Postgres.py Documentation

Release 4.0

Chad Whitacre et al.

Sep 20, 2021

Contents

This is a PostgreSQL client library for humans.

CHAPTER 1

Installation

postgres is available on GitHub and on PyPI:

```
$ pip install postgres
```

postgres requires psycopg2 version 2.8 or higher.

We currently test against Python 3.6, 3.7, 3.8 and 3.9. We don't have a testing matrix for different versions of psycopg2 or PostgreSQL.

postgres is released under the MIT license.

CHAPTER 2

See Also

The sql library provides a run / one / all API for any DB API 2.0 driver.

The Records library provides a similar top-level API, and integration with SQLAlchemy and Tablib.

CHAPTER 3

Tutorial

Instantiate a *Postgres* object when your application starts:

```
>>> from postgres import Postgres
>>> db = Postgres()
```

Use *run* to run SQL statements:

```
>>> db.run("CREATE TABLE foo (bar text, baz int)")
>>> db.run("INSERT INTO foo VALUES ('buz', 42)")
>>> db.run("INSERT INTO foo VALUES ('bit', 537)")
```

Use *one* to run SQL and fetch one result or `None`:

```
>>> db.one("SELECT * FROM foo WHERE bar='buz'")
Record(bar='buz', baz=42)
>>> db.one("SELECT * FROM foo WHERE bar='blam'")
```

Use *all* to run SQL and fetch all results:

```
>>> db.all("SELECT * FROM foo ORDER BY bar")
[Record(bar='bit', baz=537), Record(bar='buz', baz=42)]
```

If your queries return one column then you get just the value or a list of values instead of a record or list of records:

```
>>> db.one("SELECT baz FROM foo WHERE bar='buz'")
42
>>> db.all("SELECT baz FROM foo ORDER BY bar")
[537, 42]
```

Jump ahead for the *ORM Tutorial*.

3.1 Bind Parameters

In case you're not familiar with bind parameters in DB-API 2.0, the basic idea is that you put `%(foo)s` in your SQL strings, and then pass in a second argument, a `dict`, containing parameters that `psycopg2` (as an implementation of DB-API 2.0) will bind to the query in a way that is safe against SQL injection. (This is inspired by old-style Python string formatting, but it is not the same.)

```
>>> db.one("SELECT * FROM foo WHERE bar=%(bar)s", {"bar": "buz"})
Record(bar='buz', baz=42)
```

As a convenience, passing parameters as keyword arguments is also supported:

```
>>> db.one("SELECT * FROM foo WHERE bar=%(bar)s", bar="buz")
Record(bar='buz', baz=42)
```

Never build SQL strings out of user input!

Always pass user input as bind parameters!

3.2 Context Managers

Eighty percent of your database usage should be covered by the simple *run*, *one*, *all* API introduced above. For the other 20%, *postgres* provides two context managers for working at increasingly lower levels of abstraction. The lowest level of abstraction in *postgres* is a psycopg2 connection pool that we configure and manage for you. Everything in *postgres*, both the simple API and the context managers, uses this connection pool.

Use the *get_cursor* context manager to work directly with a *simple cursor*, while still taking advantage of connection pooling and automatic transaction management:

```
>>> with db.get_cursor() as cursor:
...     cursor.run("INSERT INTO foo VALUES ('blam')")
...     cursor.all("SELECT * FROM foo ORDER BY bar")
...
[Record(bar='bit', baz=537), Record(bar='blam', baz=None), Record(bar='buz', baz=42)]
```

Note that other calls won't see the changes on your transaction until the end of your code block, when the context manager commits the transaction for you:

```
>>> db.run("DELETE FROM foo WHERE bar='blam'")
>>> with db.get_cursor() as cursor:
...     cursor.run("INSERT INTO foo VALUES ('blam')")
...     db.all("SELECT * FROM foo ORDER BY bar")
...
[Record(bar='bit', baz=537), Record(bar='buz', baz=42)]
>>> db.all("SELECT * FROM foo ORDER BY bar")
[Record(bar='bit', baz=537), Record(bar='blam', baz=None), Record(bar='buz', baz=42)]
```

The *get_cursor* method gives you a context manager that wraps a *simple cursor*. It has `autocommit` turned off on its connection. If the block under management raises an exception, the connection is rolled back. Otherwise it's committed. Use this when you want a series of statements to be part of one transaction, but you don't need fine-grained control over the transaction. For fine-grained control, use *get_connection* to get a connection straight from the connection pool:

```
>>> db.run("DELETE FROM foo WHERE bar='blam'")
>>> with db.get_connection() as connection:
...     cursor = connection.cursor()
...     cursor.all("SELECT * FROM foo ORDER BY bar")
...
[Record(bar='bit', baz=537), Record(bar='buz', baz=42)]
```

A connection gotten in this way will have `autocommit` turned off, and it'll never be implicitly committed otherwise. It'll actually be rolled back when you're done with it, so it's up to you to explicitly commit as needed. This is the lowest-level abstraction that *`postgres`* provides, basically just a pre-configured connection pool from `psycopg2` that uses *simple cursors*.

CHAPTER 4

The Postgres Object

exception postgres.**NotASimpleCursor**

exception postgres.**NotAModel**

exception postgres.**NoTypeSpecified**

exception postgres.**NoSuchType**

exception postgres.**AlreadyRegistered**

exception postgres.**NotRegistered**

class postgres.**Postgres**(*url=''*, *minconn=1*, *maxconn=10*, *idle_timeout=600*, *readonly=False*, *cursor_factory=<class 'postgres.cursors.SimpleNamedTupleCursor'>*, *back_as_registry={<class 'tuple'>: <function return_tuple_as_is>, 'tuple': <function return_tuple_as_is>, <class 'dict'>: <function make_dict>, 'dict': <function make_dict>, <function namedtuple>: <function make_namedtuple>, 'namedtuple': <function make_namedtuple>, <class 'postgres.cursors.Row'>: <class 'postgres.cursors.Row'>, 'Row': <class 'postgres.cursors.Row'>}*, *pool_class=<class 'psycopg2_pool.ThreadSafeConnectionPool'>*, *cache=None*)

Interact with a PostgreSQL database.

Parameters

- **url** (*str*) – A `postgres://` URL or a PostgreSQL connection string
- **minconn** (*int*) – The minimum size of the connection pool
- **maxconn** (*int*) – The maximum size of the connection pool
- **idle_timeout** (*int*) – How many seconds to wait before closing an idle connection.
- **readonly** (*bool*) – Setting this to `True` makes all connections and cursors readonly by default.
- **cursor_factory** (*type*) – The type of cursor to use when none is specified. Defaults to *SimpleNamedTupleCursor*.

- **back_as_registry** (*dict*) – Defines the values that can be passed to various methods as a `back_as` argument.
- **pool_class** (*type*) – The type of pool to use. Defaults to `ThreadSafeConnectionPool`.
- **cache** (`Cache`) – An instance of *postgres.cache.Cache*.

This is the main object that *postgres* provides, and you should have one instance per process for each PostgreSQL database your process wants to talk to using this library.

```
>>> import postgres
>>> db = postgres.Postgres()
```

(Note that importing *postgres* under Python 2 will cause the registration of typecasters with `psycopg2` to ensure that you get unicode instead of bytestrings for text data, according to this advice.)

The libpq environment variables are used to determine the connection parameters which are not explicitly passed in the `url` argument.

When instantiated, this object creates a connection pool by calling *pool_class* with the *minconn*, *maxconn* and *idle_timeout* arguments. Everything this object provides runs through this connection pool. See the documentation of the `ConnectionPool` class for more information.

`cursor_factory` sets the default cursor that connections managed by this *Postgres* instance will use. See the *Simple Cursors* documentation below for additional options. Whatever default you set here, you can override that default on a per-call basis by passing `cursor_factory` to *get_cursor*.

The names in our simple API, *run*, *one*, and *all*, were chosen to be short and memorable, and to not directly conflict with the DB-API 2.0 `execute`, `fetchone`, and `fetchall` methods, which have slightly different semantics (under DB-API 2.0 you call `execute` on a cursor and then call one of the `fetch*` methods on the same cursor to retrieve records; with our simple API there is no second `fetch` step, and we also provide automatic dereferencing). See issues 16 and 20 for more of the rationale behind these names. The context managers on this class are named starting with `get_` to set them apart from the simple-case API.

run (*sql*, *parameters=None*, ***kw*)

Execute a query and discard any results.

Returns `None`

This is a convenience method, it passes all its arguments to *SimpleCursorBase.run* like this:

```
with self.get_cursor() as cursor:
    cursor.run(sql, parameters, **kw)
```

one (*sql*, *parameters=None*, ***kw*)

Execute a query and return a single result or a default value.

Returns a single record or value, or `default` (if `default` is not an `Exception`)

Raises `TooFew` or `TooMany`, or `default` (if `default` is an `Exception`)

This is a convenience method, it passes all its arguments to *SimpleCursorBase.one* like this:

```
with self.get_cursor() as cursor:
    return cursor.one(sql, parameters, **kw)
```

all (*sql*, *parameters=None*, ***kw*)

Execute a query and return all results.

Returns `list` of records or `list` of single values

This is a convenience method, it passes all its arguments to *SimpleCursorBase.all* like this:

```
with self.get_cursor() as cursor:
    return cursor.all(sql, parameters, **kw)
```

get_cursor(*cursor=None*, ***kw*)

Return a *CursorContextManager* that uses our connection pool.

Parameters

- **cursor** – use an existing cursor instead of creating a new one (see the explanations and caveats below)
- **kw** – passed through to *CursorContextManager* or *CursorSubcontextManager*

```
>>> with db.get_cursor() as cursor:
...     cursor.all("SELECT * FROM foo")
...
[Record(bar='buz', baz=42), Record(bar='bit', baz=537)]
```

You can use our simple *run*, *one*, *all* API, and you can also use the traditional DB-API 2.0 methods:

```
>>> with db.get_cursor() as cursor:
...     cursor.execute("SELECT * FROM foo")
...     cursor.fetchall()
...
[Record(bar='buz', baz=42), Record(bar='bit', baz=537)]
```

By default the cursor will have `autocommit` turned off on its connection. If your code block inside the `with` statement raises an exception, the transaction will be rolled back. Otherwise, it'll be committed. The context manager closes the cursor when the block ends and puts the connection back in the pool. The cursor is destroyed after use.

Use this when you want a series of statements to be part of one transaction, but you don't need fine-grained control over the transaction.

The *cursor* argument enables running queries in a subtransaction. The major difference between a transaction and a subtransaction is that the changes in the database are **not** committed (nor rolled back) at the end of a subtransaction.

The *cursor* argument is typically used inside functions which have an optional *cursor* argument themselves, like this:

```
>>> def do_something(cursor=None):
...     with db.get_cursor(cursor=cursor) as c:
...         foo = c.one("...")
...         # ... do more stuff
...     # Warning: At this point you cannot assume that the changes have
...     # been committed, so don't do anything that could be problematic
...     # or incoherent if the transaction ends up being rolled back.
```

When the *cursor* argument isn't `None`, the *back_as* argument is still supported, but the other arguments (*autocommit*, *readonly*, and the arguments of the `connection.cursor` method) are **not** supported.

get_connection(***kw*)

Return a `ConnectionContextManager` that uses our connection pool.

Parameters **kw** – passed through to *ConnectionContextManager*

```
>>> with db.get_connection() as connection:
...     cursor = connection.cursor()
...     cursor.all("SELECT * FROM foo")
...
[Record(bar='buz', baz=42), Record(bar='bit', baz=537)]
```

Use this when you want to take advantage of connection pooling and our simple *run*, *one*, *all* API, but otherwise need full control, for example, to do complex things with transactions.

Cursors from connections gotten this way also support the traditional DB-API 2.0 methods:

```
>>> with db.get_connection() as connection:
...     cursor = connection.cursor()
...     cursor.execute("SELECT * FROM foo")
...     cursor.fetchall()
...
[Record(bar='buz', baz=42), Record(bar='bit', baz=537)]
```

register_model (*ModelSubclass*, *typname=None*)
Register an ORM model.

Parameters

- **ModelSubclass** – the *Model* subclass to register with this *Postgres* instance
- **typname** – a string indicating the Postgres type to register this model for (`typname`, without an "e," is the name of the relevant column in the underlying `pg_type` table). If `None`, we'll look for `ModelSubclass.typname`.

Raises *NotAModel*, *NoTypeSpecified*, *NoSuchType*, *AlreadyRegistered*

Note: See the *orm* docs for instructions on subclassing *Model*.

---

unregister_model (*ModelSubclass*)
Unregister an ORM model.

Parameters **ModelSubclass** – the *Model* subclass to unregister

Raises *NotAModel*, *NotRegistered*

If `ModelSubclass` is registered for multiple types, it is unregistered for all of them.

check_registration (*ModelSubclass*, *include_subsubclasses=False*)
Check whether an ORM model is registered.

Parameters

- **ModelSubclass** – the *Model* subclass to check for
- **include_subsubclasses** (*bool*) – whether to also check for subclasses of `ModelSubclass` or just `ModelSubclass` itself

Returns the type names (`typname`) for which this model is registered

Return type list

Raises *NotAModel*, *NotRegistered*

postgres.**make_Connection** (*postgres*)
Define and return a subclass of `psycopg2.extensions.connection`.

Parameters **postgres** – the *Postgres* instance to bind to

Returns a `Connection` class

The class defined and returned here will be linked to the instance of *`Postgres`* that is passed in as *`postgres`*, which will use this class as the `connection_factory` for its connection pool.

The `cursor` method of this class accepts a `back_as` keyword argument. By default the valid values for `back_as` are `tuple`, `namedtuple`, `dict` and `Row` (or the strings `tuple`, `namedtuple`, `dict`, `Row`), and `None`. If `back_as` is not `None`, then it modifies the default row type of the cursor.

We also set client encoding to `UTF-8`.

class postgres.**ModelCaster**(*name*, *oid*, *attrs*, *array_oid=None*, *schema=None*)

A `CompositeCaster` subclass for *`Model`*.

CHAPTER 5

The Context Managers

class postgres.context_managers.**CursorContextManager**(*pool*, *autocommit=False*, *readonly=False*, ***cursor_kwargs*)

Instantiated once per *get_cursor* call.

Parameters

- **pool** – see psycopg2_pool
- **autocommit** (*bool*) – see connection.autocommit
- **readonly** (*bool*) – see connection.readonly
- **cursor_kwargs** – passed to connection.cursor

During construction, a connection is checked out of the connection pool and its autocommit and readonly attributes are set, then a cursor is created from that connection.

Upon exit of the with block, the connection is rolled back if an exception was raised, or committed otherwise. There are two exceptions to this:

1. if autocommit is True, then the connection is neither rolled back nor committed;
2. if readonly is True, then the connection is always rolled back, never committed.

In all cases the cursor is closed and the connection is put back in the pool.

class postgres.context_managers.**ConnectionCursorContextManager**(*conn*, *autocommit=False*, *readonly=False*, ***cursor_kwargs*)

Creates a cursor from the given connection, then wraps it in a context manager that automatically commits or rolls back the changes on exit.

Parameters

- **conn** – a connection
- **autocommit** (*bool*) – see connection.autocommit

- **readonly** (*bool*) – see connection.readonly
- **cursor_kwargs** – passed to connection.cursor

During construction, the connection's autocommit and readonly attributes are set, then connection.cursor is called with *cursor_kwargs*.

Upon exit of the with block, the connection is rolled back if an exception was raised, or committed otherwise. There are two exceptions to this:

1. if autocommit is True, then the connection is neither rolled back nor committed;
2. if readonly is True, then the connection is always rolled back, never committed.

In all cases the cursor is closed.

class postgres.context_managers.**CursorSubcontextManager**(*cursor*, *back_as=<object object>*)

Wraps a cursor so that it can be used for a subtransaction.

See *get_cursor* for an explanation of subtransactions.

Parameters

- **cursor** – the cursor to wrap
- **back_as** – temporarily overwrites the cursor's *back_as* attribute

class postgres.context_managers.**ConnectionContextManager**(*pool*, *autocommit=False*, *readonly=False*)

Instantiated once per *get_connection* call.

Parameters

- **pool** – see psycopg2_pool
- **autocommit** (*bool*) – see connection.autocommit
- **readonly** (*bool*) – see connection.readonly

This context manager checks out a connection out of the specified pool, sets its autocommit and readonly attributes.

The __enter__ method returns the Connection.

The __exit__ method rolls back the connection and puts it back in the pool.

CHAPTER 6

Simple Cursors

The *postgres* library extends the cursors provided by psycopg2 to add simpler API methods: run, one, and all.

exception postgres.cursors.**BadBackAs**(*bad_value*, *back_as_registry*)

exception postgres.cursors.**OutOfBounds**(*n*, *lo*, *hi*)

exception postgres.cursors.**TooFew**(*n*, *lo*, *hi*)

exception postgres.cursors.**TooMany**(*n*, *lo*, *hi*)

class postgres.cursors.**SimpleCursorBase**

This is a mixin to provide a simpler API atop the usual DB-API 2.0 API provided by psycopg2. Any custom cursor class you would like to use as the cursor_factory argument to *Postgres* must subclass this base.

```
>>> from psycopg2.extras import LoggingCursor
>>> from postgres.cursors import SimpleCursorBase
>>> class SimpleLoggingCursor(LoggingCursor, SimpleCursorBase):
...     pass
...
>>> from postgres import Postgres
>>> db = Postgres(cursor_factory=SimpleLoggingCursor)
```

If you try to use a cursor that doesn't subclass *SimpleCursorBase* as the default cursor_factory for a *Postgres* instance, we won't let you:

```
>>> db = Postgres(cursor_factory=LoggingCursor)  # doctest: +NORMALIZE_WHITESPACE
...
Traceback (most recent call last):
    ...
postgres.NotASimpleCursor: We can only work with subclasses of SimpleCursorBase,
LoggingCursor doesn't fit the bill.
```

However, we do allow you to use whatever you want as the cursor_factory argument for individual calls:

```
>>> with db.get_cursor(cursor_factory=LoggingCursor) as cursor:
...     cursor.all("SELECT * FROM foo")
Traceback (most recent call last):
    ...
AttributeError: 'LoggingCursor' object has no attribute 'all'
```

back_as
Determines which type of row is returned by the various methods. The valid values are the keys of the `back_as_registry`.

execute (*sql*, ***kw*)
This method is an alias of *run*.

run (*sql*, *parameters=None*, ***kw*)
Execute a query, without returning any results.

Parameters

- **sql** (*str*) – the SQL statement to execute
- **parameters** (*dict or tuple*) – the *bind parameters* for the SQL statement
- **kw** – alternative to passing a `dict` as *parameters*

Example usage:

```
>>> db.run("DROP TABLE IF EXISTS foo CASCADE")
>>> db.run("CREATE TABLE foo (bar text, baz int)")
>>> bar, baz = 'buz', 42
>>> db.run("INSERT INTO foo VALUES (%s, %s)", (bar, baz))
>>> db.run("INSERT INTO foo VALUES (%(bar)s, %(baz)s)", dict(bar=bar,␣
↪baz=baz))
>>> db.run("INSERT INTO foo VALUES (%(bar)s, %(baz)s)", bar=bar, baz=baz)
```

one (*sql*, *parameters=None*, *default=None*, *back_as=None*, *max_age=None*, ***kw*)
Execute a query and return a single result or a default value.

Parameters

- **sql** (*str*) – the SQL statement to execute
- **parameters** (*dict or tuple*) – the *bind parameters* for the SQL statement
- **default** – the value to return or raise if no results are found
- **back_as** (*type or string*) – the type of record to return
- **max_age** (*float*) – how long to keep the result in the cache (in seconds)
- **kw** – alternative to passing a `dict` as *parameters*

Returns a single record or value, or `default` (if `default` is not an `Exception`)

Raises `TooFew` or `TooMany`, or `default` (if `default` is an `Exception`)

Use this for the common case where there should only be one record, but it may not exist yet.

```
>>> db.one("SELECT * FROM foo WHERE bar='buz'")
Record(bar='buz', baz=42)
```

If the record doesn't exist, we return `None`:

```
>>> record = db.one("SELECT * FROM foo WHERE bar='blam'")
>>> if record is None:
...     print("No blam yet.")
...
No blam yet.
```

If you pass `default` we'll return that instead of `None`:

```
>>> db.one("SELECT * FROM foo WHERE bar='blam'", default=False)
False
```

If you pass an `Exception` instance or subclass for `default`, we will raise that for you:

```
>>> db.one("SELECT * FROM foo WHERE bar='blam'", default=Exception)
Traceback (most recent call last):
    ...
Exception
```

We specifically stop short of supporting lambdas or other callables for the `default` parameter. That gets complicated quickly, and it's easy to just check the return value in the caller and do your extra logic there.

You can use *`back_as`* to override the type associated with the default `cursor_factory` for your *`Postgres`* instance:

```
>>> db.default_cursor_factory
<class 'postgres.cursors.SimpleNamedTupleCursor'>
>>> db.one( "SELECT * FROM foo WHERE bar='buz'"
...        , back_as=dict
...         )
{'bar': 'buz', 'baz': 42}
```

That's a convenience so you don't have to go to the trouble of remembering where *`SimpleDictCursor`* lives and importing it in order to get dictionaries back.

If the query result has only one column, then we dereference that for you.

```
>>> db.one("SELECT baz FROM foo WHERE bar='buz'")
42
```

And if the dereferenced value is `None`, we return the value of `default`:

```
>>> db.one("SELECT sum(baz) FROM foo WHERE bar='nope'", default=0)
0
```

Dereferencing isn't performed if a *`back_as`* argument is provided:

```
>>> db.one("SELECT null as foo", back_as=dict)
{'foo': None}
```

`all`(*sql*, *parameters=None*, *back_as=None*, *max_age=None*, ***kw*)
Execute a query and return all results.

Parameters

- **`sql`** (*`str`*) – the SQL statement to execute
- **`parameters`** (*`dict or tuple`*) – the *bind parameters* for the SQL statement
- **`back_as`** (*`type or string`*) – the type of record to return

- **max_age** (*float*) – how long to keep the results in the cache (in seconds)
- **kw** – alternative to passing a `dict` as *parameters*

Returns `list` of records or `list` of single values

Use it like this:

```
>>> db.all("SELECT * FROM foo ORDER BY bar")
[Record(bar='bit', baz=537), Record(bar='buz', baz=42)]
```

You can use *back_as* to override the type associated with the default `cursor_factory` for your *Postgres* instance:

```
>>> db.default_cursor_factory
<class 'postgres.cursors.SimpleNamedTupleCursor'>
>>> db.all("SELECT * FROM foo ORDER BY bar", back_as=dict)
[{'bar': 'bit', 'baz': 537}, {'bar': 'buz', 'baz': 42}]
```

That's a convenience so you don't have to go to the trouble of remembering where *SimpleDictCursor* lives and importing it in order to get dictionaries back.

If the query results in records with a single column, we return a list of the values in that column rather than a list of records of values.

```
>>> db.all("SELECT baz FROM foo ORDER BY bar")
[537, 42]
```

Unless a *back_as* argument is provided:

```
>>> db.all("SELECT baz FROM foo ORDER BY bar", back_as=dict)
[{'baz': 537}, {'baz': 42}]
```

class postgres.cursors.**Row**(*cols*, *values*)
A versatile row type.

class postgres.cursors.**SimpleTupleCursor**
A *simple cursor* that returns tuples.

This type of cursor is especially well suited if you need to fetch and process a large number of rows at once, because tuples occupy less memory than dicts.

class postgres.cursors.**SimpleNamedTupleCursor**
A *simple cursor* that returns namedtuples.

This type of cursor is especially well suited if you need to fetch and process a large number of similarly-structured rows at once, and you also need the row objects to be more evolved than simple tuples.

class postgres.cursors.**SimpleDictCursor**
A *simple cursor* that returns dicts.

This type of cursor is especially well suited if you don't care about the order of the columns and don't need to access them as attributes.

class postgres.cursors.**SimpleRowCursor**
A *simple cursor* that returns *Row* objects.

This type of cursor is especially well suited if you want rows to be mutable.

The Row class implements both dict-style and attribute-style lookups and assignments, in addition to index-based lookups. However, index-based assigments aren't allowed.

```
>>> from postgres import Postgres
>>> from postgres.cursors import SimpleRowCursor
>>> db = Postgres(cursor_factory=SimpleRowCursor)
>>> row = db.one("SELECT 1 as key, 'foo' as value")
>>> row[0] == row['key'] == row.key == 1
True
>>> key, value = row
>>> (key, value)
(1, 'foo')
>>> row.value = 'bar'
>>> row.timestamp = '2019-09-20 13:15:22.060537+00'
>>> row
Row(key=1, value='bar', timestamp='2019-09-20 13:15:22.060537+00')
```

Although Row objects support item lookups and assigments, they are not instances of the `dict` class and they don't have its methods (`get`, `items`, etc.).

`postgres.cursors.`**`isexception`**(*obj*)

Given an object, return a boolean indicating whether it is an instance or subclass of `Exception`.

CHAPTER 7

Caching

A query's results can be stored in the `cache` attribute of the *Postgres* object to avoid burdening the database with redundant requests. The caching is enabled by the *max_age* argument of the *one* and *all* methods. For example, this call fetches a row from the *foo* table and caches it for 5 seconds:

```
>>> db.one("SELECT * FROM foo WHERE bar = 'bit'", max_age=5)
Record(bar='bit', baz=537)
```

Any other thread trying to send the same query while it's still being processed waits for the results to be received instead of sending a duplicate query.

The cache key is the query as it's sent to the server, so any difference in the parameter values, casing or even whitespace, will result in a cache miss. You might need to refactor your code if it sends queries that are similar but not exactly identical.

The *max_age* argument doesn't interfere with the *back_as* argument. Moreover, the *one* and *all* methods can each use the results cached by the other. Consequently, the following call hits the cache if it's executed within 5 seconds of the previous call above:

```
>>> db.all("SELECT * FROM foo WHERE bar = 'bit'", back_as=dict, max_age=5)
[{'bar': 'bit', 'baz': 537}]
```

It's also possible to use different *max_age* values for the same query. If a specified *max_age* is greater than the previous one, then the lifetime of the cache entry is extended accordingly.

7.1 The Cache class

class postgres.cache.**Cache**(*max_size=128*)

A cache for rows fetched from a database.

Parameters **max_size** (*int*) – The maximum number of entries allowed in the cache.

> **Warning:** This cache is only designed to be thread-safe in CPython >= 3.6 and similar Python implementations in which the `OrderedDict` class is thread-safe.

A separate lock is used for each entry so that unrelated queries don't block each other.

After inserting a new entry, the oldest one is removed if the cache now has more than *max_size* entries.

clear()
Empty the cache. This method isn't used internally.

get_lock(*key*)
Get the lock object for the specified key.

lookup(*key*, *max_age*)
Look up a cache entry and check its age.

This function returns `None` if there isn't an entry in the cache for the specified key or if the entry is older than *max_age*.

pop_entry(*entry*, *blocking=True*)
Remove the specified entry from the cache.

If *blocking* is *False*, then the entry will only be removed if its lock isn't currently held by another thread.

prune()
Drop stale entries from the cache. This method isn't used internally.

CHAPTER 8

An Object-Relational Mapper (ORM)

It's somewhat of a fool's errand to introduce a Python ORM in 2013, with SQLAlchemy ascendant (Django's ORM not-withstanding). And yet here we are. SQLAlchemy is mature and robust and full-featured. This makes it complex, difficult to learn, and kind of scary. The ORM we introduce here is simpler: it targets PostgreSQL only, it depends on raw SQL (it has no object model for schema definition nor one for query construction), and it never updates your database for you. You are in full, direct control of your application's database usage.

The fundamental technique we employ, introduced by Michael Robbelard at PyOhio 2013, is to write SQL queries that "typecast" results to table types, and then use a `CompositeCaster` subclass to map these to Python objects. This means we get to define our schema in SQL, and we get to write our queries in SQL, and we get to explicitly indicate in our SQL queries how Python should map the results to objects, and then we can write Python objects that contain only business logic and not schema definitions.

8.1 Introducing Table Types

Every table in PostgreSQL has a type associated with it, which is the column definition for that table. These are composite types just like any other composite type in PostgreSQL, meaning we can use them to cast query results. When we do, we get a single field that contains our query result, nested one level:

```
test=# CREATE TABLE foo (bar text, baz int);
CREATE TABLE
test=# INSERT INTO foo VALUES ('blam', 42);
INSERT 0 1
test=# INSERT INTO foo VALUES ('whit', 537);
INSERT 0 1
test=# SELECT * FROM foo;
+------+-----+
| bar  | baz |
+------+-----+
| blam |  42 |
| whit | 537 |
+------+-----+
(2 rows)
```

(continues on next page)

(continued from previous page)

```
test=# SELECT foo FROM foo;
+------------+
|    foo     |
+------------+
| (blam,42)  |
| (whit,537) |
+------------+
(2 rows)

test=#
```

The same thing works for views:

```
test=# CREATE VIEW bar AS SELECT bar FROM foo;
CREATE VIEW
test=# SELECT * FROM bar;
+------+
| bar  |
+------+
| blam |
| whit |
+------+
(2 rows)

test=# SELECT bar FROM bar;
+--------+
|  bar   |
+--------+
| (blam) |
| (whit) |
+--------+
(2 rows)

test=#
```

psycopg2 provides a register_composite function that lets us map PostgreSQL composite types to Python objects. This includes table and view types, and that is the basis for *postgres.orm*. We map based on types, not tables.

8.2 ORM Tutorial

First, write a Python class that subclasses *Model*:

```
>>> from postgres.orm import Model
>>> class Foo(Model):
...     typname = "foo"
...
```

Your model must have a typname attribute, which is the name of the PostgreSQL type for which this class is an object mapping. (typname, spelled without an "e," is the name of the relevant column in the pg_type table in your database.)

Second, register your model with your *Postgres* instance:

```
>>> db.register_model(Foo)
```

That will plug your model into the psycopg2 composite casting machinery, and you'll now get instances of your model back from *one* and *all* when you cast to the relevant type in your query. If your query returns more than one column, you'll need to dereference the column containing the model just as with any other query:

```
>>> rec = db.one("""
...     SELECT foo, bar.*
...       FROM foo
...       JOIN bar ON foo.bar = bar.bar
...   ORDER BY foo.bar
...      LIMIT 1
... """)
>>> rec.foo.bar
'blam'
>>> rec.bar
'blam'
```

And as usual, if your query only returns one column, then *one* and *all* will do the dereferencing for you:

```
>>> foo = db.one("SELECT foo FROM foo WHERE bar='blam'")
>>> foo.bar
'blam'
>>> [foo.bar for foo in db.all("SELECT foo FROM foo")]
['blam', 'whit']
```

To update your database, add a method to your model:

```
>>> db.unregister_model(Foo)
>>> class Foo(Model):
...
...     typname = "foo"
...
...     def update_baz(self, baz):
...         self.db.run("UPDATE foo SET baz=%s WHERE bar=%s", (baz, self.bar))
...         self.set_attributes(baz=baz)
...
>>> db.register_model(Foo)
```

Then use that method to update the database:

```
>>> db.one("SELECT baz FROM foo WHERE bar='blam'")
42
>>> foo = db.one("SELECT foo FROM foo WHERE bar='blam'")
>>> foo.update_baz(90210)
>>> foo.baz
90210
>>> db.one("SELECT baz FROM foo WHERE bar='blam'")
90210
```

We never update your database for you. We also never sync your objects for you: note the use of the *set_attributes* method to sync our instance after modifying the database.

8.3 The Model Base Class

exception postgres.orm.**ReadOnlyAttribute**

exception postgres.orm.**UnknownAttributes**

class postgres.orm.**Model**(*values*)

This is the base class for models in *postgres.orm*.

Instances of subclasses of *Model* will have an attribute for each field in the composite type for which the subclass is registered (for table and view types, these will be the columns of the table or view). These attributes are read-only. We don't update your database. You are expected to do that yourself in methods on your subclass. To keep instance attributes in sync after a database update, use the *set_attributes* helper.

set_attributes(***kw*)

Set instance attributes, according to `kw`.

Raises *UnknownAttributes*

Call this when you update state in the database and you want to keep instance attributes in sync. Note that the only attributes we can set here are the ones that were given to us by the `psycopg2` composite caster machinery when we were first instantiated. These will be the fields of the composite type for which we were registered, which will be column names for table and view types.

CHAPTER 9

Changelog

4.0 (Sep 20, 2021)

- implemented caching query results (#97)
- **BREAKING**: the methods `one` and `all` now have a *max_age* argument; make sure your code doesn't use it as a parameter name
- dropped support for Python 2.7 and 3.5 (#96)

3.0.0 (Oct 19, 2019)

- the `ReadOnly` exception has been renamed to `ReadOnlyAttribute`, and the `_read_only_attributes` attribute of the `Model` class has been renamed to `attnames` (#91)
- the ORM has been optimized and now supports __slots__ (#88)
- **BREAKING**: the `check_registration` method now always returns a list (#87)
- PostgreSQL versions older than 9.2 are no longer supported (#83)
- idle connections are now kept open for up to 10 minutes by default (#81)
- the methods `run`, `one` and `all` now support receiving query parameters as keyword arguments (#79)
- **BREAKING**: the methods `run`, `one` and `all` no longer pass extra arguments to `get_cursor` (#79 and #67)
- subtransactions are now supported (#78 and #90)
- **BREAKING**: single-column rows aren't unpacked anymore when the *back_as* argument is provided (#77)
- the cursor methods now also support the *back_as* argument (#77)
- a new row type and cursor subclass are now available, see `SimpleRowCursor` for details (#75)
- the ORM now supports non-default schemas (#74)
- connections now also have a `get_cursor` method (#73 and #82)
- the values accepted by the *back_as* argument can now be customized (#72)
- the `one` and `all` no longer fail when a row is made up of a single column named `values` (#71)

- any `InterfaceError` exception raised during an automatic rollback is now suppressed (#70)
- the *`get_cursor`* method has two new optional arguments: *autocommit* and *readonly* (#69)
- *`Postgres`* objects now have a `readonly` attribute (#69)
- the *url* argument is no longer required when creating a `Postgres` object (#68)

2.2.2 (Sep 12, 2018)

- the only dependency was changed from `psycopg2 >= 2.5.0` to `psycopg2-binary >= 2.7.5` (#64)
- the license was changed from CC0 to MIT (#62)

2.2.1 (Oct 10, 2015)

- a bug in the URL-to-DSN conversion function was fixed (#53)

2.2.0 (Sep 12, 2015)

- the ORM was modified to detect some schema changes (#43)

Python Module Index

p

Index

T

U