Python PostgreSQL

last modified July 6, 2020

Python PostgreSQL tutorial with psycopg2 module shows how to program PostgreSQL databases in Python with psycopg2 module.

PostgreSQL

PostgreSQL is a powerful, open source object-relational database system. It is a multi-user database management system. It runs on multiple platforms including Linux, FreeBSD, Solaris, Microsoft Windows and Mac OS X. PostgreSQL is developed by the PostgreSQL Global Development Group.

The psycopg2 module

There are several Python libraries for PostgreSQL. language. In this tutorial we use the psycopg2 module. It is a PostgreSQL database adapter for the Python programming language. It is mostly implemented in C as a libpq wrapper.

```
$ pip install psycopg2
```

We install the psycopg2 module.

Python psycopg2 version example

In the first code example, we get the version of the PostgreSQL database.

version.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2
import sys
```

```
con = None

try:

    con = psycopg2.connect(database='testdb', user='postgres',
        password='s$cret')

    cur = con.cursor()
    cur.execute('SELECT version()')

    version = cur.fetchone()[0]
    print(version)

except psycopg2.DatabaseError as e:

    print(f'Error {e}')
    sys.exit(1)

finally:

    if con:
        con.close()
```

In the program we connect to the previously created testdb database. We execute an SQL statement which returns the version of the PostgreSQL database.

```
import psycopg2
```

The psycopg2 is a Python module which is used to work with the PostgreSQL database.

```
con = None
```

We initialize the con variable to None. In case we could not create a connection to the database (for example the disk is full), we would not have a connection variable defined. This would lead to an error in the finally clause.

```
con = psycopg2.connect(database='testdb', user='postgres',
    password='s$cret')
```

The connect() method creates a new database session and returns a connection object. The user was created without a password. On localhost, we can omit the password option. Otherwise, it must be specified.

```
cur = con.cursor()
cur.execute('SELECT version()')
```

From the connection, we get the cursor object. The cursor is used to traverse the records from the result set. We call the execute() method of the cursor and execute the SQL statement.

```
version = cur.fetchone()[0]
```

We fetch the data. Since we retrieve only one record, we call the fetchone() method.

```
print(version)
```

We print the data that we have retrieved to the console.

```
except psycopg2.DatabaseError as e:

    print(f'Error {e}')
    sys.exit(1)
```

In case of an exception, we print an error message and exit the program with an error code 1.

```
finally:

    if con:
        con.close())
```

In the final step, we release the resources.

```
$ version.py
PostgreSQL 11.1, compiled by Visual C++ build 1914, 64-bit
```

This is the output.

In the second example, we again get the version of the PostgreSQL database. This time we use the with keyword.

version2.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2


con = psycopg2.connect(database='testdb', user='postgres',
    password='s$cret')
```

```
with con:

    cur = con.cursor()
    cur.execute('SELECT version()')

    version = cur.fetchone()[0]
    print(version)
```

The program returns the current version of the PostgreSQL database. With the use of the with keyword. The code is more compact.

```
with con:
```

With the with keyword, Python automatically releases the resources. It also provides error handling.

Python psycopg2 execute

We create the cars table and insert several rows to it. The execute() executes a database operation (query or command).

create_table.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2

con = psycopg2.connect(database='testdb', user='postgres',
                       password='s$cret')

with con:

    cur = con.cursor()

    cur.execute("DROP TABLE IF EXISTS cars")
    cur.execute("CREATE TABLE cars(id SERIAL PRIMARY KEY, name VARCHAR(255), price INT
    cur.execute("INSERT INTO cars(name, price) VALUES('Audi', 52642)")
    cur.execute("INSERT INTO cars(name, price) VALUES('Mercedes', 57127)")
    cur.execute("INSERT INTO cars(name, price) VALUES('Skoda', 9000)")
    cur.execute("INSERT INTO cars(name, price) VALUES('Volvo', 29000)")
    cur.execute("INSERT INTO cars(name, price) VALUES('Bentley', 350000)")
    cur.execute("INSERT INTO cars(name, price) VALUES('Citroen', 21000)")
    cur.execute("INSERT INTO cars(name, price) VALUES('Hummer', 41400)")
    cur.execute("INSERT INTO cars(name, price) VALUES('Volkswagen', 21600)")
```

The program creates the cars table and inserts eight rows into the table.

```
cur.execute("CREATE TABLE cars(id SERIAL PRIMARY KEY, name VARCHAR(20), price INT)")
```

This SQL statement creates a new cars table. The table has three columns.

```
cur.execute("INSERT INTO cars(name, price) VALUES('Audi', 52642)")
cur.execute("INSERT INTO cars(name, price) VALUES('Mercedes', 57127)")
```

These two lines insert two cars into the table.

```
$ psql -U postgres testdb
psql (11.1)
Type "help" for help.

testdb=# SELECT * FROM cars;
    id |    name    | price
----+------------+--------
    1 | Audi       |  52642
    2 | Mercedes   |  57127
    3 | Skoda      |   9000
    4 | Volvo      |  29000
    5 | Bentley    | 350000
    6 | Citroen    |  21000
    7 | Hummer     |  41400
    8 | Volkswagen |  21600
(8 rows)
```

We verify the written data with the psql tool.

Python psycopg2 executemany

The executemany() method is a convenience method for launching a database operation (query or command) against all parameter tuples or mappings found in the provided sequence. The function is mostly useful for commands that update the database: any result set returned by the query is discarded.

execute_many.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2

cars = (
    (1, 'Audi', 52642),
    (2, 'Mercedes', 57127),
    (3, 'Skoda', 9000),
    (4, 'Volvo', 29000),
    (5, 'Bentley', 350000),
```

```
    (6, 'Citroen', 21000),
    (7, 'Hummer', 41400),
    (8, 'Volkswagen', 21600)
)

con = psycopg2.connect(database='testdb', user='postgres',
                       password='s$cret')

with con:

    cur = con.cursor()

    cur.execute("DROP TABLE IF EXISTS cars")
    cur.execute("CREATE TABLE cars(id SERIAL PRIMARY KEY, name VARCHAR(255), price INT

    query = "INSERT INTO cars (id, name, price) VALUES (%s, %s, %s)"
    cur.executemany(query, cars)

    con.commit()
```

This example drops the cars table if it exists and (re)creates it.

```
cur.execute("DROP TABLE IF EXISTS cars")
cur.execute("CREATE TABLE cars(id SERIAL PRIMARY KEY, name VARCHAR(255), price INT)")
```

The first SQL statement drops the cars table if it exists. The second SQL statement creates the cars table.

```
query = "INSERT INTO cars (id, name, price) VALUES (%s, %s, %s)"
```

This is the query that we use.

```
cur.executemany(query, cars)
```

We insert eight rows into the table using the convenience executemany() method. The first parameter of this method is a parameterized SQL statement. The second parameter is the data, in the form of a tuple of tuples.

Python psycopg2 last inserted row id

The psycopg2 does not support the lastrowid attribute. To return the id of the last inserted row, we have to use PostgreSQL's RETURNING id clause.

lastrowid.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2

con = psycopg2.connect(database='testdb', user='postgres',
                       password='s$cret')

with con:

    cur = con.cursor()

    cur.execute("DROP TABLE IF EXISTS words")
    cur.execute("CREATE TABLE words(id SERIAL PRIMARY KEY, word VARCHAR(255))")
    cur.execute("INSERT INTO words(word) VALUES('forest') RETURNING id")
    cur.execute("INSERT INTO words(word) VALUES('cloud') RETURNING id")
    cur.execute("INSERT INTO words(word) VALUES('valley') RETURNING id")

    last_row_id = cur.fetchone()[0]

    print(f"The last Id of the inserted row is {last_row_id}")
```

The program creates a new words table and prints the Id of the last inserted row.

```
$ lastrowid.py
The last Id of the inserted row is 3
```

This is the output.

Python psycopg2 fetchall

The fetchall() fetches all the (remaining) rows of a query result, returning them as a list of tuples. An empty list is returned if there is no more record to fetch.

fetch_all.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2

con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM cars")
```

```
    rows = cur.fetchall()

    for row in rows:
        print(f"{row[0]} {row[1]} {row[2]}")
```

In this example, we retrieve all data from the cars table.

```
cur.execute("SELECT * FROM cars")
```

This SQL statement selects all data from the cars table.

```
rows = cur.fetchall()
```

The fetchall() method gets all records. It returns a result set. Technically, it is a tuple of tuples. Each of the inner tuples represents a row in the table.

```
for row in rows:
    print(f"{row[0]} {row[1]} {row[2]}")
```

We print the data to the console, row by row.

```
$ fetch_all.py
1 Audi 52642
2 Mercedes 57127
3 Skoda 9000
4 Volvo 29000
5 Bentley 350000
6 Citroen 21000
7 Hummer 41400
8 Volkswagen 21600
```

This is the output of the example.

Python psycopg2 fetchone

The fetchone() returns the next row of a query result set, returning a single tuple, or None when no more data is available.

fetchone.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2

con = psycopg2.connect(database='testdb', user='postgres',
```

```
                    password='s$cret')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM cars")

    while True:

        row = cur.fetchone()

        if row == None:
            break

        print(f"{row[0]} {row[1]} {row[2]}")
```

In this example we connect to the database and fetch the rows of the cars table one by one.

```
while True:
```

We access the data from the while loop. When we read the last row, the loop is terminated.

```
row = cur.fetchone()

if row == None:
    break
```

The fetchone() method returns the next row from the table. If there is no more data left, it returns None. In this case we break the loop.

```
print(f"{row[0]} {row[1]} {row[2]}")
```

The data is returned in the form of a tuple. Here we select records from the tuple. The first is the Id, the second is the car name and the third is the price of the car.

Python psycopg2 dictionary cursor

The default cursor retrieves the data in a tuple of tuples. With a dictionary cursor, the data is sent in a form of Python dictionaries. We can then refer to the data by their column names.

dictionary_cursor

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2
import psycopg2.extras

con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

with con:

    cursor = con.cursor(cursor_factory=psycopg2.extras.DictCursor)
    cursor.execute("SELECT * FROM cars")

    rows = cursor.fetchall()

    for row in rows:
        print(f"{row['id']} {row['name']} {row['price']}")
```

In this example, we print the contents of the cars table using the dictionary cursor.

```
import psycopg2.extras
```

The dictionary cursor is located in the extras module.

```
cursor = con.cursor(cursor_factory=psycopg2.extras.DictCursor)
```

We create a DictCursor.

```
for row in rows:
    print(f"{row['id']} {row['name']} {row['price']}")
```

The data is accessed by the column names. The column names are folded to lowercase in PostgreSQL (unless quoted) and are case sensitive. Therefore, we have to provide the column names in lowercase.

Python psycopg2 parameterized queries

When we use parameterized queries, we use placeholders instead of directly writing the values into the statements. Parameterized queries increase security and performance.

The Python psycopg2 module supports two types of placeholders: ANSI C printf format and the Python extended format.

parameterized_query.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2

con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

uId = 1
uPrice = 62300

con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

with con:

    cur = con.cursor()
    cur.execute("UPDATE cars SET price=%s WHERE id=%s", (uPrice, uId))

    print(f"Number of rows updated: {cur.rowcount}")
```

We update a price of one car. In this code example, we use the question mark placeholders.

```
cur.execute("UPDATE cars SET price=%s WHERE id=%s", (uPrice, uId))
```

The characters (%s) are placeholders for values. The values are added to the placeholders.

```
print(f"Number of rows updated: {cur.rowcount}")
```

The rowcount property returns the number of updated rows. In our case one row was updated.

```
$ parameterized_query.py
Number of rows updated: 1

testdb=> SELECT * FROM cars WHERE id=1;
 id | name | price
----+------+-------
  1 | Audi | 62300
(1 row)
```

The price of the car was updated. We check the change with the psql tool.

The second example uses parameterized statements with Python extended format.

parameterized_query2.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2

uid = 3

con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

with con:

    cur = con.cursor()

    cur.execute("SELECT * FROM cars WHERE id=%(id)s", {'id': uid } )

    row = cur.fetchone()

    print(f'{row[0]} {row[1]} {row[2]}')
```

We select a name and a price of a car using pyformat parameterized statement.

```
cur.execute("SELECT * FROM cars WHERE id=%(id)s", {'id': uid } )
```

The named placeholders start with a colon character.

```
$ parameterized_query2.py
3 Skoda 9000
```

This is the output.

Python psycopg2 mogrify

The mogrify is a psycopg2 extension to the Python DB API that returns a query string after arguments binding. The returned string is exactly the one that would be sent to the database running the execute() method or similar.

mogrify.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2

con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')
```

```
cur = None

with con:

    cur = con.cursor()

    print(cur.mogrify("SELECT name, price FROM cars WHERE id=%s", (2,)))

    # cur.execute("SELECT name, price FROM cars WHERE id=%s", (2,) )
    # row = cur.fetchone()
    # print(f"{row[0]} {row[1]}")
```

The program shows a SELECT query string after binding the arguments with mogrify().

```
$ mogrify.py
b'SELECT name, price FROM cars WHERE id=2'
```

This is the output.

Python psycopg2 insert image

In this section we are going to insert an image to the PostgreSQL database.

```
testdb=> CREATE TABLE images(id SERIAL PRIMARY KEY, data BYTEA);
```

For this example, we create a new table called images. For the images, we use the BYTEA data type. It allows to store binary strings.

insert_image.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2
import sys

def readImage():

    fin = None

    try:
        fin = open("sid.jpg", "rb")
        img = fin.read()
        return img

    except IOError as e:

        print(f'Error {e.args[0]}, {e.args[1]}')
        sys.exit(1)
```

```
    finally:

        if fin:
            fin.close()

con = None

try:
    con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

    cur = con.cursor()
    data = readImage()
    binary = psycopg2.Binary(data)
    cur.execute("INSERT INTO images(data) VALUES (%s)", (binary,))

    con.commit()

except psycopg2.DatabaseError as e:

    if con:
        con.rollback()

    print(f'Error {e}')
    sys.exit(1)

finally:

    if con:
        con.close()
```

In the program, we read an image from the current working directory and write it into the images table of the PostgreSQL testdb database.

```
try:
    fin = open("sid.jpg", "rb")
    img = fin.read()
    return img
```

We read binary data from the filesystem. We have a JPEG image called sid.jpg.

```
binary = psycopg2.Binary(data)
```

The data is encoded using the psycopg2 Binary object.

```
cur.execute("INSERT INTO images(data) VALUES (%s)", (binary,))
```

This SQL statement is used to insert the image into the database.

Python psycopg2 read image

In this section, we are going to perform the reverse operation. We read an image from the database table.

read_image.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2
import sys

def writeImage(data):

    fout = None

    try:
        fout = open('sid2.jpg', 'wb')
        fout.write(data)

    except IOError as e:

        print(f"Error {0}")
        sys.exit(1)

    finally:

        if fout:
            fout.close()


try:
    con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

    cur = con.cursor()
    cur.execute("SELECT data FROM images LIMIT 1")
    data = cur.fetchone()[0]

    writeImage(data)

except psycopg2.DatabaseError as e:

    print(f'Error {e}')
    sys.exit(1)

finally:

    if con:
        con.close()
```

We read image data from the images table and write it to another file, which we call sid2.jpg.

```
try:
    fout = open('sid2.jpg', 'wb')
    fout.write(data)
```

We open a binary file in a writing mode. The data from the database is written to the file.

```
cur.execute("SELECT data FROM images LIMIT 1")
data = cur.fetchone()[0]
```

These two lines select and fetch data from the images table. We obtain the binary data from the first row.

Python PostgreSQL metadata

Metadata is information about the data in the database. Metadata in a PostgreSQL database contains information about the tables and columns, in which we store data. Number of rows affected by an SQL statement is a metadata. Number of rows and columns returned in a result set belong to metadata as well.

Metadata in PostgreSQL can be obtained using from the description property of the cursor object or from the information_schema table.

Next we print all rows from the cars table with their column names.

column_names.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2

con = psycopg2.connect(database='testdb', user='postgres',
                       password='s$cret')

with con:

    cur = con.cursor()

    cur.execute('SELECT * FROM cars')

    col_names = [cn[0] for cn in cur.description]
    rows = cur.fetchall()
```

```
    print(f'{col_names[0]} {col_names[1]} {col_names[2]}')
```

We print the contents of the cars table to the console. Now, we include the names of the columns too.

```
col_names = [cn[0] for cn in cur.description]
```

We get the column names from the description property of the cursor object.

```
print(f'{col_names[0]} {col_names[1]} {col_names[2]}')
```

This line prints three column names of the cars table.

We print the rows using the for loop. The data is aligned with the column names.

```
$ column_names.py
id name price
```

This is the output.

In the following example we list all tables in the testdb database.

list_tables.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2

con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

with con:

    cur = con.cursor()
    cur.execute("""SELECT table_name FROM information_schema.tables
        WHERE table_schema = 'public'""")

    rows = cur.fetchall()

    for row in rows:
        print(row[0])
```

The code example prints all available tables in the current database to the terminal.

```
cur.execute("""SELECT table_name FROM information_schema.tables
    WHERE table_schema = 'public'""")
```

The table names are stored inside the system information_schema table.

```
$ list_tables.py
cars
countries
projects
employees
users
tasks
images
```

These were the tables on our system.

Python psycopg2 export and import of data

We can export and import data using copy_to() and copy_from().

copy_to.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2
import sys

con = None
fout = None

try:

    con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

    cur = con.cursor()
    fout = open('cars.csv', 'w')
    cur.copy_to(fout, 'cars', sep="|")

except psycopg2.DatabaseError as e:

    print(f'Error {e}')
    sys.exit(1)

except IOError as e:

    print(f'Error {e}')
    sys.exit(1)

finally:
```

```
    if con:
        con.close()

    if fout:
        fout.close()
```

The code example copies data from the cars table into the cars.csv file.

```
fout = open('cars.csv', 'w')
```

We open a file where we write the data from the cars table.

```
cur.copy_to(fout, 'cars', sep="|")
```

The copy_to method copies data from the cars table to the opened file. The columns are separated with the | character.

```
$ cat cars.csv
2|Mercedes|57127
3|Skoda|9000
4|Volvo|29000
5|Bentley|350000
6|Citroen|21000
7|Hummer|41400
8|Volkswagen|21600
1|Audi|62300
```

These are the contents of the cars file.

Now we are going to perform a reverse operation. We import the dumped table back into the database table.

```
testdb=> DELETE FROM cars;
DELETE 8
```

We delete the data from the cars table.

copy_from.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2
import sys

con = None
f = None
```

```
try:

    con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

    cur = con.cursor()
    f = open('cars.csv', 'r')

    cur.copy_from(f, 'cars', sep="|")
    con.commit()

except psycopg2.DatabaseError as e:

    if con:
        con.rollback()

    print(f'Error {e}')
    sys.exit(1)

except IOError as e:

    if con:
        con.rollback()

    print(f'Error {e}')
    sys.exit(1)

finally:

    if con:
        con.close()

    if f:
        f.close()
```

In the program we read the contents of the cars file and copy it back to the cars table.

```
f = open('cars.csv', 'r')
cur.copy_from(f, 'cars', sep="|")
con.commit()
```

We open the cars.csv file for reading and copy the contents to the cars table. The changes are committed.

```
SELECT * FROM cars;
 id |    name    | price
----+------------+--------
  2 | Mercedes   |  57127
  3 | Skoda      |   9000
  4 | Volvo      |  29000
  5 | Bentley    | 350000
```

```
   6 | Citroen    |  21000
   7 | Hummer     |  41400
   8 | Volkswagen |  21600
   1 | Audi       |  62300
(8 rows)
```

The output shows that we have successfully recreated the saved cars table.

Python psycopg2 transactions

A transaction is an atomic unit of database operations against the data in one or more databases. The effects of all the SQL statements in a transaction can be either all committed to the database or all rolled back.

In psycopg2 module transactions are handled by the connection class. The first command of a connection cursor starts a transaction. (We do not need to enclose our SQL commands by BEGIN and END statements to create a transaction. This is handled automatically by psycopg2.) The following commands are executed in the context of this new transaction. In case of an error, the transaction is aborted and no further commands are executed until the rollback() method.

The documentation to the psycopg2 module says that the connection is responsible to terminate its transaction, calling either the commit() or rollback() method. The committed changes are immediately made persistent into the database. Closing the connection using the close() method or destroying the connection object (using del or letting it fall out of scope) will result in an implicit rollback() call.

The psycopg2 module also supports an autocommit mode, where all changes to the tables are immediately effective. To run in autocommit mode, we set the autocommit property of the connection object to True.

no_commit.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2
import sys

con = None

try:
```

```
    con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

    cur = con.cursor()

    cur.execute("DROP TABLE IF EXISTS friends")
    cur.execute("CREATE TABLE friends(id SERIAL PRIMARY KEY, name VARCHAR(255))")
    cur.execute("INSERT INTO friends(name) VALUES ('Tom')")
    cur.execute("INSERT INTO friends(name) VALUES ('Rebecca')")
    cur.execute("INSERT INTO friends(name) VALUES ('Jim')")
    cur.execute("INSERT INTO friends(name) VALUES ('Robert')")

    con.commit()

except psycopg2.DatabaseError as e:

    if con:
        con.rollback()

    print('Error {e}')
    sys.exit(1)


finally:

    if con:
        con.close()
```

We create the friends table and try to fill it with data. However, as we will see, the data will be not committed.

```
#con.commit()
```

The commit() method is commented. If we uncomment the line, the data will be written to the table.

```
finally:

    if con:
        con.close()
```

The finally block is always executed. If we have not committed the changes and no error occurs (which would roll back the changes) the transaction is still opened. The connection is closed with the close() method and the transaction is terminated with an implicit call to the rollback() method.

```
testdb=# \dt
        List of relations
```

```
Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
public | cars      | table | postgres
public | countries | table | postgres
public | employees | table | postgres
public | images    | table | postgres
public | projects  | table | postgres
public | tasks     | table | postgres
public | users     | table | postgres
(7 rows)
```

Only after we have uncommented the line, the friends table is created.

Python psycopg2 autocommit

In the autocommit mode, an SQL statement is executed immediately.

autocommit.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import psycopg2
import sys

con = None

try:

    con = psycopg2.connect(database='testdb', user='postgres',
                    password='s$cret')

    con.autocommit = True

    cur = con.cursor()

    cur.execute("DROP TABLE IF EXISTS friends")
    cur.execute("CREATE TABLE friends(id serial PRIMARY KEY, name VARCHAR(10))")
    cur.execute("INSERT INTO friends(name) VALUES ('Jane')")
    cur.execute("INSERT INTO friends(name) VALUES ('Tom')")
    cur.execute("INSERT INTO friends(name) VALUES ('Rebecca')")
    cur.execute("INSERT INTO friends(name) VALUES ('Jim')")
    cur.execute("INSERT INTO friends(name) VALUES ('Robert')")
    cur.execute("INSERT INTO friends(name) VALUES ('Patrick')")


except psycopg2.DatabaseError as e:

    print(f'Error {e}')
    sys.exit(1)

finally:
```

```
    if con:
        con.close()
```

In this example, we connect to the database in the autocommit mode. We do not call neither commit() nor rollback() methods.

```
 con.autocommit = True
```

We set the connection to the autocommit mode.

```
$ autocommit.py

testdb=# select * from friends;
 id |  name
----+---------
  1 | Jane
  2 | Tom
  3 | Rebecca
  4 | Jim
  5 | Robert
  6 | Patrick
(6 rows)
```

The data was successfully committed to the friends table.

Visit Python tutorial or list all Python tutorials.