# sqlpy Documentation

HUSSTECH

Nov 24, 2019

# Contents

With SQLpy you can work directly with advanced SQL from the comfort of your Python code. Write SQL in *.sql* files and Python in *.py* files, with all the correct highlighting, linting and maintainability that comes with it.

- Write and run the *exact* SQL code you want
- **Use advanced SQL techniques such as**
  - CTEs
  - subqueries
  - recursion
  - distinct on, partition over (), etc. . .
- Dynamically build SQL queries for different purposes
- Use the latest features available in your database system

# CHAPTER 1

## Party like it's ANSI 1999!

SQL has been around since the mid 1970's in RDMS systems as the bedrock of many critical systems and applications. SQL is easy to start with but is quickly perceived as complex when you go beyond `"SELECT * FROM table;"`. Especially in the age of web applications where the persistence layer (both relational and No-SQL) have been treated as simple stores of data, and are often behind abstraction to bring data in and out of your application. But when you need to do something a bit more custom with your data, you often find yourself reaching to SQL.

However there has not really been a simple and straightforward way to do this directly from the application code itself for large projects. Having SQL strings dotted all over source files does not help maintainability or readability.

The solution to using SQL directly from your application code is to... *use SQL directly from your application code!* Following from the original insight of YeSQL, and learning from anosql, SQLpy is the solution for working directly with SQL in Python projects.

Why SQLpy? Read more on background here: SQLpy Blog

CHAPTER 2

# Installation

```
$ pip install sqlpy
```

You'll also need a Database DBAPI driver. See *compatibility*.

# CHAPTER 3

# Quickstart

Full documentation can be found at readthedocs.

Getting started is simple! All you need is a SQL database running and accessible to you. Let's assume a PostgreSQL database for our example.

Assume we have a database table `hello` with the following data.

| id | message |
|---|---|
| 1 | hello |
| 2 | SQLpy |
| 3 | PostgreSQL! |

First install **sqlpy** and **psycopg2**

```
$ pip install sqlpy psycopg2
```

Create a *queries.sql* file in your project directory, containing the following. (The name of the SQL snippet is how to link the query to the Python code.)

```
-- name: test_select
-- selection from database
SELECT * FROM hello
```

Set up the application and run

```
from __future__ import print_function  # Python 2-3 compatibility
from sqlpy import Queries
import psycopg2

sql = Queries('queries.sql')


def connect_db():
```

(continues on next page)

(continued from previous page)

```
    return psycopg2.connect(dbname='postgres',
            user=<user>,
            password=<password>,
            host=<host>,
            port=<port>)


db = connect_db()

with db:
    with db.cursor() as cur:
        output = sql.TEST_SELECT(cur)

print(output)

db.close()
```

. . . prints

```
[(1, u'hello'), (2, u'SQLpy'), (3, u'PostgreSQL!')]
```

You can also pass variables to the query via format strings `%s` or pyformat strings `%(name)s` and an argument tuple or dictionary respectively.

```
-- name: select_by_id
SELECT * FROM hello
WHERE id = %s;

-- name: select_by_msg
SELECT * FROM hello
WHERE id = %(msg)s;
```

```
>> sql.SELECT_BY_ID(cur, (1,))

[(1, u'hello')]

>> kwargs = {'msg': 'SQLpy'}
>> sql.SELECT_BY_MSG(cur, kwargs)

[(2, u'SQLpy')]
```

CHAPTER 4

# Database Compatibility/Limitations

SQLpy was written as a lightweight helper around your already existing Python DB API 2.0 library, with no assumptions made about the underlying library of choice.

As long as you write valid SQL for *your* database system and Python DB API library, then you should have no problems.

> For example PostgreSQL implements the `RETURNING` clause, this may be called something else or not implemented in a different system. So if you are using a With RETURNING query, then make sure you have the correct SQL syntax for your system.

Other explicit compatibility points detailed below.

## 4.1 paramstyle

**The Python DB API specifies 5 types of parameter style**

- qmark: Question mark style, e.g. ... WHERE name=?
- numeric: Numeric, positional style, e.g. ... WHERE name=:1
- named: Named style, e.g. ... WHERE name=:name
- format: ANSI C printf format codes, e.g. ... WHERE name=%s
- pyformat: Python extended format codes, e.g. ... WHERE name=%(name)s

SQLpy supports all of the *positional* paramstyles, for all query types other than `BUILT`, since the SQL code is simply passed straight through to the DB API library.

As of version 0.2.0 SQLpy only supports `pyformat` as the named paramstyle for `BUILT` query types.

Below is a non-exhaustive, possibly incomplete, probably out of date list, of popular Python DB API libaries and their paramstyle support.

| paramstyle | library |
|---|---|
| format, pyformat | psycopg2 |
| format, pyformat | pg8000 |
| format, pyformat | mysqldb |
| format, pyformat | mysqlconnector |
| format, pyformat | pymssql |
| qmark | oursql |
| qmark | pyodbc |
| qmark | sqlite3 |
| numeric, named | cx_oracle |

## 4.2 quote_ident

**PostgreSQL/psycopg 2.7+ Only**

Due to SQL parameter escaping (see Bobby Tables), many DB API libraries won't allow you to correctly pass in variables to set `idendity` values in your query. These are things like column names in the SELECT, or ORDER BY clauses. The psycopg libary for PostgreSQL provides the `quote_ident()` function to solve this. To use it, pass a tuple of strings to your SQLpy function where the SQL contains a `{}` replacement field for each tuple item.

```
-- name: select_by_id
SELECT * FROM hello
WHERE {} = %s;
```

```
>> sql.SELECT_BY_ID(cur, identifiers=('id',), (1,))

[(1, u'hello')]
```

It is also possible to define identifiers in multiple parts of the query by passing a named identifier group(s) in a dictionary. This allows multiple identifiers to be contained within the same format token slot.

```
-- name: select_orderd_group
SELECT * FROM hello
ORDER BY {order_group};
```

```
>> sql.SELECT_BY_ID(cur, identifiers={'order_group': ('col_1', 'col_2', 'col_3')}
```

CHAPTER 5

# Tests

Tests are provided through the excellent pytest, and CI via Travis CI. SQLpy is tested against a real PostgreSQL database loaded with the pagila dataset.

**To run the test suite locally without a database, simply run `make test` from the root of the project. To run with a database (gi**

- load the pagila data by copying the commands in the `before_script` block in the `.travis.yml` file.
- modify the `test_sqlpy.py` file to enable running of the skipped test blocks and add any credentials to the `db_cur()` fixture.
- run with `make test` as before

CHAPTER 6

# Development

Team work makes the dream work!

We welcome contributions! You can open an Issue to report a bug or ask a question. If you would like to submit changes for review, please follow these steps:

1. Fork the repository
2. Make your changes
3. Install the requirements in `dev-requirements.txt`
4. Submit a pull request after running `make check` (ensure it does not error!)

CHAPTER 7

# License

MIT

# CHAPTER 8

# How it works

Python PEP249 is the current standard for implementing APIs between Python and relational databases. From the PEP text

```
This API has been defined to encourage similarity between
the Python modules that are used to access databases.
```

The standard defines a number of common objects, methods and operations that any library must implement to be compliant. A slew of Python libraries exist for most if not all relational database systems, all adhering to the same specification. All of them boiling down to sending SQL commands to and returning their results from the database server to the Python runtime.

SQLpy was written as a lightweight helper around your already existing Python DB API 2.0 library, with no assumptions made about the underlying library of choice. Essentially only wrapping the `cur.execute()`, `cur.executemany()`, `cur.fetchone()`, `cur.fetchmany()` and `cur.fetchall()` methods. Connection and Cursor object creation preferences are left up to you.

SQLpy leverages the powerful `functools` module within Python and creates prepared functions reading SQL statements by reading from a queries file(s) within your project. The SQL statements have a special form that SQLpy recognises and this determines how the statement is prepared and eventually executed. Depending on if it is a simple select, a delete or a different type of statement.

For the following sections, let's assume we have a database table `hello` with the following data.

| id | message |
|---|---|
| 1 | hello |
| 2 | SQLpy |
| 3 | PostgreSQL! |

## 8.1 Initialising the Queries object

The first thing to do is to intialise the `sqlpy.Queries` object with your SQL queries.

```
sql = sqlpy.Queries(filepath, strict_parse=False, uppercase_name=True))
```

**Parameters**

- filepath (`list` of `str` or `str`): List of file locations containing the SQL statements or a single filepath to the queries file.
- strict_parse (`bool`, optional): Weather to strictly enforce matching the expected and supplied parameters to a SQL statement function.
- uppercase_name (`bool`, optional): Weather to cast the names of the SQL statement functions to uppercase.

## 8.2 Executing the functions

To execute a SQL statement and get results, just call the method by name on the `sqlpy.Queries` object. Note: The name is cast to uppercase (if this causes an uproar it can be made optional in a patch release).

```
sql = sqlpy.Queries('queries.sql')
....
results = sql.SQL_STATEMENT(cur, args=dict()|tuple(), n=None, identifiers=None, log_
↪query_params=LOG_QUERY_PARAMS)
```

**Parameters**

- `cur`: A Cursor object. Can be any cursor type you want.
- args (`tuple`) or (`dict`): A sequence of positional parameters in the query. Or a dictonary with named parameters in the query.
- n (`int`): How many results to fetch back. By default it is set to `None` which and the underlying cursor performs a `fetchall()` and all the results are returned. For `n=1` a `fetchone()` is performed and for `n>1` a `fetchmany(n)` is perfromed.
- identifiers (`tuple`): A sequence of positional strings to use to format the query before execution. Used with *identity strings*. Default is `None`.
- log_query_params (`boolean`): A flag to enable or disable logging out of the parameters sent to the query. Some data is sensitive and should not be visible in log entries. Default is `sqlpy.config.LOG_QUERY_PARAMS` which is `True`.

## 8.3 Query types

The type of query executed is determined by a token SQLpy searches for at the end of the `-- name:` special comment string in the SQL file. This can be `!`, `<!>`, `$`, `@` or not present.

Comments are detected and added to the `__doc__` attribute of the returned function.

```
-- name: sql_statement
-- a regular select statement
SELECT * FROM hello
WHERE id = %s;

-- name: insert_statement!
-- an insert statement
INSERT INTO hello (message)
```

(continues on next page)

(continued from previous page)

```
VALUES (%s);

-- name: insert_statement2<!>
-- an insert statement with return
INSERT INTO hello (message)
VALUES (%s)
RETURNING id;

-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE id = %(id_low)s
OR id = %(id_high)s;

-- name: function_name@
-- a procedure/function being called
function_name
```

**SELECT** There is no token at the end of the name string

**INSERT/UPDATE/DELETE** There is a `!` token at the end of the name string

**With RETURNING** There is a `<!>` token at the end of the name string

**Built SQL** There is a `$` token at the end of the name string Can only use `pyformat` named parameters

**Function Call** There is a `@` token at the end of the name string

### 8.3.1 Built SQL

In your application you will likely want to take different paths retrieving data depending on the current values or the variables you have available. One example could be looking up values from a table, using a varying number of search parameters. Writing a separate query for each case would be repetitive, and difficult as you need to know ahead of time the possible combinations.

**SQLpy offers the functionality to dynamically build SQL queries based on the query parameters passed to the prepared functio**

- An internal lookup map is created when the query is being parsed.
- Each line of the query is collected and inserted into a dictionary with information on its position (line number) in the overall query, and the query line itself.
- The key for each entry is the parameter contained within that line.
- Any lines with no parameter (most of the stuff before there `WHERE` clause), are collected under the same key.

When executed the query is reassembled in the correct line order, and lines containing parameters that have also been passed to the function as `kwargs` are included. (Note, not to be confused with the `**kwargs` convention in Python. Here we mean key-word arguments that are for the query.) Then the final SQL is sent to the database driver as normal.

Example.

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'id_low': 1}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

Would execute the SQL.

```
SELECT * FROM hello
WHERE id = 1;
```

**This design leads to some minor restrictions in how to write your queries that are explained below.**

**Single line per clause** It's best to lay out queries with a newline for each filter clause you use. This is to ensure that the resulting SQL query is built with the correct lines in place, and not with extra parameters for which there are no matching function inputs for.

```
-- name: built_sql_statement$
-- a built up sql statement
-- newline for each parameter
SELECT * FROM hello
WHERE id = %(id_low)s
OR id = %(id_high)s
AND message = %(msg)s;
```

**Careful with `WHERE`** SQL queries are asymmetrical, you always start the filtering clauses with `WHERE`, but after that you use `AND` or `OR`. This means that if the parameter contained within the `WHERE` clause is not passed to the function, the query will be built without it, and that is invalid SQL. To solve this, you can use `WHERE 1=1`. This always evaluates to `True` and is effectively a pass-through value, always ensuring the `WHERE` clause is present in your queries.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND id = %(id_low)s
OR id = %(id_high)s
AND message = %(msg)s;
```

**Multiple parameters per line** Sometimes you can not avoid multiple parameters that must be grouped together, such as in compound `AND-OR` clauses. Ensure you supply all the necessary argument to the function to get the correct output.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND (id = %(id_low)s OR id = %(id_high)s);
```

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'id_low': 1, 'id_high': 3}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (id = 1 OR id = 3);
```

**Missing parameters** In oder to maintain valid SQL output SQLpy will replace missing parameters with `None`, and this usually transforms to `NULL` when passed to the database. In this next example the result will still be correct, as the `id` column would not contain any `NULL` values, so the `OR` clause has no effect.

**Note:** PostgrSQL does not correctly evaluate the syntax `column = NULL`, instead you should use `column is NULL` or `column is not NULL`. When searching columns that could contain `NULL` values, it's best to use the `ANY()` operator, where an array of values to check is passed to it. It behaves like `IN ()`, and it correctly handles `NULL` values. The added benefit is that you can test for multiple conditions in clauses too, so it's a useful pattern regardless. Check out PostgreSQL Arrays for more info.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND (id = ANY(%(id_low)s) OR id = ANY(%(id_high)s));
```

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'id_low': [1]}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (id = ANY('{1}') OR id = ANY('{NULL}'));
```

**Switching off parameters** The philosophical discussion on the merits/lack of on the use of `NULL` in SQL systems is well known, but the value (or is it a Type?) is used everywhere. This just means you need to take this into account when writing your data retrieval queries with `NULL` values.

Following from the example above, say you have a compound `OR` clause on a column that can have `NULL` values. At certain times, you may not supply all the values required to the function, so `None` is substituted in its place. This is a problem because you don't want the case where extra results are returned that match the other side of the `OR`.

We have new data...

| id | message | message2 |
|---|---|---|
| 1 | hello | there |
| 2 | SQLpy | NULL |
| 3 | PostgreSQL! | rules! |
| 4 | hello | friend |

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND (message = ANY(%(msg)s) OR message2 = ANY(%(msg2)s));
```

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'message': ['hello']}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (message = ANY('{"hello"}') OR message2 = ANY('{NULL}'));
```

returns...

| id | message | message2 |
|---|---|---|
| 1 | hello | there |
| 2 | SQLpy | NULL |
| 4 | hello | friend |

We don't want row 2 in this case. To solve this, you can use a little SQL syntax gymnastics to write the `OR` clause in such a way that `NULL` does not bring in incorrect results.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND ((FALSE OR message = ANY(%(msg)s)) OR (FALSE OR message2 = ANY(%(msg2)s)));
```

The clauses are enclosed in a second set of parenthesis in the form `(FALSE OR column=%(name)s)`. If the parameter is replaced with a `NULL` then this "switches-off" that entire check, because `SELECT FALSE OR NULL --> NULL`. So `(NULL OR (FALSE OR column=VALUE))` only evaluates the right hand side of the statement. This would produce the correct output.

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (FALSE OR message = ANY('{"hello"}' OR (FALSE OR message2 = ANY('{NULL}')));
-- this reduces to
-- AND (message = ANY('{"hello"}' OR (NULL));
-- and again to
-- AND (message = ANY('{"hello"}'));
```

returns...

| id | message | message2 |
|---|---|---|
| 1 | hello | there |
| 4 | hello | friend |

**Warning: only tested in PostgreSQL**

**NULL with care** As you can see this is very tricky and also very database specific. It's probably best to avoid writing such queries in the first place, and taking a second look at your data model could also reveal a better design.

But you could still come across and need this pattern. However now that the problem is exposed purely as a SQL problem, you can now seek help in SQL Q&A forums in which there is about 50 years (and counting) of SQL language experience!

**Strict parse** If you don't like to live dangerously, then you can enable a safety mechanism around Built queries. If you initialise the `sqlpy.Queries` object as `sqlpy.Queries(..., strict_parse=True)`, a `sqlpy.exceptions.SQLArgumentException` is raised when a named argument is supplied which does not match a SQL clause.

**Built queries are limited to only SELECT queries at the moment.** There will definitely be some interesting edge cases arising from the layout and use of Built queries! If you see anything odd and think it should be handled, then do open an issue on GitHub.

### 8.3.2 Identity strings

**PostgreSQL/psycopg 2.7+ Only**

Due to SQL parameter escaping (see Bobby Tables), many DB API libraries won't allow you to correctly pass in variables to set `identity` values in your query. These are things like column names in the SELECT, or ORDER BY clauses. The psycopg libary for PostgreSQL provides the `quote_ident()` function to solve this. To use it, pass a tuple of strings to your SQLpy function where the SQL contains a `{}` replacement field for each tuple item.

```
-- name: select_by_id
SELECT * FROM hello
WHERE {} = %s;
```

```
>> sql.SELECT_BY_ID(cur, identifiers=('id',), (1,))

[(1, u'hello')]
```

It is also possible to define identifiers in multiple parts of the query by passing a named identifier group(s) in a dictionary. This allows multiple identifiers to be contained within the same format token slot.

```
-- name: select_orderd_group
SELECT * FROM hello
ORDER BY {order_group};
```

```
>> sql.SELECT_BY_ID(cur, identifiers={'order_group': ('col_1', 'col_2', 'col_3')}
```

# CHAPTER 9

# Showing Off

## 9.1 Sudoku

SQL is more than just a declarative data retrieval language, it's a fully Turing complete language in its own right. So it should be able to compute anything that a more typical application language could... although it may not be the most syntactically concise bit of code out there. SQL does shine in performing set operations, that is evaluating functions over groups (sets) of data (relations/tables) all at once.

Sets of data... numbers... grids. Did someone say Sudoku solver? A 9x9 Sudoku solving SQL snippet was recently added to the postgresql wiki page. Using a recursive window function, it implements a brute-force backtracking algorithm to solve the puzzle. Taking up only 32 lines, it could be less as SQL does not depend on whitespace heavily, it solves the example puzzle in under 10 seconds on a mid-range quad-core laptop from 2014 running postgres 9.6.

Imagine trying to program this to be done in SQL in a similar way but via an ORM!? With SQLpy it wold be easy. (let's gloss over the amount of energy that went into writing the SQL in the first place!)

```
-- name: sudoku_solver
-- a sudoku solver
-- in SQL why not
-- note the query param %s 3 lines below
WITH recursive board(b, p) AS (
  -- sudoku board expressed in column-major order, so substr() can be used to fetch a␣
↪column
  VALUES (%s::CHAR(81), 0)
  UNION ALL SELECT b, p FROM (
    -- generate boards:
    SELECT overlay(b placing new_char FROM strpos(b, '_') FOR 1)::CHAR(81), strpos(b,
↪'_'), new_char
    FROM board, (SELECT chr(n+ascii('b')) FROM generate_series(0, 8) n) new_char_
↪table(new_char)
    WHERE strpos(b, '_') > 0
  ) r(b, p, new_char) WHERE
    -- make sure the new_char doesn't appear twice in its column
    -- (there are two checks because we are excluding p itself):
```

(continues on next page)

(continued from previous page)

```
    strpos(substr(b, 1+(p-1)/9*9, (p-1)%9), new_char) = 0 AND
    strpos(substr(b, p+1, 8-(p-1)%9), new_char) = 0 AND
    -- make sure the new_char doesn't appear twice in its row:
    new_char NOT IN (SELECT substr(b, 1+i*9+(p-1)%9, 1)
                     FROM generate_series(0, 8) i
                     WHERE p <> 1+i*9+(p-1)%9) AND
    -- make sure the new_char doesn't appear twice in its 3x3 block:
    new_char NOT IN (SELECT substr(b, 1+i%3+i/3*9+(p-1)/27*27+(p-1)%9/3*3, 1)
                     FROM generate_series(0, 8) i
                     WHERE p <> 1+i%3+i/3*9+(p-1)/27*27+(p-1)%9/3*3)
) SELECT
    -- the following subquery is used to represent the board in a '\n' separated␣
↪human-readable form:
    ( SELECT string_agg((
        SELECT string_agg(chr(ascii('1')+ascii(substr(b, 1+y+x*9, 1))-ascii('b')), '
↪') r
        FROM generate_series(0, 8) x), E'\n')
      FROM generate_series(0, 8) y
    ) human_readable,
    b board,
    p depth,
    (SELECT COUNT(*) FROM board) steps
  FROM board WHERE strpos(b,'_') = 0 LIMIT 5000;
```

```
board = '__g_cd__bf_____j____c__e___c__i___jd__b__h___id____e__g__b__f_e_f____g____j_
↪h__c_'
results, _ = sql.SUDOKU_SOLVER(cur, (board,), n=1)

print(results[0])
```

```
457298631
819763254
632415879
975832146
261549387
384671925
798124563
543986712
126357498
```

CHAPTER 10

# Changelog

## 10.1 Current release

### 10.1.1 0.3.4 0.3.5

**Bugfix**

- use a more broad import for `quote_ident` function, to allow instrumentation libraries to correctly use the patched version #24

## 10.2 Previous releases

### 10.2.1 0.3.3

**Bugfix**

- identifier formatting in built sql not happening #21

### 10.2.2 0.3.2

**Major Updates**

- Support sets of identifier groups and using named parameters #20

### 10.2.3 0.3.1

**Breaking Changes**

- revert returning cursor as part of query execution #17

**Major Updates**

- expose logging query args configuration in Queries object initialisation #18

### 10.2.4 0.3.0

**Breaking Changes**

- updated API design for the query function #15

**Major Updates**

- add separated `cur.fetchone` cursor method #15
- add `cur.callproc` #12
- transparently switch to using the more efficient `execute_values` with Psycopg2 #16
- updated docs

**Minor Fixes**

- fix type in `load_queries` #14
- initial strip of whitespace on input query files
- ensure multi-file queries joined correctly with double newline #10
- make uppercase of function name optional
- moved logging helper back into main module #8

### 10.2.5 0.2.0

**Major update**

- Remove coupling to PostgreSQL and psycopg2 by conditionally importing from psycopg2
- Changed the logger to add `NullHandler()` by default
- Changed `fetchone()` to `fetchmany()`
- Improved exception handling with better Exceptions
- A lot of internal code refactoring
- Documentation, a lot of Documentation

### 10.2.6 0.1.0

First pypi upload of the project. Lacking good documentation and behavior was tied to PostgreSQL and psycopg2.

# CHAPTER 11

# How it works

Python PEP249 is the current standard for implementing APIs between Python and relational databases. From the PEP text

```
This API has been defined to encourage similarity between
the Python modules that are used to access databases.
```

The standard defines a number of common objects, methods and operations that any library must implement to be compliant. A slew of Python libraries exist for most if not all relational database systems, all adhering to the same specification. All of them boiling down to sending SQL commands to and returning their results from the database server to the Python runtime.

SQLpy was written as a lightweight helper around your already existing Python DB API 2.0 library, with no assumptions made about the underlying library of choice. Essentially only wrapping the `cur.execute()`, `cur.executemany()`, `cur.fetchone()`, `cur.fetchmany()` and `cur.fetchall()` methods. Connection and Cursor object creation preferences are left up to you.

SQLpy leverages the powerful `functools` module within Python and creates prepared functions reading SQL statements by reading from a queries file(s) within your project. The SQL statements have a special form that SQLpy recognises and this determines how the statement is prepared and eventually executed. Depending on if it is a simple select, a delete or a different type of statement.

For the following sections, let's assume we have a database table `hello` with the following data.

| id | message |
|---|---|
| 1 | hello |
| 2 | SQLpy |
| 3 | PostgreSQL! |

## 11.1 Initialising the Queries object

The first thing to do is to intialise the `sqlpy.Queries` object with your SQL queries.

```
sql = sqlpy.Queries(filepath, strict_parse=False, uppercase_name=True))
```

**Parameters**

- filepath (`list` of `str` or `str`): List of file locations containing the SQL statements or a single filepath to the queries file.
- strict_parse (`bool`, optional): Weather to strictly enforce matching the expected and supplied parameters to a SQL statement function.
- uppercase_name (`bool`, optional): Weather to cast the names of the SQL statement functions to uppercase.

## 11.2 Executing the functions

To execute a SQL statement and get results, just call the method by name on the `sqlpy.Queries` object. Note: The name is cast to uppercase (if this causes an uproar it can be made optional in a patch release).

```
sql = sqlpy.Queries('queries.sql')
....
results = sql.SQL_STATEMENT(cur, args=dict()|tuple(), n=None, identifiers=None, log_
↪query_params=LOG_QUERY_PARAMS)
```

**Parameters**

- `cur`: A Cursor object. Can be any cursor type you want.
- args (`tuple`) or (`dict`): A sequence of positional parameters in the query. Or a dictonary with named parameters in the query.
- n (`int`): How many results to fetch back. By default it is set to `None` which and the underlying cursor performs a `fetchall()` and all the results are returned. For `n=1` a `fetchone()` is performed and for `n>1` a `fetchmany(n)` is perfromed.
- identifiers (`tuple`): A sequence of positional strings to use to format the query before execution. Used with *identity strings*. Default is `None`.
- log_query_params (`boolean`): A flag to enable or disable logging out of the parameters sent to the query. Some data is sensitive and should not be visible in log entries. Default is `sqlpy.config.LOG_QUERY_PARAMS` which is `True`.

## 11.3 Query types

The type of query executed is determined by a token SQLpy searches for at the end of the `-- name:` special comment string in the SQL file. This can be `!`, `<!>`, `$`, `@` or not present.

Comments are detected and added to the `__doc__` attribute of the returned function.

```
-- name: sql_statement
-- a regular select statement
SELECT * FROM hello
WHERE id = %s;

-- name: insert_statement!
-- an insert statement
INSERT INTO hello (message)
```

(continues on next page)

(continued from previous page)

```
VALUES (%s);

-- name: insert_statement2<!>
-- an insert statement with return
INSERT INTO hello (message)
VALUES (%s)
RETURNING id;

-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE id = %(id_low)s
OR id = %(id_high)s;

-- name: function_name@
-- a procedure/function being called
function_name
```

**SELECT** There is no token at the end of the name string

**INSERT/UPDATE/DELETE** There is a `!` token at the end of the name string

**With RETURNING** There is a `<!>` token at the end of the name string

**Built SQL** There is a `$` token at the end of the name string Can only use `pyformat` named parameters

**Function Call** There is a `@` token at the end of the name string

### 11.3.1 Built SQL

In your application you will likely want to take different paths retrieving data depending on the current values or the variables you have available. One example could be looking up values from a table, using a varying number of search parameters. Writing a separate query for each case would be repetitive, and difficult as you need to know ahead of time the possible combinations.

**SQLpy offers the functionality to dynamically build SQL queries based on the query parameters passed to the prepared functio**

- An internal lookup map is created when the query is being parsed.
- Each line of the query is collected and inserted into a dictionary with information on its position (line number) in the overall query, and the query line itself.
- The key for each entry is the parameter contained within that line.
- Any lines with no parameter (most of the stuff before there `WHERE` clause), are collected under the same key.

When executed the query is reassembled in the correct line order, and lines containing parameters that have also been passed to the function as `kwargs` are included. (Note, not to be confused with the `**kwargs` convention in Python. Here we mean key-word arguments that are for the query.) Then the final SQL is sent to the database driver as normal.

Example.

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'id_low': 1}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

Would execute the SQL.

```
SELECT * FROM hello
WHERE id = 1;
```

**This design leads to some minor restrictions in how to write your queries that are explained below.**

**Single line per clause** It's best to lay out queries with a newline for each filter clause you use. This is to ensure that the resulting SQL query is built with the correct lines in place, and not with extra parameters for which there are no matching function inputs for.

```
-- name: built_sql_statement$
-- a built up sql statement
-- newline for each parameter
SELECT * FROM hello
WHERE id = %(id_low)s
OR id = %(id_high)s
AND message = %(msg)s;
```

**Careful with `WHERE`** SQL queries are asymmetrical, you always start the filtering clauses with `WHERE`, but after that you use `AND` or `OR`. This means that if the parameter contained within the `WHERE` clause is not passed to the function, the query will be built without it, and that is invalid SQL. To solve this, you can use `WHERE 1=1`. This always evaluates to `True` and is effectively a pass-through value, always ensuring the `WHERE` clause is present in your queries.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND id = %(id_low)s
OR id = %(id_high)s
AND message = %(msg)s;
```

**Multiple parameters per line** Sometimes you can not avoid multiple parameters that must be grouped together, such as in compound `AND-OR` clauses. Ensure you supply all the necessary argument to the function to get the correct output.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND (id = %(id_low)s OR id = %(id_high)s);
```

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'id_low': 1, 'id_high': 3}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (id = 1 OR id = 3);
```

**Missing parameters** In oder to maintain valid SQL output SQLpy will replace missing parameters with `None`, and this usually transforms to `NULL` when passed to the database. In this next example the result will still be correct, as the `id` column would not contain any `NULL` values, so the `OR` clause has no effect.

**Note:** PostgrSQL does not correctly evaluate the syntax `column = NULL`, instead you should use `column is NULL` or `column is not NULL`. When searching columns that could contain `NULL` values, it's best to use the `ANY()` operator, where an array of values to check is passed to it. It behaves like `IN ()`, and it correctly handles `NULL` values. The added benefit is that you can test for multiple conditions in clauses too, so it's a useful pattern regardless. Check out PostgreSQL Arrays for more info.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND (id = ANY(%(id_low)s) OR id = ANY(%(id_high)s));
```

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'id_low': [1]}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (id = ANY('{1}') OR id = ANY('{NULL}'));
```

**Switching off parameters** The philosophical discussion on the merits/lack of on the use of `NULL` in SQL systems is well known, but the value (or is it a Type?) is used everywhere. This just means you need to take this into account when writing your data retrieval queries with `NULL` values.

Following from the example above, say you have a compound `OR` clause on a column that can have `NULL` values. At certain times, you may not supply all the values required to the function, so `None` is substituted in its place. This is a problem because you don't want the case where extra results are returned that match the other side of the `OR`.

We have new data...

| id | message | message2 |
|---|---|---|
| 1 | hello | there |
| 2 | SQLpy | NULL |
| 3 | PostgreSQL! | rules! |
| 4 | hello | friend |

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND (message = ANY(%(msg)s) OR message2 = ANY(%(msg2)s));
```

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'message': ['hello']}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (message = ANY('{"hello"}') OR message2 = ANY('{NULL}'));
```

returns...

| id | message | message2 |
|---|---|---|
| 1 | hello | there |
| 2 | SQLpy | NULL |
| 4 | hello | friend |

We don't want row 2 in this case. To solve this, you can use a little SQL syntax gymnastics to write the `OR` clause in such a way that `NULL` does not bring in incorrect results.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND ((FALSE OR message = ANY(%(msg)s)) OR (FALSE OR message2 = ANY(%(msg2)s)));
```

The clauses are enclosed in a second set of parenthesis in the form `(FALSE OR column=%(name)s)`. If the parameter is replaced with a `NULL` then this "switches-off" that entire check, because `SELECT FALSE OR NULL --> NULL`. So `(NULL OR (FALSE OR column=VALUE))` only evaluates the right hand side of the statement. This would produce the correct output.

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (FALSE OR message = ANY('{"hello"}' OR (FALSE OR message2 = ANY('{NULL}')));
-- this reduces to
-- AND (message = ANY('{"hello"}' OR (NULL));
-- and again to
-- AND (message = ANY('{"hello"}'));
```

returns...

| id | message | message2 |
|---|---|---|
| 1 | hello | there |
| 4 | hello | friend |

**Warning: only tested in PostgreSQL**

**NULL with care** As you can see this is very tricky and also very database specific. It's probably best to avoid writing such queries in the first place, and taking a second look at your data model could also reveal a better design.

But you could still come across and need this pattern. However now that the problem is exposed purely as a SQL problem, you can now seek help in SQL Q&A forums in which there is about 50 years (and counting) of SQL language experience!

**Strict parse** If you don't like to live dangerously, then you can enable a safety mechanism around Built queries. If you initialise the `sqlpy.Queries` object as `sqlpy.Queries(..., strict_parse=True)`, a `sqlpy.exceptions.SQLArgumentException` is raised when a named argument is supplied which does not match a SQL clause.

**Built queries are limited to only SELECT queries at the moment.** There will definitely be some interesting edge cases arising from the layout and use of Built queries! If you see anything odd and think it should be handled, then do open an issue on GitHub.

### 11.3.2 Identity strings

**PostgreSQL/psycopg 2.7+ Only**

Due to SQL parameter escaping (see Bobby Tables), many DB API libraries won't allow you to correctly pass in variables to set `identity` values in your query. These are things like column names in the SELECT, or ORDER BY clauses. The psycopg libary for PostgreSQL provides the `quote_ident()` function to solve this. To use it, pass a tuple of strings to your SQLpy function where the SQL contains a `{}` replacement field for each tuple item.

```
-- name: select_by_id
SELECT * FROM hello
WHERE {} = %s;
```

```
>> sql.SELECT_BY_ID(cur, identifiers=('id',), (1,))

[(1, u'hello')]
```

It is also possible to define identifiers in multiple parts of the query by passing a named identifier group(s) in a dictionary. This allows multiple identifiers to be contained within the same format token slot.

```
-- name: select_orderd_group
SELECT * FROM hello
ORDER BY {order_group};
```

```
>> sql.SELECT_BY_ID(cur, identifiers={'order_group': ('col_1', 'col_2', 'col_3')}
```

#### Showing Off

# CHAPTER 12

## Sudoku

SQL is more than just a declarative data retrieval language, it's a fully Turing complete language in its own right. So it should be able to compute anything that a more typical application language could... although it may not be the most syntactically concise bit of code out there. SQL does shine in performing set operations, that is evaluating functions over groups (sets) of data (relations/tables) all at once.

Sets of data... numbers... grids. Did someone say Sudoku solver? A 9x9 Sudoku solving SQL snippet was recently added to the postgresql wiki page. Using a recursive window function, it implements a brute-force backtracking algorithm to solve the puzzle. Taking up only 32 lines, it could be less as SQL does not depend on whitespace heavily, it solves the example puzzle in under 10 seconds on a mid-range quad-core laptop from 2014 running postgres 9.6.

Imagine trying to program this to be done in SQL in a similar way but via an ORM!? With SQLpy it wold be easy. (let's gloss over the amount of energy that went into writing the SQL in the first place!)

```
-- name: sudoku_solver
-- a sudoku solver
-- in SQL why not
-- note the query param %s 3 lines below
WITH recursive board(b, p) AS (
  -- sudoku board expressed in column-major order, so substr() can be used to fetch a␣
↪column
  VALUES (%s::CHAR(81), 0)
  UNION ALL SELECT b, p FROM (
    -- generate boards:
    SELECT overlay(b placing new_char FROM strpos(b, '_') FOR 1)::CHAR(81), strpos(b,
↪'_'), new_char
    FROM board, (SELECT chr(n+ascii('b')) FROM generate_series(0, 8) n) new_char_
↪table(new_char)
    WHERE strpos(b, '_') > 0
  ) r(b, p, new_char) WHERE
    -- make sure the new_char doesn't appear twice in its column
    -- (there are two checks because we are excluding p itself):
    strpos(substr(b, 1+(p-1)/9*9, (p-1)%9), new_char) = 0 AND
    strpos(substr(b, p+1, 8-(p-1)%9), new_char) = 0 AND
    -- make sure the new_char doesn't appear twice in its row:
```

(continues on next page)

(continued from previous page)

```
    new_char NOT IN (SELECT substr(b, 1+i*9+(p-1)%9, 1)
                     FROM generate_series(0, 8) i
                     WHERE p <> 1+i*9+(p-1)%9) AND
    -- make sure the new_char doesn't appear twice in its 3x3 block:
    new_char NOT IN (SELECT substr(b, 1+i%3+i/3*9+(p-1)/27*27+(p-1)%9/3*3, 1)
                     FROM generate_series(0, 8) i
                     WHERE p <> 1+i%3+i/3*9+(p-1)/27*27+(p-1)%9/3*3)
) SELECT
    -- the following subquery is used to represent the board in a '\n' separated␣
↪human-readable form:
    ( SELECT string_agg((
        SELECT string_agg(chr(ascii('1')+ascii(substr(b, 1+y+x*9, 1))-ascii('b')), '
↪') r
        FROM generate_series(0, 8) x), E'\n')
      FROM generate_series(0, 8) y
    ) human_readable,
    b board,
    p depth,
    (SELECT COUNT(*) FROM board) steps
  FROM board WHERE strpos(b,'_') = 0 LIMIT 5000;
```

```
board = '__g_cd__bf_____j____c__e___c__i___jd__b__h___id____e__g__b__f_e_f____g____j_
↪h__c_'
results, _ = sql.SUDOKU_SOLVER(cur, (board,), n=1)

print(results[0])
```

```
457298631
819763254
632415879
975832146
261549387
384671925
798124563
543986712
126357498
```

CHAPTER 13

# Source Code Documentation

For advanced details on the internals of SQLpy, or if you're looking for information on a specific function.

## 13.1 sqlpy package

### 13.1.1 Submodules

### 13.1.2 sqlpy.config module

### 13.1.3 sqlpy.exceptions module

### 13.1.4 sqlpy.sqlpy module

### 13.1.5 Module contents

## 13.2 Module index

- modindex
- genindex

# CHAPTER 14

# SQLpy - it's just SQL

With SQLpy you can work directly with advanced SQL from the comfort of your Python code. Write SQL in *.sql* files and Python in *.py* files, with all the correct highlighting, linting and maintainability that comes with it.

- Write and run the *exact* SQL code you want
- **Use advanced SQL techniques such as**
  - CTEs
  - subqueries
  - recursion
  - distinct on, partition over (), etc...
- Dynamically build SQL queries for different purposes
- Use the latest features available in your database system

## 14.1 Party like it's ANSI 1999!

SQL has been around since the mid 1970's in RDMS systems as the bedrock of many critical systems and applications. SQL is easy to start with but is quickly perceived as complex when you go beyond `"SELECT * FROM table;"`. Especially in the age of web applications where the persistence layer (both relational and No-SQL) have been treated as simple stores of data, and are often behind abstraction to bring data in and out of your application. But when you need to do something a bit more custom with your data, you often find yourself reaching to SQL.

However there has not really been a simple and straightforward way to do this directly from the application code itself for large projects. Having SQL strings dotted all over source files does not help maintainability or readability.

The solution to using SQL directly from your application code is to... *use SQL directly from your application code!* Following from the original insight of YeSQL, and learning from anosql, SQLpy is the solution for working directly with SQL in Python projects.

Why SQLpy? Read more on background here: SQLpy Blog

## 14.2 Installation

```
$ pip install sqlpy
```

You'll also need a Database DBAPI driver. See *compatibility*.

## 14.3 Quickstart

Full documentation can be found at readthedocs.

Getting started is simple! All you need is a SQL database running and accessible to you. Let's assume a PostgreSQL database for our example.

Assume we have a database table `hello` with the following data.

| id | message |
|---|---|
| 1 | hello |
| 2 | SQLpy |
| 3 | PostgreSQL! |

First install **sqlpy** and **psycopg2**

```
$ pip install sqlpy psycopg2
```

Create a *queries.sql* file in your project directory, containing the following. (The name of the SQL snippet is how to link the query to the Python code.)

```
-- name: test_select
-- selection from database
SELECT * FROM hello
```

Set up the application and run

```
from __future__ import print_function  # Python 2-3 compatibility
from sqlpy import Queries
import psycopg2

sql = Queries('queries.sql')


def connect_db():
    return psycopg2.connect(dbname='postgres',
            user=<user>,
            password=<password>,
            host=<host>,
            port=<port>)


db = connect_db()

with db:
    with db.cursor() as cur:
        output = sql.TEST_SELECT(cur)
```

(continues on next page)

(continued from previous page)

```
print(output)

db.close()
```

. . . prints

```
[(1, u'hello'), (2, u'SQLpy'), (3, u'PostgreSQL!')]
```

You can also pass variables to the query via format strings `%s` or pyformat strings `%(name)s` and an argument tuple or dictionary respectively.

```
-- name: select_by_id
SELECT * FROM hello
WHERE id = %s;

-- name: select_by_msg
SELECT * FROM hello
WHERE id = %(msg)s;
```

```
>> sql.SELECT_BY_ID(cur, (1,))

[(1, u'hello')]

>> kwargs = {'msg': 'SQLpy'}
>> sql.SELECT_BY_MSG(cur, kwargs)

[(2, u'SQLpy')]
```

## 14.4 Database Compatibility/Limitations

SQLpy was written as a lightweight helper around your already existing Python DB API 2.0 library, with no assumptions made about the underlying library of choice.

As long as you write valid SQL for *your* database system and Python DB API library, then you should have no problems.

> For example PostgreSQL implements the `RETURNING` clause, this may be called something else or not implemented in a different system. So if you are using a With RETURNING query, then make sure you have the correct SQL syntax for your system.

Other explicit compatibility points detailed below.

### 14.4.1 paramstyle

**The Python DB API specifies 5 types of parameter style**

- qmark: Question mark style, e.g. . . . WHERE name=?
- numeric: Numeric, positional style, e.g. . . . WHERE name=:1
- named: Named style, e.g. . . . WHERE name=:name
- format: ANSI C printf format codes, e.g. . . . WHERE name=%s
- pyformat: Python extended format codes, e.g. . . . WHERE name=%(name)s

SQLpy supports all of the *positional* paramstyles, for all query types other than `BUILT`, since the SQL code is simply passed straight through to the DB API library.

As of version 0.2.0 SQLpy only supports `pyformat` as the named paramstyle for `BUILT` query types.

Below is a non-exhaustive, possibly incomplete, probably out of date list, of popular Python DB API libaries and their paramstyle support.

| paramstyle | library |
|---|---|
| format, pyformat | psycopg2 |
| format, pyformat | pg8000 |
| format, pyformat | mysqldb |
| format, pyformat | mysqlconnector |
| format, pyformat | pymssql |
| qmark | oursql |
| qmark | pyodbc |
| qmark | sqlite3 |
| numeric, named | cx_oracle |

### 14.4.2 quote_ident

**PostgreSQL/psycopg 2.7+ Only**

Due to SQL parameter escaping (see Bobby Tables), many DB API libraries won't allow you to correctly pass in variables to set `identity` values in your query. These are things like column names in the SELECT, or ORDER BY clauses. The psycopg libary for PostgreSQL provides the `quote_ident()` function to solve this. To use it, pass a tuple of strings to your SQLpy function where the SQL contains a `{}` replacement field for each tuple item.

```
-- name: select_by_id
SELECT * FROM hello
WHERE {} = %s;
```

```
>> sql.SELECT_BY_ID(cur, identifiers=('id',), (1,))

[(1, u'hello')]
```

It is also possible to define identifiers in multiple parts of the query by passing a named identifier group(s) in a dictionary. This allows multiple identifiers to be contained within the same format token slot.

```
-- name: select_orderd_group
SELECT * FROM hello
ORDER BY {order_group};
```

```
>> sql.SELECT_BY_ID(cur, identifiers={'order_group': ('col_1', 'col_2', 'col_3')}
```

## 14.5 Tests

Tests are provided through the excellent pytest, and CI via Travis CI. SQLpy is tested against a real PostgreSQL database loaded with the pagila dataset.

**To run the test suite locally without a database, simply run `make test` from the root of the project. To run with a database (gi**

- load the pagila data by copying the commands in the `before_script` block in the `.travis.yml` file.
- modify the `test_sqlpy.py` file to enable running of the skipped test blocks and add any credentials to the `db_cur()` fixture.
- run with `make test` as before

## 14.6 Development

Team work makes the dream work!

We welcome contributions! You can open an Issue to report a bug or ask a question. If you would like to submit changes for review, please follow these steps:

1. Fork the repository
2. Make your changes
3. Install the requirements in `dev-requirements.txt`
4. Submit a pull request after running `make check` (ensure it does not error!)

## 14.7 License

MIT

## 14.8 How it works

Python PEP249 is the current standard for implementing APIs between Python and relational databases. From the PEP text

```
This API has been defined to encourage similarity between
the Python modules that are used to access databases.
```

The standard defines a number of common objects, methods and operations that any library must implement to be compliant. A slew of Python libraries exist for most if not all relational database systems, all adhering to the same specification. All of them boiling down to sending SQL commands to and returning their results from the database server to the Python runtime.

SQLpy was written as a lightweight helper around your already existing Python DB API 2.0 library, with no assumptions made about the underlying library of choice. Essentially only wrapping the `cur.execute()`, `cur.executemany()`, `cur.fetchone()`, `cur.fetchmany()` and `cur.fetchall()` methods. Connection and Cursor object creation preferences are left up to you.

SQLpy leverages the powerful `functools` module within Python and creates prepared functions reading SQL statements by reading from a queries file(s) within your project. The SQL statements have a special form that SQLpy recognises and this determines how the statement is prepared and eventually executed. Depending on if it is a simple select, a delete or a different type of statement.

For the following sections, let's assume we have a database table `hello` with the following data.

| id | message |
|---|---|
| 1 | hello |
| 2 | SQLpy |
| 3 | PostgreSQL! |

### 14.8.1 Initialising the Queries object

The first thing to do is to intialise the `sqlpy.Queries` object with your SQL queries.

```
sql = sqlpy.Queries(filepath, strict_parse=False, uppercase_name=True))
```

**Parameters**

- filepath (`list` of `str` or `str`): List of file locations containing the SQL statements or a single filepath to the queries file.
- strict_parse (`bool`, optional): Weather to strictly enforce matching the expected and supplied parameters to a SQL statement function.
- uppercase_name (`bool`, optional): Weather to cast the names of the SQL statement functions to uppercase.

### 14.8.2 Executing the functions

To execute a SQL statement and get results, just call the method by name on the `sqlpy.Queries` object. Note: The name is cast to uppercase (if this causes an uproar it can be made optional in a patch release).

```
sql = sqlpy.Queries('queries.sql')
....
results = sql.SQL_STATEMENT(cur, args=dict()|tuple(), n=None, identifiers=None, log_
↪query_params=LOG_QUERY_PARAMS)
```

**Parameters**

- `cur`: A Cursor object. Can be any cursor type you want.
- args (`tuple`) or (`dict`): A sequence of positional parameters in the query. Or a dictonary with named parameters in the query.
- n (`int`): How many results to fetch back. By default it is set to `None` which and the underlying cursor performs a `fetchall()` and all the results are returned. For `n=1` a `fetchone()` is performed and for `n>1` a `fetchmany(n)` is perfromed.
- identifiers (`tuple`): A sequence of positional strings to use to format the query before execution. Used with *identity strings*. Default is `None`.
- log_query_params (`boolean`): A flag to enable or disable logging out of the parameters sent to the query. Some data is sensitive and should not be visible in log entries. Default is `sqlpy.config.LOG_QUERY_PARAMS` which is `True`.

### 14.8.3 Query types

The type of query executed is determined by a token SQLpy searches for at the end of the `-- name:` special comment string in the SQL file. This can be `!`, `<!>`, `$`, `@` or not present.

Comments are detected and added to the `__doc__` attribute of the returned function.

```
-- name: sql_statement
-- a regular select statement
SELECT * FROM hello
WHERE id = %s;

-- name: insert_statement!
-- an insert statement
INSERT INTO hello (message)
VALUES (%s);

-- name: insert_statement2<!>
-- an insert statement with return
INSERT INTO hello (message)
VALUES (%s)
RETURNING id;

-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE id = %(id_low)s
OR id = %(id_high)s;

-- name: function_name@
-- a procedure/function being called
function_name
```

**SELECT** There is no token at the end of the name string

**INSERT/UPDATE/DELETE** There is a `!` token at the end of the name string

**With RETURNING** There is a `<!>` token at the end of the name string

**Built SQL** There is a `$` token at the end of the name string Can only use `pyformat` named parameters

**Function Call** There is a `@` token at the end of the name string

### Built SQL

In your application you will likely want to take different paths retrieving data depending on the current values or the variables you have available. One example could be looking up values from a table, using a varying number of search parameters. Writing a separate query for each case would be repetitive, and difficult as you need to know ahead of time the possible combinations.

**SQLpy offers the functionality to dynamically build SQL queries based on the query parameters passed to the prepared functio**

- An internal lookup map is created when the query is being parsed.
- Each line of the query is collected and inserted into a dictionary with information on its position (line number) in the overall query, and the query line itself.
- The key for each entry is the parameter contained within that line.
- Any lines with no parameter (most of the stuff before there `WHERE` clause), are collected under the same key.

When executed the query is reassembled in the correct line order, and lines containing parameters that have also been passed to the function as `kwargs` are included. (Note, not to be confused with the `**kwargs` convention in Python. Here we mean key-word arguments that are for the query.) Then the final SQL is sent to the database driver as normal.

Example.

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'id_low': 1}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

Would execute the SQL.

```
SELECT * FROM hello
WHERE id = 1;
```

**This design leads to some minor restrictions in how to write your queries that are explained below.**

**Single line per clause** It's best to lay out queries with a newline for each filter clause you use. This is to ensure that the resulting SQL query is built with the correct lines in place, and not with extra parameters for which there are no matching function inputs for.

```
-- name: built_sql_statement$
-- a built up sql statement
-- newline for each parameter
SELECT * FROM hello
WHERE id = %(id_low)s
OR id = %(id_high)s
AND message = %(msg)s;
```

**Careful with `WHERE`** SQL queries are asymmetrical, you always start the filtering clauses with `WHERE`, but after that you use `AND` or `OR`. This means that if the parameter contained within the `WHERE` clause is not passed to the function, the query will be built without it, and that is invalid SQL. To solve this, you can use `WHERE 1=1`. This always evaluates to `True` and is effectively a pass-through value, always ensuring the `WHERE` clause is present in your queries.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND id = %(id_low)s
OR id = %(id_high)s
AND message = %(msg)s;
```

**Multiple parameters per line** Sometimes you can not avoid multiple parameters that must be grouped together, such as in compound `AND-OR` clauses. Ensure you supply all the necessary argument to the function to get the correct output.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND (id = %(id_low)s OR id = %(id_high)s);
```

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'id_low': 1, 'id_high': 3}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (id = 1 OR id = 3);
```

**Missing parameters** In oder to maintain valid SQL output SQLpy will replace missing parameters with `None`, and this usually transforms to `NULL` when passed to the database. In this next example the result will still be correct, as the `id` column would not contain any `NULL` values, so the `OR` clause has no effect.

> **Note:** PostgrSQL does not correctly evaluate the syntax `column = NULL`, instead you should use `column is NULL` or `column is not NULL`. When searching columns that could contain `NULL` values, it's best to use the `ANY()` operator, where an array of values to check is passed to it. It behaves like `IN ()`, and it correctly handles `NULL` values. The added benefit is that you can test for multiple conditions in clauses too, so it's a useful pattern regardless. Check out PostgreSQL Arrays for more info.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND (id = ANY(%(id_low)s) OR id = ANY(%(id_high)s));
```

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'id_low': [1]}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (id = ANY('{1}') OR id = ANY('{NULL}'));
```

**Switching off parameters** The philosophical discussion on the merits/lack of on the use of `NULL` in SQL systems is well known, but the value (or is it a Type?) is used everywhere. This just means you need to take this into account when writing your data retrieval queries with `NULL` values.

Following from the example above, say you have a compound `OR` clause on a column that can have `NULL` values. At certain times, you may not supply all the values required to the function, so `None` is substituted in its place. This is a problem because you don't want the case where extra results are returned that match the other side of the `OR`.

We have new data...

| id | message | message2 |
|---|---|---|
| 1 | hello | there |
| 2 | SQLpy | NULL |
| 3 | PostgreSQL! | rules! |
| 4 | hello | friend |

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND (message = ANY(%(msg)s) OR message2 = ANY(%(msg2)s));
```

```
sql = sqlpy.Queries('queries.sql')
....
kwargs = {'message': ['hello']}
results = sql.BUILT_SQL_STATEMENT(cur, kwargs)
```

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (message = ANY('{"hello"}') OR message2 = ANY('{NULL}'));
```

returns...

| id | message | message2 |
|---|---|---|
| 1 | hello | there |
| 2 | SQLpy | NULL |
| 4 | hello | friend |

We don't want row 2 in this case. To solve this, you can use a little SQL syntax gymnastics to write the `OR` clause in such a way that `NULL` does not bring in incorrect results.

```
-- name: built_sql_statement$
-- a built up sql statement
SELECT * FROM hello
WHERE 1=1
AND ((FALSE OR message = ANY(%(msg)s)) OR (FALSE OR message2 = ANY(%(msg2)s)));
```

The clauses are enclosed in a second set of parenthesis in the form `(FALSE OR column=%(name)s)`. If the parameter is replaced with a `NULL` then this "switches-off" that entire check, because `SELECT FALSE OR NULL --> NULL`. So `(NULL OR (FALSE OR column=VALUE))` only evaluates the right hand side of the statement. This would produce the correct output.

executes...

```
SELECT * FROM hello
WHERE 1=1
AND (FALSE OR message = ANY('{"hello"}' OR (FALSE OR message2 = ANY('{NULL}'))));
-- this reduces to
-- AND (message = ANY('{"hello"}' OR (NULL));
-- and again to
-- AND (message = ANY('{"hello"}'));
```

returns...

| id | message | message2 |
|---|---|---|
| 1 | hello | there |
| 4 | hello | friend |

**Warning: only tested in PostgreSQL**

**NULL with care** As you can see this is very tricky and also very database specific. It's probably best to avoid writing such queries in the first place, and taking a second look at your data model could also reveal a better design.

But you could still come across and need this pattern. However now that the problem is exposed purely as a SQL problem, you can now seek help in SQL Q&A forums in which there is about 50 years (and counting) of SQL language experience!

**Strict parse** If you don't like to live dangerously, then you can enable a safety mechanism around Built queries. If you initialise the `sqlpy.Queries` object as `sqlpy.Queries(..., strict_parse=True)`, a `sqlpy.exceptions.SQLArgumentException` is raised when a named argument is supplied which does not match a SQL clause.

**Built queries are limited to only SELECT queries at the moment.** There will definitely be some interesting edge cases arising from the layout and use of Built queries! If you see anything odd and think it should be handled, then do open an issue on GitHub.

### Identity strings

#### PostgreSQL/psycopg 2.7+ Only

Due to SQL parameter escaping (see Bobby Tables), many DB API libraries won't allow you to correctly pass in variables to set `identity` values in your query. These are things like column names in the SELECT, or ORDER BY clauses. The psycopg libary for PostgreSQL provides the `quote_ident()` function to solve this. To use it, pass a tuple of strings to your SQLpy function where the SQL contains a `{}` replacement field for each tuple item.

```
-- name: select_by_id
SELECT * FROM hello
WHERE {} = %s;
```

```
>> sql.SELECT_BY_ID(cur, identifiers=('id',), (1,))

[(1, u'hello')]
```

It is also possible to define identifiers in multiple parts of the query by passing a named identifier group(s) in a dictionary. This allows multiple identifiers to be contained within the same format token slot.

```
-- name: select_orderd_group
SELECT * FROM hello
ORDER BY {order_group};
```

```
>> sql.SELECT_BY_ID(cur, identifiers={'order_group': ('col_1', 'col_2', 'col_3')}
```

# CHAPTER 15

# Showing Off

## 15.1 Sudoku

SQL is more than just a declarative data retrieval language, it's a fully Turing complete language in its own right. So it should be able to compute anything that a more typical application language could... although it may not be the most syntactically concise bit of code out there. SQL does shine in performing set operations, that is evaluating functions over groups (sets) of data (relations/tables) all at once.

Sets of data... numbers... grids. Did someone say Sudoku solver? A 9x9 Sudoku solving SQL snippet was recently added to the postgresql wiki page. Using a recursive window function, it implements a brute-force backtracking algorithm to solve the puzzle. Taking up only 32 lines, it could be less as SQL does not depend on whitespace heavily, it solves the example puzzle in under 10 seconds on a mid-range quad-core laptop from 2014 running postgres 9.6.

Imagine trying to program this to be done in SQL in a similar way but via an ORM!? With SQLpy it wold be easy. (let's gloss over the amount of energy that went into writing the SQL in the first place!)

```
-- name: sudoku_solver
-- a sudoku solver
-- in SQL why not
-- note the query param %s 3 lines below
WITH recursive board(b, p) AS (
  -- sudoku board expressed in column-major order, so substr() can be used to fetch a␣
↪column
  VALUES (%s::CHAR(81), 0)
  UNION ALL SELECT b, p FROM (
    -- generate boards:
    SELECT overlay(b placing new_char FROM strpos(b, '_') FOR 1)::CHAR(81), strpos(b,
↪'_'), new_char
    FROM board, (SELECT chr(n+ascii('b')) FROM generate_series(0, 8) n) new_char_
↪table(new_char)
    WHERE strpos(b, '_') > 0
  ) r(b, p, new_char) WHERE
    -- make sure the new_char doesn't appear twice in its column
    -- (there are two checks because we are excluding p itself):
```

(continues on next page)

(continued from previous page)

```
    strpos(substr(b, 1+(p-1)/9*9, (p-1)%9), new_char) = 0 AND
    strpos(substr(b, p+1, 8-(p-1)%9), new_char) = 0 AND
    -- make sure the new_char doesn't appear twice in its row:
    new_char NOT IN (SELECT substr(b, 1+i*9+(p-1)%9, 1)
                     FROM generate_series(0, 8) i
                     WHERE p <> 1+i*9+(p-1)%9) AND
    -- make sure the new_char doesn't appear twice in its 3x3 block:
    new_char NOT IN (SELECT substr(b, 1+i%3+i/3*9+(p-1)/27*27+(p-1)%9/3*3, 1)
                     FROM generate_series(0, 8) i
                     WHERE p <> 1+i%3+i/3*9+(p-1)/27*27+(p-1)%9/3*3)
) SELECT
    -- the following subquery is used to represent the board in a '\n' separated␣
↪human-readable form:
    ( SELECT string_agg((
        SELECT string_agg(chr(ascii('1')+ascii(substr(b, 1+y+x*9, 1))-ascii('b')), '
↪') r
        FROM generate_series(0, 8) x), E'\n')
      FROM generate_series(0, 8) y
    ) human_readable,
    b board,
    p depth,
    (SELECT COUNT(*) FROM board) steps
  FROM board WHERE strpos(b,'_') = 0 LIMIT 5000;
```

```
board = '__g_cd__bf_____j____c__e___c__i___jd__b__h___id____e__g__b__f_e_f____g____j_
↪h__c_'
results, _ = sql.SUDOKU_SOLVER(cur, (board,), n=1)

print(results[0])
```

```
457298631
819763254
632415879
975832146
261549387
384671925
798124563
543986712
126357498
```

CHAPTER 16

# Changelog

## 16.1 Current release

### 16.1.1 0.3.4 0.3.5

**Bugfix**

- use a more broad import for `quote_ident` function, to allow instrumentation libraries to correctly use the patched version #24

## 16.2 Previous releases

### 16.2.1 0.3.3

**Bugfix**

- identifier formatting in built sql not happening #21

### 16.2.2 0.3.2

**Major Updates**

- Support sets of identifier groups and using named parameters #20

### 16.2.3 0.3.1

**Breaking Changes**

- revert returning cursor as part of query execution #17

**Major Updates**

- expose logging query args configuration in Queries object initialisation #18

### 16.2.4 0.3.0

**Breaking Changes**

- updated API design for the query function #15

**Major Updates**

- add separated `cur.fetchone` cursor method #15
- add `cur.callproc` #12
- transparently switch to using the more efficient `execute_values` with Psycopg2 #16
- updated docs

**Minor Fixes**

- fix type in `load_queries` #14
- initial strip of whitespace on input query files
- ensure multi-file queries joined correctly with double newline #10
- make uppercase of function name optional
- moved logging helper back into main module #8

### 16.2.5 0.2.0

**Major update**

- Remove coupling to PostgreSQL and psycopg2 by conditionally importing from psycopg2
- Changed the logger to add `NullHandler()` by default
- Changed `fetchone()` to `fetchmany()`
- Improved exception handling with better Exceptions
- A lot of internal code refactoring
- Documentation, a lot of Documentation

### 16.2.6 0.1.0

First pypi upload of the project. Lacking good documentation and behavior was tied to PostgreSQL and psycopg2.

## 16.3 Source Code Documentation

*Source Code Documentation*