Basic examples

In this section, we demonstrate how to use some of the very basic features of PostgreSQL using the classic PyGreSQL interface.

Creating a connection to the database

We start by creating a **connection** to the PostgreSQL database:

```
>>> from pg import DB
>>> db = DB()
```

If you pass no parameters when creating the **DB** instance, then PyGreSQL will try to connect to the database on the local host that has the same name as the current user, and also use that name for login.

You can also pass the database name, host, port and login information as parameters when creating the **DB** instance:

```
>>> db = DB(dbname='testdb', host='pgserver', port=5432,
...     user='scott', passwd='tiger')
```

The **DB** class of which `db` is an object is a wrapper around the lower level **Connection** class of the **pg** module. The most important method of such connection objects is the `query` method that allows you to send SQL commands to the database.

Creating tables

The first thing you would want to do in an empty database is creating a table. To do this, you need to send a **CREATE TABLE** command to the database. PostgreSQL has its own set of built-in types that can be used for the table

columns. Let us create two tables “weather” and “cities”:

```
>>> db.query("""CREATE TABLE weather (
...     city varchar(80),
...     temp_lo int, temp_hi int,
...     prcp float8,
...     date date)""")
>>> db.query("""CREATE TABLE cities (
...     name varchar(80),
...     location point)""")
```

Note: Keywords are case-insensitive but identifiers are case-sensitive.

You can get a list of all tables in the database with:

```
>>> db.get_tables()
['public.cities', 'public.weather']
```

Insert data

Now we want to fill our tables with data. An **INSERT** statement is used to insert a new row into a table. There are several ways you can specify what columns the data should go to.

Let us insert a row into each of these tables. The simplest case is when the list of values corresponds to the order of the columns specified in the CREATE TABLE command:

```
>>> db.query("""INSERT INTO weather
...     VALUES ('San Francisco', 46, 50, 0.25, '11/27/1994')""")
>>> db.query("""INSERT INTO cities
...     VALUES ('San Francisco', '(-194.0, 53.0)')""")
```

You can also specify the columns to which the values correspond. The columns can be specified in any order. You may also omit any number of columns, such as with unknown precipitation, below:

```
>>> db.query("""INSERT INTO weather (date, city, temp_hi, temp_lo)
```

```
...     VALUES ('11/29/1994', 'Hayward', 54, 37)""")
```

If you get errors regarding the format of the date values, your database is probably set to a different date style. In this case you must change the date style like this:

```
>>> db.query("set datestyle = MDY")
```

Instead of explicitly writing the INSERT statement and sending it to the database with the **DB.query()** method, you can also use the more convenient **DB.insert()** method that does the same under the hood:

```
>>> db.insert('weather',
...     date='11/29/1994', city='Hayward', temp_hi=54, temp_lo=37)
```

And instead of using keyword parameters, you can also pass the values to the **DB.insert()** method in a single Python dictionary.

If you have a Python list with many rows that shall be used to fill a database table quickly, you can use the **DB.inserttable()** method.

Retrieving data

After having entered some data into our tables, let's see how we can get the data out again. A **SELECT** statement is used for retrieving data. The basic syntax is:

```
SELECT columns FROM tables WHERE predicates
```

A simple one would be the following query:

```
>>> q = db.query("SELECT * FROM weather")
>>> print(q)
    city     |temp_lo|temp_hi|prcp|   date
-------------+-------+-------+----+----------
San Francisco|     46|     50|0.25|1994-11-27
Hayward      |     37|     54|    |1994-11-29
```

```
(2 rows)
```

You may also specify expressions in the target list. (The 'AS column' specifies the column name of the result. It is optional.)

```
>>> print(db.query("""SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date
...     FROM weather"""))
    city     |temp_avg|   date
-------------+--------+----------
San Francisco|      48|1994-11-27
Hayward      |      45|1994-11-29
(2 rows)
```

If you want to retrieve rows that satisfy certain condition (i.e. a restriction), specify the condition in a WHERE clause. The following retrieves the weather of San Francisco on rainy days:

```
>>> print(db.query("""SELECT * FROM weather
...     WHERE city = 'San Francisco' AND prcp > 0.0"""))
    city     |temp_lo|temp_hi|prcp|   date
-------------+-------+-------+----+----------
San Francisco|     46|     50|0.25|1994-11-27
(1 row)
```

Here is a more complicated one. Duplicates are removed when DISTINCT is specified. ORDER BY specifies the column to sort on. (Just to make sure the following won't confuse you, DISTINCT and ORDER BY can be used separately.)

```
>>> print(db.query("SELECT DISTINCT city FROM weather ORDER BY city"))
    city
-------------
Hayward
San Francisco
(2 rows)
```

So far we have only printed the output of a SELECT query. The object that is returned by the query is an instance of the **Query** class that can print itself in the nicely formatted way we saw above. But you can also retrieve the results as a list of tuples, by using the **Query.getresult()** method:

```
>>> from pprint import pprint
>>> q = db.query("SELECT * FROM weather")
>>> pprint(q.getresult())
[('San Francisco', 46, 50, 0.25, '1994-11-27'),
 ('Hayward', 37, 54, None, '1994-11-29')]
```

Here we used pprint to print out the returned list in a nicely formatted way.

If you want to retrieve the results as a list of dictionaries instead of tuples, use the **Query.dictresult()** method instead:

```
>>> pprint(q.dictresult())
[{'city': 'San Francisco',
  'date': '1994-11-27',
  'prcp': 0.25,
  'temp_hi': 50,
  'temp_lo': 46},
 {'city': 'Hayward',
  'date': '1994-11-29',
  'prcp': None,
  'temp_hi': 54,
  'temp_lo': 37}]
```

Finally, you can also retrieve the results as a list of named tuples, using the **Query.namedresult()** method. This can be a good compromise between simple tuples and the more memory intensive dictionaries:

```
>>> for row in q.namedresult():
...     print(row.city, row.date)
...
San Francisco 1994-11-27
Hayward 1994-11-29
```

If you only want to retrieve a single row of data, you can use the more convenient **DB.get()** method that does the same under the hood:

```
>>> d = dict(city='Hayward')
>>> db.get('weather', d, 'city')
>>> pprint(d)
{'city': 'Hayward',
 'date': '1994-11-29',
```

```
 'prcp': None,
 'temp_hi': 54,
 'temp_lo': 37}
```

As you see, the **DB.get()** method returns a dictionary with the column names as keys. In the third parameter you can specify which column should be looked up in the WHERE statement of the SELECT statement that is executed by the **DB.get()** method. You normally don't need it when the table was created with a primary key.

Retrieving data into other tables

A SELECT ... INTO statement can be used to retrieve data into another table:

```
>>> db.query("""SELECT * INTO TEMPORARY TABLE temptab FROM weather
...     WHERE city = 'San Francisco' and prcp > 0.0""")
```

This fills a temporary table "temptab" with a subset of the data in the original "weather" table. It can be listed with:

```
>>> print(db.query("SELECT * from temptab"))
    city     |temp_lo|temp_hi|prcp|   date
-------------+-------+-------+----+----------
San Francisco|     46|     50|0.25|1994-11-27
(1 row)
```

Aggregates

Let's try the following query:

```
>>> print(db.query("SELECT max(temp_lo) FROM weather"))
max
---
 46
(1 row)
```

You can also use aggregates with the GROUP BY clause:

```
>>> print(db.query("SELECT city, max(temp_lo) FROM weather GROUP BY city
    city     |max
-------------+---
Hayward      | 37
San Francisco| 46
(2 rows)
```

Joining tables

Queries can access multiple tables at once or access the same table in such a way that multiple instances of the table are being processed at the same time.

Suppose we want to find all the records that are in the temperature range of other records. W1 and W2 are aliases for weather. We can use the following query to achieve that:

```
>>> print(db.query("""SELECT W1.city, W1.temp_lo, W1.temp_hi,
...     W2.city, W2.temp_lo, W2.temp_hi FROM weather W1, weather W2
...     WHERE W1.temp_lo < W2.temp_lo and W1.temp_hi > W2.temp_hi"""))
 city  |temp_lo|temp_hi|    city     |temp_lo|temp_hi
-------+-------+-------+-------------+-------+-------
Hayward|     37|     54|San Francisco|     46|     50
(1 row)
```

Now let's join two different tables. The following joins the "weather" table and the "cities" table:

```
>>> print(db.query("""SELECT city, location, prcp, date
...     FROM weather, cities
...     WHERE name = city"""))
    city     |location |prcp|   date
-------------+---------+----+----------
San Francisco|(-194,53)|0.25|1994-11-27
(1 row)
```

Since the column names are all different, we don't have to specify the table name. If you want to be clear, you can do the following. They give identical results, of course:

```
>>> print(db.query("""SELECT w.city, c.location, w.prcp, w.date
```

```
...     FROM weather w, cities c WHERE c.name = w.city"""))
    city     |location |prcp|   date
-------------+---------+----+----------
San Francisco|(-194,53)|0.25|1994-11-27
(1 row)
```

Updating data

It you want to change the data that has already been inserted into a database table, you will need the **UPDATE** statement.

Suppose you discover the temperature readings are all off by 2 degrees as of Nov 28, you may update the data as follow:

```
>>> db.query("""UPDATE weather
...     SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
...     WHERE date > '11/28/1994'""")
'1'
>>> print(db.query("SELECT * from weather"))
    city     |temp_lo|temp_hi|prcp|   date
-------------+-------+-------+----+----------
San Francisco|     46|     50|0.25|1994-11-27
Hayward      |     35|     52|    |1994-11-29
(2 rows)
```

Note that the UPDATE statement returned the string `'1'`, indicating that exactly one row of data has been affected by the update.

If you retrieved one row of data as a dictionary using the **`DB.get()`** method, then you can also update that row with the **`DB.update()`** method.

Deleting data

To delete rows from a table, a **DELETE** statement can be used.

Suppose you are no longer interested in the weather of Hayward, you can do the following to delete those rows from the table:

```
>>> db.query("DELETE FROM weather WHERE city = 'Hayward'")
```

```
'1'
```

Again, you get the string `'1'` as return value, indicating that exactly one row of data has been deleted.

You can also delete all the rows in a table by doing the following. This is different from DROP TABLE which removes the table itself in addition to the removing the rows, as explained in the next section.

```
>>> db.query("DELETE FROM weather")
'1'
>>> print(db.query("SELECT * from weather"))
city|temp_lo|temp_hi|prcp|date
----+-------+-------+----+----
(0 rows)
```

Since only one row was left in the table, the DELETE query again returns the string `'1'`. The SELECT query now gives an empty result.

If you retrieved a row of data as a dictionary using the **`DB.get()`** method, then you can also delete that row with the **`DB.delete()`** method.

Removing the tables

The **DROP TABLE** command is used to remove tables. After you have done this, you can no longer use those tables:

```
>>> db.query("DROP TABLE weather, cities")
>>> db.query("select * from weather")
pg.ProgrammingError: Error:  Relation "weather" does not exist
```