Postgres.py Documentation

Release 2.1.0

Gittip, LLC

December 18, 2013

Contents

The postgres Python library is a high-value abstraction over the psycopg2 database driver.

CHAPTER 1

Installation

`postgres` is available on GitHub and on PyPI:

```
$ pip install postgres
```

`postgres` requires `psycopg2` version 2.5 or higher.

We test against Python 2.6, 2.7, 3.2, and 3.3. We don’t yet have a testing matrix for different versions of `psycopg2` or PostgreSQL.

`postgres` is in the public domain.

Tutorial

Instantiate a `Postgres` object when your application starts:

```
>>> from postgres import Postgres
>>> db = Postgres("postgres://jrandom@localhost/test")
```

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*.

2.1 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)
```

Never build SQL strings out of user input!

Always pass user input as bind parameters!

2.2 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 3

The Postgres Object

class `postgres.Postgres`(*url*, *minconn=1*, *maxconn=10*, *cursor_factory=<class 'postgres.cursors.SimpleNamedTupleCursor'>*)

Interact with a PostgreSQL database.

Parameters

- **url** (*unicode*) – 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
- **cursor_factory** – Defaults to `SimpleNamedTupleCursor`

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("postgres://jrandom@localhost/test")
```

(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.)

When instantiated, this object creates a thread-safe connection pool, which opens `minconn` connections immediately, and up to `maxconn` according to demand. Everything this object provides runs through this connection pool.

`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 `back_as` or `cursor_factory` to `one`, `all`, and `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*, **a*, ***kw*)

Execute a query and discard any results.

Parameters

- **sql** (*string*) – the SQL statement to execute
- **parameters** (*dict or tuple*) – the bind parameters for the SQL statement
- **a** – passed through to `get_cursor`
- **kw** – passed through to `get_cursor`

Returns `None`

This is a convenience method. Here is what it does:

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

Use it like this:

```
>>> db.run("DROP TABLE IF EXISTS foo CASCADE")
>>> 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)")
```

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

Parameters

- **sql** (*string*) – the SQL statement to execute
- **parameters** (*dict or tuple*) – the bind parameters for the SQL statement
- **default** – the value to return if no results are found
- **back_as** (*type or string*) – the type of record to return
- **a** – passed through to `get_cursor`
- **kw** – passed through to `get_cursor`

Returns a single record or value or the value of the `default` argument

Raises `TooFew` or `TooMany`

This is a convenience method. Here is what it does:

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

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 you do need more control (maybe you have a custom cursor class), you can pass `cursor_factory` explicitly, and that will override any `back_as`:

```
>>> from postgres.cursors import SimpleTupleCursor
>>> db.one( "SELECT * FROM foo WHERE bar='buz'"
...       , back_as=dict
...       , cursor_factory=SimpleTupleCursor
...        )
('buz', 42)
```

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 will use `values` if it exists on the record, so it should work for both mappings and sequences.

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

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

Parameters

- **sql** (*string*) – 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
- **a** – passed through to `get_cursor`

- **kw** – passed through to `get_cursor`

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

This is a convenience method. Here is what it does:

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

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 you do need more control (maybe you have a custom cursor class), you can pass `cursor_factory` explicitly, and that will override any `back_as`:

```
>>> from postgres.cursors import SimpleTupleCursor
>>> db.all( "SELECT * FROM foo ORDER BY bar"
...       , back_as=dict
...       , cursor_factory=SimpleTupleCursor
...        )
[('bit', 537), ('buz', 42)]
```

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]
```

This works for record types that are mappings (anything with a `__len__` and a `values` method) as well those that are sequences:

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

get_cursor(**a*, ***kw*)

Return a `CursorContextManager` that uses our connection pool.

Parameters

- **a** – passed through to the `cursor` method of instances of the class returned by `make_Connection`
- **kw** – passed through to the `cursor` method of instances of the class returned by `make_Connection`

```
>>> 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)]
```

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, resets `autocommit` to off on the connection, 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.

`get_connection``()`

Return a `ConnectionContextManager` that uses our connection pool.

```
>>> 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 `NotRegistered`

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

`check_registration``(`*ModelSubclass*`)`

Check whether an ORM model is registered.

Parameters **ModelSubclass** – the `Model` subclass to check for

Returns the `typname` (a string) for which this model is registered, or a list of strings if it's registered for multiple types

Rettype string

Raises `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. If a `cursor_factory` keyword argument is also given, then any `back_as` is ignored and discarded. Valid values for `back_as` are `tuple`, `namedtuple`, `dict` (or the strings `tuple`, `namedtuple`, and `dict`), and `None`. If the value of `back_as` is `None`, then we'll use the default `cursor_factory` with which our parent `Postgres` instance was instantiated. If `back_as` is not `None`, then we'll specify a `cursor_factory` that will result in records of the designated type: `postgres.cursor.SimpleTupleCursor` for `tuple`, `postgres.cursor.SimpleNamedTupleCursor` for `namedtuple`, and `postgres.cursor.SimpleDictCursor` for `dict`.

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

`postgres.`**`make_DelegatingCaster`**(*postgres*)

Define a `CompositeCaster` subclass that delegates to `model_registry`.

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

Returns a `DelegatingCaster` class

The class we return will use the `model_registry` of the given `Postgres` instance to look up a `Model` subclass to use in mapping `psycopg2` return values to higher-order Python objects. Yeah, it's a little squirrelly. :-/

The Context Managers

class `postgres.context_managers.`**`CursorContextManager`**`(`*pool*, **a*, ***kw*`)`

Instantiated once per `get_cursor` call.

Parameters pool – a `psycopg2.pool.*ConnectionPool`

The return value of `CursorContextManager.__enter__` is a `psycopg2` cursor. Any positional and keyword arguments to our constructor are passed through to the cursor constructor.

When the block starts, a connection is checked out of the connection pool and `autocommit` is set to `False`. Then a cursor is constructed, and the `one` and `all` methods are scabbed on (this allows us to provide our simple API no matter the `cursor_factory`). The cursor is returned to the `with` statement. If the block raises an exception, the connection is rolled back. Otherwise, it's committed. In either case, the cursor is closed, `autocommit` is reset to `False` (just in case) and the connection is put back in the pool.

class `postgres.context_managers.`**`ConnectionContextManager`**`(`*pool*`)`

Instantiated once per `get_connection` call.

Parameters pool – a `psycopg2.pool.*ConnectionPool`

The return value of `ConnectionContextManager.__enter__` is a `postgres.Connection`. When the block starts, a `Connection` is checked out of the connection pool and `autocommit` is set to `False`. When the block ends, the `Connection` is rolled back before being put back in the pool.

CHAPTER 5

Simple Cursors

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

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( "postgres://jrandom@localhost/test"
...              , 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( "postgres://jrandom@localhost/test"
...              , cursor_factory=LoggingCursor
...               )
...
Traceback (most recent call last):
    ...
postgres.NotASimpleCursor: We can only work with subclasses of postgres.cursors.SimpleCursorBase
```

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

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

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

Execute a query and discard any results.

Note: See the documentation at postgres.Postgres.run.

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

Note: See the documentation at `postgres.Postgres.one`.

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

Note: See the documentation at `postgres.Postgres.all`.

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

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

class `postgres.cursors.`**`SimpleDictCursor`**(**args*, ***kwargs*)
A simple cursor that returns dicts.

`postgres.cursors.`**`isexception`**(*obj*)
Given an object, return a boolean indicating whether it is an instance or subclass of `Exception`.

CHAPTER 6

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 `psycopg2 CompositeCaster` 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.

6.1 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)

test=# SELECT foo.*::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.*::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.

6.2 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.*::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.*::foo FROM foo WHERE bar='blam'")
>>> foo.bar
'blam'
>>> [foo.bar for foo in db.all("SELECT foo.*::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.*::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.

6.3 The Model Base Class

class `postgres.orm.`**`Model`**(*record*)

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

Parameters **record** (*dict*) – The raw query result

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.

Python Module Index