МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Вятский государственный университет»
(ФГБОУ ВПО «ВятГУ»)
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Методические указания
к самостоятельным и лабораторным работам
по дисциплине «Базы данных»

Киров 2019

Разработка приложения на Python с базой данных под управлением PostgreSQL

Цели лабораторной работы:

- Познакомиться с библиотекой в языке Python для связывания приложения с БД
- Освоить на практике основы взаимодействия с БД под управлением PostgreSQL в приложении на Python

Задание на лабораторную работу:

Создать приложение с графическим приложением на языке Python. Приложение должно использовать БД, разработанную в предыдущих лабораторных работах.

Для любой одной таблицы, которая содержит внешний ключ на другую таблицу, приложение должно выполнять следующее:

- Выводить строки таблицы
- Предоставлять любой фильтр по значением строк. (Например, «Дата с … по …» или «Имя содержит …»)
- Предоставлять возможность добавления новых строк
- Предоставлять возможность удаления строки

Требования к реализации:

- Заголовки должны быть осмысленными. Например, вместо «name» в таблице должен быть заголовок «Имя»
- При добавлении новой строки внешний ключ выбирается из списка
- Сохранение или удаление строки должно быть реализовано с помощью функции PL/pgSQL
- Фильтрация значений при поиске должна производиться через запрос, а не в полученной коллекции
- Разрешается использование любого фреймворка
- При разработке нужно использовать шаблоны проектирования, связанные с работой с БД

Отчет должен содержать:

- Диаграмму классов, организующих работу с БД

- Их исходный код
- Экранные формы

Подготовка среды

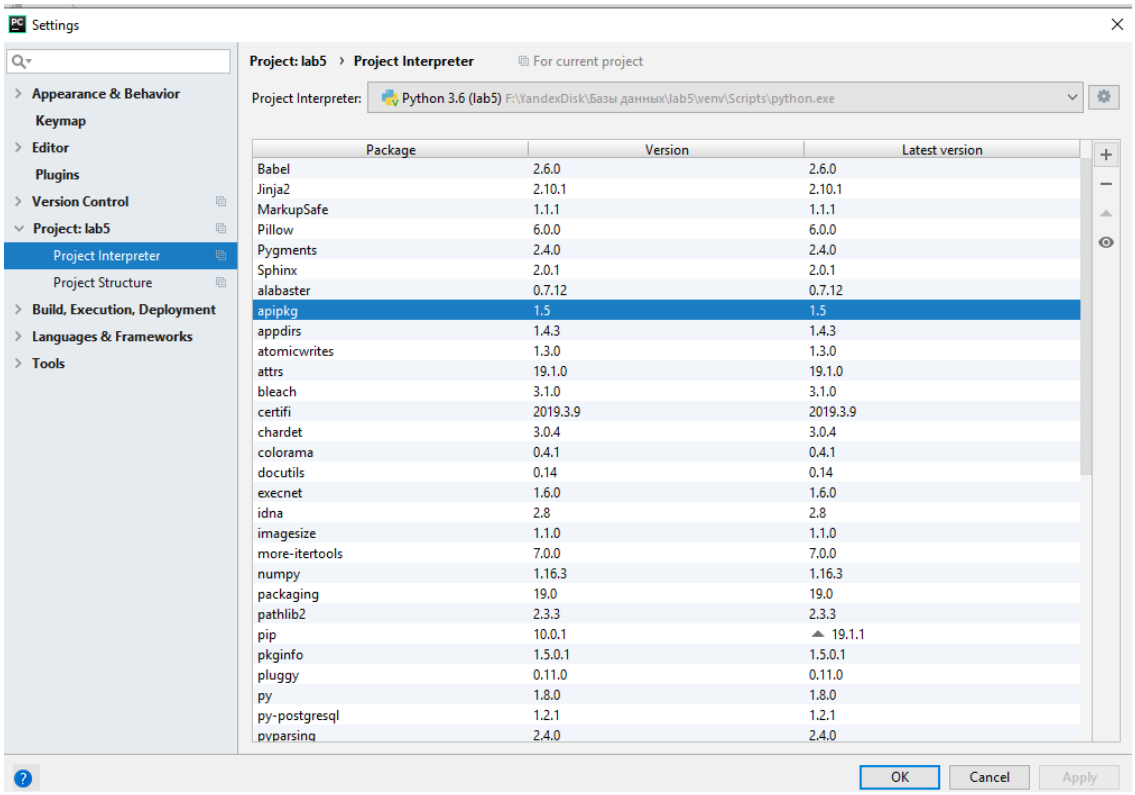
Для начала работы нам нужно установить библиотеку для работы с базой данных.

В примерах используется IDE PyCharm. Необходимые действия могут изменятся в зависимости от среды разработки.

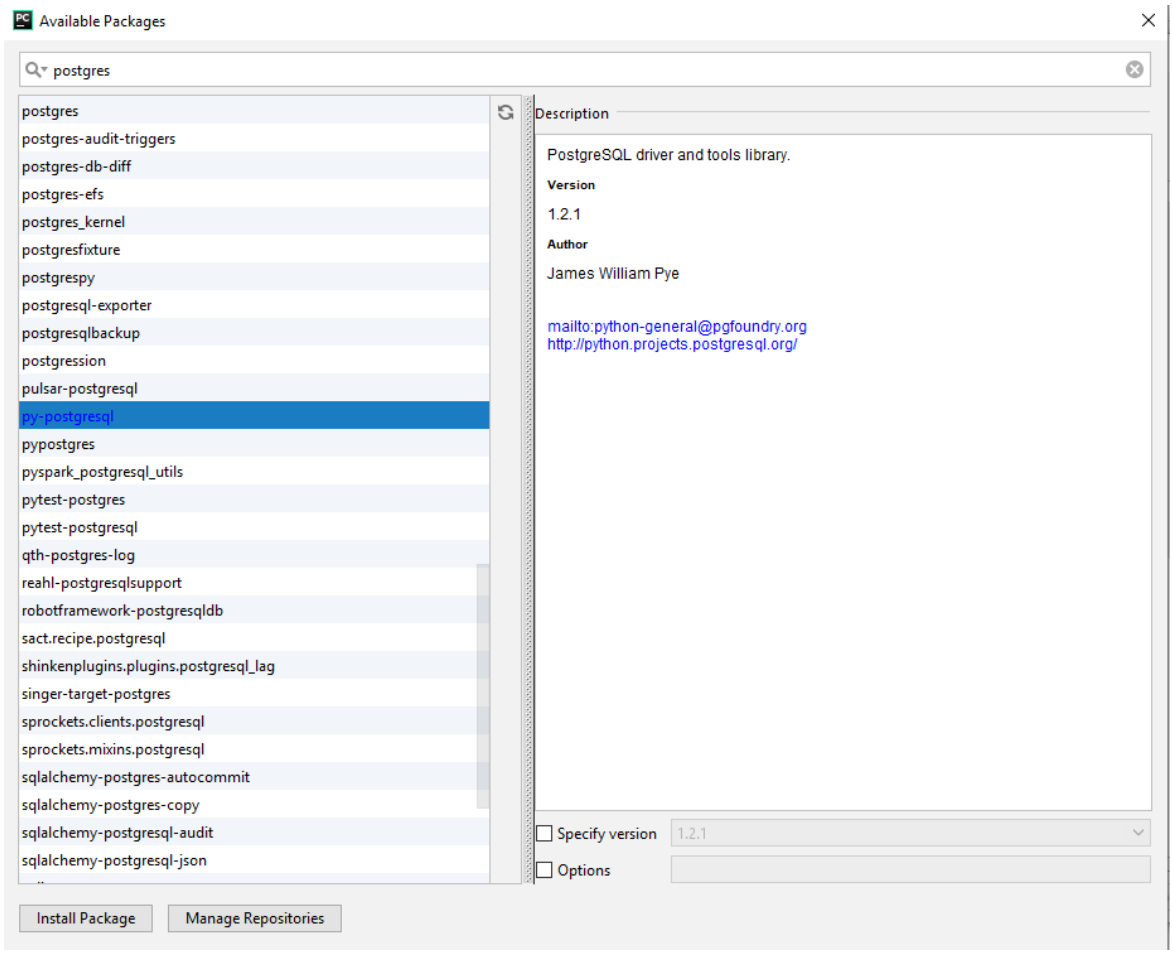
Для работы с базой данных будет использоваться библиотека py-postgresql

После создания проекта или открытия существующего нужно установить библиотеку. Для этого нужно зайти в свойства проекта через пункт Settings (File -> Settings).

Далее Project:<*Имя проекта*> -> Project Interpreter. Откроется список установленных библиотек.



Для добавления новой нужно нажать «+» в верхней правой части. В поле поиска ввести имя библиотеки, в результатах поиска выбрать нужную и нажать кнопку Install Package. В результате в списке установленных библиотек должна появиться выбранная нами библиотека.


Available Packages
postgres
postgres
postgres-audit-triggers
postgres-db-diff
postgres-efs
postgres_kernel
postgresfixture
postgrespy
postgresql-exporter
postgresqlbackup
postgression
pulsar-postgresql
py-postgresql
pypostgres
pyspark_postgresql_utils
pytest-postgres
pytest-postgresql
qth-postgres-log
reahl-postgresqlsupport
robotframework-postgresqldb
sact.recipe.postgresql
shinkenplugins.plugins.postgresql_lag
singer-target-postgres
sprockets.clients.postgresql
sprockets.mixins.postgresql
sqlalchemy-postgres-autocommit
sqlalchemy-postgres-copy
sqlalchemy-postgresql-audit
sqlalchemy-postgresql-json
Description
PostgreSQL driver and tools library.
Version
1.2.1
Author
James William Pye
mailto:python-general@pgfoundry.org
http://python.projects.postgresql.org/
Specify version
1.2.1
Options
Install Package
Manage Repositories

Подключение к базе данных

Перед тем как подключиться к базе данных, создадим класс database и сохраним его в файле с именем database.py. Через этот класс мы будем осуществлять все взаимодействие с базой данных. Чтобы начать работу с базой данных нужно подключить (импортировать) библиотеку в программе, для этого нужно написать:

```
import postgresql
```

Чтобы выполнить подключение к базе данных нужно выполнить команду подключения и передать в качестве параметров данные о базе данных. Строка подключения выглядит следующим образом:

```
postgresql.open('pq:// first_user: first_user @localhost:5432/lab')
```

first_user: first_user – Данные пользователя (логин и пароль)

localhost – адрес базы данных

5432 – Порт для подключения

lab – имя БД

В итоге для подключения потребуется следующий код:

```
import postgresql
from postgresql import exceptions    #для обработки исключений

class database:
    # Объект соединения с БД
    _connection = None

    # Строка подключения к БД
    _db_url = "pq:// first_user: first_user @localhost:5432/lab"

    def openConnection(self):
        try:
            self._connection = postgresql.open(self._db_url)
            print("Успешное подключение")
        except postgresql.exceptions.AuthenticationMethodError:  #
неподдерживаемый тип авторизации
            print("Ошибка! Неподдерживаемый тип авторизации.")
        except postgresql.exceptions.ClientCannotConnectError:  # клиенту не
удалось установить соединение с сервером
            print("Ошибка! Клиенту не удалось установить соединение с
сервером.")
        except postgresql.exceptions.ConnectTimeoutError:  # клиенту не
удалось установить соединение в заданное время
            print("Ошибка! Клиенту не удалось установить соединение в
заданное время.")
        except postgresql.exceptions.DriverError:  # Ошибка в реализации
драйвера
            print("Ошибка в реализации драйвера")
        else:
            print("Ошибка! Неизвестная ошибка при попытке подключения.")

    def closeConnection(self):
        self._connection.close()
```

Чтобы использовать созданный класс database, нужно импортировать его в другой класс. Создадим так же класс, который будет обращаться к БД через созданный только что класс.

```
import database

db = database.database()       #создание экземпляра класса database

db.openConnection()            #подключение к базе данных
```

В результате будет происходить следующее:

- Создается объект db
- После создания выполняем попытку подключиться к БД.
- Если соединение успешно, то будет выведено сообщение об успехе.
- Если подключиться не выйдет, то будет выведено сообщение об ошибке.

Всегда нужно закрывать соединение, после окончания работой с базой данных. В примере для этого используется метод closeConnection(). Если не закрыть соединение, оно может остаться активным на стороне СУБД, и привести к ошибкам., в том числе проблемы с авторизацией, если установлено ограничение на количество одновременных подключений.

Если все сделано правильно, вы получите в консоли сообщение: "Успешное подключение"

Выполнение запросов к БД

Для выполнения обращений к БД используется специальный интерфейс. Создать его можно через существующее подключение.

```
Connection.query(sql)

Sql – Текст запроса к базе данных
```

Query() – возвращает результат выполнения запроса

После выполнения запроса необходимо закрыть соединение!

Выполним простой запрос к базе данных. Например, считаем данные из таблицы group. Запрос будет иметь следующий вид:
"SELECT * FROM public.group"

Добавим в класс database следующий код:

```
    def getGroups(self):
    try:
        return self._connection.query("SELECT * FROM public.group")
    except postgresql.exceptions.ConnectionDoesNotExistError:   #Соединение закрыто или никогда не было подключено
        print("Ошибка! Соединение закрыто или никогда не было подключено.")
    except:
        print("Ошибка! Неизвестная ошибка при попытке запроса.")
```

Добавим в главный класс/скрипт следующую строчку для вызова функции, в итоге получим следующий код:

```
import database         #импортируем класс для работы с базой данных

db = database.database()       #создаём экземпляр класса

db.openConnection()    #подключаемся к базе
print(db.getGroups())  #вызываем функцию и получаем список группиз базы
db.closeConnection()   #закрываем соединение с базой
```

Если все сделано правильно, будет выведен результат запроса, в котором будет список групп

Существует такой тип запросов, как подготовленные запросы. В такие запросы можно передавать параметры, и они могут дать выигрыш в скорости работы при частом использовании. Пример кода приведен ниже:

```
    def prepareSelect(self):
    try:
        ps = self._connection.prepare("SELECT * FROM public.group WHERE id = $1")
        return ps(25, 1)    #вместо $1 подставиться 25
```

```
    except:
        print("Ошибка! Неизвестная ошибка при подготовленном запросе")
```

При работе с СУБД для обозначения дат и времени используется библиотека datetime. Рекомендуется использовать datetime, поскольку функции библиотеки py-postgresql возвращают типы данных datetime

Выполнение запросов на изменение/удаление

Запросы для удаления или изменения элементов таблиц не сильно отличаются от select запроса, и выполняются с использованием тех же объектов query/ prepare.

Для примера выполнения запросов insert/update/delete поработаем с таблицей group.

Для начала, добавим новый элемент в таблицу. Ниже представлен код для добавления:

```
        def insertGroup(self, name, number_of_students):
    try:
        self._connection.query("INSERT INTO group(name, number_of_students) 
VALUES(%s, %s)" % (name,number_of_students))
    except postgresql.exceptions.ConnectionDoesNotExistError:   #Соединение 
закрыто или никогда не было подключено
        print("Ошибка! Соединение закрыто или никогда не было подключено.")
    except:
        print("Ошибка! Неизвестная ошибка при попытке запроса.")
```

ИЛИ

```
        def insert2Group(self, name, number_of_students):
    try:
        ps = self._connection.prepare("INSERT INTO group(name, 
number_of_students) VALUES($1, $2)")
        ps(name, number_of_students)
    except postgresql.exceptions.ConnectionDoesNotExistError:   #Соединение 
закрыто или никогда не было подключено
        print("Ошибка! Соединение закрыто или никогда не было подключено.")
    except:
        print("Ошибка! Неизвестная ошибка при попытке запроса.")
```

Формат метода не изменился относительно select запроса.

Чтобы изменить какие-либо данные в таблице необходимо выполнить запрос update. Ниже представлен код для изменения данных.

```
        def updateGroup(self,id, name, number_of_students):
    try:
        self._connection.query("UPDATE group SET name=%s,
number_of_students=%s WHERE id=%d" % (name,number_of_students,id))
    except postgresql.exceptions.ConnectionDoesNotExistError:   #Соединение
закрыто или никогда не было подключено
        print("Ошибка! Соединение закрыто или никогда не было подключено.")
    except:
        print("Ошибка! Неизвестная ошибка при попытке запроса.")
```

ИЛИ

```
def update2Group(self, name, number_of_students):
    try:
        ps = self._connection.prepare("UPDATE group SET name=$1,
number_of_students=$2 WHERE id=$3")
        ps(name, number_of_students, id)
    except postgresql.exceptions.ConnectionDoesNotExistError:   #Соединение
закрыто или никогда не было подключено
        print("Ошибка! Соединение закрыто или никогда не было подключено.")
    except:
        print("Ошибка! Неизвестная ошибка при попытке запроса.")
```

Аналогичным образом можно выполнить и delete запрос.

Вызовы функций

Теперь попробуем вызвать функцию, которая была добавлена в базу в рамках предыдущей работы. Допустим есть функция добавления группы в базу : insert_group, в качестве параметров принимает имя группы и количество студентов.

Вызывать функции можно и с использованием обычного select.

```
        def insert3(self, name, number_of_students):
    try:
        self._connection.query("SELECT insert_group(%s, %s)" %
(name,number_of_students))
    except postgresql.exceptions.ConnectionDoesNotExistError:   #Соединение
закрыто или никогда не было подключено
        print("Ошибка! Соединение закрыто или никогда не было подключено.")
    except:
        print("Ошибка! Неизвестная ошибка при попытке запроса.")
```

В таком случае в select запрос просто подставляется вызов функции.

Другой вариант.

```
        def insert4(self, name, number_of_students):
    try:
        func = self._connection.proc("insert_group(varchar, varchar)")
        func(name, number_of_students)
    except postgresql.exceptions.ConnectionDoesNotExistError:   #Соединение
закрыто или никогда не было подключено
        print("Ошибка! Соединение закрыто или никогда не было подключено.")
    except:
        print("Ошибка! Неизвестная ошибка при попытке запроса.")
```

Также в вызовы функций можно подставлять свои параметры, и получать значения.

Немного дополнительной информации

Транзакции

При работе с СУБД есть такое понятие как транзакция. Это по сути любое действие, совершаемое с БД. Проблема в том, что создание транзакций – достаточно тяжелая для СУБД операция, и при большом количестве запросов может снижать ее быстродействие.

Например, у вас есть 5 запросов к БД, которые выполняются друг за другом, и если в одном из них возникнет ошибка, откатить все изменения.

Если каждый запрос будет проходить в отдельной транзакции, то СУБД создаст 5 транзакций, а нам по сути надо сделать одно действие. Объединение всех действий в одну транзакцию позволит повысить быстродействие.

Если в одном из запросов возникнет ошибка, откатить все предыдущие вызовы будет нелегко, если они были в отдельных транзакциях. Но если все выполняется в пределах одной, то достаточно будет выполнить функцию *rollback()*, которая откатит изменения, произошедшие в транзакции.

Итак, использование транзакций позволяет достичь как минимум двух вещей:

- Безопасность при выполнении нескольких запросов, если между ними есть зависимость
- Повышение быстродействия путем упаковки всех действий в одну транзакцию.

Для того, чтобы начать управлять транзакциями вручную нужно создать объект xact, через который в дальнейшем будут управляться транзакции:

tr = self._connection.xact()

Чтобы применить выполненные изменения, нужно вызвать

```
tr.commit();
```

А если у вас что-то пошло не так и надо все вернуть назад, то нужно вызвать

```
tr.rollback();
```

И, конечно же, *не забывайте закрывать за собой обращения к БД.*