**Краткое руководство по использованию базы данных PostgreSQL совместно с Python при помощи модуля Psycopg2**

*в рамках дисциплины ОП. 08 Основы проектирования баз данных*

*для специальности 09.02.07 Информационные системы и программирование*

Оглавление

[Введение 3](#_Toc134948692)

[Установка Psycopg2 с помощью pip 3](#_Toc134948693)

[Подключение к базе данных PostgreSQL из Python 4](#_Toc134948694)

[Создание базы данных PostgreSQL с Psycopg2 4](#_Toc134948695)

[Пример кода для подключения к базе данных PostgreSQL из Python 5](#_Toc134948696)

[Разбор процесса подключения в деталях 5](#_Toc134948697)

[Создание таблицы PostgreSQL из Python 6](#_Toc134948698)

[Соответствие типов данных Python и PostgreSQL 7](#_Toc134948699)

[Выполнение CRUD-операций из Python 8](#_Toc134948700)

[Работа с датой и временем из PostgreSQL 9](#_Toc134948701)

[Получение данных из таблицы PosgreSQL 11](#_Toc134948702)

[Подготовка. 11](#_Toc134948703)

[Пример получения данных с помощью fetchall() 12](#_Toc134948704)

[Передача переменных Python в качестве параметров запроса 13](#_Toc134948705)

[Получение определенного количества строк 14](#_Toc134948706)

[Использование cursor.fetchone 15](#_Toc134948707)

[CRUD-функции в базе данных PostgreSQL на Python 16](#_Toc134948708)

[Операция Insert 16](#_Toc134948709)

[Операция Update 18](#_Toc134948710)

[Удаление строк и колонок 19](#_Toc134948711)

[Cursor.executemany() запросов нескольких строк 20](#_Toc134948712)

[**Вставка нескольких строк в таблицу PostgreSQL** 20](#_Toc134948713)

[**Обновление нескольких строк в одном запросе** 21](#_Toc134948714)

[**Удаление нескольких строк из таблицы** 22](#_Toc134948715)

[Выполнение хранимых процедур и функций PostgreSQL 23](#_Toc134948716)

[Подготовка базы данных 23](#_Toc134948717)

[Пример выполнения функции и хранимой процедуры 24](#_Toc134948718)

[Управление транзакциями в PostgreSQL 25](#_Toc134948719)

[Подготовка 26](#_Toc134948720)

[Управление транзакциями psycopg2 26](#_Toc134948721)

[Connection.autocommit 26](#_Toc134948722)

[Connection.commit 26](#_Toc134948723)

[Connection.rollback 26](#_Toc134948724)

[Пример управления транзакциями PostgreSQL из Python 26](#_Toc134948725)

[Инструкция With для управления транзакциями 28](#_Toc134948726)

[Уровни изоляции psycopg2 29](#_Toc134948727)

[Пример пула соединений в PostgreSQL с помощью Psycopg2 29](#_Toc134948728)

[Классы psycopg2 для управления пулом соединений 30](#_Toc134948729)

[AbstractConnectionPool 30](#_Toc134948730)

[SimpleConnectionPool 30](#_Toc134948731)

[ThreadedConnectionPool 30](#_Toc134948732)

[PersistentConnectionPool 30](#_Toc134948733)

[Методы psycopg2 для управления пулом соединений 31](#_Toc134948734)

[Создание пула соединений с помощью psycopg2 31](#_Toc134948735)

[Пример создания и управления пулом соединений PostgreSQL 31](#_Toc134948736)

[Создание многопоточного пула соединений 32](#_Toc134948737)

# **Введение**

Это руководство по PostgreSQL в Python описывает, как использовать модуль Psycopg2 для подключения к PostgreSQL, выполнения SQL-запросов и других операций с базой данных.

Вот список разных модулей Python для работы с сервером базы данных PostgreSQL:

* [Psycopg2](https://www.psycopg.org/docs/)
* [pg8000](https://pypi.org/project/pg8000/)
* [py-postgreql](https://pypi.org/project/py-postgresql/)
* [PyGreSQL](http://www.pygresql.org/)
* [ocpgdb](https://pypi.org/project/ocpgdb/)
* [SQLAlchemy.](https://sqlalchemy.org/) Для работы SQLAlchemy нужно, чтобы хотя бы одно из перечисленных выше решений было установлено.

**Примечание:** все модули придерживаются спецификации Python Database API Specification v2.0 (PEP 249). Этот API разработан с целью обеспечить сходство разных модулей для доступа к базам данных из Python. Другими словами, синтаксис, методы и прочее очень похожи во всех этих модулях.

В этом руководстве будем использовать Psycopg2, потому что это один из самых популярных и стабильных модулей для работы с PostgreSQL:

* Он используется в большинстве фреймворков Python и Postgres;
* Он активно поддерживается и работает как с Python 3, так и с Python 2;
* Он потокобезопасен и спроектирован для работы в многопоточных приложениях. Несколько потоков могут работать с одним подключением.

Стандартные библиотеки Python не предоставляют встроенного функционала для работы с PostgreSQL, однако есть большое количество сторонних библиотек. Наиболее популярной из них является Psycopg 2 ([официальный сайт Psycopg](https://www.psycopg.org/)). Данная библиотека реализована на языке C, благодаря чему обладает сравнительно большой производительностью.

В этом руководстве пройдемся по следующим пунктам:

* Установка Psycopg2 и использование его API для доступа к базе данных PostgreSQL;
* Вставка, получение, обновление и удаление данных в базе данных из приложения Python;
* Дальше рассмотрим управление транзакциями PostgreSQL, пул соединений и методы обработки исключений, что понадобится для разработки сложных программ на Python с помощью PostgreSQL.

## **Установка Psycopg2 с помощью pip**

Для начала нужно установить текущую версию Psycopg2 для использования PostgreSQL в Python. С помощью команды pip можно установить модуль в любую операцию систему: Windows, macOS, Linux:

* pip install psycopg2

Также можно установить конкретную версию программы с помощью такой команды:

* pip install psycopg2=2.8.6

**Примечание:** если возникает ошибка установки, например «connection error: [SSL: CERTIFICATE\_VERIFY\_FAILED] certificate verify failed (\_ssl.c:598)», то ее можно решить, сделав files.pythonhosted.org доверенным хостом:

* python -m pip install --trusted-host pypi.org --trusted-host files.pythonhosted.org --trusted-host pypi.python.org psycopg2

Модуль psycopg2 поддерживает:

* Python 2.7 и Python 3, начиная с версии 3.4.
* Сервер PostgreSQL от 7.4 до 12.
* Клиентскую библиотеку PostgreSQL от 9.1.

**Проверка установки Psycopg2**

После запуска команды должны появиться следующие сообщения:

* *Collecting psycopg2*
* *Downloading psycopg2-2.8.6*
* *Installing collected packages: psycopg2*
* *Successfully installed psycopg2-2.8.6*

При использовании anaconda подойдет следующая команда:

* conda install -c anaconda psycopg2

## **Подключение к базе данных PostgreSQL из Python**

В этом разделе рассмотрим, как подключиться к PostgreSQL из Python с помощью модуля Psycopg2.

Вот какие аргументы потребуются для подключения:

* Имя пользователя: значение по умолчанию для базы данных PostgreSQL – postgres.
* Пароль: пользователь получает пароль при установке PostgreSQL.
* Имя хоста: имя сервера или IP-адрес, на котором работает база данных. Если она запущена локально, то нужно использовать localhost или 127.0.0.0.
* Имя базы данных: в этом руководстве будем использовать базу postgres\_db.

Шаги для подключения:

* Использовать метод connect() с обязательными параметрами для подключения базы данных.
* Создать объект cursor с помощью объекта соединения, который возвращает метод connect. Он нужен для выполнения запросов.
* Закрыть объект cursor и соединение с базой данных после завершения работы.
* Перехватить исключения, которые могут возникнуть в процессе.

## **Создание базы данных PostgreSQL с Psycopg2**

Для начала создадим базу данных на сервере. Во время установки PostgreSQL вы указывали пароль, его нужно использовать при подключении.

**import psycopg2**

**from psycopg2 import Error**

**from psycopg2.extensions import ISOLATION\_LEVEL\_AUTOCOMMIT**

**try:**

**# Подключение к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432")**

**connection.set\_isolation\_level(ISOLATION\_LEVEL\_AUTOCOMMIT)**

**# Курсор для выполнения операций с базой данных**

**cursor = connection.cursor()**

**sql\_create\_database = 'create database postgres\_db'**

**cursor.execute(sql\_create\_database)**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

## **Пример кода для подключения к базе данных PostgreSQL из Python**

Для подключения к базе данных PostgreSQL и выполнения SQL-запросов нужно знать название базы данных. Ее нужно создать прежде чем пытаться выполнить подключение.

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключение к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**# Курсор для выполнения операций с базой данных**

**cursor = connection.cursor()**

**# Распечатать сведения о PostgreSQL**

**print("Информация о сервере PostgreSQL")**

**print(connection.get\_dsn\_parameters(), "\n")**

**# Выполнение SQL-запроса**

**cursor.execute("SELECT version();")**

**# Получить результат**

**record = cursor.fetchone()**

**print("Вы подключены к - ", record, "\n")**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

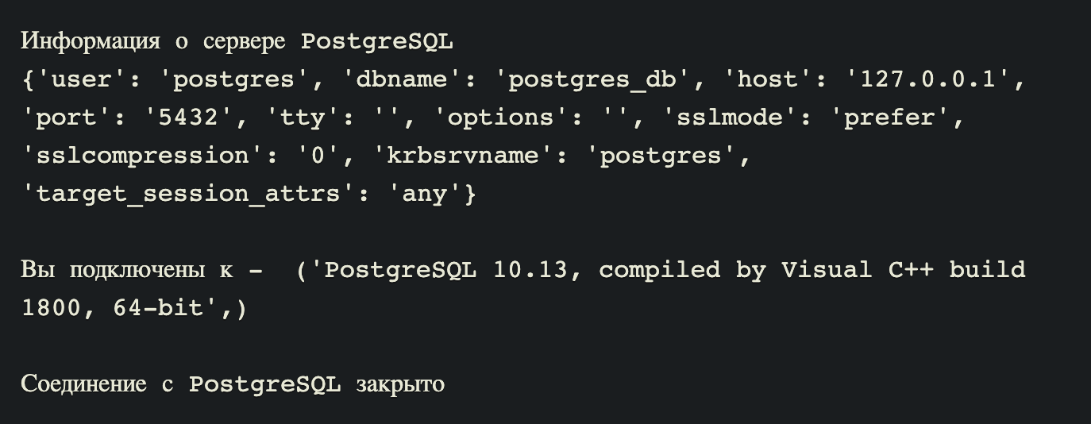
**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

После подключения появится следующий вывод:



## **Разбор процесса подключения в деталях**

**import psycopg2** — Эта строка импортирует модуль Psycopg2 в программу. С помощью классов и методов модуля можно взаимодействовать с базой.

**from psycopg2 import Error** — С помощью класса Error можно обрабатывать любые ошибки и исключения базы данных. Это сделает приложение более отказоустойчивым. Этот класс также поможет понять ошибку в подробностях. Он возвращает сообщение об ошибке и ее код.

**psycopg2.connect()** — С помощью метода connect() создается подключение к экземпляру базы данных PostgreSQL. Он возвращает объект подключения. Этот объект является потокобезопасным и может быть использован на разных потоках.

**Метод connect()** принимает разные аргументы, рассмотренные выше. В этом примере в метод были переданы следующие аргументы: user = "postgres", password = "1111", host = "127.0.0.1", port = "5432", database = "postgres\_db".

**cursor = connection.cursor()** — С базой данных можно взаимодействовать с помощью класса cursor. Его можно получить из метода cursor(), который есть у объекта соединения. Он поможет выполнять SQL-команды из Python.

Из одного объекта соединения можно создавать неограниченное количество объектов cursor. Они не изолированы, поэтому любые изменения, сделанные в базе данных с помощью одного объекта, будут видны остальным. Объекты cursor не являются потокобезопасными.

После этого выведем свойства соединения с помощью **connection.get\_dsn\_parameters()**.

**cursor.execute()** — С помощью метода execute объекта cursor можно выполнить любую операцию или запрос к базе данных. В качестве параметра этот метод принимает SQL-запрос. Результаты запроса можно получить с помощью fetchone(), fetchmany(), fetchall().

В этом примере выполняем **SELECT version()**; для получения сведений о версии PosgreSQL.

**Блок try-except-finally** — Разместим код в блоке try-except для перехвата исключений и ошибок базы данных.

**cursor.close() и connection.close()** — Правильно всегда закрывать объекты cursor и connection после завершения работы, чтобы избежать проблем с базой данных.

## **Создание таблицы PostgreSQL из Python**

В этом разделе разберем, как создавать таблицу в PostgreSQL из Python. В качестве примера создадим таблицу Mobile.

Выполним следующие шаги:

* Подготовим запрос для базы данных
* Подключимся к PosgreSQL с помощью psycopg2.connect().
* Выполним запрос с помощью cursor.execute().
* Закроем соединение с базой данных и объект cursor.

**Теперь рассмотрим пример.**

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**# Создайте курсор для выполнения операций с базой данных**

**cursor = connection.cursor()**

**# SQL-запрос для создания новой таблицы**

**create\_table\_query = '''CREATE TABLE mobile**

**(ID INT PRIMARY KEY NOT NULL,**

**MODEL TEXT NOT NULL,**

**PRICE REAL); '''**

**# Выполнение команды: это создает новую таблицу**

**cursor.execute(create\_table\_query)**

**connection.commit()**

**print("Таблица успешно создана в PostgreSQL")**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

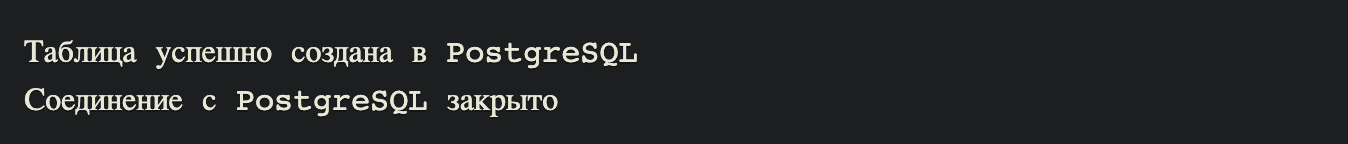
**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

Вывод:



**Примечание:** наконец, коммитим изменения с помощью метода commit().

## **Соответствие типов данных Python и PostgreSQL**

Есть стандартный маппер для конвертации типов Python в их эквиваленты в PosgreSQL и наоборот. Каждый раз при выполнении запроса PostgreSQL из Python с помощью psycopg2 результат возвращается в виде объектов Python.

|  |  |
| --- | --- |
| **Python** | **PostgreSQL** |
| None | NULL |
| bool | bool |
| float | real double |
| int long | smallint integer bigint |
| Decimal | numeric |
| str unicode | varchar text |
| date | date |
| time | time timetz |
| datetime | timestamp timestamptz |
| timedelta | interval |
| list | ARRAY |
| tuple namedtuple | Composite types IN syntax |
| dict | hstore |

**Константы и числовые преобразования**

При попытке вставить значения None и boolean (True, False) из Python в PostgreSQL, они конвертируются в соответствующие литералы SQL. То же происходит и с числовыми типами. Они конвертируются в соответствующие типы PostgreSQL.

Например, при выполнении запроса на вставку числовые объекты, такие как int, long, float и Decimal, конвертируются в числовые представления из PostgreSQL. При чтении из таблицы целые числа конвертируются в int, числа с плавающей точкой — во float*,* а десятичные — в Decimal.

## **Выполнение CRUD-операций из Python**

Таблица mobile уже есть. Теперь рассмотрим, как выполнять запросы для вставки, обновления, удаления или получения данных из таблицы в Python.

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**# Выполнение SQL-запроса для вставки данных в таблицу**

**insert\_query = """ INSERT INTO mobile (ID, MODEL, PRICE) VALUES (1, 'Iphone12', 1100)"""**

**cursor.execute(insert\_query)**

**connection.commit()**

**print("1 запись успешно вставлена")**

**# Получить результат**

**cursor.execute("SELECT \* from mobile")**

**record = cursor.fetchall()**

**print("Результат", record)**

**# Выполнение SQL-запроса для обновления таблицы**

**update\_query = """Update mobile set price = 1500 where id = 1"""**

**cursor.execute(update\_query)**

**connection.commit()**

**count = cursor.rowcount**

**print(count, "Запись успешно удалена")**

**# Получить результат**

**cursor.execute("SELECT \* from mobile")**

**print("Результат", cursor.fetchall())**

**# Выполнение SQL-запроса для удаления таблицы**

**delete\_query = """Delete from mobile where id = 1"""**

**cursor.execute(delete\_query)**

**connection.commit()**

**count = cursor.rowcount**

**print(count, "Запись успешно удалена")**

**# Получить результат**

**cursor.execute("SELECT \* from mobile")**

**print("Результат", cursor.fetchall())**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

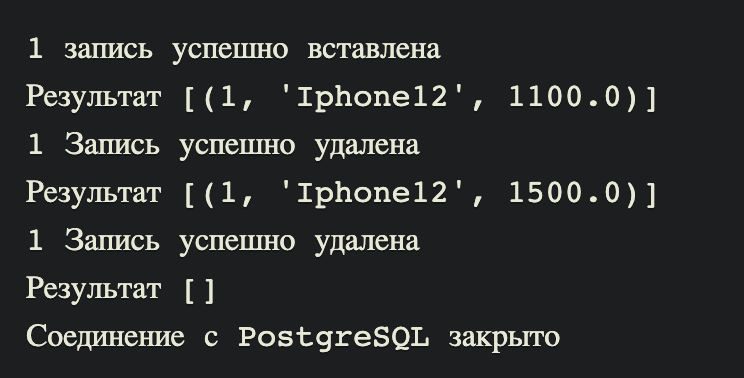
**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

Вывод:



**Примечание:** не забывайте сохранять изменения в базу данных с помощью connection.commit() после успешного выполнения операции базы данных.

## **Работа с датой и временем из PostgreSQL**

В этом разделе рассмотрим, как работать с типами date и timestamp из PostgreSQL в Python и наоборот.

Обычно при выполнении вставки объекта datetime модуль psycopg2 конвертирует его в формат timestamp PostgreSQL. По аналогии при чтении значений timestamp из таблицы PostgreSQL модуль psycopg2 конвертирует их в объекты datetime Python. Для этого примера используем таблицу Item. Выполните следующий код, чтобы подготовить таблицу.

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**# Создайте курсор для выполнения операций с базой данных**

**cursor = connection.cursor()**

**# SQL-запрос для создания новой таблицы**

**create\_table\_query = '''CREATE TABLE item (**

**item\_id serial NOT NULL PRIMARY KEY,**

**item\_name VARCHAR (100) NOT NULL,**

**purchase\_time timestamp NOT NULL,**

**price INTEGER NOT NULL**

**);'''**

**# Выполнение команды: это создает новую таблицу**

**cursor.execute(create\_table\_query)**

**connection.commit()**

**print("Таблица успешно создана в PostgreSQL")**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

Рассмотрим сценарий на простом примере. Здесь мы читаем колонку purchase\_time из таблицы и конвертируем значение в объект datetime Python.

**import psycopg2**

**import datetime**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**# Выполнение SQL-запроса для вставки даты и времени в таблицу**

**insert\_query = """ INSERT INTO item (item\_Id, item\_name, purchase\_time, price)**

**VALUES (%s, %s, %s, %s)"""**

**item\_purchase\_time = datetime.datetime.now()**

**item\_tuple = (12, "Keyboard", item\_purchase\_time, 150)**

**cursor.execute(insert\_query, item\_tuple)**

**connection.commit()**

**print("1 элемент успешно добавлен")**

**# Считать значение времени покупки PostgreSQL в Python datetime**

**cursor.execute("SELECT purchase\_time from item where item\_id = 12")**

**purchase\_datetime = cursor.fetchone()**

**print("Дата покупки товара", purchase\_datetime[0].date())**

**print("Время покупки товара", purchase\_datetime[0].time())**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

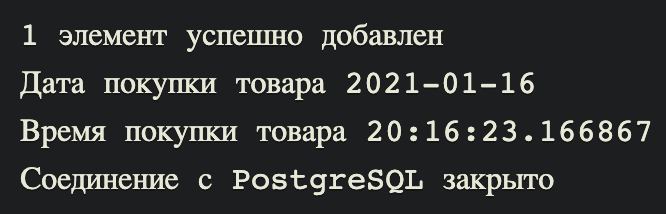
**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

Вывод:



# **Получение данных из таблицы PosgreSQL**

В этом руководстве разберем, как выполнять запрос SELECT в базе данных PostgreSQL, используя приложение на Python и Psycopg2. Также разберем, как обрабатывать и использовать результат, возвращаемый запросом.

Цели:

* Получить все строки из базы данных PostgreSQL с помощью fetchall() и ограниченное количество записей, используя fetchmany() и fetchone().
* Использовать переменные Python в операторе where для передачи динамических значений.

## **Подготовка.**

Перед началом работы нужно убедиться, что у вас есть следующее:

* Имя пользователя и пароль для подключения к PostgreSQL
* Название базы данных, из которой требуется получить данные

В этом материале воспользуемся таблицей «mobile», которая была создана в первом руководстве по работе с PostgreSQL в Python. Если таблицы нет, то ее нужно создать.

**Шаги для выполнения запроса SELECT из Python-программы**

1. Установить psycopg2 с помощью pip.
2. Создать соединение с базой данных PostgreSQL.
3. Создать инструкцию с запросом SELECT для получения данных из таблицы PostgreSQL.
4. Выполнить запрос с помощью cursor.execute() и получить результат.
5. Выполнить итерацию по объекту с помощью цикла и получить значения всех полей (колонок) базы данных для каждой строки.
6. Закрыть объекты cursor и connection.
7. Перехватить любые SQL-исключения, которые могут произойти в процессе.

**Запить тестовых данных.** Запишем несколько строк в базу данных, что бы потренироваться их получать.

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**# Выполнение SQL-запроса для вставки данных в таблицу**

**insert\_query = """ INSERT INTO mobile (ID, MODEL, PRICE) VALUES**

**(1, 'IPhone 12', 1000),**

**(2, 'Google Pixel 2', 700),**

**(3, 'Samsung Galaxy S21', 900),**

**(4, 'Nokia', 800)"""**

**cursor.execute(insert\_query)**

**connection.commit()**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

## **Пример получения данных с помощью fetchall()**

В этом примере рассмотрим, как получить все строки из таблицы:

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**postgreSQL\_select\_Query = "select \* from mobile"**

**cursor.execute(postgreSQL\_select\_Query)**

**print("Выбор строк из таблицы mobile с помощью cursor.fetchall")**

**mobile\_records = cursor.fetchall()**

**print("Вывод каждой строки и ее столбцов")**

**for row in mobile\_records:**

**print("Id =", row[0], )**

**print("Модель =", row[1])**

**print("Цена =", row[2], "\n")**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

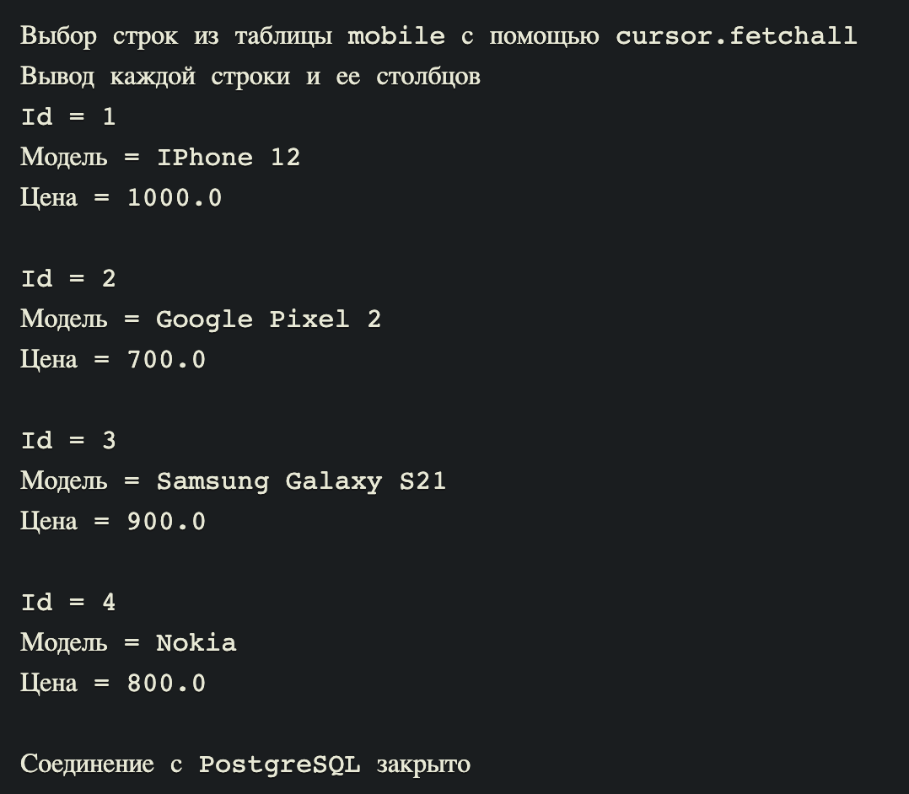
**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

Вывод:



**Примечание:** в этом примере использовалась команда cursor.fetchall() для получения всех строк из базы данных.

Используйте cursor.execute() для выполнения запроса:

* cursor.fetchall() — для всех строк.
* cursor.fetchone() — для одной.
* cursor.fetchmany(SIZE) — для определенного количества.

## **Передача переменных Python в качестве параметров запроса**

В большинстве случаев требуется передавать переменные Python в запросы SQL для получения нужного результата. Например, приложение может передать ID пользователя для получения подробностей о нем из базы данных. Для этого требуется использовать запрос с параметрами.

Запрос с параметрами — это такой запрос, где применяются заполнители (%s) на месте параметров, а значения подставляются уже во время работы программы. Таким образом эти запросы компилируются всего один раз.

**import psycopg2**

**from psycopg2 import Error**

**def get\_mobile\_details(mobile\_id):**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**postgresql\_select\_query = "select \* from mobile where id = %s"**

**cursor.execute(postgresql\_select\_query, (mobile\_id,))**

**mobile\_records = cursor.fetchall()**

**for row in mobile\_records:**

**print("Id =", row[0], )**

**print("Модель =", row[1])**

**print("Цена =", row[2])**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

**if connection:**

**cursor.close()**

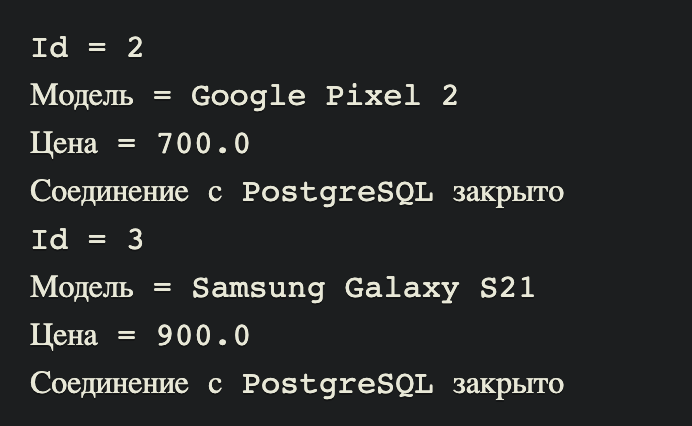
**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

**get\_mobile\_details(2)**

**get\_mobile\_details(3)**

Вывод:



## **Получение определенного количества строк**

Если таблица содержит тысячи строк, то получение всех из них может занять много времени. Но существует альтернатива в виде cursor.fetchmany().

Вот ее синтаксис:

*cursor.fetchmany([size=cursor.arraysize])*

* size — это количество строк, которые требуется получить.
* этот метод делает запрос на определенное количество строк из результата запроса. fetchmany() возвращает список кортежей, содержащих строки.
* fetchmany() возвращает пустой список, если строки не были найдены. Количество строк зависит от аргумента SIZE. Ошибка ProgrammingError возникает в том случае, если предыдущий вызов execute() не дал никаких результатов.

**fetchmany()**вернет меньше строк, если в таблице их меньше, чем было указано в аргументе SIZE.

Пример получения ограниченного количества строк из таблицы PostgreSQL с помощью **cursor.fetchmany():**

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**postgresql\_select\_query = "select \* from mobile"**

**cursor.execute(postgresql\_select\_query)**

**mobile\_records = cursor.fetchmany(2)**

**print("Вывод двух строк")**

**for row in mobile\_records:**

**print("Id =", row[0], )**

**print("Модель =", row[1])**

**print("Цена =", row[2], "\n")**

**mobile\_records = cursor.fetchmany(2)**

**print("Вывод следующих двух строк")**

**for row in mobile\_records:**

**print("Id =", row[0], )**

**print("Модель =", row[1])**

**print("Цена =", row[2], "\n")**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

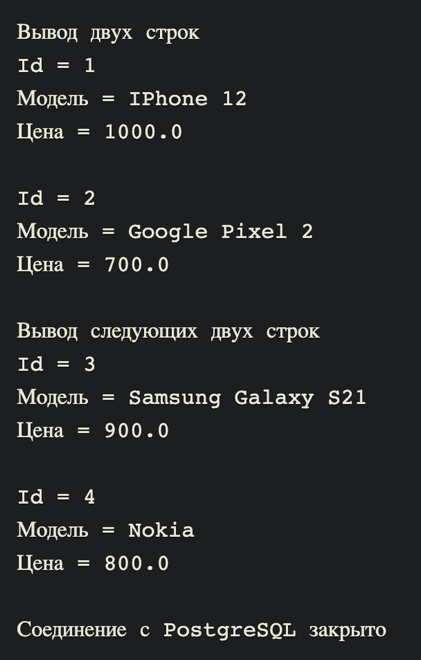
**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

Вывод:



## **Использование cursor.fetchone**

* Используйте cursor.fetchone() для получения одной строки из таблицы PostgreSQL.
* Также можно использовать этот же метод для получения следующей строки из результатов запроса.
* Он может вернуть и none, если в результате не оказалось записей
* cursor.fetchall() и fetchmany() внутри также используют этот метод.

Пример получения одной строки из базы данных PostgreSQL:

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**postgresql\_select\_query = "select \* from mobile"**

**cursor.execute(postgresql\_select\_query)**

**mobile\_records\_one = cursor.fetchone()**

**print ("Вывод первой записи", mobile\_records\_one)**

**mobile\_records\_two = cursor.fetchone()**

**print("Вывод второй записи", mobile\_records\_two)**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

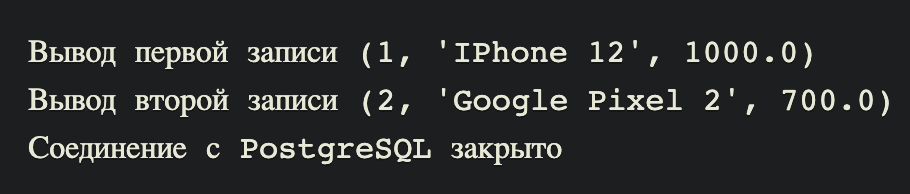
**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

Вывод:



# **CRUD-функции в базе данных PostgreSQL на Python**

В этом руководстве рассмотрим, как выполнять операции Insert, Update и Delete в базе данных PostgreSQL из Python-скриптов. Их еще называют DML-операциями. Также научимся передавать параметры в SQL-запросы.

В итоге разберем, как использовать cursor.executemany() для выполнения вставки, обновления или удаления нескольких строк в один запрос.

## **Операция Insert**

В этом разделе рассмотрим, как выполнять команду Insert для вставки одной или нескольких записей в таблицу PostgreSQL из Python с помощью Psycopg2.

Для выполнения запроса нужно сделать следующее:

* Установить psycopg2 с помощью pip.
* Установить соединение с базой данных из Python.
* Создать запрос Insert. Для этого требуется знать название таблицы и ее колонок.
* Выполнить запрос с помощью cursor.execute(). В ответ вы получите количество затронутых строк.
* После выполнения запроса нужно закоммитить изменения в базу данных.
* Закрыть объект cursor и соединение с базой данных.
* Также важно перехватить любые исключения, которые могут возникнуть в процессе.
* Наконец, можно проверить результаты, запросив данные из таблицы.

Теперь посмотрим реальный пример.

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**postgres\_insert\_query = """ INSERT INTO mobile (ID, MODEL, PRICE)**

**VALUES (%s,%s,%s)"""**

**record\_to\_insert = (5, 'One Plus 6', 950)**

**cursor.execute(postgres\_insert\_query, record\_to\_insert)**

**connection.commit()**

**count = cursor.rowcount**

**print (count, "Запись успешно добавлена в таблицу mobile")**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

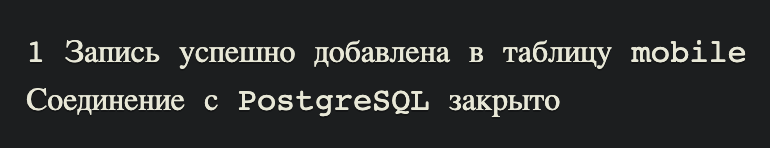
**if connection:**

**cursor.close()**

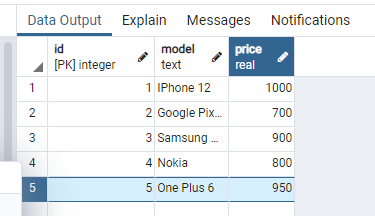
**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

Вывод:



* В этом примере использовался запрос с параметрами для передачи значений во время работы программы. А в конце изменения сохранились с помощью cursor.commit.
* С помощью запроса с параметрами можно передавать переменные python в качестве параметров на месте %s.



## **Операция Update**

В этом разделе вы узнаете, как обновлять значение в одной или нескольких колонках для одной или нескольких строк таблицы. Для этого нужно изменить запрос к базе данных.

# Задать новое значение price в строке с id для таблицы mobile

***Update*** *mobile* ***set*** *price = %s* ***where*** *id = %s*

Посмотрим на примере обновления одной строки таблицы:

**import psycopg2**

**from psycopg2 import Error**

**def update\_table(mobile\_id, price):**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**print("Таблица до обновления записи")**

**sql\_select\_query = """select \* from mobile where id = %s"""**

**cursor.execute(sql\_select\_query, (mobile\_id,))**

**record = cursor.fetchone()**

**print(record)**

**# Обновление отдельной записи**

**sql\_update\_query = """Update mobile set price = %s where id = %s"""**

**cursor.execute(sql\_update\_query, (price, mobile\_id))**

**connection.commit()**

**count = cursor.rowcount**

**print(count, "Запись успешно обновлена")**

**print("Таблица после обновления записи")**

**sql\_select\_query = """select \* from mobile where id = %s"""**

**cursor.execute(sql\_select\_query, (mobile\_id,))**

**record = cursor.fetchone()**

**print(record)**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

**if connection:**

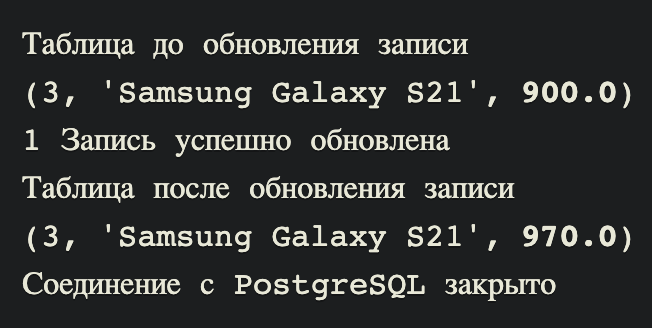
**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

**update\_table(3, 970)**

Убедимся, что обновление сработало. Вывод:



## **Удаление строк и колонок**

В этом разделе рассмотрим, как выполнять операцию удаления данных из таблицы с помощью программы на Python и Psycopg2.

# Удалить из таблицы … в строке с id …

***Delete from*** *mobile* ***where*** *id = %s*

Можно сразу перейти к примеру. Он выглядит следующим образом:

**import psycopg2**

**from psycopg2 import Error**

**def delete\_data(mobile\_id):**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**# Удаление записи**

**sql\_delete\_query = """Delete from mobile where id = %s"""**

**cursor.execute(sql\_delete\_query, (mobile\_id,))**

**connection.commit()**

**count = cursor.rowcount**

**print(count, "Запись успешно удалена")**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

**if connection:**

**cursor.close()**

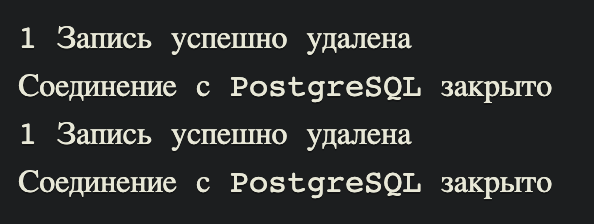
**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

**delete\_data(4)**

**delete\_data(5)**

Убедимся, что запись исчезла из таблицы.



## **Cursor.executemany() запросов нескольких строк**

**Метод cursor.executemany()** делает запрос в базу данных со всеми параметрами.

Очень часто нужно выполнить один и тот же запрос с разными данными. Например, обновить информацию о посещаемости студентов. Скорее всего, данные будут разные, но SQL останется неизменным.

**Используйте cursor.executemany() для вставки, обновления и удаления нескольких строк в один запрос.**

Синтаксис executemany():

*executemany(query, vars\_list)*

* В этом случае запросом может быть любая DML-операция (вставка, обновление, удаление).
* vars\_list — это всего лишь список кортежей, которые передаются в запрос.
* Каждый кортеж содержит одну строку для вставки или удаления.

Теперь посмотрим, как использовать этот метод.

### **Вставка нескольких строк в таблицу PostgreSQL**

Можно выполнить вставку нескольких строк с помощью SQL-запроса. Для этого используется запрос с параметрами и метод executemany().

**import psycopg2**

**from psycopg2 import Error**

**def bulk\_insert(records):**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**sql\_insert\_query = """ INSERT INTO mobile (id, model, price)**

**VALUES (%s,%s,%s) """**

**# executemany() для вставки нескольких строк**

**result = cursor.executemany(sql\_insert\_query, records)**

**connection.commit()**

**print(cursor.rowcount, "Запись(и) успешно вставлена(ы) ​​в таблицу mobile")**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

**if connection:**

**cursor.close()**

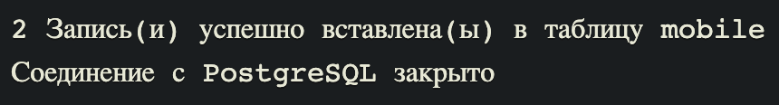
**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

**records\_to\_insert = [ (4,'LG', 800) , (5,'One Plus 6', 950)]**

**bulk\_insert(records\_to\_insert)**

Проверим результат, вернув данные из таблицы.



**Примечание:** для этого запроса был создан список записей, включающий два кортежа. Также использовались заменители. Они позволяют передать значения в запрос уже во время работы программы.

### **Обновление нескольких строк в одном запросе**

Чаще всего требуется выполнить один и тот же запрос, но с разными данными. Например, обновить зарплату сотрудников. Сумма будет отличаться, но не запрос.

Обновить несколько колонок таблицы можно с помощью cursor.executemany() и запроса с параметрами (%). Посмотрим на примере.

**import psycopg2**

**from psycopg2 import Error**

**def update\_in\_bulk(records):**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**# Обновить несколько записей**

**sql\_update\_query = """Update mobile set price = %s where id = %s"""**

**cursor.executemany(sql\_update\_query, records)**

**connection.commit()**

**row\_count = cursor.rowcount**

**print(row\_count, "Записи обновлены")**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

**if connection:**

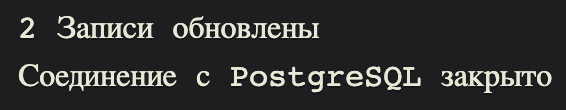
**cursor.close()**

**connection.close()**

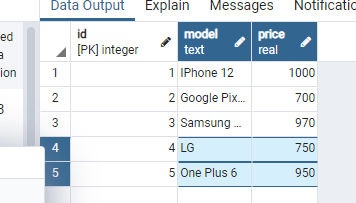
**print("Соединение с PostgreSQL закрыто")**

**update\_in\_bulk([(750, 4), (950, 5)])**

Вывод:



Проверим результат.



*Используйте cursor.rowcount, чтобы получить общее количество строк, измененных методом executemany().*

### **Удаление нескольких строк из таблицы**

В этом примере используем запрос Delete с заменителями, которые подставляют ID записей для удаления. Также есть список записей для удаления. В списке есть кортежи для каждой строки. В примере их два, что значит, что удалены будут две строки.

**import psycopg2**

**from psycopg2 import Error**

**def delete\_in\_bulk(records):**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**delete\_query = """Delete from mobile where id = %s"""**

**cursor.executemany(delete\_query, records)**

**connection.commit()**

**row\_count = cursor.rowcount**

**print(row\_count, "Записи удалены")**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

**if connection:**

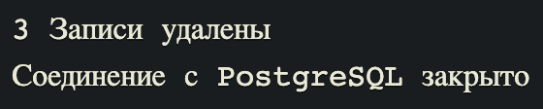
**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

**delete\_in\_bulk([(5,), (4,), (3,)])**

Убедимся, что запрос был выполнен успешно.



# **Выполнение хранимых процедур и функций PostgreSQL**

В этом руководстве рассмотрим, как выполнять хранимые процедуры и функции PostgreSQL из Python. Эти функции могут отвечать как за получение, так и за управление данными.

## **Подготовка базы данных**

Перед выполнением следующей программы убедитесь, что у вас есть следующее:

* Имя пользователя и пароль для подключения к базе данных.
* Название хранимой процедуры или функции, которые нужно выполнить.

Для этого примера была создана функция get\_production\_deployment, которая возвращает список записей о сотрудниках, участвовавших в написании кода.

**CREATE OR REPLACE FUNCTION filter\_by\_price(max\_price integer)**

**RETURNS TABLE(id INT, model TEXT, price REAL) AS $$**

**BEGIN**

**RETURN QUERY**

**SELECT \* FROM mobile where mobile.price <= max\_price;**

**END;**

**$$ LANGUAGE plpgsql;**

Теперь посмотрим, как это выполнить.

**import psycopg2**

**from psycopg2 import Error**

**def create\_func(query):**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**cursor.execute(query)**

**connection.commit()**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

**postgresql\_func = """**

**CREATE OR REPLACE FUNCTION filter\_by\_price(max\_price integer)**

**RETURNS TABLE(id INT, model TEXT, price REAL) AS $$**

**BEGIN**

**RETURN QUERY**

**SELECT \* FROM mobile where mobile.price <= max\_price;**

**END;**

**$$ LANGUAGE plpgsql;**

**"""**

**create\_func(postgresql\_func)**

Этот код добавит функцию в базу данных.

**Детали вызова функции и хранимой процедуры PostgreSQL**

Используем модуль psycopg2 для вызова функции PostgreSQL в Python. Дальше требуется выполнить следующие шаги:

* Метод connect() вернет новый объект соединения. С его помощью и можно общаться с базой.
* Создать объект Cursor с помощью соединения. Именно он позволит выполнять запросы к базе данных.
* Выполнить функцию или хранимую процедуру с помощью метода cursor.callproc(). На этом этапе нужно знать ее название, а также параметры IN и OUT. Синтаксис метода следующий.

cursor.callproc('filter\_by\_price',[IN and OUT parameters,])

Параметры следует разделить запятыми.

* Этот метод возвращает либо строки базы данных, либо количество измененных строк. Все зависит от назначения самой функции.

## **Пример выполнения функции и хранимой процедуры**

Посмотрим на демо. Процедура filter\_by\_price уже создана. В качестве параметра IN она принимает идентификатор приложения, а возвращает ID, модель и цену телефона.

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**cursor = connection.cursor()**

**# хранимая процедура**

**cursor.callproc('filter\_by\_price',[999,])**

**print("Записи с ценой меньше или равной 999")**

**result = cursor.fetchall()**

**for row in result:**

**print("Id = ", row[0], )**

**print("Model = ", row[1])**

**print("Price = ", row[2])**

**except (Exception, Error) as error:**

**print("Ошибка при работе с PostgreSQL", error)**

**finally:**

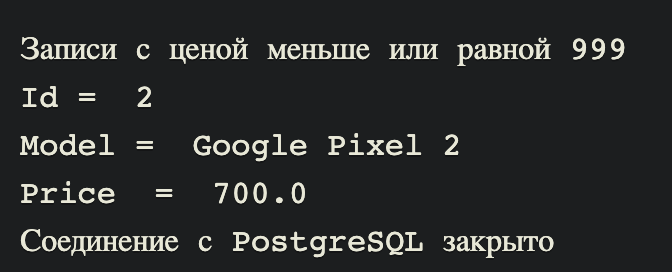
**if connection:**

**cursor.close()**

**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

После предыдущего урока у меня осталось всего 2 записи, вывод вернул одну. Вторая с ценой 1000:



После этого обрабатываем результаты с помощью fetchone. Также можно использовать методы fetchall, fetchmany() в зависимости от того, что должна вернуть функция.

Также cursor.callproc() внутри вызывает метод execute() для вызова процедуры. Так что ничего не мешает использовать его явно:

*cursor.execute("SELECT \* FROM filter\_by\_price({price});".format(price=1000))*

# **Управление транзакциями в PostgreSQL**

В этом материале рассмотрим, как управлять транзакциями PostgreSQL из Python с помощью psycopg2. Транзакция — это единица работы (unit of work), затрагивающая набор операций базы данных.

## **Подготовка**

В большинстве случаев выполнять операцию в базе данных нужно лишь после завершения некой другой операции. Например, в банковской системе перевести сумму со счета А на счет Б можно только после снятия средств со счета А.

В транзакции или все операции выполняются, или не выполняется ни одна из них. Таким образом требуется выполнить все операции в одной транзакции, чтобы она была успешной.

Вот о чем дальше пойдет речь:

* Включение и выключение режима автокомита (автосохранения).
* Транзакции для сохранения изменений в базе данных.
* Поддержка свойства ACID транзакции.
* Откат всех операций в случае неудачи.
* Изменение уровня изоляции транзакции PostgreSQL из Python.

## **Управление транзакциями psycopg2**

Транзакции PostgreSQL обрабатываются объектом соединения. Он отвечает за сохранение изменений или откат в случае неудачи.

С помощью объекта cursor выполняются операции в базе данных. Можно создать неограниченное количество объектов cursor из объекта connection. Если любая из команд объекта cursor завершается неудачно, или же транзакция отменяется, то следующие команды не выполняются вплоть до вызова метода connection.rollback().

* Объект соединения отвечает за остановку транзакции. Это можно сделать с помощью методов commit() ли rollback().
* После использования метода commit() изменения сохраняются в базе данных.
* С помощью метода rollback() можно откатить изменения.

**Примечание:** вызов метода closе() или любого другого, отвечающего за уничтожение объекта соединения, приводит к неявному вызову rollback(), вследствие чего все изменения откатываются.

### **Connection.autocommit**

По умолчанию соединение работает в режиме автоматического сохранения, то есть свойство auto-commit равно True. Это значит, что при успешном выполнении запроса изменения немедленно сохраняются в базу данных, а откат становится невозможным.

Для выполнения запросов в транзакции это свойство нужно отключить. Для этого нужно сделать connection.autocommit=False. В этом случае будет возможность откатить выполненный запрос к оригинальному состоянию в случае неудачи.

* Синтаксис autocommit:

*connection.autocommit=True or Falseс*

### **Connection.commit**

Если все операции в транзакции завершены, используйте connection.commit() для сохранения изменений в базе данных. Если метод не использовать, то все эффекты взаимодействия с данными не будут применены.

* Синтаксис commit:

*connection.commit()*

### **Connection.rollback**

Если в транзакции хотя бы одна операция завершается неудачно, то отменить изменения можно с помощью connection.rollback().

* Синтаксис rollback:

*connection.rollback()*

## **Пример управления транзакциями PostgreSQL из Python**

1. Отключите режим автосохранения (auto-commit).
2. Если все операции были выполнены успешно, используйте connection.commit() для их сохранения в базе данных.
3. Если какая-то из операций была завершена неудачно, то откатиться к последнему состоянию можно с помощью connection.rollback().

**Примечание:** транзакция остается открытой до явного вызова commit() или rollback().

Давайте понизим цену на один телефон и повысим на второй.

**import psycopg2**

**from psycopg2 import Error**

**try:**

**# Подключиться к существующей базе данных**

**connection = psycopg2.connect(user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**connection.autocommit=False**

**cursor = connection.cursor()**

**amount = 200**

**query = """select price from mobile where id = 1"""**

**cursor.execute(query)**

**record = cursor.fetchone() [0]**

**price\_a = int(record)**

**price\_a -= amount**

**# Понизить цену у первого**

**sql\_update\_query = """update mobile set price = %s where id = 1"""**

**cursor.execute(sql\_update\_query,(price\_a,))**

**query = """select price from mobile where id = 2"""**

**cursor.execute(query)**

**record = cursor.fetchone() [0]**

**price\_b = int(record)**

**price\_b += amount**

**# Повысить цену у второго**

**sql\_update\_query = """Update mobile set price = %s where id = 2"""**

**cursor.execute(sql\_update\_query, (price\_b,))**

**# совершение транзакции**

**connection.commit()**

**print("Транзакция успешно завершена")**

**except (Exception, psycopg2.DatabaseError) as error :**

**print ("Ошибка в транзакции. Отмена всех остальных операций транзакции", error)**

**connection.rollback()**

**finally:**

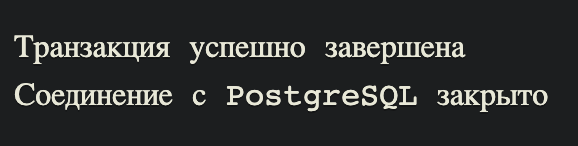
**if connection:**

**cursor.close()**

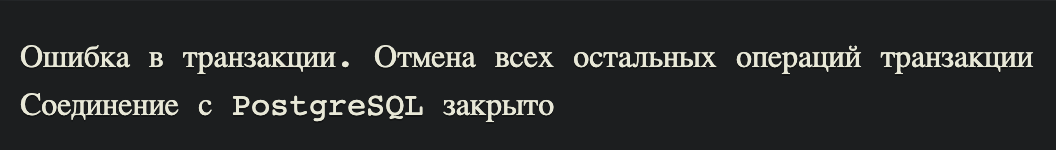
**connection.close()**

**print("Соединение с PostgreSQL закрыто")**

Вы получите следующий вывод после успешного завершения всех операций в транзакции.



Если хотя бы одна из операций будет завершена с ошибкой, то вывод будет таким.



## **Инструкция With для управления транзакциями**

Для создания транзакции внутри определенного блока в коде используйте инструкцию with.

Что делает инструкция with*?*Соединения и объекта cursor в psycopg2 — это всего лишь контекстные менеджеры, которые можно использовать с with. Основное преимущество в том, что это позволяет не писать явно commit или rollback.

* Синтаксис:

*with psycopg2.connect(connection\_arguments) as conn:*

*with conn.cursor() as cursor:*

*cursor.execute(Query)*

Когда соединение выходит из блока with, а запрос выполняется без ошибок или исключений, транзакция сохраняется. Если же в процессе возникла проблема, то транзакция откатывается.

При выходе за пределы блока соединение не закрывается, но объекты cursor и любые другие связанные объекты — да. Соединение можно использовать в более чем одном with, и каждый блок with— это всего лишь транзакция.

В этом примере выполним транзакцию в платформе для онлайн-торговки. Приобретем один товар, уменьшим баланс покупателя и переведем соответствующую сумму на счет компании.

**import psycopg2**

**from psycopg2 import Error**

**connection = psycopg2.connect(\*\*params)**

**with connection:**

**with connection.cursor() as cursor:**

**# Поиск цены товара**

**query = """select price from itemstable where itemid = 876"""**

**cursor.execute(query)**

**record = cursor.fetchone() [0]**

**item\_price = int(record)**

**# Получение остатка на счету**

**query = """select balance from ewallet where userId = 23"""**

**cursor.execute(query)**

**record = cursor.fetchone() [0]**

**ewallet\_balance = int(record)**

**new\_ewallet\_balance -= item\_price**

**# Обновление баланса с учетом расхода**

**sql\_update\_query = """Update ewallet set balance = %s where id = 23"""**

**cursor.execute(sql\_update\_query,(new\_ewallet\_balance,))**

**# Зачисление на баланс компании**

**query = """select balance from account where accountId = 2236781258763"""**

**cursor.execute(query)**

**record = cursor.fetchone()**

**account\_balance = int(record)**

**new\_account\_balance += item\_price**

**# Обновление счета компании**

**sql\_update\_query = """Update account set balance = %s where id = 132456"""**

**cursor.execute(sql\_update\_query, (new\_account\_balance,))**

**print("Транзакция успешно завершена")**

## **Уровни изоляции psycopg2**

В системах баз данных с помощью уровней изоляции можно определить какой уровень целостности транзакции будет виден остальным пользователям и системам.

Например, когда пользователь выполняет действие или операцию, которые еще не завершены, подробности о них доступны другим пользователям, которые могут выполнить последовательные действия. Например, потребитель приобретает товар. Информация об этом становится известна другим системам, что нужно для подготовки счета, квитанции и подсчетов скидки, что ускоряет весь процесс. Если уровень изоляции низкий, множество пользователей может получить доступ к одним и тем же данным одновременно. Однако это же может привести и к некоторым проблемам вплоть до утраченных обновлений. Поэтому нужно учитывать все. Более высокий уровень изоляции может заблокировать других пользователей или транзакции до завершения.

**psycopg2.extensions предоставляет следующие уровня изоляции:**

* psycopg2.extensions.ISOLATION\_LEVEL\_AUTOCOMMIT,
* psycopg2.extensions.ISOLATION\_LEVEL\_READ\_UNCOMMITTED,
* psycopg2.extensions.ISOLATION\_LEVEL\_READ\_COMMITTED,
* psycopg2.extensions.ISOLATION\_LEVEL\_REPEATABLE\_READ,
* psycopg2.extensions.ISOLATION\_LEVEL\_SERIALIZABLE,
* psycopg2.extensions.ISOLATION\_LEVEL\_DEFAULT.

**Как задать уровни изоляции**

Это можно сделать с помощью класса connection:

*conn.set\_isolation\_level(psycopg2.extensions.ISOLATION\_LEVEL\_AUTOCOMMIT)*

Также можно использовать метод connection.set\_session.

*сonnection.set\_session(isolation\_level=None, readonly=None, deferrable=None, autocommit=None)*

В этом случае **isolation\_level** могут быть **READ UNCOMMITED**, **REPEATABLE READ**, **SERIALIZE** или другие из списка констант.

## **Пример пула соединений в PostgreSQL с помощью Psycopg2**

В этом материале рассмотрим понятие пула соединений и особенности его реализации для базы данных PostgreSQL в Python с помощью Psycopg2.

**Пул соединений** — это кэшированные соединения с базой данных, которые создаются и поддерживаются таким образом, чтобы не было необходимости пересоздавать их для новых запросов.

Реализация и использование пула соединений в приложении Python, работающим с базой данных PostgreSQL, дает несколько преимуществ.

**Основное** — улучшение времени и производительности. Как известно, создание соединения с базой данных PostgreSQL — операция, потребляющая ресурсы и время. А с помощью пула можно уменьшить время запроса и ответа для приложений, работающих с базой данных в Python. Рассмотрим, как его реализовать.

В модуле psycopg2 есть 4 класса для управления пулом соединений. С их помощью можно легко создавать пул и управлять им. Как вариант, того же результата можно добиться с помощью реализации абстрактного класса.

## **Классы psycopg2 для управления пулом соединений**

В модуле psycopg2 есть четыре класса для управления пулом соединений:

* AbstractConnectionPool.
* SimpleConnectionPool.
* ThreadedConnectionPool.
* PersistentConnectionPool.

**Примечание:** SimpleConnectionPool, ThreadedConnectionPool и PersistentConnectionPool являются подклассами AbstractConnectionPool и реализуют методы из него.

Рассмотрим каждый из них по отдельности.

### **AbstractConnectionPool**

Это базовый класс, реализующий обобщенный код пула, основанный на ключе.

*psycopg2.pool.AbstractConnectionPool(minconn, maxconn, \*args, \*\*kwargs)*

**AbstractConnectionPool** — это абстрактный класс. Наследуемые должны реализовывать объявленные в нем методы. Если хотите создать собственную реализацию пула соединений, то нужно наследоваться от него и реализовать эти методы.

**minconn** — это минимальное требуемое количество объектов соединения. \*args, \*kwargs — аргументы, которые нужны для метода connect(), отвечающего за подключение к базе данных PostgreSQL.

### **SimpleConnectionPool**

Это подкласс AbstractConnectionPool, реализующий его методы. Его уже можно использовать для пула соединений. Этот класс подходит только для однопоточных приложений. Это значит, что если вы хотите создать пул соединений с помощью этого класса, то его нельзя будет передавать между потоками.

* Синтаксис:

*psycopg2.pool.SimpleConnectionPool(minconn, maxconn, \*args, \*\*kwargs)*

### **ThreadedConnectionPool**

Он также является подклассом класса AbstractConnectionPool и реализует его методы.

Этот класс используется в многопоточной среде, т.е. пул, созданный с помощью этого класса, можно разделить между несколькими потоками.

*psycopg2.pool.ThreadedConnectionPool(minconn, maxconn, \*args, \*\*kwargs)*

### **PersistentConnectionPool**

Еще один подкласс AbstractConnectionPool, реализующий его методы.

Этот класс используется в многопоточных приложениях с пулом, распределяющим постоянные соединения разным потокам.

Как и предполагает название, каждый поток получает одно соединение из пула. Таким образом, у одного потока может быть не больше одного соединения из пула.

Пул соединений генерирует ключ с помощью идентификатора потока. Это значит, что для каждого потока при вызовах соединение не меняется.

**Примечание:** этот класс преимущественно предназначен для взаимодействия с Zope и, вероятно, не подходит для обычных приложений.

* Синтаксис:

*psycopg2.pool.PersistentConnectionPool(minconn, maxconn, \*args, \*\*kwargs)*

Посмотрим, как создать пул соединений.

## **Методы psycopg2 для управления пулом соединений**

Следующие методы представлены в модуле Psycopg2 и используются для управления.

* **getconn(key=None)** — для получения доступного соединения из пула. Параметр key необязательный. При использовании этого параметра getconn() возвращает соединение, связанное с этим ключом. *Key* используется в классе PersistentConnectionPool.
* **putconn(connection, key=None, close=False)** — для возвращения соединения обратно в пул. Если параметр close равен True, то соединение удаляется и из пула. Если при получении соединения был использован ключ, то его же нужно передать при возвращении соединения.
* **closeall()** — закрывает все используемые соединения пула.

## **Создание пула соединений с помощью psycopg2**

В этом примере используем SimpleConnectionPool для создания пула. Перед этим стоит рассмотреть аргументы, которые требуются для работы. Нужно указать минимальное и максимальное количество соединений, имя пользователя, пароль, хост и базу данных.

* **minconn** — это нижний лимит количества подключений.
* **maxconn** — максимальное количество возможных подключений.
* **\*args, \*\*kwargs** — аргументы для метода connect(), необходимые для создания объекта соединений. Тут требуется указать имя хоста, пользователя, пароль, базу данных и порт.

### **Пример создания и управления пулом соединений PostgreSQL**

Рассмотрим, как использовать класс SimpleConnectionPoolдля создания и управления пулом соединений из Python.

**import psycopg2**

**from psycopg2 import pool**

**try:**

**# Подключиться к существующей базе данных**

**postgresql\_pool = psycopg2.pool.SimpleConnectionPool(1, 20,**

**user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**if postgresql\_pool:**

**print("Пул соединений создан успешно")**

**# Используйте getconn() для получения соединения из пула соединений.**

**connection = postgresql\_pool.getconn()**

**if connection:**

**print("Соединение установлено")**

**cursor = connection.cursor()**

**cursor.execute("select \* from mobile")**

**mobile\_records = cursor.fetchall()**

**print ("Отображение строк с таблицы mobile")**

**for row in mobile\_records:**

**print(row)**

**cursor.close()**

**# Используйте этот метод, чтобы отпустить объект соединения**

**# и отправить обратно в пул соединений**

**postgresql\_pool.putconn(connection)**

**print("PostgreSQL соединение вернулось в пул")**

**except (Exception, psycopg2.DatabaseError) as error :**

**print ("Ошибка при подключении к PostgreSQL", error)**

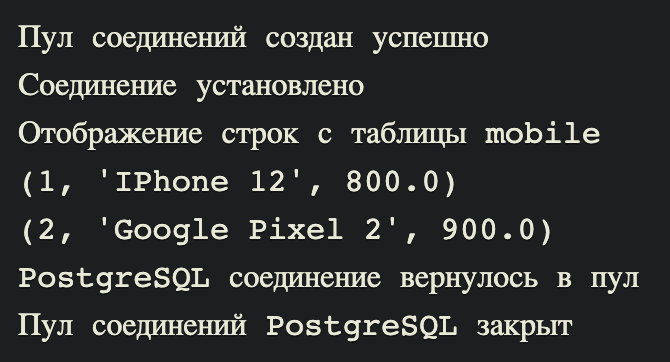
**finally:**

**if postgresql\_pool:**

**postgresql\_pool.closeall**

**print("Пул соединений PostgreSQL закрыт")**

Вывод:



Разберем пример. В метод были переданы следующие значения:

* Minimum connection = 1. Это значит, что в момент создания пула создается как минимум одно соединение.
* Maximum connection = 20. Всего можно использовать 20 соединений с PostgreSQL.
* Другие параметры для подключения.
* Конструктор класса SimpleConnectionPool возвращает экземпляр пула.
* С помощью getconn() делается запрос на подключение из пула
* После этого выполняются операции в базе данных.
* И в конце закрываются все активные и пассивные объекты соединения для закрытия приложения.

### **Создание многопоточного пула соединений**

Создадим пул соединений, который будет работать в многопоточной среде. Это можно сделать с помощью класса ThreadedConnectionPool.

**import psycopg2**

**from psycopg2 import pool**

**try:**

**# Подключиться к существующей базе данных**

**postgresql\_pool = psycopg2.pool.ThreadedConnectionPool(5, 20,**

**user="postgres",**

**# пароль, который указали при установке PostgreSQL**

**password="1111",**

**host="127.0.0.1",**

**port="5432",**

**database="postgres\_db")**

**if postgresql\_pool:**

**print("Пул соединений создан успешно")**

**# Используйте getconn() для получения соединения из пула соединений.**

**connection = postgresql\_pool.getconn()**

**if connection:**

**print("Соединение установлено")**

**cursor = connection.cursor()**

**cursor.execute("select \* from mobile")**

**mobile\_records = cursor.fetchall()**

**print ("Отображение строк с таблицы mobile")**

**for row in mobile\_records:**

**print(row)**

**cursor.close()**

**# Используйте этот метод, чтобы отпустить объект соединения**

**# и отправить обратно в пул соединений**

**postgresql\_pool.putconn(connection)**

**print("PostgreSQL соединение вернулось в пул")**

**except (Exception, psycopg2.DatabaseError) as error :**

**print ("Ошибка при подключении к PostgreSQL", error)**

**finally:**

**if postgresql\_pool:**

**postgresql\_pool.closeall**

**print("Пул соединений PostgreSQL закрыт")**

Вывод будет таким же:

