise *NotSupportedError* otherwise.

port: *int*

The port of the active connection. See `PQport()`.

dbname: *str*

The database name of the connection. See `PQdb()`.

user: *str*

The user name of the connection. See `PQuser()`.

password: *str*

The password of the connection. See `PQpass()`.

options: `str`

The command-line options passed in the connection request. See `PQoptions`.

parameter_status(*param_name*)

Return a parameter setting of the connection.

Return `None` is the parameter is unknown.

Return type
`Optional[str]`

Example of parameters are `server_version`, `standard_conforming_strings`… See `PQparameterStatus()` for all the available parameters.

encoding: `str`

The Python codec name of the connection's client encoding.

The value returned is always normalized to the Python codec `name`:

```
conn.execute("SET client_encoding TO LATIN9")
conn.info.encoding
'iso8859-15'
```

A few PostgreSQL encodings are not available in Python and cannot be selected (currently `EUC_TW`, `MULE_INTERNAL`). The PostgreSQL `SQL_ASCII` encoding has the special meaning of "no encoding": see *Strings adaptation* for details.

See also:

The PostgreSQL supported encodings.

The description `Column` object

class psycopg.**Column**(*cursor*, *index*)

An object describing a column of data from a database result, as described by the DBAPI, so it can also be unpacked as a 7-items tuple.

The object is returned by *`Cursor.description`*.

name: `str`

The name of the column.

type_code: `int`

The numeric OID of the column.

display_size: `Optional[int]`

The field size, for `varchar(n)`, None otherwise.

internal_size: `Optional[int]`

The internal field size for fixed-size types, None otherwise.

precision: `Optional[int]`

The number of digits for fixed precision types.

scale: `Optional[int]`

The number of digits after the decimal point if available.

Notifications

class psycopg.**Notify**

An asynchronous notification received from the database.

The object is usually returned by *Connection.notifies()*.

channel: str

The name of the channel on which the notification was received.

payload: str

The message attached to the notification.

pid: int

The PID of the backend process which sent the notification.

Pipeline-related objects

See *Pipeline mode support* for details.

class psycopg.**Pipeline**(*conn*)

Handler for connection in pipeline mode.

This objects is returned by *Connection.pipeline()*.

sync()

Sync the pipeline, send any pending command and receive and process all available results.

classmethod is_supported()

Return `True` if the psycopg libpq wrapper supports pipeline mode.

Return type
bool

class psycopg.**AsyncPipeline**(*conn*)

Handler for async connection in pipeline mode.

This objects is returned by *AsyncConnection.pipeline()*.

async sync()

Transaction-related objects

See *Transactions management* for details about these objects.

class psycopg.**IsolationLevel**(*value*)

Enum representing the isolation level for a transaction.

The value is usually used with the *Connection.isolation_level* property.

Check the PostgreSQL documentation for a description of the effects of the different levels of transaction isolation.

READ_UNCOMMITTED = 1

READ_COMMITTED = 2

REPEATABLE_READ = 3

SERIALIZABLE = 4

class psycopg.**Transaction**(*connection*, *savepoint_name=None*, *force_rollback=False*)

Returned by *Connection.transaction()* to handle a transaction block.

savepoint_name: Optional[str]

The name of the savepoint; None if handling the main transaction.

connection: *Connection*[Any]

The connection the object is managing.

class psycopg.**AsyncTransaction**(*connection*, *savepoint_name=None*, *force_rollback=False*)

Returned by *AsyncConnection.transaction()* to handle a transaction block.

connection: *AsyncConnection*[Any]

exception psycopg.**Rollback**(*transaction=None*)

Exit the current *Transaction* context immediately and rollback any changes made within this context.

If a transaction context is specified in the constructor, rollback enclosing transactions contexts up to and including the one specified.

It can be used as

- `raise Rollback`: roll back the operation that happened in the current transaction block and continue the program after the block.
- `raise Rollback()`: same effect as above
- `raise Rollback(tx)`: roll back any operation that happened in the *Transaction* `tx` (returned by a statement such as `with conn.transaction() as tx:` and all the blocks nested within. The program will continue after the `tx` block.

Two-Phase Commit related objects

class psycopg.**Xid**(*format_id*, *gtrid*, *bqual*, *prepared=None*, *owner=None*, *database=None*)

A two-phase commit transaction identifier.

The object can also be unpacked as a 3-item tuple (*format_id*, *gtrid*, *bqual*).

See *Two-Phase Commit protocol support* for details.

format_id: Optional[int]

Format Identifier of the two-phase transaction.

gtrid: str

Global Transaction Identifier of the two-phase transaction.

If the Xid doesn't follow the XA standard, it will be the PostgreSQL ID of the transaction (in which case *format_id* and *bqual* will be None).

bqual: Optional[str]

Branch Qualifier of the two-phase transaction.

prepared: Optional[datetime] = None

Timestamp at which the transaction was prepared for commit.

Only available on transactions recovered by *tpc_recover()*.

owner: `Optional[str] = None`

Named of the user that executed the transaction.

Only available on recovered transactions.

database: `Optional[str] = None`

Named of the database in which the transaction was executed.

Only available on recovered transactions.

1.3.6 `sql` – SQL string composition

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 `escape_identifier()`:

```
from psycopg.pq import Escaping

# This works, but it is not optimal
table_name = 'my_table'
cur.execute(
    "INSERT INTO %s VALUES (%%s, %%s)" % Escaping.escape_identifier(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. It is also still relatively dangerous: if `escape_identifier()` 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 `psycopg.sql` module allow generating SQL statements on the fly, separating clearly the variable parts of the statement from the query parameters:

```
from psycopg import sql

cur.execute(
    sql.SQL("INSERT INTO {} VALUES (%s, %s)")
        .format(sql.Identifier('my_table')),
    [10, 20])
```

Module usage

Usually you should express the template of your query as an *SQL* instance with {}-style placeholders and use *format()* to merge the variable parts into them, all of which must be *Composable* subclasses. You can still have %s-style placeholders in your query and pass values to *execute()*: such value placeholders will be untouched by format():

```
query = sql.SQL("SELECT {field} FROM {table} WHERE {pkey} = %s").format(
    field=sql.Identifier('my_name'),
    table=sql.Identifier('some_table'),
    pkey=sql.Identifier('id'))
```

The resulting object is meant to be passed directly to cursor methods such as *execute()*, *executemany()*, *copy()*, but can also be used to compose a query as a Python string, using the *as_string()* method:

```
cur.execute(query, (42,))
full_query = query.as_string(cur)
```

If part of your query is a variable sequence of arguments, such as a comma-separated list of field names, you can use the *SQL.join()* method to pass them to the query:

```
query = sql.SQL("SELECT {fields} FROM {table}").format(
    fields=sql.SQL(',').join([
        sql.Identifier('field1'),
        sql.Identifier('field2'),
        sql.Identifier('field3'),
    ]),
    table=sql.Identifier('some_table'))
```

sql objects

The sql objects are in the following inheritance hierarchy:

Composable: the base class exposing the common interface
|__ *SQL*: a literal snippet of an SQL query
|__ *Identifier*: a PostgreSQL identifier or dot-separated sequence of identifiers
|__ *Literal*: a value hardcoded into a query
|__ *Placeholder*: a %s-style placeholder whose value will be added later e.g. by *execute()*
|__ *Composed*: a sequence of Composable instances.

class psycopg.sql.**Composable**(*obj*)

Abstract base class for objects that can be used to compose an SQL string.

Composable objects can be passed directly to *execute()*, *executemany()*, *copy()* 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.

abstract as_bytes(*context*)

Return the value of the object as bytes.

Parameters
context (connection or cursor) – the context to evaluate the object into.

The method is automatically invoked by *execute()*, *executemany()*, *copy()* if a Composable is passed instead of the query string.

Return type
bytes

as_string(*context*)

Return the value of the object as string.

Parameters
context (connection or cursor) – the context to evaluate the string into.

Return type
str

class psycopg.sql.**SQL**(*obj*)

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"
```

Changed in version 3.1: The input object should be a LiteralString. See **PEP 675** for details.

format(**args*, ***kwargs*)

Merge *Composable* objects into a template.

Parameters

- **args** (Any) – parameters to replace to numbered ({0}, {1}) or auto-numbered ({}) placeholders
- **kwargs** (Any) – 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 ({}), 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.

If a Composable objects is passed to the template it will be merged according to its as_string() method. If any other Python object is passed, it will be wrapped in a *Literal* object and so escaped according to SQL rules.

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 name = {name}")
...     .format(tbl=sql.Identifier('people'), name="O'Rourke"))
...     .as_string(conn))
SELECT * FROM "people" WHERE name = 'O''Rourke'
```

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"
```

Return type
Composed

class psycopg.sql.**Identifier**(**strings*)

A *Composable* representing an SQL identifier or a dot-separated sequence.

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"
```

Multiple strings can be passed to the object to represent a qualified name, i.e. a dot-separated sequence of identifiers.

Example:

```
>>> query = sql.SQL("SELECT {} FROM {}").format(
...     sql.Identifier("table", "field"),
...     sql.Identifier("schema", "table"))
>>> print(query.as_string(conn))
SELECT "table"."field" FROM "schema"."table"
```

class psycopg.sql.**Literal**(*obj*)

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("fo'o")
>>> s2 = sql.Literal(42)
>>> s3 = sql.Literal(date(2000, 1, 1))
>>> print(sql.SQL(', ').join([s1, s2, s3]).as_string(conn))
'fo''o', 42, '2000-01-01'::date
```

Changed in version 3.1: Add a type cast to the representation if useful in ambiguous context (e.g. `'2000-01-01'::date`)

class psycopg.sql.**Placeholder**(*name=''*, *format=PyFormat.AUTO*)

A *Composable* representing a placeholder for query parameters.

If the name is specified, generate a named placeholder (e.g. `%(name)s`, `%(name)b`), otherwise generate a positional placeholder (e.g. `%s`, `%b`).

The object is useful to generate SQL queries with a variable number of arguments.

Examples:

```
>>> names = ['foo', 'bar', 'baz']

>>> q1 = sql.SQL("INSERT INTO my_table ({}) VALUES ({})").format(
...     sql.SQL(', ').join(map(sql.Identifier, names)),
...     sql.SQL(', ').join(sql.Placeholder() * len(names)))
>>> print(q1.as_string(conn))
INSERT INTO my_table ("foo", "bar", "baz") VALUES (%s, %s, %s)

>>> q2 = sql.SQL("INSERT INTO my_table ({}) VALUES ({})").format(
...     sql.SQL(', ').join(map(sql.Identifier, names)),
...     sql.SQL(', ').join(map(sql.Placeholder, names)))
>>> print(q2.as_string(conn))
INSERT INTO my_table ("foo", "bar", "baz") VALUES (%(foo)s, %(bar)s, %(baz)s)
```

class psycopg.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 objects as arguments: if they are not `Composable` they will be wrapped in a *Literal*.

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).

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"
```

Return type
Composed

Utility functions

psycopg.sql.**quote**(*obj*, *context=None*)

Adapt a Python object to a quoted SQL string.

Use this function only if you absolutely want to convert a Python string to an SQL quoted literal to use e.g. to generate batch SQL and you won't have a connection available when you will need to use it.

This function is relatively inefficient, because it doesn't cache the adaptation rules. If you pass a *context* you can adapt the adaptation rules used, otherwise only global rules are used.

Return type
str

psycopg.sql.**NULL**

psycopg.sql.**DEFAULT**

`sql.SQL` objects often useful in queries.

1.3.7 `rows` – row factory implementations

The module exposes a few generic `RowFactory` implementation, which can be used to retrieve data from the database in more complex structures than the basic tuples.

Check out *Row factories* for information about how to use these objects.

psycopg.rows.**tuple_row**(*cursor*)

Row factory to represent rows as simple tuples.

This is the default factory, used when *connect()* or *cursor()* are called without a `row_factory` parameter.

Return type
RowMaker[Tuple[Any, ...]]

psycopg.rows.**dict_row**(*cursor*)

Row factory to represent rows as dictionaries.

The dictionary keys are taken from the column names of the returned columns.

Return type
RowMaker[Dict[str, Any]]

psycopg.rows.**namedtuple_row**(*cursor*)

Row factory to represent rows as `namedtuple`.

The field names are taken from the column names of the returned columns, with some mangling to deal with invalid names.

Return type
RowMaker[NamedTuple]

psycopg.rows.**class_row**(*cls*)

Generate a row factory to represent rows as instances of the class *cls*.

The class must support every output column name as a keyword parameter.

Parameters
cls (Type[TypeVar(T, covariant=True)]) – The class to return for each row. It must support the fields returned by the query as keyword arguments.

Return type
Callable[[Cursor], *RowMaker*[~T]]

This is not a row factory, but rather a factory of row factories. Specifying `row_factory=class_row(MyClass)` will create connections and cursors returning `MyClass` objects on fetch.

Example:

```
from dataclasses import dataclass
import psycopg
from psycopg.rows import class_row

@dataclass
class Person:
    first_name: str
    last_name: str
    age: int = None

conn = psycopg.connect()
cur = conn.cursor(row_factory=class_row(Person))

cur.execute("select 'John' as first_name, 'Smith' as last_name").fetchone()
# Person(first_name='John', last_name='Smith', age=None)
```

psycopg.rows.**args_row**(*func*)

Generate a row factory calling *func* with positional parameters for every row.

Parameters
func (Callable[..., TypeVar(T, covariant=True)]) – The function to call for each row. It must support the fields returned by the query as positional arguments.

Return type
BaseRowFactory[TypeVar(T, covariant=True)]

psycopg.rows.**kwargs_row**(*func*)

Generate a row factory calling *func* with keyword parameters for every row.

Parameters
func (Callable[..., TypeVar(T, covariant=True)]) – The function to call for each row. It must support the fields returned by the query as keyword arguments.

Return type
BaseRowFactory[TypeVar(T, covariant=True)]

Formal rows protocols

These objects can be used to describe your own rows adapter for static typing checks, such as mypy.

class psycopg.rows.**RowMaker**

Callable protocol taking a sequence of value and returning an object.

The sequence of value is what is returned from a database query, already adapted to the right Python types. The return value is the object that your program would like to receive: by default (*tuple_row()*) it is a simple tuple, but it may be any type of object.

Typically, `RowMaker` functions are returned by *RowFactory*.

__call__(*values: Sequence[Any]*) → Row

Convert a sequence of values from the database to a finished object.

class psycopg.rows.**RowFactory**

Callable protocol taking a *Cursor* and returning a *RowMaker*.

A `RowFactory` is typically called when a `Cursor` receives a result. This way it can inspect the cursor state (for instance the *description* attribute) and help a `RowMaker` to create a complete object.

For instance the *dict_row()* `RowFactory` uses the names of the column to define the dictionary key and returns a `RowMaker` function which would use the values to create a dictionary for each record.

__call__(*cursor:* Cursor*[Row]*) → *RowMaker*[Row]

Inspect the result on a cursor and return a *RowMaker* to convert rows.

class psycopg.rows.**AsyncRowFactory**

Like *RowFactory*, taking an async cursor as argument.

class psycopg.rows.**BaseRowFactory**

Like *RowFactory*, taking either type of cursor as argument.

Note that it's easy to implement an object implementing both `RowFactory` and `AsyncRowFactory`: usually, everything you need to implement a row factory is to access the cursor's *description*, which is provided by both the cursor flavours.

1.3.8 errors – Package exceptions

This module exposes objects to represent and examine database errors.

DB-API exceptions

In compliance with the DB-API, all the exceptions raised by Psycopg derive from the following classes:

```
Exception
|__ Warning
|__ Error
    |__ InterfaceError
    |__ DatabaseError
        |__ DataError
        |__ OperationalError
```

```
|__ IntegrityError
|__ InternalError
|__ ProgrammingError
|__ NotSupportedError
```

These classes are exposed both by this module and the root *psycopg* module.

exception psycopg.**Error**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

Base exception for all the errors psycopg will raise.

Exception that is the base class of all other error exceptions. You can use this to catch all errors with one single `except` statement.

This exception is guaranteed to be picklable.

diag: *errors.Diagnostic*

A `Diagnostic` object to inspect details of the errors from the database.

sqlstate: `Optional[str] = None`

The code of the error, if received from the server.

This attribute is also available as class attribute on the *SQLSTATE exceptions* classes.

pgconn: `Optional[pq.PGconn]`

The connection object, if the error was raised from a connection attempt.

Most likely it will be in *BAD* state; however it might be useful to verify precisely what went wrong, for instance checking the *needs_password* and *used_password* attributes.

New in version 3.1.

pgresult: `Optional[pq.PGresult]`

The result object, if the exception was raised after a failed query.

New in version 3.1.

exception psycopg.**Warning**

Exception raised for important warnings.

Defined for DBAPI compatibility, but never raised by `psycopg`.

exception psycopg.**InterfaceError**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

An error related to the database interface rather than the database itself.

exception psycopg.**DatabaseError**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

Exception raised for errors that are related to the database.

exception psycopg.**DataError**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

An error caused by problems with the processed data.

Examples may be division by zero, numeric value out of range, etc.

exception psycopg.**OperationalError**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

An error related to the database's operation.

These errors are 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.

exception psycopg.**IntegrityError**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

An error caused when the relational integrity of the database is affected.

An example may be a foreign key check failed.

exception psycopg.**InternalError**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

An error generated when the database encounters an internal error,

Examples could be the cursor is not valid anymore, the transaction is out of sync, etc.

exception psycopg.**ProgrammingError**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

Exception raised for programming errors

Examples may be table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc.

exception psycopg.**NotSupportedError**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

A method or database API was used which is not supported by the database.

Other Psycopg errors

In addition to the standard DB-API errors, Psycopg defines a few more specific ones.

exception psycopg.errors.**ConnectionTimeout**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

Exception raised on timeout of the *connect()* method.

The error is raised if the `connect_timeout` is specified and a connection is not obtained in useful time.

Subclass of *OperationalError*.

exception psycopg.errors.**PipelineAborted**(**args*, *info=None*, *encoding='utf-8'*, *pgconn=None*)

Raised when a operation fails because the current pipeline is in aborted state.

Subclass of *OperationalError*.

Error diagnostics

class psycopg.errors.**Diagnostic**(*info*, *encoding='utf-8'*)

Details from a database error report.

The object is available as the *Error.diag* attribute and is passed to the callback functions registered with *add_notice_handler()*.

All the information available from the `PQresultErrorField()` function are exposed as attributes by the object. For instance the `severity` attribute returns the `PG_DIAG_SEVERITY` code. Please refer to the PostgreSQL documentation for the meaning of all the attributes.

The attributes available are:

column_name

constraint_name

context

datatype_name

internal_position

internal_query

message_detail

message_hint

message_primary

schema_name

severity

severity_nonlocalized

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 for errors sent by the server: not all the fields are available for all the errors and for all the server versions.

SQLSTATE exceptions

Errors coming from a database server (as opposite as ones generated client-side, such as connection failed) usually have a 5-letters error code called SQLSTATE (available in the *`sqlstate`* attribute of the error's *`diag`* attribute).

Psycopg exposes a different class for each SQLSTATE value, allowing to write idiomatic error handling code according to specific conditions happening in the database:

```
try:
    cur.execute("LOCK TABLE mytable IN ACCESS EXCLUSIVE MODE NOWAIT")
except psycopg.errors.LockNotAvailable:
    locked = True
```

The exception names are generated from the PostgreSQL source code and includes classes for every error defined by PostgreSQL in versions between 9.6 and 15. Every class in the module is named after what referred as "condition name" in the documentation, converted to CamelCase: e.g. the error 22012, `division_by_zero` is exposed by this module as the class `DivisionByZero`. There is a handful of... exceptions to this rule, required for disambiguate name clashes: please refer to the *table below* for all the classes defined.

Every exception class is a subclass of one of the *standard DB-API exception*, thus exposing the *`Error`* interface.

Changed in version 3.1.4: Added exceptions introduced in PostgreSQL 15.

`psycopg.errors.`**lookup**(*sqlstate*)

Lookup an error code or constant name and return its exception class.

Raise `KeyError` if the code is not found.

Return type
`Type`[*`Error`*]

Example: if you have code using constant names or sql codes you can use them to look up the exception class.

```
try:
    cur.execute("LOCK TABLE mytable IN ACCESS EXCLUSIVE MODE NOWAIT")
except psycopg.errors.lookup("UNDEFINED_TABLE"):
    missing = True
except psycopg.errors.lookup("55P03"):
    locked = True
```

List of known exceptions

The following are all the SQLSTATE-related error classed defined by this module, together with the base DBAPI exception they derive from.

SQLSTATE	Exception	Base exception
Class 02 - No Data (this is also a warning class per the SQL standard)		
02000	NoData	DatabaseError
02001	NoAdditionalDynamicResultSetsReturned	DatabaseError
Class 03 - SQL Statement Not Yet Complete		
03000	SqlStatementNotYetComplete	DatabaseError
Class 08 - Connection Exception		
08000	ConnectionException	OperationalError
08001	SqlclientUnableToEstablishSqlconnection	OperationalError
08003	ConnectionDoesNotExist	OperationalError
08004	SqlserverRejectedEstablishmentOfSqlconnection	OperationalError
08006	ConnectionFailure	OperationalError
08007	TransactionResolutionUnknown	OperationalError
08P01	ProtocolViolation	OperationalError
Class 09 - Triggered Action Exception		
09000	TriggeredActionException	DatabaseError
Class 0A - Feature Not Supported		
0A000	FeatureNotSupported	NotSupportedError
Class 0B - Invalid Transaction Initiation		
0B000	InvalidTransactionInitiation	DatabaseError
Class 0F - Locator Exception		
0F000	LocatorException	DatabaseError
0F001	InvalidLocatorSpecification	DatabaseError
Class 0L - Invalid Grantor		
0L000	InvalidGrantor	DatabaseError
0LP01	InvalidGrantOperation	DatabaseError
Class 0P - Invalid Role Specification		
0P000	InvalidRoleSpecification	DatabaseError
Class 0Z - Diagnostics Exception		
0Z000	DiagnosticsException	DatabaseError
0Z002	StackedDiagnosticsAccessedWithoutActiveHandler	DatabaseError
Class 20 - Case Not Found		
20000	CaseNotFound	ProgrammingError
Class 21 - Cardinality Violation		
21000	CardinalityViolation	ProgrammingError
Class 22 - Data Exception		
22000	DataException	DataError
22001	StringDataRightTruncation	DataError
22002	NullValueNoIndicatorParameter	DataError
22003	NumericValueOutOfRange	DataError
22004	NullValueNotAllowed	DataError
22005	ErrorInAssignment	DataError
22007	InvalidDatetimeFormat	DataError
22008	DatetimeFieldOverflow	DataError
22009	InvalidTimeZoneDisplacementValue	DataError
2200B	EscapeCharacterConflict	DataError

continues on next page

Table 1 – continued from previous page

SQLSTATE	Exception	Base exception
2200C	InvalidUseOfEscapeCharacter	DataError
2200D	InvalidEscapeOctet	DataError
2200F	ZeroLengthCharacterString	DataError
2200G	MostSpecificTypeMismatch	DataError
2200H	SequenceGeneratorLimitExceeded	DataError
2200L	NotAnXmlDocument	DataError
2200M	InvalidXmlDocument	DataError
2200N	InvalidXmlContent	DataError
2200S	InvalidXmlComment	DataError
2200T	InvalidXmlProcessingInstruction	DataError
22010	InvalidIndicatorParameterValue	DataError
22011	SubstringError	DataError
22012	DivisionByZero	DataError
22013	InvalidPrecedingOrFollowingSize	DataError
22014	InvalidArgumentForNtileFunction	DataError
22015	IntervalFieldOverflow	DataError
22016	InvalidArgumentForNthValueFunction	DataError
22018	InvalidCharacterValueForCast	DataError
22019	InvalidEscapeCharacter	DataError
2201B	InvalidRegularExpression	DataError
2201E	InvalidArgumentForLogarithm	DataError
2201F	InvalidArgumentForPowerFunction	DataError
2201G	InvalidArgumentForWidthBucketFunction	DataError
2201W	InvalidRowCountInLimitClause	DataError
2201X	InvalidRowCountInResultOffsetClause	DataError
22021	CharacterNotInRepertoire	DataError
22022	IndicatorOverflow	DataError
22023	InvalidParameterValue	DataError
22024	UnterminatedCString	DataError
22025	InvalidEscapeSequence	DataError
22026	StringDataLengthMismatch	DataError
22027	TrimError	DataError
2202E	ArraySubscriptError	DataError
2202G	InvalidTablesampleRepeat	DataError
2202H	InvalidTablesampleArgument	DataError
22030	DuplicateJsonObjectKeyValue	DataError
22031	InvalidArgumentForSqlJsonDatetimeFunction	DataError
22032	InvalidJsonText	DataError
22033	InvalidSqlJsonSubscript	DataError
22034	MoreThanOneSqlJsonItem	DataError
22035	NoSqlJsonItem	DataError
22036	NonNumericSqlJsonItem	DataError
22037	NonUniqueKeysInAJsonObject	DataError
22038	SingletonSqlJsonItemRequired	DataError
22039	SqlJsonArrayNotFound	DataError
2203A	SqlJsonMemberNotFound	DataError
2203B	SqlJsonNumberNotFound	DataError
2203C	SqlJsonObjectNotFound	DataError
2203D	TooManyJsonArrayElements	DataError

continues on next page

Table 1 – continued from previous page

SQLSTATE	Exception	Base exception
2203E	TooManyJsonObjectMembers	DataError
2203F	SqlJsonScalarRequired	DataError
2203G	SqlJsonItemCannotBeCastToTargetType	DataError
22P01	FloatingPointException	DataError
22P02	InvalidTextRepresentation	DataError
22P03	InvalidBinaryRepresentation	DataError
22P04	BadCopyFileFormat	DataError
22P05	UntranslatableCharacter	DataError
22P06	NonstandardUseOfEscapeCharacter	DataError
Class 23 - Integrity Constraint Violation		
23000	IntegrityConstraintViolation	IntegrityError
23001	RestrictViolation	IntegrityError
23502	NotNullViolation	IntegrityError
23503	ForeignKeyViolation	IntegrityError
23505	UniqueViolation	IntegrityError
23514	CheckViolation	IntegrityError
23P01	ExclusionViolation	IntegrityError
Class 24 - Invalid Cursor State		
24000	InvalidCursorState	InternalError
Class 25 - Invalid Transaction State		
25000	InvalidTransactionState	InternalError
25001	ActiveSqlTransaction	InternalError
25002	BranchTransactionAlreadyActive	InternalError
25003	InappropriateAccessModeForBranchTransaction	InternalError
25004	InappropriateIsolationLevelForBranchTransaction	InternalError
25005	NoActiveSqlTransactionForBranchTransaction	InternalError
25006	ReadOnlySqlTransaction	InternalError
25007	SchemaAndDataStatementMixingNotSupported	InternalError
25008	HeldCursorRequiresSameIsolationLevel	InternalError
25P01	NoActiveSqlTransaction	InternalError
25P02	InFailedSqlTransaction	InternalError
25P03	IdleInTransactionSessionTimeout	InternalError
Class 26 - Invalid SQL Statement Name		
26000	InvalidSqlStatementName	ProgrammingError
Class 27 - Triggered Data Change Violation		
27000	TriggeredDataChangeViolation	OperationalError
Class 28 - Invalid Authorization Specification		
28000	InvalidAuthorizationSpecification	OperationalError
28P01	InvalidPassword	OperationalError
Class 2B - Dependent Privilege Descriptors Still Exist		
2B000	DependentPrivilegeDescriptorsStillExist	InternalError
2BP01	DependentObjectsStillExist	InternalError
Class 2D - Invalid Transaction Termination		
2D000	InvalidTransactionTermination	InternalError
Class 2F - SQL Routine Exception		
2F000	SqlRoutineException	OperationalError
2F002	ModifyingSqlDataNotPermitted	OperationalError
2F003	ProhibitedSqlStatementAttempted	OperationalError
2F004	ReadingSqlDataNotPermitted	OperationalError

continues on next page

Table 1 – continued from previous page

SQLSTATE	Exception	Base exception
2F005	FunctionExecutedNoReturnStatement	OperationalError
Class 34 - Invalid Cursor Name		
34000	InvalidCursorName	ProgrammingError
Class 38 - External Routine Exception		
38000	ExternalRoutineException	OperationalError
38001	ContainingSqlNotPermitted	OperationalError
38002	ModifyingSqlDataNotPermittedExt	OperationalError
38003	ProhibitedSqlStatementAttemptedExt	OperationalError
38004	ReadingSqlDataNotPermittedExt	OperationalError
Class 39 - External Routine Invocation Exception		
39000	ExternalRoutineInvocationException	OperationalError
39001	InvalidSqlstateReturned	OperationalError
39004	NullValueNotAllowedExt	OperationalError
39P01	TriggerProtocolViolated	OperationalError
39P02	SrfProtocolViolated	OperationalError
39P03	EventTriggerProtocolViolated	OperationalError
Class 3B - Savepoint Exception		
3B000	SavepointException	OperationalError
3B001	InvalidSavepointSpecification	OperationalError
Class 3D - Invalid Catalog Name		
3D000	InvalidCatalogName	ProgrammingError
Class 3F - Invalid Schema Name		
3F000	InvalidSchemaName	ProgrammingError
Class 40 - Transaction Rollback		
40000	TransactionRollback	OperationalError
40001	SerializationFailure	OperationalError
40002	TransactionIntegrityConstraintViolation	OperationalError
40003	StatementCompletionUnknown	OperationalError
40P01	DeadlockDetected	OperationalError
Class 42 - Syntax Error or Access Rule Violation		
42000	SyntaxErrorOrAccessRuleViolation	ProgrammingError
42501	InsufficientPrivilege	ProgrammingError
42601	SyntaxError	ProgrammingError
42602	InvalidName	ProgrammingError
42611	InvalidColumnDefinition	ProgrammingError
42622	NameTooLong	ProgrammingError
42701	DuplicateColumn	ProgrammingError
42702	AmbiguousColumn	ProgrammingError
42703	UndefinedColumn	ProgrammingError
42704	UndefinedObject	ProgrammingError
42710	DuplicateObject	ProgrammingError
42712	DuplicateAlias	ProgrammingError
42723	DuplicateFunction	ProgrammingError
42725	AmbiguousFunction	ProgrammingError
42803	GroupingError	ProgrammingError
42804	DatatypeMismatch	ProgrammingError
42809	WrongObjectType	ProgrammingError
42830	InvalidForeignKey	ProgrammingError
42846	CannotCoerce	ProgrammingError

continues on next page

Table 1 – continued from previous page

SQLSTATE	Exception	Base exception
42883	UndefinedFunction	ProgrammingError
428C9	GeneratedAlways	ProgrammingError
42939	ReservedName	ProgrammingError
42P01	UndefinedTable	ProgrammingError
42P02	UndefinedParameter	ProgrammingError
42P03	DuplicateCursor	ProgrammingError
42P04	DuplicateDatabase	ProgrammingError
42P05	DuplicatePreparedStatement	ProgrammingError
42P06	DuplicateSchema	ProgrammingError
42P07	DuplicateTable	ProgrammingError
42P08	AmbiguousParameter	ProgrammingError
42P09	AmbiguousAlias	ProgrammingError
42P10	InvalidColumnReference	ProgrammingError
42P11	InvalidCursorDefinition	ProgrammingError
42P12	InvalidDatabaseDefinition	ProgrammingError
42P13	InvalidFunctionDefinition	ProgrammingError
42P14	InvalidPreparedStatementDefinition	ProgrammingError
42P15	InvalidSchemaDefinition	ProgrammingError
42P16	InvalidTableDefinition	ProgrammingError
42P17	InvalidObjectDefinition	ProgrammingError
42P18	IndeterminateDatatype	ProgrammingError
42P19	InvalidRecursion	ProgrammingError
42P20	WindowingError	ProgrammingError
42P21	CollationMismatch	ProgrammingError
42P22	IndeterminateCollation	ProgrammingError
Class 44 - WITH CHECK OPTION Violation		
44000	WithCheckOptionViolation	ProgrammingError
Class 53 - Insufficient Resources		
53000	InsufficientResources	OperationalError
53100	DiskFull	OperationalError
53200	OutOfMemory	OperationalError
53300	TooManyConnections	OperationalError
53400	ConfigurationLimitExceeded	OperationalError
Class 54 - Program Limit Exceeded		
54000	ProgramLimitExceeded	OperationalError
54001	StatementTooComplex	OperationalError
54011	TooManyColumns	OperationalError
54023	TooManyArguments	OperationalError
Class 55 - Object Not In Prerequisite State		
55000	ObjectNotInPrerequisiteState	OperationalError
55006	ObjectInUse	OperationalError
55P02	CantChangeRuntimeParam	OperationalError
55P03	LockNotAvailable	OperationalError
55P04	UnsafeNewEnumValueUsage	OperationalError
Class 57 - Operator Intervention		
57000	OperatorIntervention	OperationalError
57014	QueryCanceled	OperationalError
57P01	AdminShutdown	OperationalError
57P02	CrashShutdown	OperationalError

continues on next page

Table 1 – continued from previous page

SQLSTATE	Exception	Base exception
57P03	CannotConnectNow	OperationalError
57P04	DatabaseDropped	OperationalError
57P05	IdleSessionTimeout	OperationalError
Class 58 - System Error (errors external to PostgreSQL itself)		
58000	SystemError	OperationalError
58030	IoError	OperationalError
58P01	UndefinedFile	OperationalError
58P02	DuplicateFile	OperationalError
Class 72 - Snapshot Failure		
72000	SnapshotTooOld	DatabaseError
Class F0 - Configuration File Error		
F0000	ConfigFileError	OperationalError
F0001	LockFileExists	OperationalError
Class HV - Foreign Data Wrapper Error (SQL/MED)		
HV000	FdwError	OperationalError
HV001	FdwOutOfMemory	OperationalError
HV002	FdwDynamicParameterValueNeeded	OperationalError
HV004	FdwInvalidDataType	OperationalError
HV005	FdwColumnNameNotFound	OperationalError
HV006	FdwInvalidDataTypeDescriptors	OperationalError
HV007	FdwInvalidColumnName	OperationalError
HV008	FdwInvalidColumnNumber	OperationalError
HV009	FdwInvalidUseOfNullPointer	OperationalError
HV00A	FdwInvalidStringFormat	OperationalError
HV00B	FdwInvalidHandle	OperationalError
HV00C	FdwInvalidOptionIndex	OperationalError
HV00D	FdwInvalidOptionName	OperationalError
HV00J	FdwOptionNameNotFound	OperationalError
HV00K	FdwReplyHandle	OperationalError
HV00L	FdwUnableToCreateExecution	OperationalError
HV00M	FdwUnableToCreateReply	OperationalError
HV00N	FdwUnableToEstablishConnection	OperationalError
HV00P	FdwNoSchemas	OperationalError
HV00Q	FdwSchemaNotFound	OperationalError
HV00R	FdwTableNotFound	OperationalError
HV010	FdwFunctionSequenceError	OperationalError
HV014	FdwTooManyHandles	OperationalError
HV021	FdwInconsistentDescriptorInformation	OperationalError
HV024	FdwInvalidAttributeValue	OperationalError
HV090	FdwInvalidStringLengthOrBufferLength	OperationalError
HV091	FdwInvalidDescriptorFieldIdentifier	OperationalError
Class P0 - PL/pgSQL Error		
P0000	PlpgsqlError	ProgrammingError
P0001	RaiseException	ProgrammingError
P0002	NoDataFound	ProgrammingError
P0003	TooManyRows	ProgrammingError
P0004	AssertFailure	ProgrammingError
Class XX - Internal Error		
XX000	InternalError_	InternalError

continues on next page

Table 1 – continued from previous page

SQLSTATE	Exception	Base exception
XX001	DataCorrupted	InternalError
XX002	IndexCorrupted	InternalError

New in version 3.1.4: Exception SqlJsonItemCannotBeCastToTargetType, introduced in PostgreSQL 15.

1.3.9 psycopg_pool – Connection pool implementations

A connection pool is an object used to create and maintain a limited amount of PostgreSQL connections, reducing the time requested by the program to obtain a working connection and allowing an arbitrary large number of concurrent threads or tasks to use a controlled amount of resources on the server. See *Connection pools* for more details and usage pattern.

This package exposes a few connection pool classes:

- ConnectionPool is a synchronous connection pool yielding *Connection* objects and can be used by multi-thread applications.
- AsyncConnectionPool has an interface similar to ConnectionPool, but with asyncio functions replacing blocking functions, and yields *AsyncConnection* instances.
- NullConnectionPool is a ConnectionPool subclass exposing the same interface of its parent, but not keeping any unused connection in its state. See *Null connection pools* for details about related use cases.
- AsyncNullConnectionPool has the same behaviour of the NullConnectionPool, but with the same async interface of the AsyncConnectionPool.

Note: The psycopg_pool package is distributed separately from the main *psycopg* package: use pip install "psycopg[pool]", or pip install psycopg_pool, to make it available. See *Installing the connection pool*.

The version numbers indicated in this page refer to the psycopg_pool package, not to *psycopg*.

The ConnectionPool class

Pool exceptions

The AsyncConnectionPool class

AsyncConnectionPool has a very similar interface to the ConnectionPool class but its blocking methods are implemented as async coroutines. It returns instances of *AsyncConnection*, or of its subclass if specified so in the connection_class parameter.

Only the functions with different signature from ConnectionPool are listed here.

Null connection pools

New in version 3.1.

The `NullConnectionPool` is a `ConnectionPool` subclass which doesn't create connections preemptively and doesn't keep unused connections in its state. See *Null connection pools* for further details.

The interface of the object is entirely compatible with its parent class. Its behaviour is similar, with the following differences:

The `AsyncNullConnectionPool` is, similarly, an `AsyncConnectionPool` subclass with the same behaviour of the `NullConnectionPool`.

1.3.10 `conninfo` – manipulate connection strings

This module contains a few utility functions to manipulate database connection strings.

`psycopg.conninfo.`**`conninfo_to_dict`**(*conninfo=''*, ***kwargs*)

Convert the *conninfo* string into a dictionary of parameters.

Parameters

- **`conninfo`** (`str`) – A connection string as accepted by PostgreSQL.
- **`kwargs`** (`Any`) – Parameters overriding the ones specified in *conninfo*.

Return type
`Dict[str, Any]`

Returns
Dictionary with the parameters parsed from *conninfo* and *kwargs*.

Raise *`ProgrammingError`* if *conninfo* is not a a valid connection string.

```
>>> conninfo_to_dict("postgres://jeff@example.com/db", user="piro")
{'user': 'piro', 'dbname': 'db', 'host': 'example.com'}
```

`psycopg.conninfo.`**`make_conninfo`**(*conninfo=''*, ***kwargs*)

Merge a string and keyword params into a single conninfo string.

Parameters

- **`conninfo`** (`str`) – A connection string as accepted by PostgreSQL.
- **`kwargs`** (`Any`) – Parameters overriding the ones specified in *conninfo*.

Return type
`str`

Returns
A connection string valid for PostgreSQL, with the *kwargs* parameters merged.

Raise *`ProgrammingError`* if the input doesn't make a valid conninfo string.

```
>>> make_conninfo("dbname=db user=jeff", user="piro", port=5432)
'dbname=db user=piro port=5432'
```

1.3.11 adapt – Types adaptation

The psycopg.adapt module exposes a set of objects useful for the configuration of *data adaptation*, which is the conversion of Python objects to PostgreSQL data types and back.

These objects are useful if you need to configure data adaptation, i.e. if you need to change the default way that Psycopg converts between types or if you want to adapt custom data types and objects. You don't need this object in the normal use of Psycopg.

See *Data adaptation configuration* for an overview of the Psycopg adaptation system.

Dumpers and loaders

class psycopg.adapt.**Dumper**(*cls*, *context=None*)

Convert Python object of the type *cls* to PostgreSQL representation.

This is an abstract base class, partially implementing the *Dumper* protocol. Subclasses *must* at least implement the *dump()* method and optionally override other members.

abstract dump(*obj*)

Convert the object *obj* to PostgreSQL representation.

Parameters
obj (Any) – the object to convert.

Return type
Union[bytes, bytearray, memoryview]

format: *psycopg.pq.Format* **= TEXT**

Class attribute. Set it to *BINARY* if the class *dump()* methods converts the object to binary format.

quote(*obj*)

By default return the *dump()* value quoted and sanitised, so that the result can be used to build a SQL string. This works well for most types and you won't likely have to implement this method in a subclass.

Return type
Union[bytes, bytearray, memoryview]

get_key(*obj*, *format*)

Implementation of the *get_key()* member of the *Dumper* protocol. Look at its definition for details.

This implementation returns the *cls* passed in the constructor. Subclasses needing to specialise the PostgreSQL type according to the *value* of the object dumped (not only according to to its type) should override this class.

Return type
Union[type, Tuple[type, ...]]

upgrade(*obj*, *format*)

Implementation of the *upgrade()* member of the *Dumper* protocol. Look at its definition for details.

This implementation just returns *self*. If a subclass implements *get_key()* it should probably override upgrade() too.

Return type
Dumper

class psycopg.adapt.**Loader**(*oid*, *context=None*)

Convert PostgreSQL values with type OID *oid* to Python objects.

This is an abstract base class, partially implementing the *Loader* protocol. Subclasses *must* at least implement the *load()* method and optionally override other members.

abstract load(*data*)

Convert a PostgreSQL value to a Python object.

Return type
Any

format: *psycopg.pq.Format* **= TEXT**

Class attribute. Set it to *BINARY* if the class *load()* methods converts the object from binary format.

Other objects used in adaptations

class psycopg.adapt.**PyFormat**(*value*)

Enum representing the format wanted for a query argument.

The value *AUTO* allows psycopg to choose the best format for a certain parameter.

AUTO = 's'

TEXT = 't'

BINARY = 'b'

class psycopg.adapt.**AdaptersMap**(*template=None*, *types=None*)

Establish how types should be converted between Python and PostgreSQL in an *AdaptContext*.

`AdaptersMap` maps Python types to *Dumper* classes to define how Python types are converted to PostgreSQL, and maps OIDs to *Loader* classes to establish how query results are converted to Python.

Every `AdaptContext` object has an underlying `AdaptersMap` defining how types are converted in that context, exposed as the *adapters* attribute: changing such map allows to customise adaptation in a context without changing separated contexts.

When a context is created from another context (for instance when a *Cursor* is created from a *Connection*), the parent's `adapters` are used as template for the child's `adapters`, so that every cursor created from the same connection use the connection's types configuration, but separate connections have independent mappings.

Once created, `AdaptersMap` are independent. This means that objects already created are not affected if a wider scope (e.g. the global one) is changed.

The connections adapters are initialised using a global `AdptersMap` template, exposed as *psycopg.adapters*: changing such mapping allows to customise the type mapping for every connections created afterwards.

The object can start empty or copy from another object of the same class. Copies are copy-on-write: if the maps are updated make a copy. This way extending e.g. global map by a connection or a connection map from a cursor is cheap: a copy is only made on customisation.

See also:

Data adaptation configuration for an explanation about how contexts are connected.

register_dumper(*cls*, *dumper*)

Configure the context to use *dumper* to convert object of type *cls*.

If two dumpers with different *format* are registered for the same type, the last one registered will be chosen when the query doesn't specify a format (i.e. when the value is used with a `%s` "*AUTO*" placeholder).

Parameters

- **cls** (Union[type, str, None]) – The type to manage.
- **dumper** (Type[*Dumper*]) – The dumper to register for *cls*.

If *cls* is specified as string it will be lazy-loaded, so that it will be possible to register it without importing it before. In this case it should be the fully qualified name of the object (e.g. `"uuid.UUID"`).

If *cls* is None, only use the dumper when looking up using *get_dumper_by_oid()*, which happens when we know the Postgres type to adapt to, but not the Python type that will be adapted (e.g. in COPY after using *set_types()*).

register_loader(*oid*, *loader*)

Configure the context to use *loader* to convert data of oid *oid*.

Parameters

- **oid** (Union[int, str]) – The PostgreSQL OID or type name to manage.
- **loader** (Type[*Loader*]) – The loar to register for *oid*.

If `oid` is specified as string, it refers to a type name, which is looked up in the *types* registry. `

types

The object where to look up for types information (such as the mapping between type names and oids in the specified context).

Type
TypesRegistry

get_dumper(*cls*, *format*)

Return the dumper class for the given type and format.

Raise *ProgrammingError* if a class is not available.

Parameters

- **cls** (type) – The class to adapt.
- **format** (*PyFormat*) – The format to dump to. If *AUTO*, use the last one of the dumpers registered on *cls*.

Return type
Type[*Dumper*]

get_dumper_by_oid(*oid*, *format*)

Return the dumper class for the given oid and format.

Raise *ProgrammingError* if a class is not available.

Parameters

- **oid** (int) – The oid of the type to dump to.
- **format** (*Format*) – The format to dump to.

Return type
Type[*Dumper*]

get_loader(*oid*, *format*)

Return the loader class for the given oid and format.

Return `None` if not found.

Parameters

- **oid** (int) – The oid of the type to load.
- **format** (*Format*) – The format to load from.

Return type
Optional[Type[*Loader*]]

class psycopg.adapt.**Transformer**(*context=None*)

An object that can adapt efficiently between Python and PostgreSQL.

The life cycle of the object is the query, so it is assumed that attributes such as the server version or the connection encoding will not change. The object have its state so adapting several values of the same type can be optimised.

Parameters
context (*AdaptContext*) – The context where the transformer should operate.

1.3.12 types – Types information and adapters

The psycopg.types package exposes:

- objects to describe PostgreSQL types, such as *TypeInfo*, *TypesRegistry*, to help or *customise the types conversion*;
- concrete implementations of *Loader* and *Dumper* protocols to *handle builtin data types*;
- helper objects to represent PostgreSQL data types which *don't have a straightforward Python representation*, such as *Range*.

Types information

The *TypeInfo* object describes simple information about a PostgreSQL data type, such as its name, oid and array oid. TypeInfo subclasses may hold more information, for instance the components of a composite type.

You can use *TypeInfo.fetch()* to query information from a database catalog, which is then used by helper functions, such as *register_hstore()*, to register adapters on types whose OID is not known upfront or to create more specialised adapters.

The TypeInfo object doesn't instruct Psycopg to convert a PostgreSQL type into a Python type: this is the role of a *Loader*. However it can extend the behaviour of other adapters: if you create a loader for MyType, using the *TypeInfo* information, Psycopg will be able to manage seamlessly arrays of MyType or ranges and composite types using MyType as a subtype.

See also:

Data adaptation configuration describes how to convert from Python objects to PostgreSQL types and back.

```
from psycopg.adapt import Loader
from psycopg.types import TypeInfo

t = TypeInfo.fetch(conn, "mytype")
t.register(conn)

for record in conn.execute("SELECT mytypearray FROM mytable"):
    # records will return lists of "mytype" as string

class MyTypeLoader(Loader):
    def load(self, data):
        # parse the data and return a MyType instance
```

(continues on next page)

(continued from previous page)

```
conn.adapters.register_loader("mytype", MyTypeLoader)

for record in conn.execute("SELECT mytypearray FROM mytable"):
    # records will return lists of MyType instances
```

class psycopg.types.**TypeInfo**(*name*, *oid*, *array_oid*, ***, *regtype=''*, *delimiter=','*)

Hold information about a PostgreSQL base type.

classmethod fetch(*conn*, *name*)

async classmethod fetch(*aconn*, *name*)

Query a system catalog to read information about a type.

Parameters

- **conn** (*Connection or AsyncConnection*) – the connection to query
- **name** (`str` or *Identifier*) – the name of the type to query. It can include a schema name.

Returns
a `TypeInfo` object (or subclass) populated with the type information, `None` if not found.

If the connection is async, `fetch()` will behave as a coroutine and the caller will need to `await` on it to get the result:

```
t = await TypeInfo.fetch(aconn, "mytype")
```

register(*context=None*)

Register the type information, globally or in the specified *context*.

Parameters
context (*Optional[AdaptContext]*) – the context where the type is registered, for instance a *Connection* or *Cursor*. `None` registers the `TypeInfo` globally.

Registering the *TypeInfo* in a context allows the adapters of that context to look up type information: for instance it allows to recognise automatically arrays of that type and load them from the database as a list of the base type.

In order to get information about dynamic PostgreSQL types, Psycopg offers a few `TypeInfo` subclasses, whose `fetch()` method can extract more complete information about the type, such as *CompositeInfo*, *RangeInfo*, *MultirangeInfo*, *EnumInfo*.

`TypeInfo` objects are collected in *TypesRegistry* instances, which help type information lookup. Every *AdaptersMap* exposes its type map on its *types* attribute.

class psycopg.types.**TypesRegistry**(*template=None*)

Container for the information about types in a database.

`TypeRegistry` instances are typically exposed by *AdaptersMap* objects in adapt contexts such as *Connection* or *Cursor* (e.g. `conn.adapters.types`).

The global registry, from which the others inherit from, is available as *psycopg.adapters*`.types`.

__getitem__(*key: Union[str, int]*) → *TypeInfo*

__getitem__(*key: Tuple[Type[T], int]*) → T

Return info about a type, specified by name or oid

Parameters
key (Union[str, int, Tuple[type, int]]) – the name or oid of the type to look for.

Raise KeyError if not found.

Return type
TypeInfo

```
>>> import psycopg

>>> psycopg.adapters.types["text"]
<TypeInfo: text (oid: 25, array oid: 1009)>

>>> psycopg.adapters.types[23]
<TypeInfo: int4 (oid: 23, array oid: 1007)>
```

get(*key: Union[str, int]*) → Optional[*TypeInfo*]

get(*key: Tuple[Type[T], int]*) → Optional[T]

Return info about a type, specified by name or oid

Parameters
key (Union[str, int, Tuple[type, int]]) – the name or oid of the type to look for.

Unlike *__getitem__*, return None if not found.

Return type
Optional[*TypeInfo*]

get_oid(*name*)

Return the oid of a PostgreSQL type by name.

Parameters
key – the name of the type to look for.

Return the array oid if the type ends with "[]"

Raise KeyError if the name is unknown.

Return type
int

```
>>> psycopg.adapters.types.get_oid("text[]")
1009
```

get_by_subtype(*cls*, *subtype*)

Return info about a *TypeInfo* subclass by its element name or oid.

Parameters

- **cls** (Type[TypeVar(T, bound= TypeInfo)]) – the subtype of `TypeInfo` to look for. Currently supported are *RangeInfo* and *MultirangeInfo*.
- **subtype** (Union[int, str]) – The name or OID of the subtype of the element to look for.

Return type
Optional[TypeVar(T, bound= TypeInfo)]

Returns
The `TypeInfo` object of class *cls* whose subtype is *subtype*. `None` if the element or its range are not found.

JSON adapters

See *JSON adaptation* for details.

class psycopg.types.json.**Json**(*obj*, *dumps=None*)

class psycopg.types.json.**Jsonb**(*obj*, *dumps=None*)

Wrappers to signal to convert `obj` to a json or jsonb PostgreSQL value.

Any object supported by the underlying `dumps()` function can be wrapped.

If a `dumps` function is passed to the wrapper, use it to dump the wrapped object. Otherwise use the function specified by `set_json_dumps()`.

psycopg.types.json.**set_json_dumps**(*dumps*, *context=None*)

Set the JSON serialisation function to store JSON objects in the database.

Parameters

- **dumps** (`Callable[[Any], str]`) – The dump function to use.
- **context** (*Connection* or *Cursor*) – Where to use the *dumps* function. If not specified, use it globally.

By default dumping JSON uses the builtin `json.dumps`. You can override it to use a different JSON library or to use customised arguments.

If the *Json* wrapper specified a *dumps* function, use it in precedence of the one set by this function.

psycopg.types.json.**set_json_loads**(*loads*, *context=None*)

Set the JSON parsing function to fetch JSON objects from the database.

Parameters

- **loads** (`Callable[[bytes], Any]`) – The load function to use.
- **context** (*Connection* or *Cursor*) – Where to use the *loads* function. If not specified, use it globally.

By default loading JSON uses the builtin `json.loads`. You can override it to use a different JSON library or to use customised arguments.

1.3.13 abc – Psycopg abstract classes

The module exposes Psycopg definitions which can be used for static type checking.

class psycopg.abc.**Dumper**(*cls*, *context=None*)

Convert Python objects of type *cls* to PostgreSQL representation.

Parameters

- **cls** (*type*) – The type that will be managed by this dumper.
- **context** (*AdaptContext* or None) – The context where the transformation is performed. If not specified the conversion might be inaccurate, for instance it will not be possible to know the connection encoding or the server date format.

A partial implementation of this protocol (implementing everything except *dump()*) is available as *psycopg.adapt.Dumper*.

format: *Format*

The format that this class *dump()* method produces, *TEXT* or *BINARY*.

This is a class attribute.

dump(*obj*)

Convert the object *obj* to PostgreSQL representation.

Parameters
obj (Any) – the object to convert.

Return type
Union[bytes, bytearray, memoryview]

The format returned by dump shouldn't contain quotes or escaped values.

quote(*obj*)

Convert the object *obj* to escaped representation.

Parameters
obj (Any) – the object to convert.

Return type
Union[bytes, bytearray, memoryview]

Tip: This method will be used by *Literal* to convert a value client-side.

This method only makes sense for text dumpers; the result of calling it on a binary dumper is undefined. It might scratch your car, or burn your cake. Don't tell me I didn't warn you.

oid: int

The oid to pass to the server, if known; 0 otherwise (class attribute).

If the OID is not specified, PostgreSQL will try to infer the type from the context, but this may fail in some contexts and may require a cast (e.g. specifying `%s::type` for its placeholder).

You can use the *psycopg.adapters.types* registry to find the OID of builtin types, and you can use *TypeInfo* to extend the registry to custom types.

get_key(*obj*, *format*)

Return an alternative key to upgrade the dumper to represent *obj*.

Parameters

- **obj** (Any) – The object to convert
- **format** (*PyFormat*) – The format to convert to

Normally the type of the object is all it takes to define how to dump the object to the database. For instance, a Python `date` can be simply converted into a PostgreSQL `date`.

In a few cases, just the type is not enough. For example:

- A Python `datetime` could be represented as a `timestamptz` or a `timestamp`, according to whether it specifies a `tzinfo` or not.
- A Python int could be stored as several Postgres types: int2, int4, int8, numeric. If a type too small is used, it may result in an overflow. If a type too large is used, PostgreSQL may not want to cast it to a smaller type.
- Python lists should be dumped according to the type they contain to convert them to e.g. array of strings, array of ints (and which size of int?...)

In these cases, a dumper can implement `get_key()` and return a new class, or sequence of classes, that can be used to identify the same dumper again. If the mechanism is not needed, the method should return the same *cls* object passed in the constructor.

If a dumper implements *get_key()* it should also implement *upgrade()*.

Return type
Union[type, Tuple[type, ...]]

upgrade(*obj*, *format*)

Return a new dumper to manage *obj*.

Parameters

- **obj** (Any) – The object to convert
- **format** (*PyFormat*) – The format to convert to

Once `Transformer.get_dumper()` has been notified by *get_key()* that this Dumper class cannot handle *obj* itself, it will invoke `upgrade()`, which should return a new *Dumper* instance, which will be reused for every objects for which `get_key()` returns the same result.

Return type
Dumper

class psycopg.abc.**Loader**(*oid*, *context=None*)

Convert PostgreSQL values with type OID *oid* to Python objects.

Parameters

- **oid** (*int*) – The type that will be managed by this dumper.
- **context** (*AdaptContext* or None) – The context where the transformation is performed. If not specified the conversion might be inaccurate, for instance it will not be possible to know the connection encoding or the server date format.

A partial implementation of this protocol (implementing everything except *load()*) is available as *psycopg.adapt.Loader*.

format: *Format*

The format that this class *load()* method can convert, *TEXT* or *BINARY*.

This is a class attribute.

load(*data*)

Convert the data returned by the database into a Python object.

Parameters
data (Union[bytes, bytearray, memoryview]) – the data to convert.

Return type
Any

class psycopg.abc.**AdaptContext**(**args*, ***kwargs*)

A context describing how types are adapted.

Example of *AdaptContext* are *Connection*, *Cursor*, *Transformer*, *AdaptersMap*.

Note that this is a Protocol, so objects implementing AdaptContext don't need to explicitly inherit from this class.

See also:

Data adaptation configuration for an explanation about how contexts are connected.

property adapters: *AdaptersMap*

The adapters configuration that this object uses.

Return type
AdaptersMap

property connection: Optional[BaseConnection[Any]]

The connection used by this object, if available.

Return type
Connection or *AsyncConnection* or None

1.3.14 pq – libpq wrapper module

Psycopg is built around the libpq, the PostgreSQL client library, which performs most of the network communications and returns query results in C structures.

The low-level functions of the library are exposed by the objects in the `psycopg.pq` module.

pq module implementations

There are actually several implementations of the module, all offering the same interface. Current implementations are:

- `python`: a pure-python implementation, implemented using the `ctypes` module. It is less performing than the others, but it doesn't need a C compiler to install. It requires the libpq installed in the system.
- `c`: a C implementation of the libpq wrapper (more precisely, implemented in Cython). It is much better performing than the `python` implementation, however it requires development packages installed on the client machine. It can be installed using the `c` extra, i.e. running `pip install "psycopg[c]"`.
- `binary`: a pre-compiled C implementation, bundled with all the required libraries. It is the easiest option to deal with, fast to install and it should require no development tool or client library, however it may be not available for every platform. You can install it using the `binary` extra, i.e. running `pip install "psycopg[binary]"`.

The implementation currently used is available in the *__impl__* module constant.

At import time, Psycopg 3 will try to use the best implementation available and will fail if none is usable. You can force the use of a specific implementation by exporting the env var `PSYCOPG_IMPL`: importing the library will fail if the requested implementation is not available:

```
$ PSYCOPG_IMPL=c python -c "import psycopg"
Traceback (most recent call last):
   ...
ImportError: couldn't import requested psycopg 'c' implementation: No module named
↪'psycopg_c'
```

Module content

psycopg.pq.__impl__: str = 'binary'

The currently loaded implementation of the psycopg.pq package.

Possible values include python, c, binary.

The choice of implementation is automatic but can be forced setting the PSYCOPG_IMPL env var.

psycopg.pq.version()

See also:

the PQlibVersion() function

psycopg.pq.__build_version__: int = 150000

The libpq version the C package was built with.

A number in the same format of *server_version* representing the libpq used to build the speedup module (c, binary) if available.

Certain features might not be available if the built version is too old.

psycopg.pq.error_message(*obj*, *encoding='utf8'*)

Return an error message from a *PGconn* or *PGresult*.

The return value is a str (unlike pq data which is usually bytes): use the connection encoding if available, otherwise the *encoding* parameter as a fallback for decoding. Don't raise exceptions on decoding errors.

Return type
str

Objects wrapping libpq structures and functions

TODO

finish documentation

class psycopg.pq.PGconn

pgconn_ptr

get_cancel()

Return type
PGcancel

needs_password

used_password

encrypt_password(*passwd*, *user*, *algorithm=None*)

Return type
bytes

```
>>> enc = conn.info.encoding
>>> encrypted = conn.pgconn.encrypt_password(password.encode(enc), rolename.
↪encode(enc))
b'SCRAM-SHA-256$4096:...
```

trace(*fileno*)

Return type
None

set_trace_flags(*flags*)

Return type
None

untrace()

Return type
None

```
>>> conn.pgconn.trace(sys.stderr.fileno())
>>> conn.pgconn.set_trace_flags(pq.Trace.SUPPRESS_TIMESTAMPS | pq.Trace.REGRESS_
↪MODE)
>>> conn.execute("select now()")
F       13      Parse     "" "BEGIN" 0
F       14      Bind      "" "" 0 0 1 0
F       6       Describe          P ""
F       9       Execute   "" 0
F       4       Sync
B       4       ParseComplete
B       4       BindComplete
B       4       NoData
B       10      CommandComplete   "BEGIN"
B       5       ReadyForQuery     T
F       17      Query     "select now()"
B       28      RowDescription    1 "now" NNNN 0 NNNN 8 -1 0
B       39      DataRow   1 29 '2022-09-14 14:12:16.648035+02'
B       13      CommandComplete   "SELECT 1"
B       5       ReadyForQuery     T
<psycopg.Cursor [TUPLES_OK] [INTRANS] (database=postgres) at 0x7f18a18ba040>
>>> conn.pgconn.untrace()
```

class psycopg.pq.**PGresult**

pgresult_ptr

class psycopg.pq.**Conninfo**

class psycopg.pq.**Escaping**

class psycopg.pq.**PGcancel**

Enumerations

class psycopg.pq.**ConnStatus**(*value*)

Current status of the connection.

There are other values in this enum, but only *OK* and *BAD* are seen after a connection has been established. Other statuses might only be seen during the connection phase and are considered internal.

See also:

PQstatus() returns this value.

OK = 0

BAD = 1

class psycopg.pq.**PollingStatus**(*value*)

The status of the socket during a connection.

If READING or WRITING you may select before polling again.

See also:

PQconnectPoll for a description of these states.

FAILED = 0

READING = 1

WRITING = 2

OK = 3

class psycopg.pq.**TransactionStatus**(*value*)

The transaction status of a connection.

See also:

PQtransactionStatus for a description of these states.

IDLE = 0

ACTIVE = 1

INTRANS = 2

INERROR = 3

UNKNOWN = 4

class psycopg.pq.**ExecStatus**(*value*)

The status of a command.

See also:

PQresultStatus for a description of these states.

EMPTY_QUERY = 0

COMMAND_OK = 1

TUPLES_OK = 2

COPY_OUT = 3

COPY_IN = 4

BAD_RESPONSE = 5

NONFATAL_ERROR = 6

FATAL_ERROR = 7

COPY_BOTH = 8

SINGLE_TUPLE = 9

PIPELINE_SYNC = 10

PIPELINE_ABORTED = 11

class psycopg.pq.**PipelineStatus**(*value*)

Pipeline mode status of the libpq connection.

See also:

PQpipelineStatus for a description of these states.

OFF = 0

ON = 1

ABORTED = 2

class psycopg.pq.**Format**(*value*)

Enum representing the format of a query argument or return value.

These values are only the ones managed by the libpq. *psycopg* may also support automatically-chosen values: see *psycopg.adapt.PyFormat*.

TEXT = 0

BINARY = 1

class psycopg.pq.**DiagnosticField**(*value*)

Fields in an error report.

Available attributes:

SEVERITY
SEVERITY_NONLOCALIZED
SQLSTATE
MESSAGE_PRIMARY
MESSAGE_DETAIL
MESSAGE_HINT
STATEMENT_POSITION
INTERNAL_POSITION
INTERNAL_QUERY
CONTEXT
SCHEMA_NAME
TABLE_NAME

COLUMN_NAME

DATATYPE_NAME

CONSTRAINT_NAME

SOURCE_FILE

SOURCE_LINE

SOURCE_FUNCTION

See also:

PQresultErrorField for a description of these values.

class psycopg.pq.**Ping**(*value*)

Response from a ping attempt.

See also:

PQpingParams for a description of these values.

OK = 0

REJECT = 1

NO_RESPONSE = 2

NO_ATTEMPT = 3

class psycopg.pq.**Trace**(*value*)

Enum to control tracing of the client/server communication.

See also:

PQsetTraceFlags for a description of these values.

SUPPRESS_TIMESTAMPS = 1

REGRESS_MODE = 2

1.3.15 crdb – CockroachDB support

New in version 3.1.

CockroachDB is a distributed database using the same fronted-backend protocol of PostgreSQL. As such, Psycopg can be used to write Python programs interacting with CockroachDB.

Opening a connection to a CRDB database using *psycopg.connect()* provides a largely working object. However, using the *psycopg.crdb.connect()* function instead, Psycopg will create more specialised objects and provide a types mapping tweaked on the CockroachDB data model.

Main differences from PostgreSQL

CockroachDB behaviour is different from PostgreSQL: please refer to the database documentation for details. These are some of the main differences affecting Psycopg behaviour:

- *cancel()* doesn't work before CockroachDB 22.1. On older versions, you can use CANCEL QUERY instead (but from a different connection).
- *Server-side cursors* are well supported only from CockroachDB 22.1.3.

- *backend_pid* is only populated from CockroachDB 22.1. Note however that you cannot use the PID to terminate the session; use SHOW session_id to find the id of a session, which you may terminate with CANCEL SESSION in lieu of PostgreSQL's `pg_terminate_backend()`.
- Several data types are missing or slightly different from PostgreSQL (see *adapters* for an overview of the differences).
- The *two-phase commit protocol* is not supported.
- `LISTEN` and `NOTIFY` are not supported. However the CHANGEFEED command, in conjunction with *stream()*, can provide push notifications.

CockroachDB-specific objects

psycopg.crdb.**connect**(*conninfo=''*, ***kwargs*)

Connect to a database server and return a new *CrdbConnection* instance.

Return type
CrdbConnection[Any]

This is an alias of the class method `CrdbConnection.connect`.

If you need an asynchronous connection use the `AsyncCrdbConnection.connect()` method instead.

class psycopg.crdb.**CrdbConnection**(*pgconn*, *row_factory=<function tuple_row>*)

Wrapper for a connection to a CockroachDB database.

psycopg.Connection subclass.

classmethod is_crdb(*conn*)

Return `True` if the server connected to `conn` is CockroachDB.

Return type
bool

Parameters
conn (*Connection*, *AsyncConnection*, *PGconn*) – the connection to check

class psycopg.crdb.**AsyncCrdbConnection**(*pgconn*, *row_factory=<function tuple_row>*)

Wrapper for an async connection to a CockroachDB database.

psycopg.AsyncConnection subclass.

class psycopg.crdb.**CrdbConnectionInfo**(*pgconn*)

ConnectionInfo subclass to get info about a CockroachDB database.

The object is returned by the *info* attribute of *CrdbConnection* and *AsyncCrdbConnection*.

The object behaves like `ConnectionInfo`, with the following differences:

vendor: str

The `CockroachDB` string.

server_version: int

Return the CockroachDB server version connected.

Return a number in the PostgreSQL format (e.g. 21.2.10 -> 210210).

psycopg.crdb.**adapters**

The default adapters map establishing how Python and CockroachDB types are converted into each other.

The map is used as a template when new connections are created, using *psycopg.crdb.connect()* (similarly to the way *psycopg.adapters* is used as template for new PostgreSQL connections).

This registry contains only the types and adapters supported by CockroachDB. Several PostgreSQL types and adapters are missing or different from PostgreSQL, among which:

- Composite types
- `range`, `multirange` types
- The `hstore` type
- Geometric types
- Nested arrays
- Arrays of `jsonb`
- The `cidr` data type
- The `json` type is an alias for `jsonb`
- The `int` type is an alias for `int8`, not `int4`.

1.3.16 _dns – DNS resolution utilities

This module contains a few experimental utilities to interact with the DNS server before performing a connection.

Warning: This module is experimental and its interface could change in the future, without warning or respect for the version scheme. It is provided here to allow experimentation before making it more stable.

Warning: This module depends on the dnspython package. The package is currently not installed automatically as a Psycopg dependency and must be installed manually:

```
$ pip install "dnspython >= 2.1"
```

psycopg._dns.**resolve_srv**(*params*)

Apply SRV DNS lookup as defined in **RFC 2782**.

Parameters
params (`dict`) – The input parameters, for instance as returned by *conninfo_to_dict()*.

Returns
An updated list of connection parameters.

For every host defined in the `params["host"]` list (comma-separated), perform SRV lookup if the host is in the form `_Service._Proto.Target`. If lookup is successful, return a params dict with hosts and ports replaced with the looked-up entries.

Raise *OperationalError* if no lookup is successful and no host (looked up or unchanged) could be returned.

In addition to the rules defined by RFC 2782 about the host name pattern, perform SRV lookup also if the the port is the string `SRV` (case insensitive).

Warning: This is an experimental functionality.

Note: One possible way to use this function automatically is to subclass *Connection*, extending the *_get_connection_params()* method:

```
import psycopg._dns  # not imported automatically

class SrvCognizantConnection(psycopg.Connection):
    @classmethod
    def _get_connection_params(cls, conninfo, **kwargs):
        params = super()._get_connection_params(conninfo, **kwargs)
        params = psycopg._dns.resolve_srv(params)
        return params

# The name will be resolved to db1.example.com
cnn = SrvCognizantConnection.connect("host=_postgres._tcp.db.psycopg.org")
```

async psycopg._dns.**resolve_srv_async**(*params*)

Async equivalent of *resolve_srv()*.

classmethod Connection.**_get_connection_params**(*conninfo*, ***kwargs*)

Manipulate connection parameters before connecting.

Parameters

- **conninfo** (str) – Connection string as received by *connect()*.
- **kwargs** (Any) – Overriding connection arguments as received by `connect()`.

Return type

Dict[str, Any]

Returns

Connection arguments merged and eventually modified, in a format similar to *conninfo_to_dict()*.

Warning: This is an experimental method.

This method is a subclass hook allowing to manipulate the connection parameters before performing the connection. Make sure to call the `super()` implementation before further manipulation of the arguments:

```
@classmethod
def _get_connection_params(cls, conninfo, **kwargs):
    params = super()._get_connection_params(conninfo, **kwargs)
    # do something with the params
    return params
```

async classmethod AsyncConnection.**_get_connection_params**(*conninfo*, ***kwargs*)

Manipulate connection parameters before connecting.

Changed in version 3.1: Unlike the sync counterpart, perform non-blocking address resolution and populate the `hostaddr` connection parameter, unless the user has provided one themselves. See *resolve_hostaddr_async()* for details.

Return type
Dict[str, Any]

Warning: This is an experimental method.

async psycopg._dns.**resolve_hostaddr_async**(*params*)

Perform async DNS lookup of the hosts and return a new params dict.

Deprecated since version 3.1: The use of this function is not necessary anymore, because *psycopg.AsyncConnection.connect()* performs non-blocking name resolution automatically.

Parameters
params (dict) – The input parameters, for instance as returned by *conninfo_to_dict()*.

If a `host` param is present but not `hostname`, resolve the host addresses dynamically.

The function may change the input `host`, `hostname`, `port` to allow connecting without further DNS lookups, eventually removing hosts that are not resolved, keeping the lists of hosts and ports consistent.

Raise *OperationalError* if connection is not possible (e.g. no host resolve, inconsistent lists length).

See the PostgreSQL docs for explanation of how these params are used, and how they support multiple entries.

Warning: Before psycopg 3.1, this function doesn't handle the `/etc/hosts` file.

Note: Starting from psycopg 3.1, a similar operation is performed automatically by `AsyncConnection._get_connection_params()`, so this function is unneeded.

In psycopg 3.0, one possible way to use this function automatically is to subclass *AsyncConnection*, extending the *_get_connection_params()* method:

```
import psycopg._dns  # not imported automatically

class AsyncDnsConnection(psycopg.AsyncConnection):
    @classmethod
    async def _get_connection_params(cls, conninfo, **kwargs):
        params = await super()._get_connection_params(conninfo, **kwargs)
        params = await psycopg._dns.resolve_hostaddr_async(params)
        return params
```

1.4 Release notes

1.4.1 psycopg release notes

Current release

Psycopg 3.1.4

- Include *error classes* defined in PostgreSQL 15.

- Add support for Python 3.11 (ticket #305).
- Build binary packages with libpq from PostgreSQL 15.0.

Psycopg 3.1.3

- Restore the state of the connection if *Cursor.stream()* is terminated prematurely (ticket #382).
- Fix regression introduced in 3.1 with different named tuples mangling rules for non-ascii attribute names (ticket #386).
- Fix handling of queries with escaped percent signs (%%) in *ClientCursor* (ticket #399).
- Fix possible duplicated BEGIN statements emitted in pipeline mode (ticket #401).

Psycopg 3.1.2

- Fix handling of certain invalid time zones causing problems on Windows (ticket #371).
- Fix segfault occurring when a loader fails initialization (ticket #372).
- Fix invalid SAVEPOINT issued when entering *Connection.transaction()* within a pipeline using an implicit transaction (ticket #374).
- Fix queries with repeated named parameters in *ClientCursor* (ticket #378).
- Distribute macOS arm64 (Apple M1) binary packages (ticket #344).

Psycopg 3.1.1

- Work around broken Homebrew installation of the libpq in a non-standard path (ticket #364)
- Fix possible "unrecognized service" error in async connection when no port is specified (ticket #366).

Psycopg 3.1

- Add *Pipeline mode* (ticket #74).
- Add *Client-side-binding cursors* (ticket #101).
- Add CockroachDB support in *psycopg.crdb* (ticket #313).
- Add *Two-Phase Commit* support (ticket #72).
- Add *Enum adaptation* (ticket #274).
- Add `returning` parameter to *executemany()* to retrieve query results (ticket #164).
- *executemany()* performance improved by using batch mode internally (ticket #145).
- Add parameters to *copy()*.
- Add *COPY Writer objects*.
- Resolve domain names asynchronously in *AsyncConnection.connect()* (ticket #259).
- Add *pq.PGconn.trace()* and related trace functions (ticket #167).
- Add `prepare_threshold` parameter to *Connection* init (ticket #200).
- Add `cursor_factory` parameter to *Connection* init.

- Add *Error.pgconn* and *Error.pgresult* attributes (ticket #242).
- Restrict queries to be LiteralString as per **PEP 675** (ticket #323).
- Add explicit type cast to values converted by *sql.Literal* (ticket #205).
- Drop support for Python 3.6.

Psycopg 3.0.17

- Fix segfaults on fork on some Linux systems using ctypes implementation (ticket #300).
- Load bytea as bytes, not memoryview, using ctypes implementation.

Psycopg 3.0.16

- Fix missing *rowcount* after SHOW (ticket #343).
- Add scripts to build macOS arm64 packages (ticket #162).

Psycopg 3.0.15

- Fix wrong escaping of unprintable chars in COPY (nonetheless correctly interpreted by PostgreSQL).
- Restore the connection to usable state after an error in *stream()*.
- Raise *DataError* instead of OverflowError loading binary intervals out-of-range.
- Distribute manylinux2014 wheel packages (ticket #124).

Psycopg 3.0.14

- Raise *DataError* dumping arrays of mixed types (ticket #301).
- Fix handling of incorrect server results, with blank sqlstate (ticket #303).
- Fix bad Float4 conversion on ppc64le/musllinux (ticket #304).

Psycopg 3.0.13

- Fix *Cursor.stream()* slowness (ticket #286).
- Fix oid for lists of integers, which might cause the server choosing bad plans (ticket #293).
- Make *Connection.cancel()* on a closed connection a no-op instead of an error.

Psycopg 3.0.12

- Allow `bytearray`/`memoryview` data too as *`Copy.write()`* input (ticket #254).
- Fix dumping `IntEnum` in text mode, Python implementation.

Psycopg 3.0.11

- Fix *`DataError`* loading arrays with dimensions information (ticket #253).
- Fix hanging during COPY in case of memory error (ticket #255).
- Fix error propagation from COPY worker thread (mentioned in ticket #255).

Psycopg 3.0.10

- Leave the connection in working state after interrupting a query with Ctrl-C (ticket #231).
- Fix *`Cursor.description`* after a COPY … TO STDOUT operation (ticket #235).
- Fix building on FreeBSD and likely other BSD flavours (ticket #241).

Psycopg 3.0.9

- Set *`Error.sqlstate`* when an unknown code is received (ticket #225).
- Add the `tzdata` package as a dependency on Windows in order to handle time zones (ticket #223).

Psycopg 3.0.8

- Decode connection errors in the `client_encoding` specified in the connection string, if available (ticket #194).
- Fix possible warnings in objects deletion on interpreter shutdown (ticket #198).
- Don't leave connections in ACTIVE state in case of error during COPY … TO STDOUT (ticket #203).

Psycopg 3.0.7

- Fix crash in *`executemany()`* with no input sequence (ticket #179).
- Fix wrong *`rowcount`* after an *`executemany()`* returning no rows (ticket #178).

Psycopg 3.0.6

- Allow to use *`Cursor.description`* if the connection is closed (ticket #172).
- Don't raise exceptions on *`ServerCursor.close()`* if the connection is closed (ticket #173).
- Fail on *`Connection.cursor()`* if the connection is closed (ticket #174).
- Raise *`ProgrammingError`* if out-of-order exit from transaction contexts is detected (tickets #176, #177).
- Add `CHECK_STANDBY` value to *`ConnStatus`* enum.

Psycopg 3.0.5

- Fix possible "Too many open files" OS error, reported on macOS but possible on other platforms too (ticket #158).
- Don't clobber exceptions if a transaction block exit with error and rollback fails (ticket #165).

Psycopg 3.0.4

- Allow to use the module with strict strings comparison (ticket #147).
- Fix segfault on Python 3.6 running in `-W error` mode, related to `backport.zoneinfo` ticket #109.
- Build binary package with libpq versions not affected by CVE-2021-23222 (ticket #149).

Psycopg 3.0.3

- Release musllinux binary packages, compatible with Alpine Linux (ticket #141).
- Reduce size of binary package by stripping debug symbols (ticket #142).
- Include typing information in the `psycopg_binary` package.

Psycopg 3.0.2

- Fix type hint for *`sql.SQL.join()`* (ticket #127).
- Fix type hint for *`Connection.notifies()`* (ticket #128).
- Fix call to `MultiRange.__setitem__()` with a non-iterable value and a slice, now raising a `TypeError` (ticket #129).
- Fix disable cursors methods after close() (ticket #125).

Psycopg 3.0.1

- Fix use of the wrong dumper reusing cursors with the same query but different parameter types (ticket #112).

Psycopg 3.0

First stable release. Changed from 3.0b1:

- Add *Geometry adaptation using Shapely* (ticket #80).
- Add *Multirange adaptation* (ticket #75).
- Add *`pq.__build_version__`* constant.
- Don't use the extended protocol with COPY, (tickets #78, #82).
- Add `context` parameter to *`connect()`* (ticket #83).
- Fix selection of dumper by oid after *`set_types()`*.
- Drop `Connection.client_encoding`. Use *`ConnectionInfo.encoding`* to read it, and a `SET` statement to change it.

- Add binary packages for Python 3.10 (ticket #103).

Psycopg 3.0b1

- First public release on PyPI.

1.4.2 `psycopg_pool` release notes

Future releases

psycopg_pool 3.1.3 (unreleased)

- Add support for Python 3.11 (ticket #305).

Current release

psycopg_pool 3.1.2

- Fix possible failure to reconnect after losing connection from the server (ticket #370).

psycopg_pool 3.1.1

- Fix race condition on pool creation which might result in the pool not filling (ticket #230).

psycopg_pool 3.1.0

- Add *Null connection pools* (ticket #148).
- Add `ConnectionPool.open()` and *open* parameter to the pool init (ticket #151).
- Drop support for Python 3.6.

psycopg_pool 3.0.3

- Raise `ValueError` if `ConnectionPool` `min_size` and `max_size` are both set to 0 (instead of hanging).
- Raise `PoolClosed` calling `wait()` on a closed pool.

psycopg_pool 3.0.2

- Remove dependency on the internal `psycopg._compat` module.

psycopg_pool 3.0.1

- Don't leave connections idle in transaction after calling `check()` (ticket #144).

psycopg_pool 3.0

- First release on PyPI.

1.5 Indices and tables

- genindex
- modindex

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

S

T

U

V

W

X