aiopg Documentation

Release 0.15.0

Andrew Svetlov

Aug 14, 2018

Contents

aiopg is a library for accessing a *PostgreSQL* database from the asyncio (PEP-3156/tulip) framework. It wraps asynchronous features of the Psycopg database driver.

CHAPTER 1

Features

- Implements *asyncio DBAPI like* interface for *PostgreSQL*. It includes *Connection*, *Cursor* and *Pool* objects.
- Implements *optional* support for charming *sqlalchemy* functional sql layer.

CHAPTER 2

Basics

The library uses `psycopg2` connections in **asynchronous** mode internally.

Literally it is an (almost) transparent wrapper for psycopg2 connection and cursor, but with only exception.

You should use `yield from conn.f()` instead of just call `conn.f()` for every method.

Properties are unchanged, so `conn.prop` is correct as well as `conn.prop = val`.

See example:

```
import asyncio
import aiopg

dsn = 'dbname=aiopg user=aiopg password=passwd host=127.0.0.1'

async def go():
    async with aiopg.create_pool(dsn) as pool:
        async with pool.acquire() as conn:
            async with conn.cursor() as cur:
                await cur.execute("SELECT 1")
                ret = []
                async for row in cur:
                    ret.append(row)
                assert ret == [(1,)]

loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

For documentation about connection and cursor methods/properties please go to psycopg docs: http://initd.org/psycopg/docs/

Note: psycopg2 creates new connections with `autocommit=True` option in asynchronous mode. Autocommitting cannot be disabled.

See *Transactions* about transaction usage in *autocommit mode*.

Note: Throughout this documentation, examples utilize the *async/await* syntax introduced by **PEP 492** that is only valid for Python 3.5+.

If you are using Python 3.4, please replace `await` with `yield from` and `async def` with a `@coroutine` decorator. For example, this:

```
async def coro(...):
    ret = await f()
```

shoud be replaced by:

```
@asyncio.coroutine
def coro(...):
    ret = yield from f()
```

see also *yield from/@coroutine style* examples.

CHAPTER 3

SQLAlchemy and aiopg

Core API Reference provides core support for *PostgreSQL* connections.

We have found it to be very annoying to write raw SQL queries manually, so we introduce support for *sqlalchemy* query builders:

```
import asyncio
from aiopg.sa import create_engine
import sqlalchemy as sa


metadata = sa.MetaData()

tbl = sa.Table('tbl', metadata,
               sa.Column('id', sa.Integer, primary_key=True),
               sa.Column('val', sa.String(255)))

async def go():
    async with create_engine(user='aiopg',
                             database='aiopg',
                             host='127.0.0.1',
                             password='passwd') as engine:
        async with engine.acquire() as conn:
            await conn.execute(tbl.insert().values(val='abc'))

            async for row in conn.execute(tbl.select().where(tbl.c.val=='abc')):
                print(row.id, row.val)


loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

We believe constructions like `tbl.insert().values(val='abc')` and `tbl.select().where(tbl.c.val=='abc')` to be very handy and convenient.

CHAPTER 4

Installation

```
pip3 install aiopg
```

Note: *aiopg* requires *psycopg2* library.

You can use standard one from your distro like:

```
$ sudo apt-get install python3-psycopg2
```

but if you like to use virtual environments (*virtualenvwrapper*, *virtualenv* or *venv*) you probably have to install *libpq* development package:

```
$ sudo apt-get install libpq-dev
```

Also you probably want to use *aiopg.sa*.

aiopg.sa module is **optional** and requires *sqlalchemy*. You can install *sqlalchemy* by running:

```
pip3 install sqlalchemy
```

CHAPTER 5

Source code

The project is hosted on GitHub

Please feel free to file an issue on bug tracker if you have found a bug or have some suggestion for library improvement.

The library uses Travis for Continious Integration.

CHAPTER 6

Discussion list

aio-libs google group: https://groups.google.com/forum/#!forum/aio-libs

Feel free to post your questions and ideas here.

CHAPTER 7

Dependencies

- Python 3.3 and `asyncio` or Python 3.4+
- psycopg2
- aiopg.sa requires *sqlalchemy*.

CHAPTER 8

Authors and License

The `aiopg` package is written by Andrew Svetlov. It's BSD licensed and freely available.

Feel free to improve this package and send a pull request to GitHub.

Contents:

8.1 Core API Reference

8.1.1 Connection

The library provides a way to connect to *PostgreSQL* database.

Example:

```
@asyncio.coroutine
def go():
    conn = yield from aiopg.connect(database='aiopg',
                                    user='aiopg',
                                    password='secret',
                                    host='127.0.0.1')
    cur = yield from conn.cursor()
    yield from cur.execute("SELECT * FROM tbl")
    ret = yield from cur.fetchall()
```

coroutine async-with `aiopg.`**connect**(*dsn=None*, ***, *loop=None*, *timeout=60.0*, *enable_json=True*, *enable_hstore=True*, *enable_uuid=True*, *echo=False*, ***kwargs*)

Make a connection to *PostgreSQL* server.

The function accepts all parameters that `psycopg2.connect()` does plus optional keyword-only *loop* and *timeout* parameters.

Parameters

- **loop** – asyncio event loop instance or `None` for default one.
- **timeout** (*float*) – default timeout (in seconds) for connection operations.

  60 secs by default.
- **enable_json** (*bool*) – enable json column types for connection.

  `True` by default.
- **enable_hstore** (*bool*) – try to enable hstore column types for connection.

  `True` by default.

  For using HSTORE columns extension should be installed in database first:

```
CREATE EXTENSION HSTORE
```

- **enable_uuid** (*bool*) – enable uuid column types for connection.

  `True` by default.
- **echo** (*bool*) – log executed SQL statement (`False` by default).

Returns *Connection* instance.

class aiopg.**Connection**

A connection to a *PostgreSQL* database instance. It encapsulates a database session.

Its insterface is very close to `psycopg2.connection` (http://initd.org/psycopg/docs/connection.html) except all methods are coroutines.

Use *connect()* for creating connection.

The most important method is

coroutine async-with cursor(*name=None*, *cursor_factory=None*, *scrollable=None*, *withhold=False*, *, *timeout=None*)

Creates a new cursor object using the connection.

The only *cursor_factory* can be specified, all other parameters are not supported by *psycopg2* in asynchronous mode yet.

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

timeout is a timeout for returned cursor instance if parameter is not *None*.

name, *scrollable* and *withhold* parameters are not supported by *psycopg2* in asynchronous mode.

Returns *Cursor* instance.

close()

Immediatelly close the connection.

Close the connection now (rather than whenever `del` is executed). The connection will be unusable from this point forward; an `psycopg2.InterfaceError` will be raised if any operation is attempted with the connection. The same applies to all cursor objects trying to use the connection. Note that closing a connection without committing the changes first will cause any pending change to be discarded as if a `ROLLBACK` was performed.

Changed in version 0.5: *close()* is regular function now. For sake of backward compatibility the method returns `asyncio.Future` instance with result already set to `None` (you still can use `yield from conn.close()` construction.

closed
The readonly property that returns `True` if connections is closed.

echo
Return *echo mode* status. Log all executed queries to logger named `aiopg` if `True`

raw
The readonly property that underlying `psycopg2.connection` instance.

coroutine cancel(*timeout=None*)
Cancel current database operation.

The method interrupts the processing of the current operation. If no query is being executed, it does nothing. You can call this function from a different thread than the one currently executing a database operation, for instance if you want to cancel a long running query if a button is pushed in the UI. Interrupting query execution will cause the cancelled method to raise a `psycopg2.extensions.QueryCanceledError`. Note that the termination of the query is not guaranteed to succeed: see the documentation for `PQcancel()`.

Parameters **timeout** (*float*) – timeout for cancelling.

dsn
The readonly property that returns *dsn* string used by the connection.

autocommit
Autocommit mode status for connection (always `True`).

Note: *psycopg2* doesn't allow to change *autocommit* mode in asynchronous mode.

---

encoding
Client encoding for SQL operations.

Note: *psycopg2* doesn't allow to change encoding in asynchronous mode.

---

isolation_level
Get the transaction isolation level for the current session.

Note: The only value allowed in asynchronous mode value is `psycopg2.extensions.ISOLATION_LEVEL_READ_COMMITTED` (`READ COMMITTED`).

---

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

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

To avoid a leak in case excessive notices are generated, only the last 50 messages are kept.

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

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

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

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

See also:

libpq docs for PQbackendPID() for details.

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

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

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

See also:

libpq docs for PQparameterStatus() for details.

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

See also:

libpq docs for PQtransactionStatus() for details.

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

See also:

libpq docs for PQprotocolVersion() for details.

server_version
A read-only integer representing the backend version.

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

See also:

libpq docs for PQserverVersion() for details.

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

The status is undefined for *closed* connectons.

timeout
A read-only float representing default timeout for connection's operations.

notifies
An `asyncio.Queue` instance for received notifications.

See also:

Server-side notifications

The `Connection` class also has several methods not described here. Those methods are not supported in asynchronous mode (`psycopg2.ProgrammingError` is raised).

8.1.2 Cursor

class aiopg.**Cursor**

A cursor for connection.

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

Cursors that are created from the same connection are not isolated, i.e., any changes done to the database by a cursor are immediately visible by the other cursors. Cursors created from different connections can or can not be isolated, depending on the connections' isolation level.

Its insterface is very close to `psycopg2.cursor` (http://initd.org/psycopg/docs/cursor.html) except all methods are coroutines.

Use `Connection.cursor()` for getting cursor for connection.

echo

Return *echo mode* status. Log all executed queries to logger named `aiopg` if `True`

description

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

Each of these sequences is a `collections.namedtuple()` containing information describing one result column:

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

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

close()

Close the cursor now (rather than whenever `del` is executed). The cursor will be unusable from this point forward; an `psycopg2.InterfaceError` will be raised if any operation is attempted with the cursor.

Note: `close()` is not a coroutine, you don't need to wait it via `yield from curs.close()`.

---

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

raw
The readonly property that underlying `psycopg2.cursor` instance.

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

timeout
A read-only float representing default timeout for cursor's operations.

coroutine execute(*operation*, *parameters=None*, *, *timeout=None*)
Prepare and execute a database operation (query or command).

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

Parameters **timeout** (*`float`*) – overrides cursor's timeout if not `None`.

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

coroutine callproc(*procname*, *parameters=None*, *, *timeout=None*)
Call a stored database procedure with the given name. The sequence of parameters must contain one entry for each argument that the procedure expects. The result of the call is returned as modified copy of the input sequence. Input parameters are left untouched, output and input/output parameters replaced with possibly new values.

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

Parameters **timeout** (*`float`*) – overrides cursor's timeout if not `None`.

mogrify(*operation*, *parameters=None*)
Returns a query string after arguments binding. The string returned is exactly the one that would be sent to the database running the *`Cursor.execute()`* method or similar.

The returned string is always a bytes string:

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

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

Results retrieval methods

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

Cursor object supports *asynchronous iteration* starting from Python 3.5:

```
await cursor.execute('SELECT key, value FROM tbl;')
async for key, value in cursor:
    ...
```

Warning: *Cursor* objects do **not** support regular iteration (using `for` statement) since version 0.7.

Iterable protocol in *Cursor* hides `yield from` from user, witch should be explicit. Moreover iteration support is optional, according to PEP-249 (https://www.python.org/dev/peps/pep-0249/#iter).

coroutine fetchone()

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

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

A `psycopg2.ProgrammingError` is raised if the previous call to *execute()* did not produce any result set or no call was issued yet.

coroutine fetchmany(*size=cursor.arraysize*)

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

The number of rows to fetch per call is specified by the parameter. If it is not given, the cursor's *Cursor.arraysize* determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned:

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

A `psycopg2.ProgrammingError` is raised if the previous call to *execute()* did not produce any result set or no call was issued yet.

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

coroutine fetchall()

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

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

A `psycopg2.ProgrammingError` is raised if the previous call to *execute()* did not produce any result set or no call was issued yet.

coroutine scroll(*value*, *mode='relative'*)

Scroll the cursor in the result set to a new position according to mode.

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

If the scroll operation would leave the result set, a `psycopg2.ProgrammingError` is raised and the cursor position is not changed.

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> yield from cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)",␣
↪(42, 'bar'))
>>> cur.statusmessage
```

(continues on next page)

(continued from previous page)

```
'INSERT 0 1'
```

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

setoutputsize(*size*, *column=None*)
This method is exposed in compliance with the *DBAPI*. It currently does nothing but it is safe to call it.

coroutine begin()
Begin a transaction and return a transaction handle. The returned object is an instance of `_TransactionBeginContextManager`.:

```
async def begin(engine):
   async with engine.cursor() as cur:
       async with cur.begin():
           await cur.execute("insert into tbl values(1, 'data')")

       async with cur.begin():
           await cur.execute('select * from tbl')
           row = await cur.fetchall()
           assert row == [(22, 'read only'), (1, 'data'), ]
```

coroutine begin_nested()
Begin a nested transaction and return a transaction handle.

The returned object is an instance of `_TransactionBeginContextManager`.

Any transaction in the hierarchy may `commit` and `rollback`, however the outermost transaction still controls the overall `commit` or `rollback` of the transaction of a whole. It utilizes SAVEPOINT facility of *PostgreSQL* server:

```
async def begin_nested(engine):
    async with engine.cursor() as cur:
        async with cur.begin_nested():
            await cur.execute("insert into tbl values(1, 'data')")

                try:
                    async with cur.begin_nested():
                        await cur.execute("insert into tbl values(1/0, 'no␣
↪data')")
                except psycopg2.DataError:
                        pass

        async with cur.begin_nested():
            await cur.execute("insert into tbl values(2, 'data')")

        async with cur.begin_nested():
            await cur.execute('select * from tbl')
            row = await cur.fetchall()
            assert row == [(22, 'read only'), (1, 'data'),  (2, 'data'), ]
```

8.1.3 Pool

The library provides *connection pool* as well as plain *Connection* objects.

The basic usage is:

```
import asyncio
import aiopg

dsn = 'dbname=jetty user=nick password=1234 host=localhost port=5432'


@asyncio.coroutine
def test_select():
    pool = yield from aiopg.create_pool(dsn)

    with (yield from pool.cursor()) as cur:
        yield from cur.execute('SELECT 1')
        ret = yield from cur.fetchone()
        assert ret == (1,), ret
```

coroutine async-with aiopg.**create_pool**(*dsn=None*, ***, *minsize=1*, *maxsize=10*, *enable_json=True*, *enable_hstore=True*, *enable_uuid=True*, *echo=False*, *on_connect=None*, *loop=None*, *timeout=60.0*, ***kwargs*)

Create a pool of connections to *PostgreSQL* database.

The function accepts all parameters that `psycopg2.connect()` does plus optional keyword-only parameters *loop*, *minsize*, *maxsize*.

Parameters

- **loop** – is an optional *event loop* instance, `asyncio.get_event_loop()` is used if *loop* is not specified.
- **minsize** (*int*) – minimum size of the *pool*, `1` by default.
- **maxsize** (*int*) – maximum sizes of the *pool*, `10` by default. `0` means unlimited pool size.
- **timeout** (*float*) – a default timeout (in seconds) for connection operations. `60` secs by default.
- **enable_json** (*bool*) – enable json column types for connections created by the pool, enabled by default.
- **enable_hstore** (*bool*) – enable hstore column types for connections created by the pool, enabled by default.

  For using HSTORE columns extension should be installed in database first:

  ```
  CREATE EXTENSION HSTORE
  ```

- **enable_uuid** (*bool*) – enable UUID column types for connections created by the pool, enabled by default.
- **echo** (*bool*) – executed log SQL queryes (disabled by default).
- **on_connect** – a *callback coroutine* executed at once for every created connection. May be used for setting up connection level state like client encoding etc.
- **pool_recycle** (*float*) – number of seconds after which connection is recycled, helps to deal with stale connections in pool, default value is `-1`, means recycling logic is disabled.

Returns *Pool* instance.

class aiopg.**Pool**

A connection pool.

After creation pool has *minsize* free connections and can grow up to *maxsize* ones.

If *minsize* is `0` the pool doesn't creates any connection on startup.

If *maxsize* is `0` than size of pool is unlimited (but it recycles used connections of course).

The most important way to use it is getting connection in *with statement*:

```
with (yield from pool) as conn:
    cur = yield from conn.cursor()
```

and shortcut for getting *cursor* directly:

```
with (yield from pool.cursor()) as cur:
    yield from cur.execute('SELECT 1')
```

See also *Pool.acquire()* and *Pool.release()* for acquring *connection* without *with statement*.

echo
Return *echo mode* status. Log all executed queries to logger named `aiopg` if `True`

minsize
A minimal size of the pool (*read-only*), `1` by default.

maxsize
A maximal size of the pool (*read-only*), `10` by default.

size
A current size of the pool (*readonly*). Includes used and free connections.

freesize
A count of free connections in the pool (*readonly*).

timeout
A read-only float representing default timeout for operations for connections from pool.

clear()
A coroutine that closes all *free* connections in the pool. At next connection acquiring at least *minsize* of them will be recreated.

close()
Close pool.

Mark all pool connections to be closed on getting back to pool. Closed pool doesn't allow to acquire new connections.

If you want to wait for actual closing of acquired connection please call *wait_closed()* after *close()*.

> **Warning:** The method is not a coroutine.

terminate()
Terminate pool.

Close pool with instantly closing all acquired connections also.

wait_closed() should be called after *terminate()* for waiting for actual finishing.

> **Warning:** The method is not a coroutine.

coroutine wait_closed()
Wait for releasing and closing all acquired connections.

Should be called after *close()* for waiting for actual pool closing.

coroutine async-with acquire()
Acquire a connection from *free pool*. Create a new connection if needed and *size* of pool is less than *maxsize*.

Returns a *Connection* instance.

> **Warning:** nested `acquire()` might lead to deadlocks.

release(*conn*)
A coroutine that reverts connection *conn* to *free pool* for future recycling.

Changed in version 0.10: The method is converted into a coroutine to get exception context in case of errors.

The change is backward compatible though since technically it's a regular method returning a future instance.

cursor(*name=None*, *cursor_factory=None*, *scrollable=None*, *withhold=False*, ***, *timeout=None*)
A coroutine that *acquires* a connection and returns *context manager*.

The only *cursor_factory* can be specified, all other parameters are not supported by *psycopg2* in asynchronous mode yet.

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

timeout is a timeout for returned cursor instance if parameter is not *None*.

name, *scrollable* and *withhold* parameters are not supported by *psycopg2* in asynchronous mode.

The usage is:

```
with (yield from pool.cursor()) as cur:
    yield from cur.execute('SELECT 1')
```

After exiting from *with block* cursor *cur* will be closed.

8.1.4 Exceptions

Any call to library function, method or property can raise an exception.

aiopg doesn't define any exception class itself, it reuses DBAPI Exceptions from `psycopg2`

8.1.5 Transactions

While *psycopg2* asynchronous connections have to be in *autocommit mode* it is still possible to use SQL transactions executing **BEGIN** and **COMMIT** statements manually as Psycopg Asynchronous Support docs .

`Connection.commit()` and `Connection.rollback()` methods are disabled and always raises `psycopg2.ProgrammingError` exception.

8.1.6 Extension type translations

JSON

aiopg has support for JSON data type enabled by default.

For pushing data to server please wrap json dict into psycopg2.extras.Json:

```
from psycopg2.extras import Json

data = {'a': 1, 'b': 'str'}
yield from cur.execute("INSERT INTO tbl (val) VALUES (%s)", [Json(data)])
```

On receiving data from json column *psycopg2* autoconvers result into python dict object:

```
yield from cur.execute("SELECT val FROM tbl")
item = yield from cur.fetchone()
assert item == {'b': 'str', 'a': 1}
```

8.1.7 Server-side notifications

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

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

Receiving part should establish listening on notification channel by LISTEN call and wait notification events from *Connection.notifies* queue.

There is usage example:

```
import asyncio
import aiopg

dsn = 'dbname=aiopg user=aiopg password=passwd host=127.0.0.1'


async def notify(conn):
    async with conn.cursor() as cur:
        for i in range(5):
            msg = "message {}".format(i)
            print('Send ->', msg)
            await cur.execute("NOTIFY channel, %s", (msg,))

        await cur.execute("NOTIFY channel, 'finish'")


async def listen(conn):
    async with conn.cursor() as cur:
        await cur.execute("LISTEN channel")
        while True:
            msg = await conn.notifies.get()
            if msg.payload == 'finish':
                return
            else:
```

(continues on next page)

(continued from previous page)

```
                print('Receive <-', msg.payload)


async def main():
    async with aiopg.create_pool(dsn) as pool:
        async with pool.acquire() as conn1:
            listener = listen(conn1)
            async with pool.acquire() as conn2:
                notifier = notify(conn2)
                await asyncio.gather(listener, notifier)
    print("ALL DONE")


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

8.2 aiopg.sa — support for SQLAlchemy functional SQL layer

8.2.1 Intro

While *core API* provides a core support for access to *PostgreSQL* database, I found manipulations with raw SQL strings too annoying.

Fortunately we can use excellent SQLAlchemy Core as **SQL query builder**.

Example:

```
import asyncio
from aiopg.sa import create_engine
import sqlalchemy as sa


metadata = sa.MetaData()

tbl = sa.Table('tbl', metadata,
               sa.Column('id', sa.Integer, primary_key=True),
               sa.Column('val', sa.String(255)))


async def create_table(engine):
    async with engine.acquire() as conn:
        await conn.execute('DROP TABLE IF EXISTS tbl')
        await conn.execute('''CREATE TABLE tbl (
                                  id serial PRIMARY KEY,
                                  val varchar(255))''')


async def go():
    async with create_engine(user='aiopg',
                             database='aiopg',
                             host='127.0.0.1',
                             password='passwd') as engine:

        await create_table(engine)
```

(continues on next page)

(continued from previous page)

```
        async with engine.acquire() as conn:
            await conn.execute(tbl.insert().values(val='abc'))

            async for row in conn.execute(tbl.select()):
                print(row.id, row.val)


loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

So you can execute SQL query built by `tbl.insert().values(val='abc')` or `tbl.select()` expressions.

sqlalchemy has rich and very powerful set of SQL construction functions, please read tutorial for full list of available operations.

Also we provide SQL transactions support. Please take a look on *SAConnection.begin()* method and family.

8.2.2 Engine

coroutine async-with aiopg.sa.**create_engine**(*dsn=None*, ***, *minsize=1*, *maxsize=10*, *loop=None*, *dialect=dialect*, *timeout=60*, ***kwargs*)

Crate an *Engine* instance with embedded connection pool.

The pool has *minsize* opened connections to *PostgreSQL* server.

aiopg.sa.**dialect**

An instance of *SQLAlchemy* dialect set up for *psycopg2* usage.

An `sqlalchemy.engine.interfaces.Dialect` instance.

See also:

`sqlalchemy.dialects.postgresql.psycopg2` psycopg2 dialect.

class aiopg.sa.**Engine**

Connects a *aiopg.Pool* and `sqlalchemy.engine.interfaces.Dialect` together to provide a source of database connectivity and behavior.

An *Engine* object is instantiated publicly using the *create_engine()* coroutine.

dialect

A `sqlalchemy.engine.interfaces.Dialect` for the engine, readonly property.

name

A name of the dialect, readonly property.

driver

A driver of the dialect, readonly property.

dsn

DSN connection info, readonly property.

See also:

psycopg2 connection.dsn attribute.

minsize

A minimal size of the pool (*read-only*), `1` by default.

maxsize
A maximal size of the pool (*read-only*), `10` by default.

size
A current size of the pool (*readonly*). Includes used and free connections.

freesize
A count of free connections in the pool (*readonly*).

timeout
A read-only float representing default timeout for operations for connections from pool.

close()
Close engine.

Mark all engine connections to be closed on getting back to engine. Closed engine doesn't allow to acquire new connections.

If you want to wait for actual closing of acquired connection please call *wait_closed()* after *close()*.

> **Warning:** The method is not a coroutine.

terminate()
Terminate engine.

Close engine's pool with instantly closing all acquired connections also.

wait_closed() should be called after *terminate()* for waiting for actual finishing.

> **Warning:** The method is not a coroutine.

coroutine wait_closed()
A coroutine that waits for releasing and closing all acquired connections.

Should be called after *close()* for waiting for actual engine closing.

coroutine async-with acquire()
Get a connection from pool.

This method is a coroutine.

Returns a *SAConnection* instance. Result of this method could be used as async contex manager:

```
async with engine.acquire() as conn:
    await conn.execute(tbl.insert().values(val='abc'))
```

> **Warning:** nested `acquire()` might lead to deadlocks.

release()
Revert back connection *conn* to pool.

> **Warning:** The method is not a coroutine.

8.2.3 Connection

class aiopg.sa.**SAConnection**

A wrapper for *aiopg.Connection* instance.

The class provides methods for executing *SQL queries* and working with *SQL transactions*.

coroutine async-for execute(*query*, **multiparams*, ***params*)

Executes a *SQL query* with optional parameters.

Parameters

- **query** – a SQL query string or any *sqlalchemy* expression (see SQLAlchemy Core)
- ***multiparams/**params** – represent bound parameter values to be used in the execution. Typically, the format is either a dictionary passed to *multiparams:

```
await conn.execute(
    table.insert(),
    {"id":1, "value":"v1"}
)
```

...or individual key/values interpreted by **params:

```
await conn.execute(
    table.insert(), id=1, value="v1"
)
```

In the case that a plain SQL string is passed, a tuple or individual values in *multiparams may be passed:

```
await conn.execute(
    "INSERT INTO table (id, value) VALUES (%d, %s)",
    (1, "v1")
)

await conn.execute(
    "INSERT INTO table (id, value) VALUES (%s, %s)",
    1, "v1"
)
```

Result value for SELECT statements may be iterated immediately:

```
async for row conn.execute(tbl.select()):
    print(row.id, row.name, row.surname)
```

Returns *ResultProxy* instance with results of SQL query execution.

coroutine scalar(*query*, **multiparams*, ***params*)

Executes a *SQL query* and returns a scalar value.

See also:

SAConnection.execute() and *ResultProxy.scalar()*.

closed

The readonly property that returns True if connections is closed.

coroutine async-with begin()

Begin a transaction and return a transaction handle.

This method is a coroutine.

The returned object is an instance of *Transaction*. This object represents the "scope" of the transaction, which completes when either the *Transaction.rollback()* or *Transaction.commit()* method is called.

Nested calls to *begin()* on the same *SAConnection* will return new *Transaction* objects that represent an emulated transaction within the scope of the enclosing transaction, that is:

```
trans = await conn.begin()    # outermost transaction
trans2 = await conn.begin()   # "inner"
await trans2.commit()         # does nothing
await trans.commit()          # actually commits
```

Calls to *Transaction.commit()* only have an effect when invoked via the outermost *Transaction* object, though the *Transaction.rollback()* method of any of the *Transaction* objects will roll back the transaction.

See also:

SAConnection.begin_nested() - use a SAVEPOINT

SAConnection.begin_twophase() **- use a two phase (XA)** transaction

coroutine async-with begin_nested()
Begin a nested transaction and return a transaction handle.

The returned object is an instance of *NestedTransaction*.

Any transaction in the hierarchy may `commit` and `rollback`, however the outermost transaction still controls the overall `commit` or `rollback` of the transaction of a whole. It utilizes SAVEPOINT facility of *PostgreSQL* server.

See also:

SAConnection.begin(), *SAConnection.begin_twophase()*.

coroutine async-with begin_twophase(*xid=None*)
Begin a two-phase or XA transaction and return a transaction handle.

The returned object is an instance of *TwoPhaseTransaction*, which in addition to the methods provided by *Transaction*, also provides a *prepare()* method.

Parameters **xid** – the two phase transaction id. If not supplied, a random id will be generated.

See also:

SAConnection.begin(), *SAConnection.begin_twophase()*.

coroutine recover_twophase()
Return a list of prepared twophase transaction ids.

coroutine rollback_prepared(*xid*)
Rollback prepared twophase transaction *xid*.

coroutine commit_prepared(*xid*)
Commit prepared twophase transaction *xid*.

in_transaction
The readonly property that returns `True` if a transaction is in progress.

coroutine close()
Close this *SAConnection*.

This results in a release of the underlying database resources, that is, the *aiopg.Connection* referenced internally. The *aiopg.Connection* is typically restored back to the connection-holding *aiopg.Pool* referenced by the *Engine* that produced this *SAConnection*. Any transactional state present on the *aiopg.Connection* is also unconditionally released via calling *Transaction.rollback()* method.

After *close()* is called, the *SAConnection* is permanently in a closed state, and will allow no further operations.

8.2.4 ResultProxy

class aiopg.sa.**ResultProxy**

Wraps a *DB-API like* `Cursor` object to provide easier access to row columns.

Individual columns may be accessed by their integer position, case-sensitive column name, or by `sqlalchemy.schema.Column`` object. e.g.:

```
async for row in conn.execute(...):
    col1 = row[0]    # access via integer position
    col2 = row['col2']   # access via name
    col3 = row[mytable.c.mycol] # access via Column object.
```

ResultProxy also handles post-processing of result column data using `sqlalchemy.types.TypeEngine` objects, which are referenced from the originating SQL statement that produced this result set.

dialect

The readonly property that returns `sqlalchemy.engine.interfaces.Dialect` dialect for the *ResultProxy* instance.

See also:

dialect global data.

keys()

Return the current set of string keys for rows.

rowcount

The readonly property that returns the 'rowcount' for this result.

The 'rowcount' reports the number of rows *matched* by the WHERE criterion of an UPDATE or DELETE statement.

Note: Notes regarding *ResultProxy.rowcount*:

- This attribute returns the number of rows *matched*, which is not necessarily the same as the number of rows that were actually *modified* - an UPDATE statement, for example, may have no net change on a given row if the SET values given are the same as those present in the row already. Such a row would be matched but not modified.
- *ResultProxy.rowcount* is *only* useful in conjunction with an UPDATE or DELETE statement. Contrary to what the Python DBAPI says, it does *not* return the number of rows available from the results of a SELECT statement as DBAPIs cannot support this functionality when rows are unbuffered.
- Statements that use RETURNING does not return a correct rowcount.

---

returns_rows

A readonly property that returns `True` if this *ResultProxy* returns rows.

I.e. if it is legal to call the methods *ResultProxy.fetchone()*, *ResultProxy.fetchmany()*, *ResultProxy.fetchall()*.

closed
Return `True` if this *ResultProxy* is closed (no pending rows in underlying cursor).

close()
Close this *ResultProxy*.

Closes the underlying *aiopg.Cursor* corresponding to the execution.

Note that any data cached within this *ResultProxy* is still available. For some types of results, this may include buffered rows.

This method is called automatically when:

- all result rows are exhausted using the fetchXXX() methods.
- cursor.description is None.

coroutine fetchall()
Fetch all rows, just like *aiopg.Cursor.fetchall()*.

The connection is closed after the call.

Returns a list of *RowProxy*.

coroutine fetchone()
Fetch one row, just like *aiopg.Cursor.fetchone()*.

If a row is present, the cursor remains open after this is called.

Else the cursor is automatically closed and `None` is returned.

Returns an *RowProxy* instance or `None`.

coroutine fetchmany(*size=None*)
Fetch many rows, just like *aiopg.Cursor.fetchmany()*.

If rows are present, the cursor remains open after this is called.

Else the cursor is automatically closed and an empty list is returned.

Returns a list of *RowProxy*.

coroutine first()
Fetch the first row and then close the result set unconditionally.

Returns `None` if no row is present or an *RowProxy* instance.

coroutine scalar()
Fetch the first column of the first row, and close the result set.

Returns `None` if no row is present or an *RowProxy* instance.

class aiopg.sa.**RowProxy**
A `collections.abc.Mapping` for representing a row in query result.

Keys are column names, values are result values.

Individual columns may be accessed by their integer position, case-sensitive column name, or by `sqlalchemy.schema.Column`` object.

Has overloaded operators __eq__ and __ne__ for comparing two rows.

The *RowProxy* is *not hashable*.

..method:: as_tuple()

Return a tuple with values from `RowProxy.values()`.

8.2.5 Transaction objects

class aiopg.sa.**Transaction**

Represent a database transaction in progress.

The *Transaction* object is procured by calling the *SAConnection.begin()* method of *SAConnection*:

```
async with engine.acquire() as conn:
    async with conn.begin() as tr:
        await conn.execute("insert into x (a, b) values (1, 2)")
```

The object provides *rollback()* and *commit()* methods in order to control transaction boundaries. Contex manager will invoke *rollback()* in case of exception in contex managers code block and *commit()* - in case of success.

See also:

SAConnection.begin(), *SAConnection.begin_twophase()*, *SAConnection.begin_nested()*.

is_active

A readonly property that returns `True` if a transaction is active.

connection

A readonly property that returns *SAConnection* for transaction.

coroutine close()

Close this *Transaction*.

If this transaction is the base transaction in a begin/commit nesting, the transaction will *Transaction.rollback()*. Otherwise, the method returns.

This is used to cancel a *Transaction* without affecting the scope of an enclosing transaction.

coroutine rollback()

Roll back this *Transaction*.

coroutine commit()

Commit this *Transaction*.

class aiopg.sa.**NestedTransaction**

Represent a 'nested', or SAVEPOINT transaction.

A new *NestedTransaction* object may be procured using the *SAConnection.begin_nested()* method.

The interface is the same as that of *Transaction*.

See also:

PostgreSQL commands for nested transactions:

- SAVEPOINT
- RELEASE SAVEPOINT
- ROLLBACK TO SAVEPOINT

class aiopg.sa.**TwoPhaseTransaction**

Represent a two-phase transaction.

A new *TwoPhaseTransaction* object may be procured using the *SAConnection.begin_twophase()* method.

The interface is the same as that of *Transaction* with the addition of the *TwoPhaseTransaction.prepare()* method.

xid

A readonly property that returns twophase transaction id.

coroutine prepare()

Prepare this *TwoPhaseTransaction*.

After a PREPARE, the transaction can be committed.

See also:

PostgreSQL commands for two phase transactions:

- PREPARE TRANSACTION
- COMMIT PREPARED
- ROLLBACK PREPARED

8.3 Examples of aiopg usage

Below is a list of examples from aiopg/examples

Every example is a correct tiny python program.

8.3.1 async/await style

Low-level API

```
import asyncio
import aiopg

dsn = 'dbname=aiopg user=aiopg password=passwd host=127.0.0.1'


async def test_select():
    async with aiopg.create_pool(dsn) as pool:
        async with pool.acquire() as conn:
            async with conn.cursor() as cur:
                await cur.execute("SELECT 1")
                ret = []
                async for row in cur:
                    ret.append(row)
                assert ret == [(1,)]
    print("ALL DONE")


loop = asyncio.get_event_loop()
loop.run_until_complete(test_select())
```

Usage of LISTEN/NOTIFY commands

```
import asyncio
import aiopg

dsn = 'dbname=aiopg user=aiopg password=passwd host=127.0.0.1'


async def notify(conn):
    async with conn.cursor() as cur:
        for i in range(5):
            msg = "message {}".format(i)
            print('Send ->', msg)
            await cur.execute("NOTIFY channel, %s", (msg,))

        await cur.execute("NOTIFY channel, 'finish'")


async def listen(conn):
    async with conn.cursor() as cur:
        await cur.execute("LISTEN channel")
        while True:
            msg = await conn.notifies.get()
            if msg.payload == 'finish':
                return
            else:
                print('Receive <-', msg.payload)


async def main():
    async with aiopg.create_pool(dsn) as pool:
        async with pool.acquire() as conn1:
            listener = listen(conn1)
            async with pool.acquire() as conn2:
                notifier = notify(conn2)
                await asyncio.gather(listener, notifier)
    print("ALL DONE")


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

Simple sqlalchemy usage

```
import asyncio
from aiopg.sa import create_engine
import sqlalchemy as sa


metadata = sa.MetaData()

tbl = sa.Table('tbl', metadata,
               sa.Column('id', sa.Integer, primary_key=True),
               sa.Column('val', sa.String(255)))
```

(continues on next page)

(continued from previous page)

```
async def create_table(conn):
    await conn.execute('DROP TABLE IF EXISTS tbl')
    await conn.execute('''CREATE TABLE tbl (
    id serial PRIMARY KEY,
    val varchar(255))''')


async def go():
    async with create_engine(user='aiopg',
                             database='aiopg',
                             host='127.0.0.1',
                             password='passwd') as engine:
        async with engine.acquire() as conn:
            await create_table(conn)
        async with engine.acquire() as conn:
            await conn.execute(tbl.insert().values(val='abc'))

            async for row in conn.execute(tbl.select()):
                print(row.id, row.val)


loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

Complex sqlalchemy queries

```
import asyncio
from aiopg.sa import create_engine
import sqlalchemy as sa
import random
import datetime


metadata = sa.MetaData()

users = sa.Table('users', metadata,
                 sa.Column('id', sa.Integer, primary_key=True),
                 sa.Column('name', sa.String(255)),
                 sa.Column('birthday', sa.DateTime))

emails = sa.Table('emails', metadata,
                  sa.Column('id', sa.Integer, primary_key=True),
                  sa.Column('user_id', None, sa.ForeignKey('users.id')),
                  sa.Column('email', sa.String(255), nullable=False),
                  sa.Column('private', sa.Boolean, nullable=False))


async def create_tables(conn):
    await conn.execute('DROP TABLE IF EXISTS emails')
    await conn.execute('DROP TABLE IF EXISTS users')
    await conn.execute('''CREATE TABLE users (
                                       id serial PRIMARY KEY,
                                       name varchar(255),
                                       birthday timestamp)''')
    await conn.execute('''CREATE TABLE emails (
```

(continues on next page)

(continued from previous page)

```
                                id serial,
                                user_id int references users(id),
                                email varchar(253),
                                private bool)''')


names = {'Andrew', 'Bob', 'John', 'Vitaly', 'Alex', 'Lina', 'Olga',
         'Doug', 'Julia', 'Matt', 'Jessica', 'Nick', 'Dave', 'Martin',
         'Abbi', 'Eva', 'Lori', 'Rita', 'Rosa', 'Ivy', 'Clare', 'Maria',
         'Jenni', 'Margo', 'Anna'}


def gen_birthday():
    now = datetime.datetime.now()
    year = random.randint(now.year - 30, now.year - 20)
    month = random.randint(1, 12)
    day = random.randint(1, 28)
    return datetime.datetime(year, month, day)


async def fill_data(conn):
    async with conn.begin():
        for name in random.sample(names, len(names)):
            uid = await conn.scalar(
                users.insert().values(name=name, birthday=gen_birthday()))
            emails_count = int(random.paretovariate(2))
            for num in random.sample(range(10000), emails_count):
                is_private = random.uniform(0, 1) < 0.8
                await conn.execute(emails.insert().values(
                    user_id=uid,
                    email='{}+{}@gmail.com'.format(name, num),
                    private=is_private))


async def count(conn):
    c1 = (await conn.scalar(users.count()))
    c2 = (await conn.scalar(emails.count()))
    print("Population consists of", c1, "people with",
          c2, "emails in total")
    join = sa.join(emails, users, users.c.id == emails.c.user_id)
    query = (sa.select([users.c.name])
             .select_from(join)
             .where(emails.c.private == False)  # noqa
             .group_by(users.c.name)
             .having(sa.func.count(emails.c.private) > 0))

    print("Users with public emails:")
    async for row in conn.execute(query):
        print(row.name)

    print()


async def show_julia(conn):
    print("Lookup for Julia:")
    join = sa.join(emails, users, users.c.id == emails.c.user_id)
    query = (sa.select([users, emails], use_labels=True)
```

(continues on next page)

(continued from previous page)

```
            .select_from(join).where(users.c.name == 'Julia'))
    async for row in conn.execute(query):
        print(row.users_name, row.users_birthday,
              row.emails_email, row.emails_private)
    print()


async def ave_age(conn):
    query = (sa.select([sa.func.avg(sa.func.age(users.c.birthday))])
             .select_from(users))
    ave = (await conn.scalar(query))
    print("Average age of population is", ave,
          "or ~", int(ave.days / 365), "years")
    print()


async def go():
    engine = await create_engine(user='aiopg',
                                 database='aiopg',
                                 host='127.0.0.1',
                                 password='passwd')
    async with engine:
        async with engine.acquire() as conn:
            await create_tables(conn)
            await fill_data(conn)
            await count(conn)
            await show_julia(conn)
            await ave_age(conn)


loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

8.3.2 yield from/@coroutine style

Old style Low-level API

```
import asyncio
import aiopg

dsn = 'dbname=aiopg user=aiopg password=passwd host=127.0.0.1'


@asyncio.coroutine
def test_select():
    pool = yield from aiopg.create_pool(dsn)
    with (yield from pool.cursor()) as cur:
        yield from cur.execute("SELECT 1")
        ret = yield from cur.fetchone()
        assert ret == (1,)
    print("ALL DONE")


loop = asyncio.get_event_loop()
loop.run_until_complete(test_select())
```

Usage of LISTEN/NOTIFY commands using old-style API

```
import asyncio
import aiopg

dsn = 'dbname=aiopg user=aiopg password=passwd host=127.0.0.1'


@asyncio.coroutine
def notify(conn):
    cur = yield from conn.cursor()
    try:
        for i in range(5):
            msg = "message {}".format(i)
            print('Send ->', msg)
            yield from cur.execute("NOTIFY channel, %s", (msg,))

        yield from cur.execute("NOTIFY channel, 'finish'")
    finally:
        cur.close()


@asyncio.coroutine
def listen(conn):
    cur = yield from conn.cursor()
    try:
        yield from cur.execute("LISTEN channel")
        while True:
            msg = yield from conn.notifies.get()
            if msg.payload == 'finish':
                return
            else:
                print('Receive <-', msg.payload)
    finally:
        cur.close()


@asyncio.coroutine
def main():
    pool = yield from aiopg.create_pool(dsn)
    with (yield from pool) as conn1:
        listener = listen(conn1)
        with (yield from pool) as conn2:
            notifier = notify(conn2)
            yield from asyncio.gather(listener, notifier)
    print("ALL DONE")


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

Simple sqlalchemy usage commands using old-style API

```
import asyncio
from aiopg.sa import create_engine
import sqlalchemy as sa
```

(continues on next page)

(continued from previous page)

```
metadata = sa.MetaData()

tbl = sa.Table('tbl', metadata,
               sa.Column('id', sa.Integer, primary_key=True),
               sa.Column('val', sa.String(255)))


@asyncio.coroutine
def create_table(engine):
    with (yield from engine) as conn:
        yield from conn.execute('DROP TABLE IF EXISTS tbl')
        yield from conn.execute('''CREATE TABLE tbl (
                                       id serial PRIMARY KEY,
                                       val varchar(255))''')


@asyncio.coroutine
def go():
    engine = yield from create_engine(user='aiopg',
                                      database='aiopg',
                                      host='127.0.0.1',
                                      password='passwd')

    yield from create_table(engine)
    with (yield from engine) as conn:
        yield from conn.execute(tbl.insert().values(val='abc'))

        res = yield from conn.execute(tbl.select())
        for row in res:
            print(row.id, row.val)


loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

Complex sqlalchemy queries commands using old-style API

```
import asyncio
from aiopg.sa import create_engine
import sqlalchemy as sa
import random
import datetime


metadata = sa.MetaData()

users = sa.Table('users', metadata,
                 sa.Column('id', sa.Integer, primary_key=True),
                 sa.Column('name', sa.String(255)),
                 sa.Column('birthday', sa.DateTime))

emails = sa.Table('emails', metadata,
                  sa.Column('id', sa.Integer, primary_key=True),
```

(continues on next page)

(continued from previous page)

```
                  sa.Column('user_id', None, sa.ForeignKey('users.id')),
                  sa.Column('email', sa.String(255), nullable=False),
                  sa.Column('private', sa.Boolean, nullable=False))


@asyncio.coroutine
def create_tables(engine):
    with (yield from engine) as conn:
        yield from conn.execute('DROP TABLE IF EXISTS emails')
        yield from conn.execute('DROP TABLE IF EXISTS users')
        yield from conn.execute('''CREATE TABLE users (
                                        id serial PRIMARY KEY,
                                        name varchar(255),
                                        birthday timestamp)''')
        yield from conn.execute('''CREATE TABLE emails (
                                    id serial,
                                    user_id int references users(id),
                                    email varchar(253),
                                    private bool)''')


names = {'Andrew', 'Bob', 'John', 'Vitaly', 'Alex', 'Lina', 'Olga',
         'Doug', 'Julia', 'Matt', 'Jessica', 'Nick', 'Dave', 'Martin',
         'Abbi', 'Eva', 'Lori', 'Rita', 'Rosa', 'Ivy', 'Clare', 'Maria',
         'Jenni', 'Margo', 'Anna'}


def gen_birthday():
    now = datetime.datetime.now()
    year = random.randint(now.year - 30, now.year - 20)
    month = random.randint(1, 12)
    day = random.randint(1, 28)
    return datetime.datetime(year, month, day)


@asyncio.coroutine
def fill_data(engine):
    with (yield from engine) as conn:
        tr = yield from conn.begin()

        for name in random.sample(names, len(names)):
            uid = yield from conn.scalar(
                users.insert().values(name=name, birthday=gen_birthday()))
            emails_count = int(random.paretovariate(2))
            for num in random.sample(range(10000), emails_count):
                is_private = random.uniform(0, 1) < 0.8
                yield from conn.execute(emails.insert().values(
                    user_id=uid,
                    email='{}+{}@gmail.com'.format(name, num),
                    private=is_private))
        yield from tr.commit()


@asyncio.coroutine
def count(engine):
    with (yield from engine) as conn:
        c1 = (yield from conn.scalar(users.count()))
```

(continues on next page)

(continued from previous page)

```
        c2 = (yield from conn.scalar(emails.count()))
        print("Population consists of", c1, "people with",
              c2, "emails in total")
        join = sa.join(emails, users, users.c.id == emails.c.user_id)
        query = (sa.select([users.c.name])
                 .select_from(join)
                 .where(emails.c.private == False)  # noqa
                 .group_by(users.c.name)
                 .having(sa.func.count(emails.c.private) > 0))

        print("Users with public emails:")
        ret = yield from conn.execute(query)
        for row in ret:
            print(row.name)

        print()


@asyncio.coroutine
def show_julia(engine):
    with (yield from engine) as conn:
        print("Lookup for Julia:")
        join = sa.join(emails, users, users.c.id == emails.c.user_id)
        query = (sa.select([users, emails], use_labels=True)
                 .select_from(join).where(users.c.name == 'Julia'))
        res = yield from conn.execute(query)
        for row in res:
            print(row.users_name, row.users_birthday,
                  row.emails_email, row.emails_private)
        print()


@asyncio.coroutine
def ave_age(engine):
    with (yield from engine) as conn:
        query = (sa.select([sa.func.avg(sa.func.age(users.c.birthday))])
                 .select_from(users))
        ave = (yield from conn.scalar(query))
        print("Average age of population is", ave,
              "or ~", int(ave.days / 365), "years")
        print()


@asyncio.coroutine
def go():
    engine = yield from create_engine(user='aiopg',
                                      database='aiopg',
                                      host='127.0.0.1',
                                      password='passwd')

    yield from create_tables(engine)
    yield from fill_data(engine)
    yield from count(engine)
    yield from show_julia(engine)
    yield from ave_age(engine)
```

(continues on next page)

(continued from previous page)

```
loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

8.4 Instruction for contributors

8.4.1 Developer environment

First clone the git repo:

```
$ git clone git@github.com:aio-libs/aiopg.git
$ cd aiopg
```

After that you need to create and activate a virtual environment. I recommend using *virtualenvwrapper* but just *virtualenv* or *venv* will also work. For `virtualenvwrapper`:

```
$ mkvirtualenv aiopg -p `which python3`
```

For `venv` (for example; put the directory wherever you want):

```
$ python3 -m venv ../venv_directory
$ source ../venv_directory/bin/activate
```

Just as when doing a normal install, you need the *libpq* library:

```
$ sudo apt-get install libpq-dev
```

UPD

The latest `aiopg` test suite uses docker container for running Postgres server. See https://docs.docker.com/engine/installation/linux/ubuntulinux/ for instructions for Docker installing.

No local Postgres server needed.

In the virtual environment you need to install *aiopg* itself and some additional development tools (the development tools are needed for running the test suite and other development tasks):

```
$ pip install -Ue .
$ pip install -Ur requirements.txt
```

That's all.

To run all of the *aiopg* tests do:

```
$ make test
```

This command runs *pep8* and *pyflakes* first and then executes the *aiopg* unit tests.

When you are working on solving an issue you will probably want to run some specific test, not the whole suite:

```
$ py.test -s -k test_initial_empty
```

For debug sessions I prefer to use *ipdb*, which is installed as part of the development tools. Insert the following line into your code in the place where you want to start interactively debugging the execution process:

```
import ipdb; ipdb.set_trace()
```

The library is reasonably well covered by tests. There is a make target for generating the coverage report:

```
$ make cov
```

8.4.2 Contribution

I like to get well-formed Pull Requests on github. The pull request should include both the code fix and tests for the bug.

If you cannot make a good test yourself or want to report a problem, please open an issue at https://github.com/aio-libs/aiopg/issues.

8.5 Glossary

DBAPI **PEP 249** – Python Database API Specification v2.0

ipdb ipdb exports functions to access the IPython debugger, which features tab completion, syntax highlighting, better tracebacks, better introspection with the same interface as the pdb module.

libpq The standard C library to communicate with *PostgreSQL* server.

http://www.postgresql.org/docs/9.3/interactive/libpq.html

pep8 Python style guide checker

pep8 is a tool to check your Python code against some of the style conventions in **PEP 8** – Style Guide for Python Code.

PostgreSQL A popular database server.

http://www.postgresql.org/

psycopg2 A PostgreSQL database adapter for the Python programming language. psycopg2 was written with the aim of being very small and fast, and stable as a rock.

http://initd.org/psycopg/

pyflakes passive checker of Python programs

A simple program which checks Python source files for errors.

Pyflakes analyzes programs and detects various errors. It works by parsing the source file, not importing it, so it is safe to use on modules with side effects. It's also much faster.

https://pypi.python.org/pypi/pyflakes

sqlalchemy The Python SQL Toolkit and Object Relational Mapper.

http://www.sqlalchemy.org/

venv standard python module for creating lightweight "virtual environments" with their own site directories, optionally isolated from system site directories. Each virtual environment has its own Python binary (allowing creation of environments with various Python versions) and can have its own independent set of installed Python packages in its site directories.

https://docs.python.org/dev/library/venv.html

virtualenv The tool to create isolated Python environments. It's not included into python stdlib's but it is very popular instrument.

venv and *virtualenv* does almost the same, *venv* has been developed after the *virtualenv*.

https://virtualenv.readthedocs.io/en/latest/

virtualenvwrapper virtualenvwrapper is a set of extensions to Ian Bicking's *virtualenv* tool. The extensions include wrappers for creating and deleting virtual environments and otherwise managing your development workflow, making it easier to work on more than one project at a time without introducing conflicts in their dependencies.

virtualenvwrapper is my choice, highly recommend the tool to everyone.

https://virtualenvwrapper.readthedocs.io/en/latest/

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index