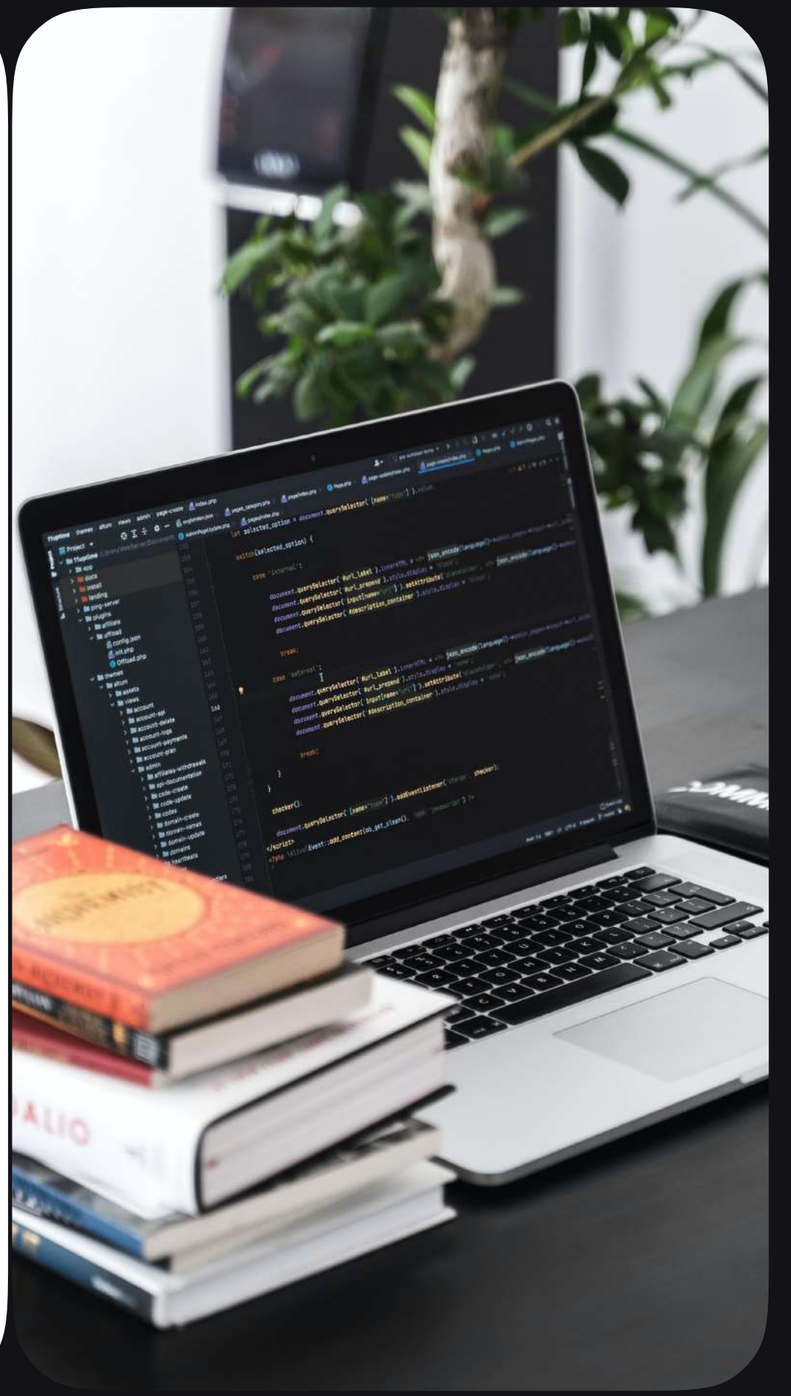
Занятие 9

SQL: простые операторы. Табличные данные в PyQT



Статические SQL запросы

Динамические SQL запросы

SQL запрос с форматированием строк

Динамические запросы с параметрами-заполнителями

Навигация по записям в запросе

Закрытие и удаление подключений к базе данных

Статические SQL запросы

На прошлом занятии мы:

- установили имя базы данных на «contacts.sqlite» и открыли соединение
- создали новую таблицу под названием contacts в вашей базе данных
- создали в таблице 4 столбца

Колонка	Содержание
id	An integer with the table’s primary key
name	Строка с именем контакта
job	Строка с названием должности контакта
email	Строка с адресом электронной почты контакта

Статические SQL запросы

```
import sys

from PyQt5.QtSql import QSqlDatabase, QSqlQuery

# Create the connection
con = QSqlDatabase.addDatabase("QSQLITE")
con.setDatabaseName("contacts.sqlite")

# Open the connection
if not con.open():
    print("Database Error: %s" % con.lastError().databaseText())
    sys.exit(1)

# Create a query and execute it right away using .exec()
createTableQuery = QSqlQuery()
createTableQuery.exec(
    """
    CREATE TABLE contacts (
        id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
        name VARCHAR(40) NOT NULL,
        job VARCHAR(50),
        email VARCHAR(40) NOT NULL
    )
    """
)

print(con.tables())
```

Статические SQL запросы

Мы использовали **статический SQL запрос** — передали в .exec() строковый SQL-запрос в качестве аргумента, такой запрос не получает никаких параметров извне запроса.

```
createTableQuery.exec(
    """
    CREATE TABLE contacts (
        id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
        name VARCHAR(40) NOT NULL,
        job VARCHAR(50),
        email VARCHAR(40) NOT NULL
    )
    """
)
```

Динамические SQL запросы

Часто нужно извлекать данные в ответ на определенные входные параметры.

Запросы, принимающие параметры во время выполнения, называются динамическими запросами.

Использование параметров позволяет точно настроить запрос и получить данные в ответ на определенные значения параметров. Разные значения приведут к разным результатам.

Подходы использования входных параметров в SQL запросах:

1. Динамический запрос с использованием форматирования строк для интерполяции значений параметров.
2. Подготовка запроса с помощью параметров-заполнителей, а затем связывание конкретных значений с параметрами.

SQL запрос с форматированием строк

Используются f-строки для создания динамического запроса путем интерполяции определенных значений в строковый SQL-запрос.

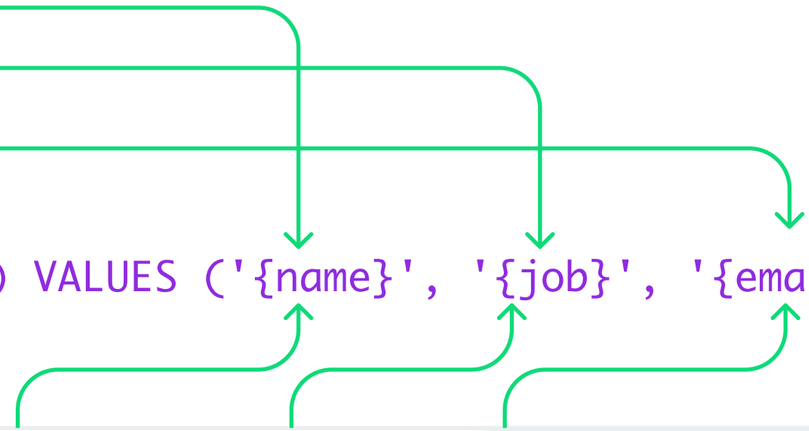
Добавим данные в нашу таблицу контактов с помощью запроса с форматированием строк.

```
from PyQt5.QtSql import QSqlQuery, QSqlDatabase
con = QSqlDatabase.addDatabase("QSQLITE")
con.setDatabaseName("contacts.sqlite")
con.open()
name = "User"
job = "Teacher"
email = "teacher@example.com"
query = QSqlQuery()
query.exec(
...     f"""INSERT INTO contacts (name, job, email)
...     VALUES ('{name}', '{job}', '{email}')"""
... )
```

Теперь таблица содержит данные о User

SQL запрос с форматированием строк

```
name = "User"
job = "Teacher"
email = "teacher@example.com"
query = QSqlQuery()
query.exec(f"""INSERT INTO contacts (name, job, email) VALUES ('{name}', '{job}', '{email}')""")
```



Подставляемые значения должны быть строковые, поэтому используются ‘ ‘ вокруг заполнителя в f-строке

SQL запрос с форматированием строк

При создании динамических запросов методом форматирования строк. Необходимо быть уверенным, что значения параметров поступают из надежного источника. В противном случае возникает риск SQL-инъекции.

SQL-инъекция — это уязвимость, которая возникает, когда у злоумышленника появляется возможность модифицировать SQL запрос. Например, если пользователь узнает имя передаваемого параметра в SQL запросе, он может заменить данные на свои.

Таким образом SQL-инъекция может привести как минимум к краже любых данных из базы (например, списка пользователей). В особо запущенных случаях она позволяет менять данные в базе, читать или создавать файлы с произвольным содержимым на сервере и даже выполнять какие-то команды.

Выполнение динамических запросов: Параметры-заполнители

Использование параметров-заполнителей для выполнения динамических запросов требует предварительной подготовки запросов с использованием шаблона с заполнителями для параметров.

PyQt поддерживает два стиля заполнителей параметров:

1. Стиль Oracle — использует именованные заполнители, такие как :name или :email.
2. Стиль ODBC — использует знак вопроса (?) в качестве позиционного заполнителя.

- Стили не могут быть смешаны в одном запросе.

ODBC означает — «Открытое подключение к базе данных»

Создание динамического запроса в PyQt:

1

Создайте шаблон с заполнителем для каждого параметра запроса и передайте этот шаблон в качестве аргумента в функцию **.prepare()**, которая анализирует, компилирует и подготавливает шаблон запроса к выполнению.

Если шаблон имеет какие-либо проблемы, такие как синтаксическая ошибка SQL, то **.prepare()** не удается скомпилировать шаблон и возвращает False.

Если процесс подготовки проходит успешно, то функция prepare() возвращает True.

2

Передайте определенное значение каждому параметру, используя **.bindValue()** с именованными или позиционными параметрами или используя **.addBindValue()** с позиционными параметрами. **.bindValue()** имеет следующие два варианта:

- .bindValue(placeholder, val)
- .bindValue(pos, val)

1. .bindValue(placeholder, val)
2. .bindValue(pos, val)

В первом варианте заполнитель представляет собой заполнитель в стиле **Oracle**. Во втором варианте pos представляет собой целое число на основе нуля с позицией параметра в запросе. В обоих вариантах **val** содержит значение, которое должно быть привязано к определенному параметру.

.addBindValue() добавляет значение в список заполнителей с помощью позиционной привязки. Это означает, что порядок вызовов функции **.addBindValue()** определяет, какое значение будет привязано к каждому параметру-заполнителю в подготовленном запросе.

Чтобы использовать подготовленные запросы, используем инструкцию INSERT INTO SQL для заполнения базы данных некоторыми образцами данных.

```
insertDataQuery = QSqlQuery()
insertDataQuery.prepare(
    """
    INSERT INTO contacts (
        name,
        job,
        email
    )
    VALUES (?, ?, ?)
    """
)

data = [
    ("Student1", "school student", "student1@example.com"),
    ("Student2", "school student", "student2@example.com"),
    ("Student3", "school student", "student3@example.com"),
    ("Student4", "school student", "student4@example.com"),
]

for name, job, email in data:
    insertDataQuery.addBindValue(name)
    insertDataQuery.addBindValue(job)
    insertDataQuery.addBindValue(email)
    insertDataQuery.exec()
```

1 Создали объект QSqlQuery.

2 Вызвали функцию .prepare() для объекта запроса.

В этом случае для заполнителей используется стиль ODBC. Запрос будет принимать значения для name, job и email, поэтому вам понадобятся три заполнителя. Поскольку столбец id является автоинкрементированным целым числом, вам не нужно вводить для него значения.

3 Создаём некоторые образцы данных для заполнения базы данных. Данные содержат список кортежей, и каждый кортеж содержит три элемента: name, job и email каждого контакта.

4 Привязываем значения, которые необходимо передать каждому заполнителю, а затем вызвать **.exec()** для выполнения запроса. Для этого **используем цикл for**. Заголовок цикла распаковывает каждый кортеж в данных на три отдельные переменные с удобными именами. Затем вызываем функцию **.addBindValue()** для объекта запроса, чтобы привязать значения к заполнителям.

используются позиционные заполнители, поэтому порядок, в котором вызывается .addBindValue (), будет определять порядок, в котором каждое значение передается соответствующему заполнителю.

Такой подход к созданию динамических запросов удобен, когда необходимо настроить свои запросы, используя значения, полученные из входных данных пользователя.

Всякий раз, когда принимается ввод пользователя для выполнения запроса к базе данных- возникает угроза безопасности SQL-инъекции.

В PyQt объединение **.prepare** (), **.bindValue** () и **.addBindValue()** полностью защищает от атак SQL-инъекций.

Практика

Структура таблицы

С помощью динамического запроса добавьте 5 записей в базу данных Animals.

id	title	group	max_speed
1	cheetah	cats	120
2	pronghorn	ungulates	115
3	gazelle	ungulates	105
4	antelope	ungulates	90
5	lion	cats	80

Навигация по записям в запросе

QSqlQuery предоставляет набор методов навигации , которые можно использовать для перемещения по записям в результате запроса:

Методика	Извлечение
.next()	Следующая запись
.previous()	Предыдущая запись
.first()	Первая запись
.last()	Последняя запись
.seek(index, relative=False)	Рекорд на позиции index

Пример: Навигация по записям в запросе

Создадим соединение с базой данных (мы ранее добавили в базу несколько записей)

```
from PyQt5.QtSql import QSqlDatabase, QSqlQuery
con = QSqlDatabase.addDatabase("QSQLITE")
con.setDatabaseName("contacts.sqlite")
con.open()
```

Создим **QSqlQuery** объект и выполним запрос

```
query = QSqlQuery()
query.exec("SELECT name, job, email FROM contacts")
```

Запрос возвращает данные о **name**, **job** и **email** все контакты, хранящиеся в **contacts** таблице. После **.exec()** является активным запросом. Можно перемещаться по записям в этом запросе, используя любой из методов навигации.

Пример: Навигация по записям в запросе

Также можно получить данные в любом столбце записи, используя .value():

```
>>> query.first()
True
>>> # Named indices for readability
>>> name, job, email = range(3)
>>> # Retrieve data from the first record
>>> query.value(name)
'User'
>>> # Next record
>>> query.next()
True
>>> query.value(job)
'Teacher'
>>> # Last record
>>> query.last()
True
>>> query.value(email)
'student4@example.com'
```

Пример: Навигация по записям в запросе

С помощью методов навигации возможно перемещаться по результату запроса.
С помощью .value() возможно получать данные в любом столбце данной записи.

Можно перебирать все записи в своем запросе, используя while цикл вместе с .next():

```
>>> query.exec()
True
>>> while query.next():
...     print(query.value(name), query.value(job), query.value(email))
...
User Teacher teacher@example.com
Student1 school student student1@example.com
…
```

Пример: Навигация по записям в запросе

С помощью **.next()** происходит перемещение по всем записям в результате запроса. После того, как перебрали записи в результате запроса, **.next()** начинает возвращаться, **False** пока вы не будет запущен **.exec()** снова.

Вызов **.exec()** извлекает данные из базы данных и помещает внутренний указатель объекта запроса на одну позицию перед первой записью, поэтому при вызове **.next()** вы снова получаете первую запись.

Вы также можете выполнить цикл в обратном порядке, используя **.previous()**.

.previous() работает аналогично **.next()**, но итерация выполняется в обратном порядке. Другими словами, цикл переходит от позиции указателя запроса обратно к первой записи.

Пример: Навигация по записям в запросе

Иногда может потребоваться получить индекс, который идентифицирует данный столбец в таблице, используя имя этого столбца. Для этого необходимо вызвать **.indexOf()** возвращаемое значение **.record()**:

```
>>> query.first()
True
>>> # Get the index of name
>>> name = query.record().indexOf("name")
>>> query.value(name)
'User'
```

Обращение к **.indexOf()** результату **.record()** возвращает индекс «**name**» столбца. Если «**name**» не существует, то **.indexOf()** возвращается –1.
Это удобно, когда вы используете **SELECT*** оператор, в котором порядок столбцов неизвестен.

Навигация по записям в запросе

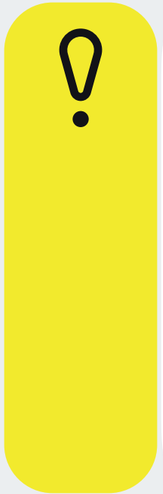
Когда работа с объектом запроса закончена, необходимо отключить его, вызвав .finish(). Это освободит системную память, связанную с рассматриваемым объектом запроса.

```
>>> # Finish the query object if unneeded
>>> query.finish()
>>> query.isActive()
False
```

Закрытие и удаление подключений к базе данных

Необходимо закрыть соединение после использования и освободить ресурсы, связанные с этим соединением, например системную память.

Чтобы закрыть соединение в PyQt, вы вызываете **.close()** соединение.



Этот метод закрывает соединение и освобождает все полученные ресурсы. Это также делает недействительными любые связанные QSqlQuery объекты, потому что они не могут работать должным образом без активного соединения.

Закрытие и удаление подключений к базе данных

```
>>> from PyQt5.QtSql import QSqlDatabase
>>> con = QSqlDatabase.addDatabase("QSQLITE")
>>> con.setDatabaseName("contacts.sqlite")
>>> con.open()
True
>>> con.isOpen()
True
>>> con.close()
>>> con.isOpen()
False
```

Чтобы закрыть соединение и освободить все связанные с ним ресурсы вызовите **.close()**. Чтобы убедиться, что соединение закрыто, вызовите **.isOpen()**.

Закрытие и удаление подключений к базе данных

Обратите внимание, что QSqlQuery объекты остаются в памяти после закрытия связанного с ними соединения, поэтому необходимо сделать запросы неактивными, вызвав **.finish()** или **.clear()**, или удалив **QSqlQuery** объект перед закрытием соединения.

- Возможно повторно открыть и повторно использовать любое ранее закрытое соединение.
- **.close()** не удаляет соединения из списка доступных соединений, поэтому они остаются пригодными для использования.

Закрытие и удаление подключений к базе данных

Для того чтобы полностью удалить соединения с базой данных, используйте.removeDatabase().

Чтобы сделать это безопасно:

1. завершите запросы с помощью .finish()
2. закройте базу данных с помощью .close()
3. удалите соединение

Вы можете использовать **.removeDatabase(connectionName)** для удаления вызываемого соединения **connectionNamec** базой данных из списка доступных соединений. Удаленные соединения больше не доступны для использования в текущем приложении.

Практика

1. Создайте запрос к базе Animals, выведите на экран первую запись в полученном ответе.
2. Создайте запрос к базе Animals, выведите на экран последнюю запись в полученном ответе.
3. Выведите на экран все записи из базы Animals.