

Пособие по MySQL на Python

[Статьи](#) / [Базы данных](#) / [MySQL](#)

Содержание

1. Начало работы с MySQL на Python

- 1.1. Введение в MySQL на Python
- 1.2. Скачивание коннектора MySQL Python
- 1.3. Установка коннектора MySQL Python
- 1.4. Проверка правильности установки MySQL Connector / Python
- 1.5. Подключение Python к базе данных MySQL
- 1.6. Подготовка экземпляра базы данных
- 1.7. Подключение к базе данных MySQL с помощью функции connect()
- 1.8. Подключение к базе данных MySQL с помощью объекта MySQLConnection
- 1.9. Запросы Python MySQL
- 1.10. Запрос данных с помощью fetchone
- 1.11. Запрос данных с помощью fetchall
- 1.12. Запрос данных с помощью fetchmany
- 1.13. Вставка данных в Python MySQL
- 1.14. Вставка одной строки в таблицу
- 1.15. Вставка нескольких строк в таблицу
- 1.16. Обновление данных в Python MySQL
- 1.17. Удаление данных в MySQL на Python
- 1.18. Вызов в Python хранимых процедур MySQL
- 1.19. Прежде чем мы начнем
- 1.20. Вызов хранимых процедур из Python
- 1.21. Работа в Python MySQL с BLOB
- 1.22. Обновление в Python BLOB-данных
- 1.23. Чтение данных BLOB в Python

Начало работы с MySQL на Python

Это руководство поможет вам начать работу с **MySQL** на **Python**. Вы узнаете об особенностях **MySQL** на **Python** и как установить **MySQL Connector / Python** на вашей локальной системе.

Введение в MySQL на Python

Для доступа к базе данных **MySQL** из **Python**, вам нужен драйвер базы данных. **MySQL Connector / Python** является стандартизированным драйвером базы данных, предоставляемым **MySQL**.

MySQL Connector / Python поддерживает почти все функции, предоставляемые **MySQL версии 5.7**. Он позволяет конвертировать значения параметров между **Python** и **MySQL**, например, **Python DateTime** и **MySQL DATETIME**.

MySQL Connector / Python разработан специально для **MySQL**. Он поддерживает все расширения **MySQL** для стандартного **SQL**, такие как условие **LIMIT**.

MySQL Connector / Python позволяет сжимать поток данных между **Python** и сервером базы данных **MySQL** с использованием сжатия протоколов. Он поддерживает соединения с использованием сокета **TCP / IP** и безопасные **TCP / IP** соединения, использующие **SSL**.

MySQL Connector / Python представляет собой **API**, реализованный с помощью чистого **Python**. Это означает, что вам не нужно устанавливать какую-либо клиентскую библиотеку **MySQL** или модули **Python**, кроме стандартной библиотеки.

В этом пособии мы будем рассматривать **MySQL / Python 2.0**, который поддерживает **Python версий 2.6, 2.7 и 3.3**.

Скачивание коннектора MySQL Python

Для работы с коннектором **MySQL Python** вам необходимо скачать и установить его в вашей локальной системе. Доступны версии для различных платформ: **Mac OS X, Microsoft Windows, Ubuntu Linux** и т.д. Вам просто нужно выбрать нужную платформу и запустить скачивание.

Установка коннектора MySQL Python

Процесс установки коннектора **MySQL Python** довольно прост. Например, чтобы установить его в среде **Windows**, нужно выполнить следующие действия:

- Распаковать загруженный файл во временный каталог, например, **C: Temp**;
- Открыть окно консоли и переключиться на папку, в которую вы распаковали коннектор:

```
1 > cd c:\temp
```

- В папке C: Temp использовать следующую команду:

```
1 c:\temp > python setup.py install
```

Проверка правильности установки MySQL Connector / Python

После установки коннектора **MySQL Python** вы должны проверить его, чтобы убедиться, что он работает правильно, и вы можете подключаться к серверу базы данных **MySQL** без каких-либо проблем. Для проверки правильности установки выполните следующие действия:

- Откройте командную строку **Python**;
- Введите следующий код:

```
1 >>> import mysql.connector
2 >>> mysql.connector.connect(host='localhost',database='mysql',user='root',password='')
```

Если на экране появится приведенный ниже текст, значит, вы успешно установили коннектор **MySQL Python** на вашей системе:

```
1 <mysql.connector.connection.MySQLConnection object at 0x0187AE50>
```

Давайте перейдем к следующему разделу, чтобы узнать, как подключаться к базе данных **MySQL** из **Python**.

Подключение Python к базе данных MySQL

В этом разделе вы узнаете о различных способах подключения к базам данных **MySQL** из **Python** с использованием **MySQL Connector / Python API**.

Подготовка экземпляра базы данных

Во-первых, для этого пособия мы создаем новую базу данных с именем **python_mysql**. Чтобы создать новую базу данных, вы можете запустить **MySQL Workbench** или клиентский инструмент **MySQL** и использовать оператор **CREATE DATABASE** следующим образом:

```
1 1 CREATE DATABASE python_mysql;
```

Во-вторых, вам нужно загрузить данные в базу данных **python_mysql.sql** из файла **python_mysql.sql**.

Подключение к базе данных MySQL с помощью функции connect()

Давайте рассмотрим следующий модуль Python (`python_mysql_connect1.py`):

```
1 import mysql.connector
2 from mysql.connector import Error
3
4 def connect():
5     """ Connect to MySQL database """
6     try:
7         conn = mysql.connector.connect(host='localhost',
8                                       database='python_mysql',
9                                       user='root',
10                                      password='secret')
11
12         if conn.is_connected():
13             print('Connected to MySQL database')
14
15     except Error as e:
16         print(e)
17
18     finally:
19         conn.close()
20
21 if __name__ == '__main__':
22     connect()
```

Давайте рассмотрим этот модуль в деталях:

- Во-первых, мы импортируем объекты `mysql.connector` и `Error` из пакета **MySQL Connector / Python**;
- Во-вторых, для подключения к базе данных **MySQL** мы используем функцию `connect()`, которая принимает следующие параметры: хост, база данных, пользователь и пароль. Функция `connect()` устанавливает соединение с базой данных `python_mysql` и возвращает объект **MySQLConnection**;
- В-третьих, мы проверяем, было ли успешно установлено соединение с базой данных **MySQL** с помощью метода `is_connected()`. В случае возникновения исключений, например, если сервер базы данных не доступен, база данных не существует, имя пользователя или пароль неверны и т.д., **Python** вызовет исключение **Error**. Мы обрабатываем это исключение, используя блок `try except`;
- В-четвертых, если не произошло исключение, мы закрываем соединение с базой данных, вызвав метод `Close()` объекта **MySQLConnection**.

Для тестирования модуля `python_mysql_connect1.py`, используется следующая команда:

```
1 >python python_mysql_connect1.py
2 Connected to MySQL database
```

В этом примере, мы жестко задали настройки базы данных, такие как `localhost`, `python_mysql`, `root`, что нельзя назвать хорошей практикой. Поэтому давайте исправим это.

Подключение к базе данных MySQL с помощью объекта MySQLConnection

В этом примере мы создадим конфигурационный файл базы данных с именем `config.ini` и определим раздел с четырьмя параметрами следующим образом:

```
1 [mysql]
2 host = localhost
3 database = python_mysql
4 user = root
5 password =
```

Мы можем создать новый модуль с именем `python_mysql_dbconfig.py`, который считывает конфигурацию базы данных из файла `config.ini` и возвращает словарь следующим образом:

```

1  from configparser import ConfigParser
2
3
4  def read_db_config(filename='config.ini', section='mysql'):
5      """ Read database configuration file and return a dictionary object
6      :param filename: name of the configuration file
7      :param section: section of database configuration
8      :return: a dictionary of database parameters
9      """
10     # create parser and read ini configuration file
11     parser = ConfigParser()
12     parser.read(filename)
13
14     # get section, default to mysql
15     db = {}
16     if parser.has_section(section):
17         items = parser.items(section)
18         for item in items:
19             db[item[0]] = item[1]
20     else:
21         raise Exception('{0} not found in the {1} file'.format(section, filename))
22
23     return db

```

Обратите внимание, что мы использовали пакет **ConfigParser**, чтобы считать файл конфигурации.

Давайте проверим этот модуль в **REPL**:

```

1  >>> from python_mysql_dbconfig import read_db_config
2  >>> read_db_config()
3  {'password': '', 'host': 'localhost', 'user': 'root', 'database': 'python_mysql'}

```

Он работает, как ожидалось.

Теперь мы можем создать новый модуль **python_mysql_connect2.py**, который использует объект **MySQLConnection** для подключения к базе данных **python_mysql**:

```

1  from mysql.connector import MySQLConnection, Error
2  from python_mysql_dbconfig import read_db_config
3
4  def connect():
5      """ Connect to MySQL database """
6
7      db_config = read_db_config()
8
9      try:
10         print('Connecting to MySQL database...')
11         conn = MySQLConnection(**db_config)
12
13         if conn.is_connected():
14             print('connection established.')
15         else:
16             print('connection failed.')
17
18     except Error as error:
19         print(error)
20
21     finally:
22         conn.close()
23         print('Connection closed.')
24
25 if __name__ == '__main__':
26     connect()

```

Давайте рассмотрим приведенный выше код более подробно:

- Во-первых, мы импортировали необходимые объекты, в том числе **MySQLConnection**, **Error** из пакета **MySQL Connector / Python** и **read_db_config** из модуля **python_mysql_dbconfig**, который мы разработали;
- Во-вторых, внутри функции **connect()**, мы считали конфигурацию базы данных и использовали ее для создания нового экземпляра объекта **MySQLConnection**. Остальная часть кода работает аналогично первому примеру.
-

Когда мы запускаем **python_mysql_connect2** в окне консоли, мы получаем следующий результат:

```

1  >python python_mysql_connect2.py
2  Connecting to MySQL database...
3  connection established.
4  Connection closed.

```

В этом разделе мы рассмотрели, как подключаться к базам данных **MySQL** с помощью функцию **connect()** и объекта **MySQLConnection**. Оба способа дают тот же результат -

устанавливают соединение с базой данных **MySQL** и возвращают объект **MySQLConnection**.

Запросы Python MySQL

В этом разделе мы покажем, как запрашивать данные из базы данных MySQL в Python с использованием MySQL Connector / Python API, таких как `fetchone()`, `fetchmany()` и `fetchall()`.

Для запроса данных из базы данных MySQL из Python вам нужно сделать следующее:

- Подключиться к базе данных **MySQL**, вы получаете объект **MySQLConnection**;
- Установить экземпляр объекта **MySQLCursor** из объекта **MySQLConnection**;
- Использовать курсора для выполнения запроса путем вызова метода **execute()**;
- Использовать методы **fetchone()**, **fetchmany()** и **fetchall()** для выборки данных из результирующего набора;
- Закрыть курсор, а также подключение к базе данных, вызвав метод **close()** соответствующего объекта.

Мы расскажем, как использовать методы **fetchone()**, **fetchmany()** и **fetchall()** более подробно в следующих разделах.

Запрос данных с помощью fetchone

Метод **fetchone()** возвращает следующую строку набора результатов запроса или **None** в случае, если строк не осталось. Давайте посмотрим на следующий код:

```
1  from mysql.connector import MySQLConnection, Error
2  from python_mysql_dbconfig import read_db_config
3
4  def query_with_fetchone():
5      try:
6          dbconfig = read_db_config()
7          conn = MySQLConnection(**dbconfig)
8          cursor = conn.cursor()
9          cursor.execute("SELECT * FROM books")
10
11         row = cursor.fetchone()
12
13         while row is not None:
14             print(row)
15             row = cursor.fetchone()
16
17     except Error as e:
18         print(e)
19
20     finally:
21         cursor.close()
22         conn.close()
23
24 if __name__ == '__main__':
25     query_with_fetchone()
```

Давайте рассмотрим его более подробно:

- Во-первых, мы подключаемся к базе данных, создав новый объект **MySQLConnection**;
- Во-вторых, из объекта **MySQLConnection** мы устанавливаем новый объект **MySQLCursor**;
- В-третьих, мы выполняем запрос, который выбирает все строки из таблицы **books**;
- В-четвертых, мы вызываем метод **fetchone()**, чтобы выбрать следующую строку из набора результатов. В блоке **while loop** мы выводим содержимое строки и переходим к следующей строке, пока все строки не будут выбраны;
- В-пятых, мы закрываем курсор и объект подключения через вызов метода **close()** соответствующего объекта.

Запрос данных с помощью fetchall

В том случае, если число строк в таблице мало, вы можете использовать для извлечения всех строк из таблицы базы данных метод **fetchall()**. Рассмотрим следующий код:

```
1 from mysql.connector import MySQLConnection, Error
2 from python_mysql_dbconfig import read_db_config
3
4 def query_with_fetchall():
5     try:
6         dbconfig = read_db_config()
7         conn = MySQLConnection(**dbconfig)
8         cursor = conn.cursor()
9         cursor.execute("SELECT * FROM books")
10        rows = cursor.fetchall()
11
12        print('Total Row(s):', cursor.rowcount)
13        for row in rows:
14            print(row)
15
16    except Error as e:
17        print(e)
18
19    finally:
20        cursor.close()
21        conn.close()
22
23 if __name__ == '__main__':
24     query_with_fetchall()
```

Логика тут почти та же, что и в примере с использованием метода **fetchone()**, за исключением вызова метода **fetchall()**. Так как мы выбрали в память все строки из таблицы **books**, мы можем получить общее количество возвращаемых строк с помощью свойства **rowcount** объекта курсора.



Материал по теме:

[MySQL - это просто!](#)

Запрос данных с помощью fetchmany

Для сравнительно больших таблиц извлечение всех строк и возвращение набора результатов может занять значительное время. Кроме того, для **fetchall()** необходимо выделение достаточного объема памяти для хранения всего набора результатов. Это не слишком эффективно.

MySQL Connector / Python предоставляет нам метод **fetchmany()**, который возвращает следующее количество строк (**n**) набора результатов, что позволяет нам эффективно использовать объем памяти за оптимальное время. Давайте рассмотрим, как используется метод **fetchmany()**.

Во-первых, мы разрабатываем генератор, который разбивает вызовы базы данных на серию вызовов **fetchmany()** следующим образом:

```
1 def iter_row(cursor, size=10):
2     while True:
3         rows = cursor.fetchmany(size)
4         if not rows:
5             break
6         for row in rows:
7             yield row
```

Во-вторых, мы можем использовать генератор **iter_row()** для извлечения 10 строк за раз, как это показано ниже:

```
1 def query_with_fetchmany():
2     try:
3         dbconfig = read_db_config()
4         conn = MySQLConnection(**dbconfig)
5         cursor = conn.cursor()
6
7         cursor.execute("SELECT * FROM books")
8
9         for row in iter_row(cursor, 10):
10            print(row)
11
12    except Error as e:
13        print(e)
14
15    finally:
16        cursor.close()
17        conn.close()
```

В этом разделе мы рассмотрели различные методы запроса данных из базы данных **MySQL на Python**. Важно понимать каждую технику, чтобы для каждого конкретного случая использовать соответствующий вариант, что позволит увеличить производительность и

оптимизировать потребление памяти.

Вставка данных в Python MySQL

В этом разделе мы расскажем, как вставлять данные в таблицы **MySQL** с использованием **MySQL Connector / Python API**.

Чтобы вставить новые строки в таблицу **MySQL** необходимо выполнить следующие действия:

- Подключиться к серверу базы данных **MySQL**, создав новый объект **MySQLConnection**;
- Инициировать объект **MySQLCursor** из объекта **MySQLConnection**;
- Выполнить оператор **INSERT** для вставки данных в соответствующую таблицу;
- Закрыть соединение с базой данных.

MySQL Connector / Python предоставляет **API**, который позволяет вставить за один раз одну или несколько строк в таблицу. Давайте рассмотрим каждый метод более подробно.

Вставка одной строки в таблицу

Следующий код вставляет новую книгу в таблицу **books**:

```
1  from mysql.connector import MySQLConnection, Error
2  from python_mysql_dbconfig import read_db_config
3
4  def insert_book(title, isbn):
5      query = "INSERT INTO books(title,isbn) "
6             "VALUES(%s,%s)"
7      args = (title, isbn)
8
9      try:
10         db_config = read_db_config()
11         conn = MySQLConnection(**db_config)
12
13         cursor = conn.cursor()
14         cursor.execute(query, args)
15
16         if cursor.lastrowid:
17             print('last insert id', cursor.lastrowid)
18         else:
19             print('last insert id not found')
20
21         conn.commit()
22     except Error as error:
23         print(error)
24
25     finally:
26         cursor.close()
27         conn.close()
28
29 def main():
30     insert_book('A Sudden Light','9781439187036')
31
32 if __name__ == '__main__':
33     main()
```

В приведенном выше коде мы:

- Во-первых, импортируем объекты **MySQLConnection** и **Error** из пакета **MySQL Connector / Python** и функцию **read_db_config()** из модуля **python_mysql_dbconfig**;
- Во-вторых, определяем новую функцию под названием **insert_book()**, которая принимает два аргумента: название и **ISBN**. Внутри функции **insert_book()**, мы готовим оператор **INSERT** (*запрос*) и данные (*аргументы*), которые мы будем вставлять в таблицу **books**. Обратите внимание, что данные, которые мы передаем в функцию, это кортеж;
- В-третьих, внутри блока **try except** мы создаем новое подключение, выполняем оператор и утверждаем изменения. Обратите внимание, что вы должны вызвать метод **commit()** явно для того, чтобы изменения в базу данных были внесены. В случае, если новая строка была вставлена успешно, мы можем получить последний вставленный **id** столбца **AUTO INCREMENT** с помощью свойство **lastrowid** объекта **MySQLCursor**;
- В-четвертых, в конце функции **insert_book()** мы закрываем курсор и соединение с базой данных;
- В-пятых, в функции **main()** мы вызываем функцию **insert_book()** и передаем **title** и **isbn**, чтобы вставить новую строку в таблицу **books**.

Вставка нескольких строк в таблицу

Оператор **MySQL INSERT** позволяет вставить сразу несколько строк с помощью синтаксиса **VALUES**. Вам просто нужно включить несколько списков значений столбцов. Каждый список заключен в скобки и разделен запятыми. Например, чтобы вставить несколько книг в таблицу **books** используется следующий оператор:

```
1  INSERT INTO books(title,isbn)
2  VALUES('Harry Potter And The Order Of The Phoenix', '9780439358071'),
3         ('Gone with the Wind', '9780446675536'),
4         ('Pride and Prejudice (Modern Library Classics)', '9780679783268');
5  Чтобы вставить несколько строк в таблицу в Python используется метод executemany() объекта
MySQLCursor. Смотрите следующий код:
6  from mysql.connector import MySQLConnection, Error
7  from python_mysql_dbconfig import read_db_config
8
9  def insert_books(books):
10     query = "INSERT INTO books(title,isbn) "
11            "VALUES(%,%s)"
12
13     try:
14         conn = MySQLConnection(**db_config)
15
16         cursor = conn.cursor()
17         cursor.executemany(query, books)
18
19         conn.commit()
20     except Error as e:
21         print('Error:', e)
22
23     finally:
24         cursor.close()
25         conn.close()
26
27 def main():
28     books = [('Harry Potter And The Order Of The Phoenix', '9780439358071'),
29             ('Gone with the Wind', '9780446675536'),
30             ('Pride and Prejudice (Modern Library Classics)', '9780679783268')]
31     insert_books(books)
32
33 if __name__ == '__main__':
34     main()
```

Логика в этом примере аналогична логике первого примера. Только вместо вызова метода **execute()** мы используем метод **executemany()**.

В функции **main()** мы передаем список кортежей, каждый из которых содержит название и **ISBN** книги.

Вызвав метод **executemany()** объекта **MySQLCursor**, **MySQL Connector / Python** переводит оператор **INSERT** в оператор, который содержит несколько списков значений.

В этом разделе мы рассмотрели, как вставить одну или несколько строк в таблицу **MySQL** в **Python**.

Обновление данных в Python MySQL

В этом разделе мы рассмотрим действия, необходимые для обновления данных в таблице **MySQL** с помощью **MySQL Connector / Python API**.

Для обновления данных в таблице **MySQL** в **Python**, вам нужно выполнить следующие действия:

- Подключиться к серверу базы данных **MySQL**, создав новый объект **MySQLConnection**;
- Создать новый объект **MySQLCursor** из объекта **MySQLConnection** и вызвать метод **execute()** объекта **MySQLCursor**. Чтобы утвердить изменения, нужно вызвать метод **commit()** объекта **MySQLConnection** после вызова метода **execute()**. В противном случае никакие изменения в базу данных внесены не будут;
- Закрыть курсор и соединение с базой данных.

В следующем примере, мы будем обновлять название книги, указанной через **ID** книги:


```
1 from mysql.connector import MySQLConnection, Error
2 from python_mysql_dbconfig import read_db_config
3
4 def update_book(book_id, title):
5     # read database configuration
6     db_config = read_db_config()
7
8     # prepare query and data
9     query = """ UPDATE books
10                SET title = %s
11                WHERE id = %s """
12
13     data = (title, book_id)
14
15     try:
16         conn = MySQLConnection(**db_config)
17
18         # update book title
19         cursor = conn.cursor()
20         cursor.execute(query, data)
21
22         # accept the changes
23         conn.commit()
24
25     except Error as error:
26         print(error)
27
28     finally:
29         cursor.close()
30         conn.close()
31
32 if __name__ == '__main__':
33     update_book(37, 'The Giant on the Hill *** TEST ***')
```

В этом модуле мы использовали функцию `read_db_config()` из модуля `python_mysql_dbconfig`, который мы создали в разделе Подключение к базе данных через Python.

Внутри оператора **UPDATE** мы размещаем два заполнителя (%), один для названия книги, второй - для **ID** книги. Мы передали оба кортежа оператора **UPDATE** (*query*) и (*title,id*) в метод `execute()`. Коннектор интерпретирует запрос следующим образом:

```
1 UPDATE books
2 SET title = 'The Giant on the Hill *** TEST ***'
3 WHERE id = 37
```



Материал по теме:

Репликация MySQL

Важно помнить, что мы всегда должны использовать заполнители (%) внутри любых операторов **SQL**, которые содержат информацию пользователей. Это помогает нам предотвратить потенциально вредоносные действия.

Давайте проверим наш новый модуль, чтобы убедиться, если он работает.

Во-первых, мы выбираем книгу с **ID 37**:

```
1 SELECT * FROM books
2 WHERE id = 37;
```

id	title	isbn
37	The Giant on the Hill	1235644620578

Во-вторых, мы запускаем модуль.

В-третьих, мы выбираем запись книги, снова выполнив оператор **SELECT**, чтобы увидеть, действительно ли запись изменилась.

id	title	isbn
37	The Giant on the Hill	1235644620578

Все работает, как ожидалось.

В этом разделе вы рассказали, как обновлять данные с помощью **MySQL Connector / Python API**.

Удаление данных в MySQL на Python

В этом разделе мы рассмотрим этапы удаления данных из базы данных **MySQL** с помощью

MySQL Python.

Для удаления строк в таблице **MySQL** через **Python** вам нужно совершить следующие действия:

- Подключиться к серверу базы данных **MySQL**, создав новый объект **MySQLConnection**;
- Создать новый объект **MySQLCursor** из объекта **MySQLConnection** и вызвать метод **execute()** объекта **MySQLCursor**. Чтобы утвердить изменения, нужно вызвать метод **commit()** объекта **MySQLConnection** после вызова метода **execute()**;
- Закрыть курсора и соединение с базой данных, вызвав метод **close()** соответствующего объекта.

В следующем примере показано, как удалить книгу с указанным **ID**:

```
1 from mysql.connector import MySQLConnection, Error
2 from python_mysql_dbconfig import read_db_config
3
4 def delete_book(book_id):
5     db_config = read_db_config()
6
7     query = "DELETE FROM books WHERE id = %s"
8
9     try:
10        # connect to the database server
11        conn = MySQLConnection(**db_config)
12
13        # execute the query
14        cursor = conn.cursor()
15        cursor.execute(query, (book_id,))
16
17        # accept the change
18        conn.commit()
19
20    except Error as error:
21        print(error)
22
23    finally:
24        cursor.close()
25        conn.close()
26
27 if __name__ == '__main__':
28     delete_book(102)
```

Обратите внимание, что мы используем функцию **read_db_config()** из модуля **python_mysql_dbconfig**, который мы разработали в предыдущих разделах.

Так как нам нужно удалить из таблицы **books** конкретную строку, мы должны разместить заполнитель (%) на стороне оператора **DELETE**.

Когда мы вызываем метод **execute()**, мы передаем ему и оператор **DELETE** и кортеж (**book_id**). Коннектор интерпретирует оператор **DELETE** в следующую форму:

```
1 DELETE FROM books WHERE id = 102
```

Вы должны всегда использовать заполнители внутри любого запроса, который вы передаете в метод **execute()**.

Это помогает нам предотвратить потенциально вредоносные действия.

Перед запуском кода, давайте проверим таблицу **books**, чтобы просмотреть данные, прежде чем мы удалим запись:

```
1 SELECT * FROM books
2 WHERE id = 102;
```

id	title	isbn
102	Pride and Prejudice (Modern Library Classics)	9780679783268

После запуска приведенного выше модуля, мы снова выполняем оператор **SELECT**. Строка не возвращается. Это означает, что модуль успешно удалил запись.

В этом разделе мы рассмотрели, как удалить данные из таблицы **MySQL** с использованием **MySQL Connector / Python API**.

Вызов в Python хранимых процедур MySQL

В этом разделе мы покажем, как вызывать в **Python** хранимые процедуры **MySQL** с использованием **MySQL Connector / Python API**.

Прежде чем мы начнем

В этом разделе в качестве демонстрации мы создадим две хранимые процедуры. Первая - для получения всех книг с информацией об авторе из таблиц **books** и **authors**:

```
1 DELIMITER $$
2
3 USE python_mysql$$
4
5 CREATE PROCEDURE find_all()
6 BEGIN
7 SELECT title, isbn, CONCAT(first_name, ' ', last_name) AS author
8 FROM books
9 INNER JOIN book_author ON book_author.book_id = books.id
10 INNER JOIN authors ON book_author.author_id = authors.id;
11 END$$
12
13 DELIMITER ;
```

Хранимая процедура **find_all()** содержит оператор **SELECT** с условием **JOIN**, который извлекает название, **ISBN** и полное имя автора из таблиц **books** и **authors**. Когда мы выполняем хранимую процедуру **find_all()**, она возвращает следующий результат:

```
1 CALL find_all();
2 [IMG=http://www.mysqltutorial.org/wp-content/uploads/2014/10
/python_mysql_stored_procedure_example.png?cf1f9d]
3 Вторая хранимая процедура с именем find_by_isbn() используется, чтобы найти книгу по ISBN
следующим образом:
4 DELIMITER $$
5
6 CREATE PROCEDURE find_by_isbn(IN p_isbn VARCHAR(13),OUT p_title VARCHAR(255))
7 BEGIN
8 SELECT title INTO p_title FROM books
9 WHERE isbn = p_isbn;
10 END$$
11
12 DELIMITER ;
```

find_by_isbn() принимает два параметра: первый параметр **ISBN** (*параметр IN*), второй - заголовок (*OUT параметр*). Когда вы передаете в хранимую процедуру **ISBN**, вы получаете название книги, например:

```
1 CALL find_by_isbn('1235927658929',@title);
2 SELECT @title;
```

Вызов хранимых процедур из Python

Для вызова хранимой процедуры в **Python**, вам нужно выполнить следующие действия:

- Подключиться к серверу базы данных **MySQL**, создав новый объект **MySQLConnection**;
- Создать новый объект **MySQLCursor** из объекта **MySQLConnection**, вызвав метод **cursor()**;
- Вызвать метод **callproc()** объекта **MySQLCursor**. Вы передаете имя хранимой процедуры в качестве первого аргумента метода **callproc()**. Если для хранимой процедуры требуются параметры, вы должны передать их список в качестве второго аргумента метода **callproc()**. В случае, если хранимая процедура возвращает набор результатов, вы можете сослаться на метод **stored_results()** объекта **MySQLCursor**, чтобы получить итератор списка и перебрать этот набор результатов с помощью метода **fetchall()**;
- Закрыть курсор и подключение к базе данных, как всегда.

Следующий пример демонстрирует, как вызывать хранимую процедуру **find_all()** в **Python** и выводить набор результатов:

```

1  from mysql.connector import MySQLConnection, Error
2  from python_mysql_dbconfig import read_db_config
3
4  def call_find_all_sp():
5      try:
6          db_config = read_db_config()
7          conn = MySQLConnection(**db_config)
8          cursor = conn.cursor()
9
10         cursor.callproc('find_all')
11
12         # print out the result
13         for result in cursor.stored_results():
14             print(result.fetchall())
15
16     except Error as e:
17         print(e)
18
19     finally:
20         cursor.close()
21         conn.close()
22
23 if __name__ == '__main__':
24     call_find_all_sp()

```

В следующем примере показано, как вызвать хранимую процедуру **find_by_isbn()**:

```

1  from mysql.connector import MySQLConnection, Error
2  from python_mysql_dbconfig import read_db_config
3
4  def call_find_by_isbn():
5      try:
6          db_config = read_db_config()
7          conn = MySQLConnection(**db_config)
8          cursor = conn.cursor()
9
10         args = ['1236400967773', 0]
11         result_args = cursor.callproc('find_by_isbn', args)
12
13         print(result_args[1])
14
15     except Error as e:
16         print(e)
17
18     finally:
19         cursor.close()
20         conn.close()
21
22 if __name__ == '__main__':
23     call_find_by_isbn()

```

Для хранимой процедуры **find_by_isbn()** требуются два параметра, следовательно, мы должны передать список (**args**), который содержит два элемента: первый из них **ISBN** (**1236400967773**), а второй **0**. Вторым элементом списка аргументов (**0**) - это просто заполнитель, содержащий параметр **p_title**.

Метод **callproc()** возвращает список (**result_args**), который содержит два элемента, где вторым элементом (**result_args[1]**) содержит значение параметра **p_title**.

В этом разделе мы рассмотрели, как вызываются хранимые процедуры через Python с использованием метода **callproc()** объекта **MySQLCursor**.

Работа в Python MySQL с BLOB

В этом разделе мы рассмотрим, как работать в **Python** с данными **MySQL BLOB**, а именно примеры обновления и чтения данных **BLOB**.

В таблице **authors** содержится столбец с именем **photo**, в котором хранятся данные типа **BLOB**. Мы считаем данные из файла изображения и обновим ими столбец **photo**.

Обновление в Python BLOB-данных

Во-первых, мы разрабатываем функцию с именем **read_file()**, которая считывает файл и возвращает содержимое файла:

```

1  def read_file(filename):
2      with open(filename, 'rb') as f:
3          photo = f.read()
4      return photo

```

Во-вторых, мы создаем новую функцию под названием **update_blob()**, которая обновляет фото автора, указанного с помощью **author_id**.

```

1  from mysql.connector import MySQLConnection, Error
2  from python_mysql_dbconfig import read_db_config
3
4  def update_blob(author_id, filename):
5      # read file
6      data = read_file(filename)
7
8      # prepare update query and data
9      query = "UPDATE authors "
10             "SET photo = %s "
11             "WHERE id = %s"
12
13     args = (data, author_id)
14
15     db_config = read_db_config()
16
17     try:
18         conn = MySQLConnection(**db_config)
19         cursor = conn.cursor()
20         cursor.execute(query, args)
21         conn.commit()
22     except Error as e:
23         print(e)
24     finally:
25         cursor.close()
26         conn.close()

```

Давайте подробно рассмотрим этот код:

- Во-первых, мы вызываем функцию **read_file()**, которая считывает данные из файла и возвращает их;
- Во-вторых, мы составляем оператор **UPDATE**, который обновляет столбец фото автора, указанного с помощью **author_id**. Переменная **args** - это кортеж, который содержит данные файла и **author_id**. Мы передаем эту переменную в метод **execute()** вместе с **query**;
- В-третьих, внутри блока **try except** мы подключаемся к базе данных, устанавливаем курсор и выполняем запрос с **args**. Для того чтобы изменения вступили в силу, мы вызываем метод **commit()** объекта **MySQLConnection**;
- В-четвертых, мы закрываем курсор и соединение с базой данных в блоке **finally**.

Обратите внимание, что мы импортировали объекты **MySQLConnection** и **Error** из пакета **MySQL Connector / Python** и функцию **read_db_config()** из модуля **python_mysql_dbconfig**, который мы разработали в предыдущих разделах.

Давайте протестируем функцию **update_blob()**:

```

1  def main():
2      update_blob(144, "picturesgarth_stein.jpg")
3
4  if __name__ == '__main__':
5      main()

```

Обратите внимание, что для тестирования вы можете использовать следующую фотографию и поместить ее в папку изображений:




Внутри основной функции, мы вызываем функцию **update_blob()** для обновления столбца фото для автора с идентификатором **144**. Чтобы проверить результат, мы выбираем данные из таблицы **authors**:

```

1  SELECT * FROM authors
2  WHERE id = 144;

```

id	first_name	last_name	photo
144	Garth	Stein	

Все работает, как ожидалось.

Чтение данных BLOB в Python

В этом примере мы выбираем **BLOB**-данные из таблицы авторов и записываем их в файл.

Во-первых, мы разрабатываем функцию **write_file()**, которая записывает двоичные данные в

файл следующим образом:

```
1 def write_file(data, filename):
2     with open(filename, 'wb') as f:
3         f.write(data)
```

Во-вторых, мы создаем новую функцию под названием `read_blob()`:

```
1 def read_blob(author_id, filename):
2     # select photo column of a specific author
3     query = "SELECT photo FROM authors WHERE id = %s"
4
5     # read database configuration
6     db_config = read_db_config()
7
8     try:
9         # query blob data form the authors table
10        conn = MySQLConnection(**db_config)
11        cursor = conn.cursor()
12        cursor.execute(query, (author_id,))
13        photo = cursor.fetchone()[0]
14
15        # write blob data into a file
16        write_file(photo, filename)
17
18    except Error as e:
19        print(e)
20
21    finally:
22        cursor.close()
23        conn.close()
```

Функция `read_blob()` считывает **BLOB**-данные из таблицы **authors** и записывает их в файл, указанный в параметре имени файла.

Этот код действует очень просто:

- Во-первых, мы составляем оператор **SELECT**, который извлекает фотографию конкретного автора;
- Во-вторых, мы получаем конфигурацию базы данных, вызвав функцию `read_db_config()`;
- В-третьих, внутри блока **try except** мы подключаемся к базе данных, устанавливаем курсор и выполняем запрос. После того, как мы получили **BLOB**-данные, мы используем функцию `write_file()`, чтобы записать их в файл, указанный в имени файла;
- В-четвертых, в конечном блоке мы закрываем курсор и соединение с базой данных.

Теперь, давайте проверим функцию `read_blob()`:

```
1 def main():
2     read_blob(144, "outputgarth_stein.jpg")
3
4 if __name__ == '__main__':
5     main()
```

Если вы откроете папку вывода в проекте и увидите там картинку, это означает, что вы успешно считали **BLOB**-данные.

В этом разделе, мы рассказали, как обновлять и считывать **BLOB**-данные в **MySQL** из **Python** с использованием **MySQL Connector / API**.