pgmock Documentation

Release 1.3.3

Clover Health

Sep 25, 2020

Contents

CHAPTER 1

pgmock

`pgmock` provides utilities for obtaining and mocking out expressions in Postgres queries. This allows for testing smaller portions of larger queries and alleviates issues of having to set up state in the database for more traditional (and faster) SQL unit tests.

`pgmock` has three primary use cases:

1. Obtaining expressions in a query
2. Patching expressions in a query
3. Patching queries executed by SQLAlchemy

A quickstart for each of these is below. To skip the quickstart and go straight to the tutorial, go to *Tutorial*.

1.1 Quickstart

1.1.1 Obtaining Expressions in a Query

Assume we want to test the following query:

```
query = "SELECT sub.c1, sub.c2 FROM (SELECT c1, c2 FROM test_table WHERE c1 = 'hi!')␣
↪sub;"
```

This example illustrates a query that has comparison logic in the subquery named *sub*. This subquery can be obtained with:

```
import pgmock

sub = pgmock.sql(query, pgmock.subquery('sub'))

print(sub)
"SELECT c1, c2 FROM test_table WHERE c1 = 'hi!'"
```

In the above, `pgmock.sql` was used to render the SQL targetted by the `pgmock.subquery` *selector*. A selector is a way to specify an expression inside a query. In this case, it's a subquery named *sub*.

Selectors can be chained together to refine selections. For example, `pgmock.statement(0).subquery('sub')` would reference the subquery `sub` in the first statement of the SQL. All available selectors in pgmock are listed below. For more info about the selectors and how they work, view the *Interface* section.

1. *pgmock.statement* - Obtain a statement or range of statements
2. *pgmock.cte* - Obtain or patch a common table expression
3. *pgmock.create_table_as* - Obtain or patch a `CREATE TABLE AS` expression
4. *pgmock.table* - Patch a table
5. *pgmock.insert_into* - Obtain or patch an `INSERT INTO` expression
6. *pgmock.subquery* - Obtain or patch a subquery

1.1.2 Patching Expressions in a Query

If one wanted to test the above subquery and ensure that it filters rows properly, a database table named `test_table` would need to be created along with the appropriate data inserted. This setup, however, is rather cumbersome and slow, especially in a test case that needs to tear down the database after each test.

In the spirit of Python mocking and only testing logic in unit tests, `pgmock` provides the ability to patch expressions with Postgres VALUES.

What does this look like in practice? Lets continue using our `sub` variable from above:

```
rows = [('hi!', 'val1'), ('hello!', 'val2'), ('hi!', 'val3')]

# Patch "test_table" with the rows as the return value
patch = pgmock.patch(pgmock.table('test_table'), rows=rows, cols=['c1', 'c2'])

# Apply the patch to the subquery SQL
patched = pgmock.sql(sub, patch)

print(patched)
"SELECT c1, c2 FROM (VALUES ('hi!','val1'),('hello!','val2'),('hi!','val3')) AS test_
↪table(c1,c2) WHERE c1 = 'hi!'"
```

In the above, we made a `patch` with a `table` selector and made it return a list of rows. When using `sql` to render the query, `test_table` was modified to be a `VALUES` expression.

The `patched` query can now be executed with no database setup and the filtering logic can be tested for correctness.

This approach could similarly be used on the full original query. Patching the table of the subquery would proceed as follows:

```
# Apply the patch to the full query
patched = pgmock.sql(query, patch)

print(patched)
"SELECT sub.c1, sub.c2 FROM (SELECT c1, c2 FROM (VALUES ('hi!','val1'),('hello!','val2
↪'),('hi!','val3')) AS test_table(c1,c2) WHERE c1 = 'hi!') sub;"
```

One could similarly patch out the entire subquery:

```
# Patch the "sub" subquery with the rows as the return value
patch = pgmock.patch(pgmock.subquery('sub'), rows=rows, cols=['c1', 'c2'])

# Apply the patch to the full query
patched = pgmock.sql(query, patch)

print(patched)
"SELECT sub.c1, sub.c2 from (VALUES ('hi!','val1'),('hello!','val2'),('hi!','val3'))␣
↪AS sub(c1,c2);"
```

Having a patched query like the above allows one to use a readonly database connection and execute the query while testing that it behaves as expected. For example:

```
import sqlalchemy as sqla

db_conn = sqla.create_engine('postgresql://localhost:5432/local-db')
results = db_conn.execute(patched)

# Assert only rows where c1 = "hi!" are returned
assert results == [('hi!', 'val1'), ('hi!', 'val3')]
```

Want to only patch out some of your columns? Pass dictionaries of rows as input and `null` values are filled in for everything else in the row:

```
# Patch the "sub" subquery with the dictionary rows as the return value. All missing␣
↪columns will
# be filled with nulls
rows = [{'c1': 'hi!'}, {'c2': 'hello!'}]
patch = pgmock.patch(pgmock.subquery('sub'), rows=rows, cols=['c1', 'c2'])

# Apply the patch to the full query
patched = pgmock.sql(query, patch)

print(patched)
"SELECT sub.c1, sub.c2 FROM  (VALUES ('hi!',null),(null,'hello!')) AS sub(c1,c2);"
```

1.1.3 Patching Queries Executed by SQLAlchemy

Sometimes it is not possible to have full control over the SQL being executed, such as when testing SQLAlchemy code. For this case, `pgmock` can be used as a context manager and modify executed SQLAlchemy queries on the fly. This functionality can be used like so:

```
# "connectable" is a SQLAlchemy engine, session, connection, or other connectable␣
↪object
with pgmock.mock(connectable) as mocker:
    # Apply patches
    mocker.patch(pgmock.subquery('sub'), rows=rows, cols=['c1', 'c2'])

    # Execute SQLAlchemy code
    ...

    # Assert that the queries were rendered
    assert len(mocker.renderings) == expected_number_of_queries
```

The `renderings` variable contains tuples of the original SQL and the modified SQL for every query executed within the context manager. In this example, all queries are assumed to have a *sub* subquery that is patched with provided

output rows. Patching can also be done on a per-query basis, and this is described more in the *Tutorial*.

1.2 Next Steps

- Go to *Tutorial* for a full tutorial on `pgmock`.
- Go to *Interface* for the documentation of the main `pgmock` interface.
- For `pgmock` exceptions and docs about what causes some errors, go to *Exceptions*.
- It's also good to familiarize yourself with some of the known issues and future work of `pgmock` by going to *Known Issues and Future Work*.

CHAPTER 2

Installation

pgmock can be installed with:

```
pip3 install pgmock
```

CHAPTER 3

Tutorial

This tutorial is created directly from an ipython notebook. If you'd like to interactively run this tutorial, do the following:

```
# Go to your pgmock directory with the cloned code
cd pgmock
make setup
jupyter notebook Tutorial.ipynb
# Follow the instructions to open the notebook in your browser
```

The setup for this ipython notebook is below and includes the creation of the testing database and required imports.

```
import testing.postgresql
import sqlalchemy
import pgmock
import pgmock.exceptions

test_db = testing.postgresql.Postgresql()
test_engine = sqlalchemy.create_engine(test_db.url())
```

3.1 Terminology and Key Concepts

It's useful to understand some commonly-used terminology and key concepts before going into the `pgmock` tutorial.

Selectors - `pgmock` *selectors* are objects used to obtain portions of SQL within a query. Selectors can represent subqueries, tables, select statements, and other types of SQL expressions. All selectors in `pgmock` are chainable, meaning they can be called after one another to refine a selection.

Patching - `pgmock` *patching* is concerned with converting SQL select expressions or tables into postgres `VALUES` expressions. For example, this statement:

```
SELECT c1 from test_table
```

could have `test_table` patched to be:

```
SELECT c1 from (VALUES ('hi!'),('hello!') AS test_table(c1)
```

Patching can also take place on joins, subqueries, `INSERT INTO` expressions, and other SQL expressions that allow postgres `VALUES`.

Rendering - Whenever SQL is obtained from a query or modified with a patch, it is *rendered*.

3.2 Obtaining Expressions in a Query

Obtaining specific portions of a query can be useful when wanting to execute smaller parts of your SQL or for (later in the tutorial) patching out portions of your SQL in tests.

This part of the tutorial demonstrates obtaining expressions inside a query using the `pgmock.sql` function. `pgmock.sql` takes a SQL string and a selector as input. The expression in the query referenced by the selector is then rendered and returned as a string. Uses of different selectors to obtain expressions is shown in the following.

3.2.1 Obtaining Statements

The `pgmock.statement` selector can be used to render ranges of statements in a query as shown below.

```
query = '''SELECT * from table1;
SELECT * from table2;
SELECT * from table3
'''

# Obtain the first statement
print(pgmock.sql(query, pgmock.statement(0)))
```

```
SELECT * from table1
```

```
# Obtain the first three statements using a range of 0 - 3 (3 is the exclusive ending␣
↪index)
print(pgmock.sql(query, pgmock.statement(0, 3)))
```

```
SELECT * from table1;
SELECT * from table2;
SELECT * from table3
```

```
# Going out of range will throw an exception
try:
    pgmock.sql(query, pgmock.statement(4))
except pgmock.exceptions.StatementParseError as exc:
    print(exc)
```

```
Found 3 statements. Range of [4:5] is out of bounds. The following SQL was used:

SELECT * from table1;
SELECT * from table2;
SELECT * from table3

View the docs for this exception at https://pgmock.readthedocs.io/en/latest/
↪exceptions.html for more information.
```

Note: Rendering statements splits the SQL with the semicolon character. If the semicolon appears in any comments or string literals, this can interfere with obtaining statements. Use `safe_mode=True` to *pgmock.sql* in order to fix this issue if it happens. This comes at a performance cost, and more details can be read at *pgmock.config.set_safe_mode*.

3.2.2 Obtaining Subqueries

The `pgmock.subquery` selector can be used to render subqueries in SQL as shown below.

```
query = 'SELECT sub.c1, sub.c2 FROM (SELECT * FROM test_table) sub;'

# Obtain the subquery named "sub"
print(pgmock.sql(query, pgmock.subquery('sub')))
```

```
SELECT * FROM test_table
```

```
# An exception will be raised if the subquery alias cannot be found
try:
    pgmock.sql(query, pgmock.subquery('bad'))
except pgmock.exceptions.NoMatchError as exc:
    print(exc)
```

```
No subquery found for alias "bad". The following SQL was used:

SELECT sub.c1, sub.c2 FROM (SELECT * FROM test_table) sub;

View the docs for this exception at https://pgmock.readthedocs.io/en/latest/
↪exceptions.html for more information.
```

```
# pgmock does not handle the case when the same subquery alias is used twice or nested
query = 'SELECT sub.c1, sub.c2 FROM (SELECT * FROM (SELECT * FROM test_table) sub)␣
↪sub;'

try:
    pgmock.sql(query, pgmock.subquery('sub'))
except pgmock.exceptions.MultipleMatchError as exc:
    print(exc)
```

```
Nested matches were found in your selection. The following multiple matches of SQL␣
↪were used:

SELECT * FROM test_table
---
SELECT * FROM (SELECT * FROM test_table) sub

View the docs for this exception at https://pgmock.readthedocs.io/en/latest/
↪exceptions.html for more information.
```

Note: Almost all `pgmock` selectors can only render exactly one match like the subquery selector above. These cases will not be illustrated for later examples. Note that there is a distinction between "rendering" selectors with `pgmock.sql` like above and patching them with `pgmock.patch`. `pgmock.patch` supports patching multiple occurences

in a selection

3.2.3 Obtaining Insert Into Expressions

The `pgmock.insert_into` selector can be used to render `INSERT INTO` expressions in SQL as shown below.

```
query = '''INSERT INTO table_a
SELECT * FROM other_table;

INSERT INTO table_b
SELECT * FROM other_table'''

# The insert_into selector takes the table name that is inserted into
print(pgmock.sql(query, pgmock.insert_into('table_a')))
```

```
INSERT INTO table_a
SELECT * FROM other_table
```

In order to obtain the body of the expression, the `body` selector can be chained to the `insert_into` selector.

```
print(pgmock.sql(query, pgmock.insert_into('table_a').body()))
```

```
SELECT * FROM other_table
```

Note: All selectors can be chained like the above example where it makes sense. For example, one could obtain the `INSERT INTO` expression of the first statement with `pgmock.statement(0).insert_into('table_a')`

Along with that, multiple selectors can also be provided to `pgmock.sql` or `pgmock.sql_file`, which in turn will chain them underneath the hood. For example, doing:

```
pgmock.sql(query, pgmock.insert_into('table_a'), pgmock.body())
```

is equivalent to:

```
pgmock.sql(query, pgmock.insert_into('table_a').body())
```

It is up to the user to pick which style they prefer. `pgmock` suggests using multiple arguments when applying multiple `pgmock.patch` selectors and using chaining syntax for all other cases.

3.2.4 Obtaining Create Table As Expressions

The `pgmock.create_table_as` selector can be used to render `CREATE TABLE AS` expressions in SQL as shown below. It's similar to `pgmock.insert_into`.

```
query = '''CREATE TABLE table_a AS (
  SELECT * FROM other_table
);

CREATE TABLE table_b AS SELECT * FROM other_table'''

# The insert_into selector takes the table name that is inserted into
print(pgmock.sql(query, pgmock.create_table_as('table_a')))
```

```
CREATE TABLE table_a AS (
  SELECT * FROM other_table
)
```

In order to obtain the body of the `CREATE TABLE AS` expression, the `body` selector can be chained to the `create_table_as` selector.

```
print(pgmock.sql(query, pgmock.create_table_as('table_a').body()))
```

```
 (
  SELECT * FROM other_table
)
```

3.2.5 Obtaining Common Table Expressions (CTEs)

The `pgmock.cte` selector can be used to render common table expressions (CTEs) in SQL as shown below. It's similar to `pgmock.subquery`.

```
query = '''
WITH cte1 AS (
    SELECT * FROM table1
), cte2 AS (
    SELECT * FROM table2
)
'''

# Obtain the CTE aliased "cte1"
print(pgmock.sql(query, pgmock.cte('cte1')))
```

```
SELECT * FROM table1
```

```
# Obtain the "cte2" CTE
print(pgmock.sql(query, pgmock.cte('cte2')))
```

```
SELECT * FROM table2
```

3.3 Patching Expressions in a Query

As mentioned before, patching parts of a query will transform the relevant expression into Postgres VALUES. Why is this useful?

1. When using `VALUES` lists, there is no need to create database tables and data before executing the query
2. Testing queries will run much faster in automated tests since there is no overhead of database setup and teardown
3. Only data that is relevant to the test can be patched, resulting in smaller and more readable tests. `pgmock` allows other useless columns to be filled in with nulls by default if desired

Below are some illustrations of patching queries and running some assertions on those queries. This section uses the test engine that we created at the beginning of the tutorial.

Note: In an automated pytest test case, we'd use the fixtures from pytest-pgsql when testing our queries

3.3.1 Patching Tables and Joins on Tables

Tables, whether those tables are being selected or joined, can be patched with `pgmock` by using the `pgmock.table` selector. Some examples of patching tables and joins are below.

```
# Create a query and filter a column
query = "SELECT c2 FROM my_table WHERE c1 = 'value'"

# Create a patch for the table. The patch takes a selector,
# rows (a list of lists for each column or a list of dictionaries keyed on column),
# and column names
patch = pgmock.patch(pgmock.table('my_table'),
                     [('dummy_data', 'data'), ('value', 'hello'), ('value', 'hi')],
                     ['c1', 'c2'])

# Render the patched SQL so that we can execute it
sql = pgmock.sql(query, patch)
print(sql)
```

```
SELECT c2 FROM  (VALUES ('dummy_data','data'),('value','hello'),('value','hi')) AS my_
↪table(c1,c2) WHERE c1 = 'value'
```

```
# Execute the SQL and verify that filtering happened correctly
results = list(test_engine.execute(sql))
assert results == [('hello',), ('hi',)]
```

Note: *pgmock.patch* accepts lists of dictionaries as rows of input as well. For example,

```
patch = pgmock.patch(pgmock.table('my_table'),
             [{'c1': 'dummy_data', 'c2': 'data'}, {'c1': 'value', 'c2': 'hello'}, {'c1
↪': 'value', 'c2': 'hi'}],
             ['c1', 'c2'])
```

is equivalent to the example from above.

Using this format allows for only specifying values for columns that matter. All missing columns will be filled with null values. Both formats of patching will be used throughout the rest of the tutorial.

Similar to selectors, patches are also chainable or can be provided as multiple arguments to `pgmock.sql` or `pgmock.sql_file`. This is useful for the case of patching multiple expressions in a query. For example, we can patch a table and a join on another table like so.

```
# Create a query with a join
query = 'SELECT one.c1 FROM t1 one JOIN t2 two ON one.c1 = two.c1'

# When making the patches, keep in mind these tables have aliases and
# the alias must also be provided when obtaining the table
t1_patch = pgmock.patch(
    pgmock.table('t1', alias='one'),
    [('val1.1',), ('val1.2',)],
    ['c1']
)

t2_patch = pgmock.patch(
    pgmock.table('t2', alias='two'),
    [('val1.1',), ('val1.2',), ('val1.3',)],
```

```
    ['c1']
)

# Render the SQL that has both tables patched
sql = pgmock.sql(query, t1_patch, t2_patch)
print(sql)
```

```
SELECT one.c1 FROM  (VALUES ('val1.1'),('val1.2')) AS one(c1) JOIN  (VALUES ('val1.1
↪'),('val1.2'),('val1.3')) AS two(c1) ON one.c1 = two.c1
```

```
# Execute the SQL and verify that the join happened correctly
results = list(test_engine.execute(sql))
assert results == [('val1.1',), ('val1.2',)]
```

3.3.2 Patching Multiple Occurrences of Tables

If a table appears multiple times in your SQL, it will be patched in all occurrences by default. This holds true for any selector. If you want to only patch a specific occurrence or range of tables, use list syntax. For example:

```
# Create a query and filter a column
query = "SELECT c2 FROM my_table; SELECT c3 from my_table"

# Patch both occurrences of the table
patch = pgmock.patch(pgmock.table('my_table'),
                     [('dummy_data', 'data'), ('value', 'hello'), ('value', 'hi')],
                     ['c2', 'c3'])

# Render the patched SQL so that we can execute it
sql = pgmock.sql(query, patch)
print(sql)

# Use list syntax to only patch the second occurrence of the table
patch = pgmock.patch(pgmock.table('my_table')[1],
                     [('dummy_data', 'data'), ('value', 'hello'), ('value', 'hi')],
                     ['c2', 'c3'])
print(pgmock.sql(query, patch))
```

```
SELECT c2 FROM  (VALUES ('dummy_data','data'),('value','hello'),('value','hi')) AS my_
↪table(c2,c3); SELECT c3 from  (VALUES ('dummy_data','data'),('value','hello'),(
↪'value','hi')) AS my_table(c2,c3)
SELECT c2 FROM my_table; SELECT c3 from  (VALUES ('dummy_data','data'),('value','hello
↪'),('value','hi')) AS my_table(c2,c3)
```

3.3.3 Patching Subqueries

Patching subqueries (and almost all other expressions) works in the same way as patching tables. Create a selector you want to patch and provide the data to be patched.

```
# Create a query with a subquery
query = "SELECT sub.c1, sub.c2 FROM (SELECT * FROM test_table) sub;"

# Create a patch for the subquery. Similar to patching tables, provide a subquery␣
↪selector and the data for the subquery
```

```
patch = pgmock.patch(pgmock.subquery('sub'),
                     [('val1', 'val2'), ('val3', 'val4')],
                     ['c1', 'c2'])

# Render the patched SQL so that we can execute it
sql = pgmock.sql(query, patch)
print(sql)
```

```
SELECT sub.c1, sub.c2 FROM  (VALUES ('val1','val2'),('val3','val4')) AS sub(c1,c2);
```

```
# Execute the SQL and verify that the subquery was patched properly
results = list(test_engine.execute(sql))
assert results == [('val1', 'val2'), ('val3', 'val4')]
```

3.3.4 Patching CTEs, Create Table As, Insert Into, and Other Expressions

`pgmock` can patch almost every selector available in the library, such as `pgmock.insert_into`, `pgmock.create_table_as`, and `pgmock.cte`. Patching these expressions result in different types of patches depending on what is being patched.

For example, patching an `INSERT INTO` statement will result in replacing the body of the `INSERT INTO` with a `VALUES` list that has no alias (Postgres doesn't support the syntax of `INSERT INTO table (VALUES ..) AS ...`).

When patching `CREATE TABLE AS` or a CTE, the patch will insert a `SELECT * FROM (VALUES ...) AS ...`. Doing this syntax allows column names of the patch to be preserved and gets around the restriction of not being able to do `CREATE TABLE AS (VALUES ...) AS ...`.

Note: Keep in mind that when patching statements when the table structure is defined, such as `CREATE TABLE t(col1, col2) AS` or `WITH cte_name(col1, col2) AS`, the columns provided to the patch need to be in the same order as they are defined in the alias definition.

Below is an example of patching a CTE

```
# Create an example of selecting from a CTE
query = '''
WITH cte_name AS (
    SELECT * from some_other_table
)

SELECT c1, c2, c3 from cte_name;
'''

# Patch the CTE with the data you want returned
patch = pgmock.patch(
    pgmock.cte('cte_name'),
               [('val1', 'val2', 'val3')],
               ['c1', 'c2', 'c3']
)

sql = pgmock.sql(query, patch)
print(sql)
```

```
WITH cte_name AS ( SELECT * FROM (VALUES ('val1','val2','val3')) AS pgmock(c1,c2,c3))

SELECT c1, c2, c3 from cte_name;
```

```
results = list(test_engine.execute(sql))
assert results == [('val1', 'val2', 'val3')]
```

3.3.5 Patching and Executing Smaller Components of Queries

Sometimes one may only be interested in testing a small part of their SQL. This is especially true in testing the selects of an `INSERT INTO` or a subquery. An example of pulling out a subquery and testing it is shown below.

```
# Create a query with a subquery
query = "SELECT sub.c1, sub.c2 FROM (SELECT * FROM test_table where c1 = 'value') sub;
↪"

# Obtain the subquery so that it can be patched and tested
subquery = pgmock.sql(query, pgmock.subquery('sub'))

# Create a patch for the subquery's table. Similar to patching tables, provide a␣
↪subquery selector and the data for the subquery
patch = pgmock.patch(pgmock.table('test_table'),
                     [('value', 'val2'), ('val3', 'val4')],
                     ['c1', 'c2'])

# Render the patched SQL so that we can execute it
sql = pgmock.sql(subquery, patch)
print(sql)
```

```
SELECT * FROM  (VALUES ('value','val2'),('val3','val4')) AS test_table(c1,c2) where␣
↪c1 = 'value'
```

```
# Execute the SQL and verify that the subquery performs its select properly
results = list(test_engine.execute(sql))
assert results == [('value', 'val2')]
```

3.4 Patching Queries Executed by SQLAlchemy

Sometimes it's not always possible to have full control of the SQL that's being executed. For example, one might want to test code that issues many different SQLAlchemy statements and still want to patch out the underlying tables.

For these cases, `pgmock.mock` can be used as a context manager. `pgmock.mock` takes the SQLAlchemy connectable as an argument and listens for any queries executed against the connectable. When queries are executed, they are patched on the fly before they are executed. Some examples of this are shown below.

```
with pgmock.mock(test_engine) as mocker:
    # Apply patches to the mocker object we created. For this example, we are going to
    # patch "test_table"
    mocker.patch(pgmock.table('test_table'), [('val1', 'val2', 'val3')], ['c1', 'c2',
↪'c3'])

    # When executing this query, it will be patched on the fly with the values␣
↪provided
```

```
    results = list(test_engine.execute('SELECT * from test_table'))
    assert results == [('val1', 'val2', 'val3')]
```

3.4.1 Patching Multiple Queries with Side Effects

In most testing situations, one will have more complex SQLAlchemy code that may issue multiple queries. For example, lets take the previous test example and put our SQLAlchemy code in a function that executes two different queries.

```
def my_sqla_func(engine):
    """A function that issues a couple different queries that we want to test"""
    cursors = engine.execute('SELECT * from pg_cursors')
    # Do something important with the cursors...

    # Now return results from a table
    return list(engine.execute('SELECT * from test_table'))
```

If we try to test this function the same way as before, an error will happen.

```
with pgmock.mock(test_engine) as mocker:
    # Apply patches to the mocker object we created. For this example, we are going to
    # patch "test_table"
    mocker.patch(pgmock.table('test_table'), [('val1', 'val2', 'val3')], ['c1', 'c2',
↪'c3'])

    # When executing this query, it will be patched on the fly with the values␣
↪provided
    try:
        results = my_sqla_func(test_engine)
        assert results == [('val1', 'val2', 'val3')]
    except pgmock.exceptions.NoMatchError as exc:
        print(exc)
```

```
No table "test_table" found. The following SQL was used:

SELECT * from pg_cursors

View the docs for this exception at https://pgmock.readthedocs.io/en/latest/
↪exceptions.html for more information.
```

In the above, running `my_sqla_func` with the patched "test_table" threw a `NoMatchError`. When looking at the error message, it appears that this error occurred on our first query of our function (`SELECT * from pg_cursors`).

This happens because the patch on "test_table" will be applied to every single query that's issued, including the first query that cannot be patched. Instead of silently continuing, `pgmock` will raise errors anytime something cannot be matched.

In order to get around this, use a `side_effect` argument to the patch instead of a single return value. A side effect is a list of return values to use every time the patch is applied to a query. The first side effect will be applied to the first query issued and so forth. If more queries are issued than the number of side effects, a `SideEffectExhaustedError` will be raised. If `None` is provided as a return value, the patch will be completely ignored for the query.

The previous example can be changed to use a side effect in the following way.

```
with pgmock.mock(test_engine) as mocker:
    # Apply patches to the mocker object we created. For this example, we are going to
    # patch "test_table" on the second query that is issued by using a side effect
    mocker.patch(
        pgmock.table('test_table'),
        side_effect=[
            # Ignore patching test_table for the first query
            None,
            # Use pgmock.data to construct rows and columns of return data for
            # the second query
            pgmock.data([('val1', 'val2', 'val3')], ['c1', 'c2', 'c3'])
        ])

    # When executing this query, it will be patched on the fly with the values␣
↪provided
    results = my_sqla_func(test_engine)
    assert results == [('val1', 'val2', 'val3')]

    # As a precaution, it is always good practice to assert that the number of␣
↪renderings of
    # the mocker matches the number of queries you expected your test code to issue
    assert len(mocker.renderings) == 2
```

3.5 Advanced Usage

3.5.1 Patching Custom Types and Using Type Hinting

`pgmock` allows the user to provide quite a few different types of Python objects to patched values lists. Python objects are converted into their proper postgres type. For example, a datetime object is converted to a timestamp and a dictionary is converted to a json object. An example of this is shown below.

```
import datetime as dt
query = "SELECT * FROM my_table"

# Create a patch for the table. The patch takes a selector, rows (a list of lists for␣
↪each column), and column names
patch = pgmock.patch(pgmock.table('my_table'),
                     [(dt.datetime(2017, 6, 14), {'my': 'json_data'}, None)],
                     ['c1', 'c2', 'c3'])

# Render the patched SQL so that we can execute it
sql = pgmock.sql(query, patch)
print(sql)
```

```
SELECT * FROM  (VALUES ('2017-06-14T00:00:00'::TIMESTAMP,'{"my": "json_data"}'::JSON,
↪null)) AS my_table(c1,c2,c3)
```

The amount of Python types supported out of the box in `pgmock` is rather limited. Along with that, it's impossible to specify certain datatypes one might need for their tests in Python (e.g. a null datetime). `pgmock` allows the user to specify type hints to cast their values to a particular type. The type is specified by placing `::type_name` after the column name. For example, the following illustrates how to cast patched values to datetimes and bigints.

```
# Create a patch for the table. The patch takes a selector, rows (a list of lists for
↪each column), and column names
patch = pgmock.patch(pgmock.table('my_table'),
                     [('2017, 6, 14', 10000, None)],
                     ['c1::timestamp', 'c2::bigint', 'c3::timestamp'])

# Render the patched SQL so that we can execute it
sql = pgmock.sql(query, patch)
print(sql)
```

```
SELECT * FROM  (VALUES ('2017, 6, 14'::timestamp,10000::bigint,null::timestamp)) AS
↪my_table(c1,c2,c3)
```

Note: Type hints can only be used on python strings, floats, and ints. In other words, if you use a "timestamp" type, a string must be used as the value instead of a datetime object. Otherwise a `ColumnTypeError` will be raised.

3.5.2 Testing Postgres Arrays

Postgres arrays can be modeled in pgmock but cannot be passed in as python lists instead they must be strings in the Postgres array syntax. The syntax is similar to python except for curly brackets. If it's a text array then each string should be surrounded by double quotes. Remember to cast the field as the correct array type (ex. ::text[] or integer[]).

```
# Create a patch for the table. The patch takes a selector, rows (a list of lists for
↪each column), and column names
patch = pgmock.patch(pgmock.table('my_table'),
                     [('2017, 6, 14', 10000, '{"apple", "iphone"}')],
                     ['c1::timestamp', 'c2::bigint', 'c3::text[]'])

# Render the patched SQL so that we can execute it
sql = pgmock.sql(query, patch)
print(sql)
```

```
SELECT * FROM  (VALUES ('2017, 6, 14'::timestamp,10000::bigint,'{"apple", "iphone"}
↪'::text[])) AS my_table(c1,c2,c3)
```

3.5.3 Filling in Meaningless Columns with nulls

Sometimes the logic in a query only depends on the values of a couple columns and it isn't necessary to provide values for all of the other columns. `pgmock` allows users to ignore providing values for columns and fills in the empty values with null. Below is an example that illustrates how to do this when passing in rows to `pgmock.patch`.

```
# Create a query that returns many columns
query = "SELECT c1, c2, c3, c4, c5 from test_table where c1 = 'value'"

# When patching out the table, only provide values for "c1" since we're testing the
↪filtering of the select
patch = pgmock.patch(pgmock.table('test_table'), [('value', ), ('not_filtered', )], [
↪'c1', 'c2', 'c3', 'c4', 'c5'])

# Render the patched SQL. All other values for columns will be null
sql = pgmock.sql(query, patch)
print(sql)
```

```
SELECT c1, c2, c3, c4, c5 from  (VALUES ('value',null,null,null,null),('not_filtered',
↪null,null,null,null)) AS test_table(c1,c2,c3,c4,c5) where c1 = 'value'
```

```
# The patch can also take a list of dictionaries that only specifies which column␣
↪values to use. This is another
# way to fill in meaningless values with nulls
patch = pgmock.patch(pgmock.table('test_table'), [{'c1': 'value'}, {'c1': 'not_
↪filtered'}], ['c1', 'c2', 'c3', 'c4', 'c5'])

# Render the patched SQL. All other values for columns will be null
sql = pgmock.sql(query, patch)
print(sql)
```

```
SELECT c1, c2, c3, c4, c5 from  (VALUES ('value',null,null,null,null),('not_filtered',
↪null,null,null,null)) AS test_table(c1,c2,c3,c4,c5) where c1 = 'value'
```

```
# Only one row should have matched the filter
results = list(test_engine.execute(sql))
assert len(results) == 1
```

```
# Be sure to stop the testing DB for this tutorial
test_db.stop()
test_engine.dispose()
```

3.5.4 Configuring for Accuracy and Performance

`pgmock` comes with a configuration module (`pgmock.config`) that can be used to set flags that aid in accuracy of selectors / patching. These flags come with accuracy/performance hits that should be known before using them.

Safe Mode

`pgmock` searches SQL with regular expressions. Regular expressions can get tripped up whenever special characters appear in comments of SQL or in string literals. For example, `-- this is my comment; hello!` will mess up `pgmock.statement` since it splits the SQL by the semicolon character. Turning on **safe mode** will search a pre-formatted version of the supplied SQL that is stripped of comments and string literals.

Safe mode can be turned on with `pgmock.config.set_safe_mode`. By default, it is set to `False` because it incurs a major performance hit when using it. Safe mode doesn't have to be configured globally with `pgmock.config.set_safe_mode`. It can be passed to `pgmock.sql` or `pgmock.sql_file` as an argument. It can also be used in a context manager so that it is only set during the duration of execution like so:

```
with pgmock.config.set_safe_mode(True):
    # Run SQL that cant natively be searched by pgmock because of regex issues
    ...
```

It's recommended to pass it as an argument to `pgmock.sql` when needing to be modified. For configuring it in a pytest fixture, one can use the context manager like so:

```
import pytest

@pytest.fixture(scope='module')
def use_safe_mode():
```

```
    with pgmock.config.set_safe_mode(True):
        yield
```

Replace New Patch Aliases

Since `pgmock` turns expressions into `VALUES` expressions when patching, it is not always possible to preserve the original name of what's being patched. For example, `SELECT * from schema.table_name` is impossible to patch as `SELECT * FROM (VALUES ...) AS schema.table_name` since `schema.table_name` is not a valid alias.

When this case happens in the case of the `pgmock.table` selector, `pgmock` will make an alias as the table name and then replace any refences to the old table name.

For example, `SELECT schema.table_name.col FROM schema.table_name` would be replaced with `SELECT table_name.col FROM (VALUES ...) AS table_name(...)`. Note that this only matters when the full schema and table name is used to reference columns.

By default, this mode is turned on. To turn it off globally, call `pgmock.config.set_replace_new_patch_aliases(False)`. Similar to `pgmock.config.set_safe_mode`, this function can be used as a context manager or in a pytest fixture. It can also be given as an argument to `pgmock.mock` since SQLAlchemy will use this style of selects by default when making a `sqlalchemy.insert` object from a table with a schema.

If users are never using the full schema and table name when referencing columns, it is safe to turn this option off and will improve `pgmock.table` selector performance by about 20%.

3.6 Using pgmock with pytest

The examples above illustrate programmatically making a test database and running assertions. For examples of how to use `pgmock` with pytest, check out the test_examples.py file in `pgmock`. This file shows how to use `pgmock` with pytest-pgsql. An example of using the context manager and reading from a SQL file is provided.

CHAPTER 4

Interface

4.1 pgmock

Primary `pgmock` interface

`pgmock.`**`create_table_as`**(*table*)

Obtains a selector for a `CREATE TABLE AS` statement.

Searches for `CREATE TABLE table_name(optional columns) AS` and returns the entire statement. The body of the statement (e.g. the `SELECT` or anything after `CREATE TABLE AS`) can be returned by chaining the `body()` selector.

Parameters **`table`** (*str*) – The name of the table as referenced in the expression

Returns A chainable SQL selector.

Return type Selector

Raises

- *NoMatchError* – When the expression cannot be found during rendering.
- *MultipleMatchError* – When multiple expressions are found during rendering.

Examples

Obtain the `CREATE TABLE AS` of table "t"

```
ctas = pgmock.sql(sql_string, pgmock.create_table_as('t'))
```

Obtain the body of the `CREATE TABLE AS` of table "t"

```
ctas_body = pgmock.sql(sql_string, pgmock.create_table_as('t').body())
```

Note: When patching `CREATE TABLE AS` statements, the entire body of the statement is patched with a `SELECT * FROM VALUES ... AS pgmock(columns...)`. This is because it is illegal to do `VALUES ... AS ...` after a "create table as" statement.

pgmock.**cte**(*alias*)

Obtains a selector for a common table expression (CTE)

CTEs are matched by searching for a `WITH cte_name AS` or for searching for a CTE after a comma (e.g `WITH cte_name1 AS ..., cte_name2 AS ...`)

Parameters **alias** (*str*) – The alias of the CTE

Returns A chainable SQL selector

Return type Selector

Raises

- *NoMatchError* – When the CTE cannot be found during rendering.
- *MultipleMatchError* – When multiple CTEs are found during rendering.
- *NestedMatchError* – When nested subquery matches are found during rendering.
- *InvalidSQLError* – When enclosing parentheses for a CTE cannot be found.

Examples

Obtain the CTE that has the alias "a"

```
cte = pgmock.sql(sql_string, pgmock.cte('a'))
```

pgmock.**data**(*rows=None*, *cols=None*)

Creates patch data for a side effect.

Parameters

- **rows** (*List[tuple], optional*) – A list of tuples of values to patch for each row. Each row must have the same length. If `None`, defaults to an empty list.
- **cols** (*List[str]*) – A list of columns.

Returns A data object that can be used as input to a side effect of a patch, for example `pgmock.patch(side_effect=[pgmock.data(rows=..., cols=...)])`

Return type Data

pgmock.**mock**(*connectable*, *replace_new_patch_aliases=None*)

Creates a mock selector that can be patched.

This is intended to be used as a context manager with a given SQLAlchemy connectable (e.g. an engine, session, connection, etc). For example:

```
with pgmock.mock(engine) as mocker:
    mocker.patch(pgmock.table('my_table'), rows, cols)
    mocker.patch(pgmock.table('other_table'), rows, cols)

    # Run SQLAlchemy queries...
```

```
    # Assert the mocker was rendered with as many queries executed
    assert len(mocker.renderings) == num_expected_queries
```

Any queries executed inside of the context manager will be patched by SQLAlchemy's `before_cursor_execute` event. Renderings of patched SQL can be obtained by examining the `renderings` property of the object, which is a list of tuples of the original and modified SQL of every query.

If any of the patches cannot be matched during query execution, the relevant exceptions are raised. Specific patches can be applied to specific queries by using the `side_effect` argument of *pgmock.patch*.

Parameters

- **connectable** (*SQLAlchemy connectable object*) – The connectable SQLAlchemy object (e.g engine, session, connection, etc)
- **replace_new_patch_aliases** (*bool, optional*) – If `True`, will replace any references to patch aliases when they differ from the original alias. If `None`, uses the globally-configured value that defaults to `True`. More information on this can be found at *pgmock.config.set_replace_new_patch_aliases*

Returns A chainable mock object that can be patched.

Return type Mock

Raises Any error that can happen during rendering.

pgmock.**patch**(*selector=None*, *rows=None*, *cols=None*, *side_effect=None*)

Applies a patch to a selector.

Parameters

- **selector** (*Selector, optional*) – relevant SQL.
- **rows** (*List[tuple], optional*) – A list of tuples of values to patch for each row. Each row must have the same length. If `None`, patching is ignored.
- **cols** (*List[str]*) – A list of columns. If more columns are provided than the length of the rows, `null` values are filled in for the missing values.
- **side_effect** (*List[pgmock.data]*) – A list of side effects. Side effects can only be provided when `rows` and `cols` are not provided. Each side effect is rendered on each subsequent rendering of the patch. Side effects must be instantiated with `pgmock.data` and the arguments are `rows` and `cols`. Note: providing `None` as a side effect will ignore the patch for the rendering.

Returns A chainable mock object that can be patched.

Return type Mock

Raises *UnpatchableError* – When the selector cannot be patched

Examples

Patch a table "schema.table_name" with values

```
patch = pgmock.patch(pgmock.table('schema.table_name'),
                     rows=[(1, 2), (3, 4)],
                     cols=['a', 'b'])
patched_query = pgmock.sql(sql_string, patch)
```

Patch a table "schema.table_name" with a side effect while using SQLAlchemy

```
with pgmock.mock(connetion) as mocker:
    mocker.patch(pgmock.table('schema.table_name'),
                 side_effect=[
                     None,
                     pgmock.data([(1, 2), (3, 4)], ['a', 'b'])
                 ])
    # Do no patching on the first execution of the SQLAlchemy
    # connection since the side effect returns ``None`` the
    # first time
    connection.execute(...)
    # Now apply the patch the second time
    connection.execute(...)
```

pgmock.**sql**(*query*, **selectors*, *safe_mode=None*)

Renders SQL from a query and selector.

Parameters

- **query** (*str*) – The SQL query
- ***selectors** (*Selector*) – The selector(s) of the query to render. If multiple selectors are provided as positional arguments, they are automatically chained together.
- **safe_mode** (*bool, optional*) – If `True`, used stripped SQL when using selectors. This has a performance hit but can result in more accurate searching. If `None`, defaults to the globally configured value (which defaults to `False`). More information on this can be viewed at *pgmock.config.set_safe_mode*.

Returns The rendered SQL

Return type str

Raises Any exceptions that can be thrown by the selector during rendering

Examples

Render the SQL from a subquery aliased with "a"

```
subquery_sql = pgmock.sql(sql_string, pgmock.subquery('a'))
```

pgmock.**sql_file**(*file_name*, **selectors*, *safe_mode=None*)

Renders SQL from a sql file and selector.

Parameters

- **file_name** (*str*) – The SQL file name
- ***selectors** (*Selector*) – The selector(s) of the query to render. If multiple selectors are provided as positional arguments, they are automatically chained together.
- **safe_mode** (*bool, optional*) – If `True`, used stripped SQL when using selectors. This has a performance hit but can result in more accurate searching. If `None`, defaults to the globally configured value (which defaults to `False`). More information on this can be viewed at *pgmock.config.set_safe_mode*.

Returns The rendered SQL

Return type str

Raises Any exceptions that can be thrown by the selector during rendering

Examples

Render the SQL from a file that has a subquery aliased with "a"

```
subquery_sql = pgmock.sql_file(sql_file_path, pgmock.subquery('a'))
```

pgmock.**statement**(*start*, *end=None*)

Obtains a statement selector.

Statements are naively parsed by splitting SQL based on the semicolon. If any semicolons exist in the comments or literal strings, this selector has undefined behavior.

Parameters

- **start** (*int*) – The starting statement. If `end` is `None`, obtain a single statement
- **end** (*int, optional*) – The ending statement (exclusive)

Returns A chainable SQL selector.

Return type Selector

Raises *StatementParseError* – When the statement range is invalid for the parsed statements.

Examples

Obtain the first statement in a SQL string

```
statement = pgmock.sql(sql_string, pgmock.statement(0))
```

Obtain the second and third statements in a SQL string

```
statement = pgmock.sql(sql_string, pgmock.statement(1, 3))
```

pgmock.**insert_into**(*table*)

Obtains a selector for an `INSERT INTO` expression.

Searches for `INSERT INTO table_name(optional columns)` and returns the entire statement. The body of the statement (e.g. the `SELECT` or anything after `INSERT INTO`) can be returned by chaining the `body()` selector.

Parameters **table** (*str*) – The table of the expression.

Returns A chainable SQL selector.

Return type Selector

Raises

- *NoMatchError* – When the expression cannot be found during rendering.
- *MultipleMatchError* – When multiple expressions are found during rendering.

Examples

Obtain the `INSERT INTO` of table "t"

```
insert_into = pgmock.sql(sql_string, pgmock.insert_into('t'))
```

Obtain the body of the `INSERT INTO` of table "t"

```
insert_into_body = pgmock.sql(sql_string, pgmock.insert_into('t').body())
```

Note: When patching `INSERT INTO` statements, the entire body of the statement after the `INSERT INTO` is patched

pgmock.**subquery**(*alias*)

Obtains a selector for a subquery

Subqueries are matched by an alias preceeded by an enclosing parenthesis. Once matched, the SQL is search for the starting parenthesis.

Parameters **alias** (*str*) – The alias of the subquery.

Returns A chainable SQL selector.

Return type Selector

Raises

- *NoMatchError* – When the expression cannot be found during rendering.
- *MultipleMatchError* – When multiple expressions are found during rendering.
- *NestedMatchError* – When nested subquery matches are found during rendering.
- *InvalidSQLError* – When enclosing parentheses for a subquery cannot be found.

Examples

Obtain the subquery that has the alias "a"

```
subquery = pgmock.sql(sql_string, pgmock.subquery('a'))
```

Todo:

- Support for subqueries without an alias (e.g. after an "in" keyword)

pgmock.**table**(*name*, *alias=None*)

Obtains a selector for a table

Tables are matched by searching for their name and optional aliases after a `FROM` or `JOIN` keyword. If the table has an alias but the alias isn't provided, a *NoMatchError* will be thrown.

Parameters

- **name** (*str*) – The name of the table (including the schema if in the query)
- **alias** (*str, optional*) – The alias of the table if it exists

Returns A chainable SQL selector.

Return type Selector

Raises

- *NoMatchError* – When the expression cannot be found during rendering.
- *MultipleMatchError* – When multiple expressions are found during rendering.

Examples

Obtain the table with no alias that has the name "schema.table_name"

```
table = pgmock.sql(sql_string, pgmock.table('schema.table_name'))
```

Obtain the table with the name "schema.table_name" that has the alias "a"

```
table = pgmock.sql(sql_string, pgmock.table('schema.table_name', 'a'))
```

Todo:

- Support lateral joins and other joins that have keywords after the `JOIN` keyword

4.2 pgmock.config

Configuration functions for `pgmock`

`pgmock.config.`**`get_safe_mode`**`()`
Returns the configured safe mode

`pgmock.config.`**`set_safe_mode`**`(`*safe_mode*`)`
Sets whether safe mode is turned on or off in `pgmock`.

If `safe_mode` is `True`, all selectors will be applied to a stripped version of SQL that excludes comments and string literals. This improves the accuracy of `pgmock` matching but comes with a performance hit.

Safe mode is set to `False` by default.

Examples

```
# Set the global setting
pgmock.config.set_safe_mode(True)

# Only set the configuration while in use of the
# context manager. Revert it back to the original
# value when the context manager exits
with pgmock.config.set_safe_mode(False):
    ...
```

`pgmock.config.`**`get_replace_new_patch_aliases`**`()`
Returns the configured replacement of new patch aliases

`pgmock.config.`**`set_replace_new_patch_aliases`**`(`*replace_new_patch_aliases*`)`
Sets whether new patch aliases should be replaced in SQL when found

Since `pgmock` turns expressions into `VALUES` expressions when patching, it is not always possible to preserve the original name of what's being patched.

If the name of the expression being patched cannot be used as a valid patch alias (e.g. a table with a schema name in it), this setting ensures that all references to the new patch alias will be updated.

This setting primarily applies to SQL in this style:

```
SELECT schema.table_name.col from schema.table_name
```

When patching `schema.table_name` with a `VALUES` list, it is impossible to alias the `VALUES` list with `schema.table_name` since `.` is an invalid alias character. To get around this, `pgmock` makes the alias of the `VALUES` list be the table name without the schema name. When this setting is turned on, it will ensure that the `SELECT schema.table_name.col` will also be valid. The patch will look like this when the setting is on:

```
SELECT table_name.col from (VALUES(...) AS table_name)
```

and it will look like this when off:

```
SELECT schema.table_name.col from (VALUES(...) AS table_name)
```

The latter example is invalid SQL, so this setting should be turned on if the full schema name is present when referencing columns.

Note: This setting incurs a performance overhead (10-20% slower depending on the SQL length) only when using the `pgmock.table` selector.

This setting does a global search and replace on the query. In the example above, it would replace every instance of `schema.table_name.` with `table_name.`. Keep this in mind as it could potentially have adverse side effects on other SQL that might reference the schema and table name followed by a period.

This setting does not handle the case when double quotes are used to reference anything (e.g. `"schema"."table_name"."col"`).

Examples

```
# Set the global setting
pgmock.config.set_replace_new_patch_aliases(True)

# Only set the configuration while in use of the
# context manager. Revert it back to the original
# value when the context manager exits
with pgmock.config.set_replace_new_patch_aliases(False):
    ...
```

CHAPTER 5

Exceptions

These are all of the custom exceptions thrown by `pgmock`. The descriptions of most of the exceptions talk about common scenarios that can cause the exceptions to be raised.

exception `pgmock.exceptions.`**ColumnMismatchInPatchError**
: Thrown when creating a patch with a list of dictionaries where the dictionary keys don't match with the column names provided

  For example, this code will throw this error because the "col1" column is being specified in the row data of a patch but not the columns:

```
pgmock.patch(pgmock.table('table'), rows=[{'col1': 'value'}], cols=['col2'])
```

exception `pgmock.exceptions.`**ColumnsNeededForPatchError**
: Thrown when columns are required for patching values.

  It's possible to use `pgmock.patch` without providing columns. This is standard for patching anything that takes a `VALUES` list without associated column names (e.g insert into `VALUES (...)`). However, it's illegal to not provide column names for patching expressions that require them.

  This error is thrown when trying to patch an expression that has a name or an alias associated with it and not providing the column names. For example, `pgmock.patch(pgmock.table('table'), [rows])` or patching a subquery without providing columns would throw this.

  This error is also thrown when trying to use lists of dictionaries as rows to `pgmock.patch` without providing column names

exception `pgmock.exceptions.`**Error**
: The base exception for `pgmock`

exception `pgmock.exceptions.`**InvalidSQLError**
: Thrown when invalid SQL is loaded with pgmock.

  This error is thrown when no matches are found for enclosing parentheses. For example, matching a CTE with the following SQL:

```
WITH cte_name AS (SELECT * from table
```

would produce this error since there is no matching right paren for the initial left paren.

exception pgmock.exceptions.**MultipleMatchError**
Thrown when multiple matches are rendered.

This error happens when a selector finds multiple occurrences of a SQL expression and it is either rendered or has other selectors chained to it.

For example, say your SQL is:

```
CREATE TABLE a AS ( SELECT * FROM t1 ) ;
CREATE TABLE a AS ( SELECT * FROM t2 ) ;
```

Doing:

```
pgmock.sql(sql, pgmock.create_table_as('a'))
```

Will result in a *MultipleMatchError* since multiple `CREATE TABLE AS` expressions were found and `pgmock` tried to obtain the SQL for it. One must refine the selection with list syntax to choose which one is rendered like so:

```
pgmock.sql(sql, pgmock.create_table_as('a')[0])
```

The above will return the SQL for the first `CREATE TABLE AS` expression.

The same situation holds true when chaining selectors. A selector cannot be chained to one that results in multiple matches. For example, the following selector is invalid for use in any pgmock function (including *pgmock.patch*):

```
pgmock.create_table_as('a').body()
```

Note: It is possible to select multiple occurences of some expressions when patching them (such as tables). This only holds true for selectors given to *pgmock.patch* like so:

```
pgmock.sql(sql, pgmock.patch(pgmock.create_table_as('a'), values))
```

---

exception pgmock.exceptions.**NestedMatchError**
Thrown when a selector selects a nested pattern

For example, imagine we have the following SQL:

```
SELECT * FROM (
    SELECT * FROM (
        SELECT * FROM test_table
    ) bb
) bb
```

The following code will produce this error:

```
pgmock.sql(sql, pgmock.subquery('bb'))
```

This is because subqueries (along with CTEs) can have nested patterns, and it is ambiguous how pgmock should patch or select them

exception pgmock.exceptions.**NoConnectableError**
Thrown when using a mock as a context manager with no connectable.

exception pgmock.exceptions.**NoMatchError**
Thrown when parsing an expression and not finding a match.

This error commonly happens when using a selector that takes a name (such as `pgmock.subquery('subquery_name')`) and not being able to find a match for the given name.

Please read the docs for the selector that you are using and check that you have provided all of the proper arguments first. For example, selecting a table that has an alias requires also passing its alias or this error will be raised.

If you are certain that your selector is referencing a valid name in your query, contact the authors or open up an issue at https://github.com/CloverHealth/pgmock with the entire exception message provided and code.

exception pgmock.exceptions.**PatchError**
Top-level patching error

exception pgmock.exceptions.**SQLParseError**
Top-level error thrown when parsing SQL expressions

exception pgmock.exceptions.**SelectorChainingError**
Thrown when trying to chain together selectors that can't be chained.

`pgmock` allows selectors to be chained into one single selector like so:

```
selector = pgmock.patch(...).patch(...)
sql = pgmock.sql(my_sql_string, selector)
```

Sometimes it isn't always feasible to chain together multiple selectors, so pgmock allows multiple selectors to be passed to `pgmock.sql` like so:

```
sql = pgmock.sql(my_sql_string, pgmock.patch(...), pgmock.patch(...))
```

The syntax from the latter is equivalent to the syntax from the former.

This exception is raised when using the syntax from the latter example and using selectors that are impossible to be chained together. For example, pgmock doesn't allow this selector to be constructed:

```
pgmock.patch(...).subquery(...)
```

The above isn't allowed because patches effectively stop any other selectors from further refining the view of SQL being rendered.

This error is raised when an invalid chain such as the one from above is passed to `pgmock.sql` or `pgmock.sql_file` using multiple selectors.

exception pgmock.exceptions.**SideEffectExhaustedError**
Thrown when using a side effect on a patch and the iterable has been exhausted.

This is thrown when a `side_effect` has been provided for a patch, but the number of queries executed has surpassed the number of results in the side effect. For example, this code would cause this exception:

```
with pgmock.mock(connectable):
    # Provide exactly one side effect result
    pgmock.patch(pgmock.table('table'),
                 side_effect=[pgmock.data(rows, cols)])

    # The first query will run fine since the side effect has one value
    run_code()

    # The second query will fail because it will try to use the second
```

```
    # value in the side effect
    run_code()
```

exception pgmock.exceptions.**StatementParseError**
Thrown when statements cannot be parsed.

This happens when trying to obtain a statement in a SQL query and the index is out of the bounds of the number of statements.

Keep in mind that when using *pgmock.statement* that the first argument is the index of the first statement starting at 0.

exception pgmock.exceptions.**UnpatchableError**
Thrown when an expression cannot be patched.

This error is thrown when trying to patch SQL that is not patchable or currently not supported by `pgmock`. Since patching is only applicable to expressions that can be translated into Postgres `VALUES` statements, sometimes it is not possible to patch an expression.

For example, trying to patch the first two statements of a query with `pgmock.patch(pgmock.statement(0, 2), ...)` would throw this error since it is not possible to patch two entire statements.

exception pgmock.exceptions.**ValueSerializationError**
Thrown when a Python value cannot be serialized to a postgres `VALUES` value.

pgmock supports serializing the following Python types: bool, float, int, str, dict (json), UUID, datetime, date, time (all time types support timezones).

If the python type being serialized doesn't match, one must supply a column type hint when patching values. Open an issue on pgmock or contact the authors in order to support serializing other types!

pgmock.exceptions.**throw**(*exception*, *msg*, *sql=None*)
Throws an exception with an error message that points to exception docs

CHAPTER 6

Known Issues and Future Work

pgmock has quite a few limitations and issues, some that will be addressed and some things that are not possible to implement because of limitations in the postgres VALUES expressions. These are discussed here along with future work to address issues and make pgmock better.

6.1 General Parsing Issues

pgmock relies on regular expressions to speedily find and patch relevant SQL expressions. This has limitations with parsing nested parentheses and pgmock implements this with custom code.

Using special characters/words in string literals and comments can cause issues for the regular expressions. For example, this would cause issues when parsing statements since a semicolon is in the comments:

```
STATEMENT 1;
-- I'm a comment that has a semicolon ; in it
STATEMENT 2;
```

For now, pgmock addresses this issue by allowing *safe* mode to be turned on (*pgmock.config.set_safe_mode*) or by passing safe_mode=True to *pgmock.sql* or *pgmock.sql_file*. This mode will strip all SQL of any string literals or comments to make searching more accurate, but it comes at a cost of performance. In some cases for really large SQL, performance can be impacted by 50 - 100X slowdowns in pgmock's selector performance. While this time is still rather small in the context of a test that hits a database, it should be taken into account when using safe mode.

6.2 Other Known Issues

Here's a short list of some other known issues:

- Subqueries without an alias are currently not supported (e.g. an IN expression). Future versions of pgmock plan to address this.

- Joins where another keyword comes after the join are currently not supported (e.g. `JOIN LATERAL`), but future versions plan to address this.
- Using the `replace_new_patch_aliases` mode (see *pgmock.config.set_replace_new_patch_aliases*) will not work when using double quotes around selected columns (e.g. `SELECT "schema"."table"."column" FROM ...`)

CHAPTER 7

Release Notes

```
1.3.7
-----

* INC-1 Triggered the build

1.3.6
-----

* INC-1 Update the deployment deploy\_requirements.txt

1.3.5
-----

* Add support for rendering a python list and tuple as postgres Array (#5)
* Fix column name of reserved keywords causes syntax error (#4)

1.3.4
-----

* Add UUID type to \_to\_sql\_value() (#3)

1.3.2
-----

* [DI-8] Temple update 2018-07-30 (#2)

1.3.1
-----

* Delete old README.md file

1.3.0
-----

* Initial open source release (#1)
```

```
* first commit
```

CHAPTER 8

Contributing Guide

This project was created using temple. For more information about temple, go to the Temple docs.

8.1 Setup

Set up your development environment with:

```
git clone git@github.com:CloverHealth/pgmock.git
cd pgmock
make setup
```

`make setup` will setup a virtual environment managed by pyenv and install dependencies.

Note that if you'd like to use something else to manage dependencies other than pyenv, call `make dependencies` instead of `make setup`.

8.2 Testing and Validation

Run the tests with:

```
make test
```

Validate the code with:

```
make validate
```

8.3 Documentation

Sphinx documentation can be built with:

```
make docs
```

The static HTML files are stored in the `docs/_build/html` directory. A shortcut for opening them is:

```
make open_docs
```

8.4 Releases and Versioning

Anything that is merged into the master branch will be automatically deployed to PyPI. Documentation will be published to ReadTheDocs.

The following files will be generated and should *not* be edited by a user:

- `ChangeLog` - Contains the commit messages of the releases. Please have readable commit messages in the master branch and squash and merge commits when necessary.
- `AUTHORS` - Contains the contributing authors.
- `version.py` - Automatically updated to include the version string.

This project uses Semantic Versioning through PBR. This means when you make a commit, you can add a message like:

```
sem-ver: feature, Added this functionality that does blah.
```

Depending on the sem-ver tag, the version will be bumped in the right way when releasing the package. For more information, about PBR, go the the PBR docs.

Python Module Index

p

Index