**Работаем с SQLite базой данных из Python**

Так как различных СУБД достаточно много, крайне неудобно было бы, если при переходе на новую СУБД приходилось бы с нуля изучать библиотеку для работы с ней. Чтобы избежать таких ситуаций, есть специальный стандарт **PEP 249** (Python Database API Specification v2.0), в котором, помимо всего прочего, описано, какой интерфейс должна предоставлять программисту любая библиотека для работы с базами данных. Поэтому, какую бы СУБД вы не выбрали для управления хранением данных вашего приложения, принципы работы с ней будут очень похожи. Для работы с SQLite из Python используется библиотека sqlite3, которая реализует этот стандарт.

PEP 249 оперирует такими понятиями, как подключения и курсоры:

* **Подключение** — объект, в котором чаще всего указывается либо путь к файлу, либо путь к серверу. Он отвечает только за подключение к БД и, соответственно, отключение от нее
* **Курсор** — объект, в котором непосредственно производится работа с БД

Напишем программу (пока что без графического интерфейса), которая получает результаты одного из рассмотренных выше запросов и выводит их в консоль.

# Импорт библиотеки

import sqlite3

# Подключение к БД

con = sqlite3.connect("films\_db.sqlite")

# Создание курсора

cur = con.cursor()

# Выполнение запроса и получение всех результатов

result = cur.execute("""SELECT \* FROM films

WHERE year = 2010""").fetchall()

# Вывод результатов на экран

for elem in result:

print(elem)

con.close()

(248, 'Алиса в стране чудес', 2010, 13, '')

(4382, 'Железный человек 2', 2010, 11, '')

(9138, 'Ноттингем', 2010, 11, '')

(15495, 'Утомленные солнцем: Предстояние', 2010, 2, '')

Как можно заметить, результат запроса — это список кортежей.

Метод .fetchall() возвращает все полученные элементы. Существует еще метод .fetchone(), возвращающий, как несложно догадаться, только первый элемент, и метод .fetchmany(n), возвращающий n первых записей.

Для запросов очень часто необходимо указывать какие-либо параметры, в нашем случае: год выпуска, продолжительность фильма и т. д. Для этого существует удобный синтаксис. Вместо значения в запросе указывается вопросительный знак, а затем вторым параметром в итерируемом объекте (чаще всего в кортеже) указываются необходимые значения для подстановки.

result = cur.execute("""SELECT \* FROM films

WHERE year = ? and duration > ?""", (2010, 90)).fetchall()

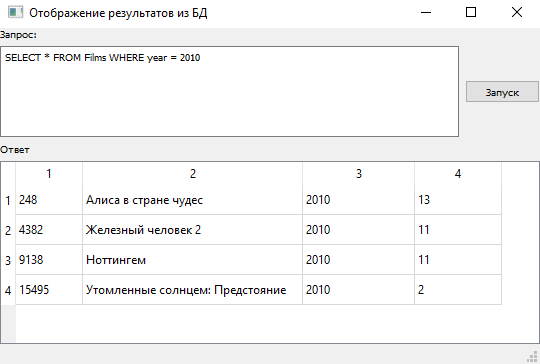
Важно не забыть, что, если мы указываем кортеж из одного элемента, нам все равно необходимо после него поставить запятую.

result = cur.execute("""SELECT \* FROM Films

WHERE year = ?""", (2009,)).fetchall()

Давайте добавим к нашему приложению графический пользовательский интерфейс. Напишем программу, которая будет отображать результаты введенного запроса в таблице QTableWidget.

С помощью QtDesigner создадим интерфейс: поле для ввода запроса, таблица для отображения результатов и кнопка для запуска выполнения запроса. Работа с данными из таблицы базы данных с помощью виджета QTableWidget полностью аналогична тому, что мы рассматривали на прошлом уроке во время работы с .csv-файлами.



import sqlite3

import sys

from PyQt5 import uic

from PyQt5.QtWidgets import QApplication

from PyQt5.QtWidgets import QMainWindow, QTableWidgetItem

class DBSample(QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

uic.loadUi('UI1.ui', self)

self.connection = sqlite3.connect("films\_db.sqlite")

self.pushButton.clicked.connect(self.select\_data)

# По умолчанию будем выводить все данные из таблицы films

self.textEdit.setPlainText("SELECT \* FROM films")

self.select\_data()

def select\_data(self):

# Получим результат запроса,

# который ввели в текстовое поле

query = self.textEdit.toPlainText()

res = self.connection.cursor().execute(query).fetchall()

# Заполним размеры таблицы

self.tableWidget.setColumnCount(5)

self.tableWidget.setRowCount(0)

# Заполняем таблицу элементами

for i, row in enumerate(res):

self.tableWidget.setRowCount(

self.tableWidget.rowCount() + 1)

for j, elem in enumerate(row):

self.tableWidget.setItem(

i, j, QTableWidgetItem(str(elem)))

def closeEvent(self, event):

# При закрытии формы закроем и наше соединение

# с базой данных

self.connection.close()

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

ex = DBSample()

ex.show()

sys.exit(app.exec())

**Возможности PyQT по работе с базами данных**

В курсе мы и дальше будем говорить про работу с базами данных как можно более обще, но нельзя не отметить то, что у PyQT (как и у некоторых других больших библиотек) есть своя универсальная надстройка для работы с БД — модуль PyQt5.QtSql. Рассмотрим простейший пример: отобразим все данные из таблицы films.

import sys

from PyQt5.QtSql import QSqlDatabase, QSqlTableModel

from PyQt5.QtWidgets import QWidget, QTableView, QApplication

class Example(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

# Зададим тип базы данных

db = QSqlDatabase.addDatabase('QSQLITE')

# Укажем имя базы данных

db.setDatabaseName('films\_db.sqlite')

# И откроем подключение

db.open()

# QTableView - виджет для отображения данных из базы

view = QTableView(self)

# Создадим объект QSqlTableModel,

# зададим таблицу, с которой он будет работать,

# и выберем все данные

model = QSqlTableModel(self, db)

model.setTable('films')

model.select()

# Для отображения данных на виджете

# свяжем его и нашу модель данных

view.setModel(model)

view.move(10, 10)

view.resize(617, 315)

self.setGeometry(300, 100, 650, 450)

self.setWindowTitle('Пример работы с QtSql')

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

ex = Example()

ex.show()

sys.exit(app.exec())

Общий принцип работы с модулем следующий:

* Создаем и настраиваем объект QSqlDatabase для связи с базой данных
* С помощью QSqlTableModel или QSqlQueryModel получаем и управляем данными из базы данных
* С помощью виджета QTableView отображаем данные в табличном виде

Подробнее почитать про работу с базами данных, использованием модуля QtSql можно почитать в официальной [документации](https://doc.qt.io/qt-5/sql-programming.html).

Давайте снова вернемся в SQLiteStudio, откроем модельную [базу данных](https://yastatic.net/s3/lyceum/content/images/second-year/pyqt-6/films_db.sqlite) с информацией о фильмах и выполним несколько запросов.

**Добавление записей**

На прошлом уроке мы познакомились только с одной частью работы с SQL — получение информации с помощью специального запроса SELECT. Но перед тем, как получать информацию из базы, необходимо ее туда поместить.

Для этого существует оператор INSERT. Его синтаксис в общем случае выглядит так:

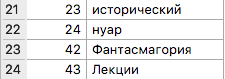
INSERT INTO имя\_таблицы(названия\_полей\*) VALUES(значения)

Названия полей могут не указываться, тогда значения по умолчанию подставятся в поля по порядку. Давайте рассмотрим несколько примеров:

INSERT INTO genres(id,title) VALUES(42,'Фантасмагория')

INSERT INTO genres(title) VALUES('Лекции')

И теперь, когда мы захотим вывести таблицу жанров, мы увидим, что там появились новые значения.



Но откуда взялось значение id, равное 43? Мы же его нигде не указывали? Дело в том, что в нашей таблице поле id является **автоинкрементным**, то есть при создании нового элемента берется максимальный из уже созданных индексов, увеличивается на единицу и присваивается новому элементу.

Часто в таблицу необходимо вставить не одно значение, а несколько. Для это не нужно вызывать оператор INSERT несколько раз, а достаточно через запятую перечислить все значения, которые необходимо добавить. Например, так:

INSERT INTO genres VALUES (45, 'Научные'), (46, 'Сказки')

**Изменение записей**

Часто случается, что информация, хранящаяся в базе данных, нуждается в изменении. Для таких запросов используется оператор UPDATE. Его синтаксис в общем случае выглядит следующим образом:

UPDATE имя\_таблицы

SET название\_колонки = новое\_значение

WHERE условие

Обратите внимание: если не указать условие обновления, изменятся **абсолютно все** записи в таблице.

Перейдем к примеру. Предположим, что мы посмотрели режиссерскую версию фильма «Аватар», и теперь хотим указать новую продолжительность фильма. Разумеется, можно сохранить информацию, удалить запись и создать новую. Можно, но это крайне неудобно. Так что обновим значение поля duration для фильма «Аватар» .

UPDATE films

SET duration = '162'

WHERE title = 'Аватар'

https://yastatic.net/s3/lyceum/content/images/second-year/pyqt-7/qt-7-2.png

**Удаление записей**

Но бывают и ситуации, когда какая-либо информация уже совершенно точно не нужна. Тогда ее необходимо удалить из БД, чтобы она не занимала лишнее пространство в памяти. Для этого используется оператор DELETE.

Например, чтобы удалить все фильмы, выпущенные до 1985 года, необходимо написать следующий запрос:

DELETE from films

where year < 1985

Обратите внимание: если не указать условие, будут удалены **абсолютно все** записи в таблице. Во многие менеджеры даже встроен запрос подтверждения действия при выполнении запроса DELETE без части WHERE.

**Модификация из Python**

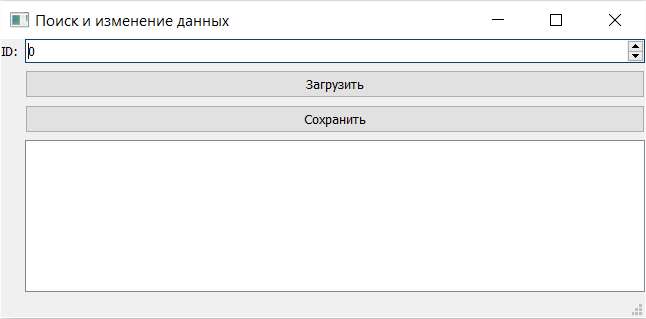
**Редактирование данных**

В прошлом уроке мы научились получать информацию из БД и отображать ее в таблице. Но достаточно часто пользователям нужно не только просматривать, но и модифицировать и удалять информацию. Напишем программу, которая позволит получить фильм по его идентификатору, изменить его поля и сохранить.

Создадим интерфейс с помощью QtDesigner. Разместим на форме:

* Виджет для ввода id записи, например, QSpinBox
* Две кнопки. Одну — для получения информации из базы данных, другую — для сохранения изменений
* QTableWidget для отображения информации

Упакуем это все с использованием какого-нибудь Layout. В результате у нас должно получиться что-то вроде такого:



Начнем писать наше приложение с создания класса и заглушек для методов.

import sqlite3

import sys

from PyQt5 import uic

from PyQt5.QtWidgets import QApplication

from PyQt5.QtWidgets import QMainWindow, QTableWidgetItem

class MyWidget(QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

uic.loadUi("UI1.ui", self)

self.con = sqlite3.connect("films\_db.sqlite")

self.pushButton.clicked.connect(self.update\_result)

self.tableWidget.itemChanged.connect(self.item\_changed)

self.pushButton\_2.clicked.connect(self.save\_results)

self.modified = {}

self.titles = None

def update\_result(self):

pass

def item\_changed(self, item):

pass

def save\_results(self):

pass

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

ex = MyWidget()

ex.show()

sys.exit(app.exec())

Для того чтобы обработать изменение содержимого ячейки, воспользуемся сигналом self.tableWidget.itemChanged.

Начнем с метода получения данных update\_result(). Тут все просто. Единственное, на что стоит обратить внимание — получение данных о заголовках столбцов из параметра курсора.

def update\_result(self):

cur = self.con.cursor()

# Получили результат запроса, который ввели в текстовое поле

result = cur.execute("SELECT \* FROM films WHERE id=?",

(item\_id := self.spinBox.text(), )).fetchall()

# Заполнили размеры таблицы

self.tableWidget.setRowCount(len(result))

# Если запись не нашлась, то не будем ничего делать

if not result:

self.statusBar().showMessage('Ничего не нашлось')

return

else:

self.statusBar().showMessage(f"Нашлась запись с id = {item\_id}")

self.tableWidget.setColumnCount(len(result[0]))

self.titles = [description[0] for description in cur.description]

# Заполнили таблицу полученными элементами

for i, elem in enumerate(result):

for j, val in enumerate(elem):

self.tableWidget.setItem(i, j, QTableWidgetItem(str(val)))

self.modified = {}

Теперь напишем метод, который будет отслеживать изменения ячеек. Будем записывать в словарь пару, состоящую из наименования поля и нового значения.

def item\_changed(self, item):

# Если значение в ячейке было изменено,

# то в словарь записывается пара: название поля, новое значение

self.modified[self.titles[item.column()]] = item.text()

Обратите внимание на параметр item. При изменении ячейки в него будет попадать объект класса QTableWidgetItem, у которого можно получить интересующую нас информацию с помощью методов column() и text().

Затем нам необходимо сохранить полученные результаты в БД. Так как заранее мы не знаем, какие поля были модифицированы, то мы не можем использовать конструкцию с вопросительным знаком. Так что просто «склеим» необходимый нам запрос, например, используя обыкновенную конкатенацию строк.

def save\_results(self):

if self.modified:

cur = self.con.cursor()

que = "UPDATE films SET\n"

que += ", ".join([f"{key}='{self.modified.get(key)}'"

for key in self.modified.keys()])

que += "WHERE id = ?"

print(que)

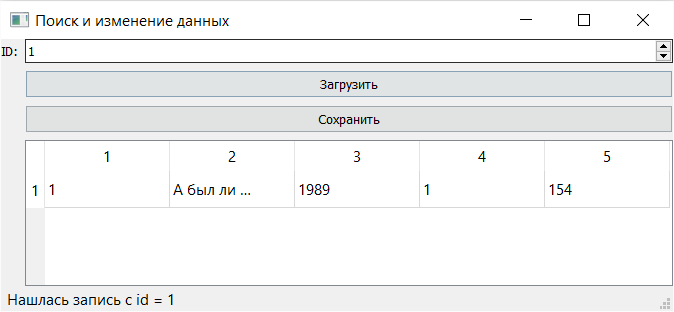
cur.execute(que, (self.spinBox.text(),))

self.con.commit()

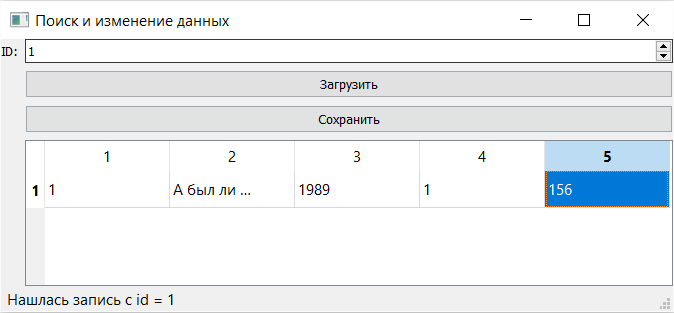
self.modified.clear()

Одна из самых важных строк в save\_results — вызов метода commit() у объекта-соединения с базой данных. По умолчанию все изменения с базой данных проводятся в памяти и не записываются в сам файл, поэтому если мы совершим манипуляцию с данными и перезапустим приложение, все правки пропадут. Чтобы такого не происходило, изменения надо зафиксировать. За это как раз и отвечает метод commit().

Изначальное отображение:



После изменения и сохранения:



На практике изменение записи непосредственно в таблице используется не так часто. Обычно для редактирования существующей или создания новой записи производится в отдельной форме. Так лучше делать по нескольким причинам:

1. Более широкие возможности для пользователя благодаря правильному подбору виджетов для каждого отдельного типа данных.
2. Меньше возможности пользователю отредактировать не те данные.
3. Проще отслеживать и сохранять изменения.

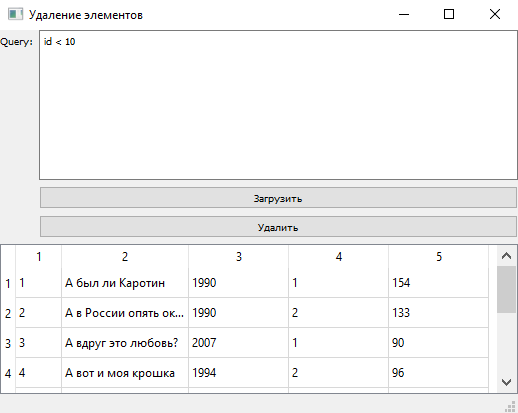
**Удаление данных**

Теперь рассмотрим программу для удаления выделенного элемента.

Часто перед удалением какой-либо информации различные программы запрашивают у пользователя подтверждение. Сделаем программу с похожей функциональностью. Будем работать только с таблицей films. Создадим интерфейс нашей программы. Разместим на форме следующие виджеты:

* Виджет для ввода параметров фильтрации, например, например, QTextEdit
* Две кнопки. Одну — для получения информации из базы данных, другую — для удаления записей
* QTableWidget для отображения информации

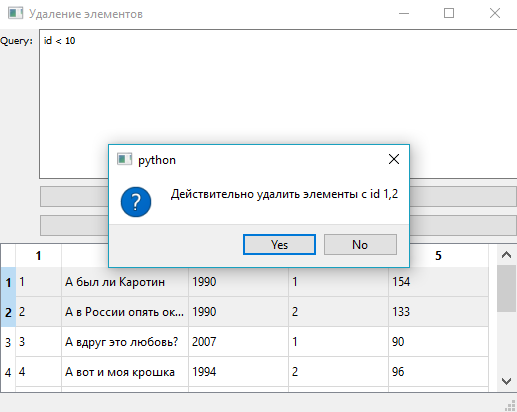
Снова запакуем в какой-нибудь Layout. Результат будет выглядеть как-то так:



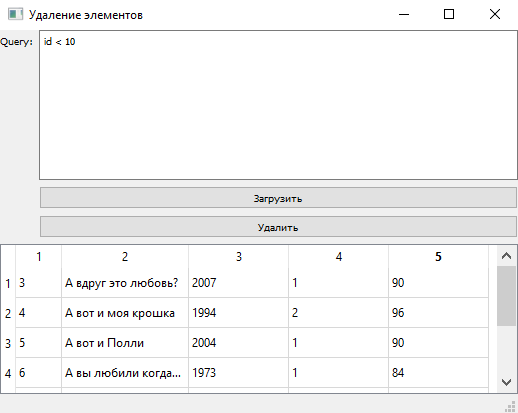
В поле для ввода будем вводить только условие (то, что будет идти после оператора WHERE). А сам запрос будет формироваться так:

queue = "SELECT \* FROM films WHERE " + self.textEdit.toPlainText()

Пользователь может выделить одно или несколько значений, а затем нажать на кнопку «Удалить». После этого открывается окно с подтверждением, в котором указаны все ID, которые были выделены.



В случае положительного ответа при повторном запросе удаленные элементы уже не отобразятся.



Основной метод, отличающий этот пример от предыдущего — delete\_elem() для удаления выделенных элементов. Рассмотрим его поподробнее.

def delete\_elem(self):

# Получаем список элементов без повторов и их id

rows = list(set([i.row() for i in self.tableWidget.selectedItems()]))

ids = [self.tableWidget.item(i, 0).text() for i in rows]

# Спрашиваем у пользователя подтверждение на удаление элементов

valid = QMessageBox.question(

self, '', "Действительно удалить элементы с id " + ",".join(ids),

QMessageBox.Yes, QMessageBox.No)

# Если пользователь ответил утвердительно, удаляем элементы.

# Не забываем зафиксировать изменения

if valid == QMessageBox.Yes:

cur = self.con.cursor()

cur.execute("DELETE FROM films WHERE id IN (" + ", ".join(

'?' \* len(ids)) + ")", ids)

self.con.commit()

В первой строке мы получаем список строк без повторов. Откуда взялись повторы? Выделяя строку, мы выделяем все элементы строки (все ячейки), соответственно, в список добавляется номер строки столько раз, сколько в ней элементов.

Поскольку номера строк очень часто не совпадают с идентификаторами, во второй строке мы получаем id записей, по которым мы и будем производить удаление, непосредственно из значений ячейки в нулевой колонке.

Теперь нам необходимо показать пользователю форму с подтверждением своего действия. Можно воспользоваться уже знакомым нам QInputDialog и написать что-то вроде:

answer, ok\_pressed = QInputDialog.getItem(

self, "Подтверждение удаления",

"Вы точно хотите удалить элементы с id " + ",".join(ids),

("нет", "да"), 1, False)

Однако для отображения пользователю диалога о подтверждении какого-либо действия или информационного сообщения о каком-нибудь системном событии (например, об ошибке) в PyQT есть более привычный и подходящий инструмент. Это класс QMessageBox, у которого с QInputDialog общий родитель — QDialog. Поэтому импортируем его из PyQt5.QtWidgets, вызовем метод question(), в который передаются следующие параметры:

* Родитель — self
* Заголовок — обычно передается пустое поле, если мы хотим задать пользователю вопрос
* Текст вопроса
* Варианты ответов — QMessageBox.Yes, QMessageBox.No

Возможности QMessageBox достаточно широки, рекомендуем ознакомиться с ними в [документации](https://doc.qt.io/qt-5/qmessagebox.html).

После того как пользователь нажмет на одну из кнопок, результат будет занесен в переменную valid. А затем будет выполнена проверка и удаление.

Важно обратить внимание на то, что текст запроса формируется с использованием и конкатенации строк, и оператора "?". В данной задаче мы также столкнулись с методом commit() у соединения с базой данных. Не забывайте фиксировать изменения после изменения данных или их удаления.