МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«МИРЭА - Российский технологический университет»
(РТУ МИРЭА)

Тарланов А. Т.

Базы данных и дополнительные компоненты библиотеки PyQT

Учебно-методическое пособие

Москва 2021

УДК 004.432
ББК 32.973
Т 20

Тарланов А.Т. Базы данных и дополнительные компоненты библиотеки PyQT [Электронный ресурс]: Учебно-методическое пособие / Тарланов А.Т. – М.: МИРЭА – Российский технологический университет, 2021. –1 электрон.опт. диск (CD-ROM).

Целью настоящего учебно-методического пособия является изучение кроссплатформенной библиотеки QT, в частности, ее реализации под язык программирования Python – PyQT. Рассматриваются следующие темы:

- Технологии хранения однотипных записей в файлах, форматам с фиксированной и с произвольной длиной записи (DSV, TSV, CSV). Форматы с произвольной длиной записи, методы работы с ними при помощи строковых функций и специализированной библиотеки csv;
- Базы данных и язык SQL;
- Взаимодействие с пользователем на новом уровне: работа с клавиатурой и мышкой. Сборка программы в exe-файл;
- Дополнительные компоненты PyQT на примере PyQT Graph — библиотеки для построения графиков. Проигрывание музыки в своих приложениях;

Учебно-методическое пособие предназначено для бакалавров, обучающихся по направлению 09.03.02 «Информационные системы и технологии», и специалистов, обучающихся по специальности 10.05.05 «Безопасность информационных технологий в правоохранительной сфере».

Учебно-методическое пособие издаётся в авторской редакции.

Авторский коллектив: Тарланов Арслан Тарланович.

Рецензенты:

1.Оцоков Шамиль Алиевич, д.т.н., профессор кафедры вычислительных машин, систем и сетей, ФГБУ ВО «Национальный исследовательский университет МЭИ»
2. Никонов Вячеслав Викторович, к.т.н., и.о. заведующего кафедрой КБ-9 «Предметно-ориентированные информационные системы» института КБСП ФГБУ ВО РТУ МИРЭА

Системные требования:
Наличие операционной системы Windows, поддерживаемой производителем.
Наличие свободного места в оперативной памяти не менее 128 Мб.
Наличие свободного места в памяти постоянного хранения (на жестком диске) не менее 30 Мб.
Наличие интерфейса ввода информации.
Дополнительные программные средства: программа для чтения pdf-файлов (Adobe Reader).
Подписано к использованию по решению Редакционно-издательского совета
МИРЭА — Российский технологический университет.
Обьем: 1.85 мб
Тираж: 10

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Данное учебно-методическое пособие посвящено кроссплатформенной библиотеке QT, в частности, ее реализации под язык программирования Python – PyQT. Рассматриваются следующие темы:

- Технологии хранения однотипных записей в файлах, форматам с фиксированной и с произвольной длиной записи (DSV, TSV, CSV). Форматы с произвольной длиной записи, методы работы с ними при помощи строковых функций и специализированной библиотеки csv;
- Базы данных и язык SQL;
- Взаимодействие с пользователем на новом уровне: работа с клавиатурой и мышкой. Сборка программы в exe-файл;
- Дополнительные компоненты PyQT на примере PyQTGraph — библиотеки для построения графиков. Проигрывание музыки в своих приложениях;
- Использование QThread для предотвращения зависания графического интерфейса пользователя.

Для работы с данным учебно-методическим пособием необходимо обладать базовыми знаниями по языку программирования Python и библиотеке PyQT.

1. Работа с простыми таблицами (csv). Работа с табличными данными в PyQT

Глава посвящена технологии хранения однотипных записей в файлах, форматам с фиксированной и с произвольной длиной записи (DSV, TSV, CSV) [1-3]. Особое внимание уделено форматам с произвольной длиной записи, методам работы с ними при помощи строковых функций и специализированной библиотеки csv.

1.1. Хранение записей в файлах

Любой файл может содержать последовательность байт, которая интерпретируется человеком и прикладными программами различным образом. Например, байт с десятичным значением 65 может означать и число 65, и код латинской буквы A в зависимости от договоренности по формату. Очень быстро стало нужно хранить в файлах массивы однородных объектов (записей), имеющих некоторый набор полей или характеристик. По сути такой массив представляет собой таблицу со строками и столбцами. Каждый столбец имеет свой, как правило, примитивный тип: целое или вещественное число, строка и т. д.

Пример таких структур — информация об учениках класса с указанием фамилии, имени, даты рождения, роста, среднего балла и т. д.

Очень важно, что формат при этом может быть совершенно различным. Главное, чтобы все программы, которые с ним работают, были написаны в соответствии со спецификацией этого формата, то есть **понимали** его.

В процессе эволюции из всех форматов для хранения массивов записей прижились два: с фиксированной и произвольной длиной записи. А подлинного искусства обработка и хранение таблиц достигли в реляционных базах данных, которые мы не рассматриваем на этой лекции.

1.2. Форматы с фиксированной длиной записи

Такие форматы практически всегда имеют байтовую, а не текстовую структуру.

Размер каждого поля фиксируется таким образом, чтобы иметь минимальную, но достаточную длину в байтах для представления всего диапазона значений. Как правило, на этом этапе возникают проблемы с выбором длины строковых данных.

Для примера рассмотрим расшифровку записи для хранения информации о школьнике:

```
Имя: строка(50)

Фамилия: строка(50)

Возраст: однобайтовое число

Пол: строка(1)

Адрес: строка(100)
```

Это прямая аналогия таких типов данных, как **записи** в Паскале и **структуры** в Си.

Как видно из примера, в мире нашей программы существует ряд ограничений, например, нет адресов, которые содержат более 100 символов.

Преимущества бинарных форматов с фиксированной длиной записи аналогичны преимуществам по работе с массивами: мы за константное время можем взять любую по счету запись, просто вычислив, по какому смещению в файле она располагается относительно его начала.

Фиксированный формат имеет и недостатки. Один из самых больших в том, что, как только появляется запись, одно из полей которой **не влезает** в предопределенную длину, мы не сможем использовать данный формат или вынуждены будем его видоизменить.

Причем, расширяя размер поля для данной записи, мы расширяем ее и для всех остальных, расходуя понапрасну ресурсы памяти.

Одна из самых широко известных ситуаций, связанных с фиксированным форматом, — проблема 2000 года. [4]

1.3. Форматы с произвольной длиной записи

Если длина полей в одной записи не фиксирована (например, мы говорим, что адрес клиента может быть любой длины), нам нужно решить две проблемы:

- Как мы будем понимать, где **границы полей**.
- Как мы будем понимать, где **границы записей.**

Решать их можно по-разному: например, перед каждым полем делать служебное поле из 4 байт, которое показывает размер в байтах следующего поля. Таким образом, в нашем случае поле не сможет содержать более 4 Гб данных.

Можно пойти другим путем и ввести специальные разделители. Поговорим про этот случай. Воспользуемся модельным примером и рассмотрим прайс-лист товаров в магазине «Икея».

Вот небольшой фрагмент этого файла:

```
> keywords      price   product_name

> МОРУМ, Ковёр, безворсовый    6999    МОРУМ

> МОРУМ, Ковёр, безворсовый    6999    МОРУМ

> ИДБИ, Придверный коврик      649     ИДБИ

> ХОДДЕ, Ковёр, безворсовый    1399    ХОДДЕ

> ОПЛЕВ, Придверный коврик     599     ОПЛЕВ

> ОПЛЕВ, Придверный коврик     599     ОПЛЕВ
```

Разделителем записей был выбран символ **новой строки**, а разделителем полей — **табуляция** (хотя на самом деле увидеть символы табуляции можно с трудом). В результате мы получили текстовый файл, который легко читается и человеком, и компьютерной программой.

Назовем такой формат **TSV**.

TSV

TSV (англ. tab separated values — «значения, разделенные табуляцией») — текстовый формат для представления таблиц баз данных. Каждая запись в таблице — строка текстового файла. Каждое поле записи отделяется от других символом табуляции, а точнее — горизонтальной табуляции.

TSV — форма более общего формата **DSV** («значения, разграниченные разделителем», от англ. delimiter separated values).

Программы для работы с электронными таблицами (MS Excel, LibreOffice Calc) поддерживают импорт из такого формата.

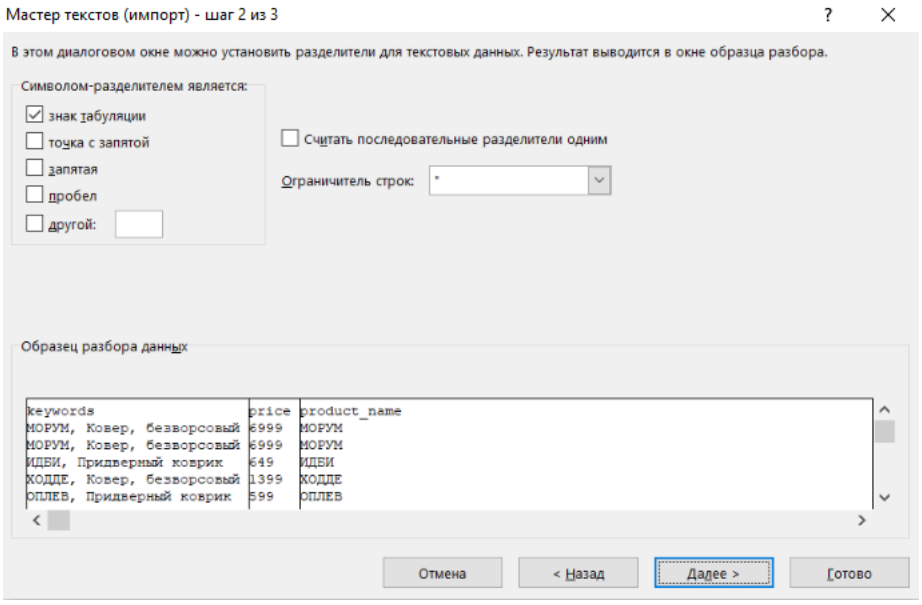


Рисунок 1.1

Посмотрим, как работать с таким форматом в Python. Тут нет ничего сложного, поскольку у нас есть мощная функциональность по работе со строками, в частности, метод `split`.

```
data = open('files/ikea.txt', encoding='utf-8').read()
for row in data.split('\n')[:10]:
    print(row.split('\t'))
```

```
['keywords', 'price', 'product_name']
['МОРУМ, Ковёр, безворсовый', '6999', 'МОРУМ']
['МОРУМ, Ковёр, безворсовый', '6999', 'МОРУМ']
['ИДБИ, Придверный коврик', '649', 'ИДБИ']
['ХОДДЕ, Ковёр, безворсовый', '1399', 'ХОДДЕ']
['ОПЛЕВ, Придверный коврик', '599', 'ОПЛЕВ']
['ОПЛЕВ, Придверный коврик', '599', 'ОПЛЕВ']
['ЮНКЭН, Брикеты', '89', 'ЮНКЭН']
['БУНСЁ, Детское садовое кресло', '1199', 'БУНСЁ']
['ИКЕА ПС ВОГЭ, Садовое лёгкое кресло', '1999', 'ИКЕА ПС ВОГЭ']
```

Вспомним списочные выражения и сразу сделаем «двумерный массив», а потом обратимся к цене пятого по счету товара:

```
table = [r.split('\t') for r in data.split('\n')]
print(table[5][1])
```

```
599
```

Мы можем также отсортировать элементы по цене и напечатать 10 самых дешевых товаров:

```
table = table[1:]
table.sort(key=lambda x: int(x[1]))
```

```
for r in table[:10]:
    print(r)
```

```
['СМОРИСКА, Стопка', '6', 'СМОРИСКА']
['СМОРИСКА, Стакан', '12', 'СМОРИСКА']
['ДИСТАНС, Контейнер', '12', 'ДИСТАНС']
['ДИСТАНС, Контейнер', '12', 'ДИСТАНС']
['ОППЕН, Миска', '25', 'ОППЕН']
['ДАРРОКА, Стакан', '25', 'ДАРРОКА']
['АНТАГЕН, Щетка для мытья посуды', '25', 'АНТАГЕН']
['ВАНКИВА, Рама', '25', 'ВАНКИВА']
['ХОППЛЁС, Доска разделочная', '27', 'ХОППЛЁС']
['ДАРРОКА, Стакан д/виски', '29', 'ДАРРОКА']
```

Зачем в примере выше необходимо было убрать первую строку из таблицы? И что получится, если ее там оставить?

Есть много встроенных функций, заменяющих стандартные лямбды, например, `operator.itemgetter` из модуля `operator`. Их удобно указывать в параметре `key`.

Одним из самых распространенных форматов **DSV** стал **CSV** — формат с разделителем полей-запятой (англ. comma separated values). Наш файл будет выглядеть в нем вот так:

```
> keywords,price,product_name

>"МОРУМ, Ковёр, безворсовый",6999,МОРУМ

>"МОРУМ, Ковёр, безворсовый",6999,МОРУМ

>"ИДБИ, Придверный коврик",649,ИДБИ
```

```
>"ХОДДЕ, Ковёр, безворсовый",1399,ХОДДЕ

>"ОПЛЕВ, Придверный коврик",599,ОПЛЕВ

>"ОПЛЕВ, Придверный коврик",599,ОПЛЕВ

>"ЮНКЭН, Брикеты",89,ЮНКЭН

>"БУНСЁ, Детское садовое кресло",1199,БУНСЁ

>"ИКЕА ПС ВОГЭ, Садовое лёгкое кресло",1999,ИКЕА ПС ВОГЭ
```

Для всех форматов DSV проблемой является символ-разделитель полей в данных. В этом случае вводят так называемый **разделитель текста**, в качестве которого выступают двойные кавычки, а если в поле встречается сам разделитель текста, то его удваивают. Например, **ООО "Светлана"** при записи в файл превращается в **"ООО ""Светлана"""**.

Вот почему в приведенном фрагменте CSV-файла первое поле в кавычках — внутри него есть запятые.

Нужно отметить, что в CSV могут быть другие разделители, например, точка с запятой. Очень часто это регулируется настройками операционной системы (параметр «Разделитель элементов списка» в ОС Windows):

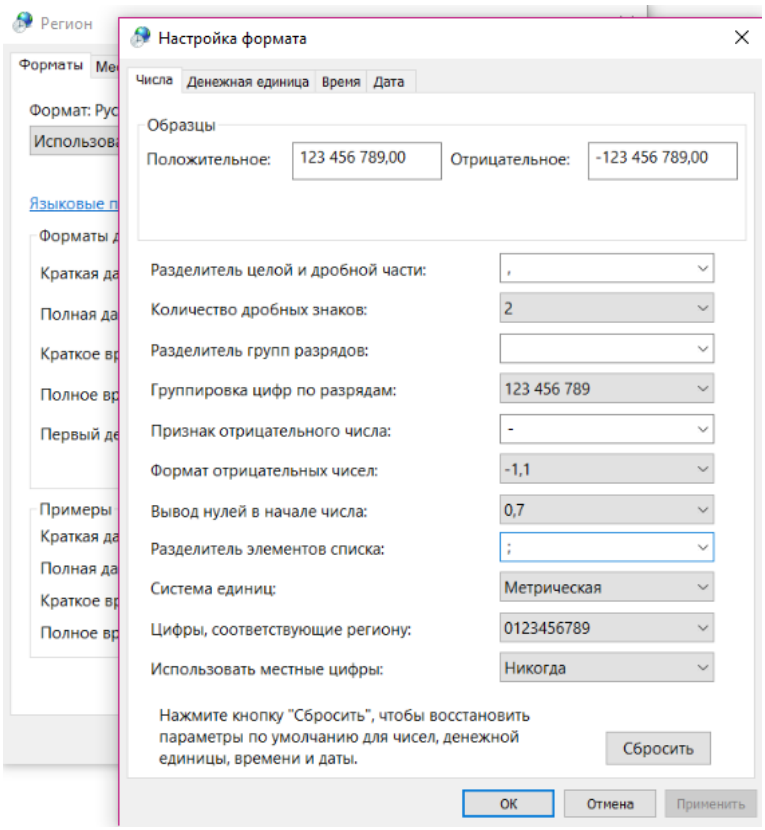


Рисунок 1.2

1.4. Библиотека CSV

Несмотря на то, что DSV-форматы просты, отсутствие четких стандартов в выборе разделителей и экранировании символов привели к тому, что с ними лучше работать при помощи специализированных библиотек, а не в стиле «использования функции» `split`.

Для работы с такими форматами в Python есть модуль csv.

В модуле есть два основных объекта: `reader` и `writer`, созданные, чтобы читать и создавать csv-файлы соответственно.

Приведем пример использования **читателя** с почти полным набором значений, указав:

- Кодировку файла
- Символ-разделитель
- Разделитель текста

Объект `reader` дает доступ к построчному итератору полностью аналогично работе с файлом или списком.

Общности ради в следующем примере мы покажем, что разделителем может быть любой символ.

Мы выводим только 10 строк из таблицы, чтобы не «засорять» экран.

```
import csv
with open('files/ikea.csv', encoding="utf8") as csvfile:
    reader = csv.reader(csvfile, delimiter=';', quotechar='"')
    for index, row in enumerate(reader):
        if index > 10:
            break
        print(row)
```

```
['keywords', 'price', 'product_name']
['МОРУМ, Ковёр, безворсовый', '6999', 'МОРУМ']
['МОРУМ, Ковёр, безворсовый', '6999', 'МОРУМ']
['ИДБИ, Придверный коврик', '649', 'ИДБИ']
['ХОДДЕ, Ковёр, безворсовый', '1399', 'ХОДДЕ']
['ОПЛЕВ, Придверный коврик', '599', 'ОПЛЕВ']
['ОПЛЕВ, Придверный коврик', '599', 'ОПЛЕВ']
['ЮНКЭН, Брикеты', '89', 'ЮНКЭН']
['БУНСЁ, Детское садовое кресло', '1199', 'БУНСЁ']
['ИКЕА ПС ВОГЭ, Садовое лёгкое кресло', '1999', 'ИКЕА ПС ВОГЭ']
['КУНГСХОЛЬМЕН, Садовый табурет', '5500', 'КУНГСХОЛЬМЕН']
```

Давайте разберем построчно, что происходит в этом коде.

Мы пользуемся `with`, чтобы просто открыть наш файл с кодировкой UTF-8, а потом создаем объект `reader`, говоря ему про символы-разделители полей и строк.

Объект `reader` может служить итератором (и использоваться в цикле `for`) по строкам, каждая из которых представляет собой список. При создании `reader`

мы указываем, что символ-разделитель записей `delimiter` в нашем файле — точка с запятой, а символ кавычек `quotechar` — двойные кавычки. Кроме того, мы используем `enumerate`, чтобы посчитать строки.

Отметим, что исходный файл содержит подписи полей в первой строке, что будет снова нам мешать (например, при сортировке строк). Для корректной работы мы должны были бы исключить первую строку из обработки.

Но в модуле csv есть специальный объект **DictReader**, который поддерживает создание объекта-словаря на основе подписей к полям.

DictReader не просто словарь, а словарь, который отслеживает порядок ключей после их добавления, — **OrderDict**, что будет дальше видно в примере. Дополнительно почитать про **OrderDict** можно тут.

Теперь мы можем обращаться к полям не по индексу, а по **названию**, что делает программу еще более понятной.

Найдем топ-10 самых дорогих товаров (как вы думаете, какая запись более понятна: `int(x['price'])` или `int(x[1])`?):

```
with open('files/ikea.csv', encoding="utf8") as csvfile:

    reader = csv.DictReader(csvfile, delimiter=';', quotechar='"')

    expensive = sorted(reader, key=lambda x: int(x['price']), reverse=True)


for record in expensive[:10]:

    print(record)
```

```
OrderedDict([('keywords',     'ГРИЛЬЕРА,     Плита'),     ('price',     '99999'),
('product_name', 'ГРИЛЬЕРА')])

OrderedDict([('keywords',     'ГРИЛЬЕРА,     Плита'),     ('price',     '99999'),
('product_name', 'ГРИЛЬЕРА')])

OrderedDict([('keywords', 'КИВИК, Диван-кровать 3-местный'), ('price', '79999'),
('product_name', 'КИВИК')])

OrderedDict([('keywords', 'КИВИК, Диван-кровать 3-местный'), ('price', '79999'),
('product_name', 'КИВИК')])

OrderedDict([('keywords',  'СТОКГОЛЬМ,  Диван  3-местный'),  ('price',  '69999'),
('product_name', 'СТОКГОЛЬМ')])
```

```
OrderedDict([('keywords', 'ИСАНДЕ, Встраив холодильник/морозильник A++'), ('price', '59999'), ('product_name', 'ИСАНДЕ')])

OrderedDict([('keywords', 'КУЛИНАРИСК, Комбинир СВЧ с горячим обдувом'), ('price', '54999'), ('product_name', 'КУЛИНАРИСК')])

OrderedDict([('keywords', 'ХОГКЛАССИГ, Индукц варочн панель'), ('price', '49999'), ('product_name', 'ХОГКЛАССИГ')])

OrderedDict([('keywords', 'ГРЭНСЛЁС, Комбинир СВЧ с горячим обдувом'), ('price', '49999'), ('product_name', 'ГРЭНСЛЁС')])

OrderedDict([('keywords', 'КУЛИНАРИСК, Духовка/пиролитическая самоочистка'), ('price', '49999'), ('product_name', 'КУЛИНАРИСК')])
```

Мы привели цены к типу `int`, потому что строки сравниваются в лексикографическом порядке (по алфавиту). Например:

```
print('11' > '100')
```

```
True
```

```
print(11 > 100)
```

```
False
```

Использование объекта для записи (`writer`) аналогично «читателю» (`reader`):

```
with open('files/квадраты.csv', 'w', newline='') as csvfile:
    writer = csv.writer(
        csvfile, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)
    for i in range(10):
        writer.writerow([i, i ** 2, f"Квадрат числа {i} равен {i ** 2}"])
```

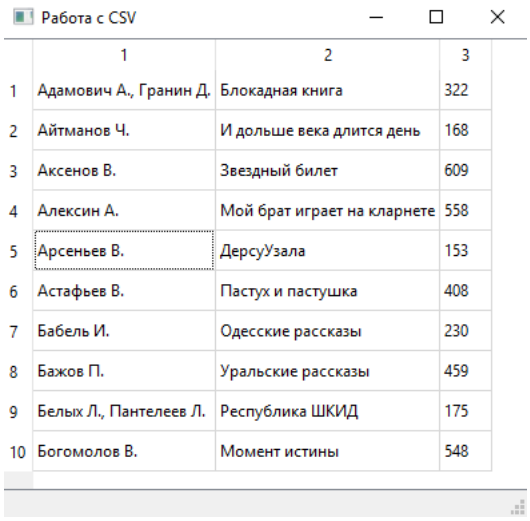
1.5. PyQT. Чтение из .csv файла

Поскольку большинство программ используются людьми без опыта работы в командной строке, в пакетах, отвечающих за графический интерфейс,

существуют специальные модули для работы с табличными данными. Для работы с таблицами в PyQT существует класс **Table Widget**.

Создадим простое приложение, которое будет отображать содержимое таблицы, которая хранится в формате .csv.

Для этого создадим в QtDesigner новую форму, добавим на нее **Table Widget**, а затем конвертируем полученный GUI в файл с расширением .py (ui_1.py).



	1	2	3
1	Адамович А., Гранин Д.	Блокадная книга	322
2	Айтманов Ч.	И дольше века длится день	168
3	Аксенов В.	Звездный билет	609
4	Алексин А.	Мой брат играет на кларнете	558
5	Арсеньев В.	ДерсуУзала	153
6	Астафьев В.	Пастух и пастушка	408
7	Бабель И.	Одесские рассказы	230
8	Бажов П.	Уральские рассказы	459
9	Белых Л., Пантелеев Л.	Республика ШКИД	175
10	Богомолов В.	Момент истины	548

Рисунок 1.3

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow,QTableWidgetItem
from ui_1 import Ui_Form
import csv



class MyWidget(QMainWindow, Ui_Form):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.loadTable('Data.csv')

    def loadTable(self, table_name):
```

```
        with open('Data.csv', encoding="utf8") as csvfile:
            reader = csv.reader(csvfile, delimiter=';', quotechar='"')
            title = next(reader)
            self.tableWidget.setColumnCount(len(title))
            self.tableWidget.setHorizontalHeaderLabels(title)
            self.tableWidget.setRowCount(0)
            for i, row in enumerate(reader):
                self.tableWidget.setRowCount(self.tableWidget.rowCount() + 1)
                for j, elem in enumerate(row):
                    self.tableWidget.setItem(i, j, QTableWidgetItem(elem))
        self.tableWidget.resizeColumnsToContents()


app = QApplication(sys.argv)
ex = MyWidget()
ex.show()
sys.exit(app.exec_())
```

В функции `loadTable` происходит загрузка содержимого в виджет таблицы. Разберемся, что в ней происходит.

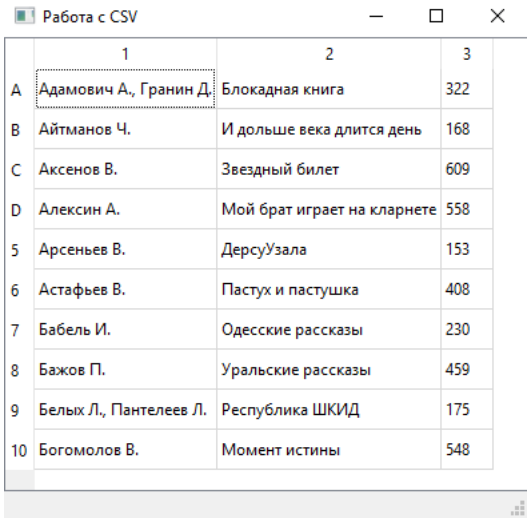
Для отображения таблицы необходимо указать ее размеры: количество строк и столбцов. Если бы у нас была фиксированная матрица, мы бы легко узнали ее размеры, но в случае работы с .csv мы используем итератор, поэтому необходимо применить некоторые лайфхаки.

Сначала, так же, как и в прошлых примерах, мы открываем файл для чтения. Поскольку `reader` является итератором, то, для того чтобы получить первую строку, необходимо воспользоваться функцией `next()`.

Чтобы отобразить заголовки в виджете, воспользуемся функцией `setHorizontalHeaderLabels`. Кстати, существует также функция

`setVerticalHeaderLabels` — для задания вертикальных заголовков. По умолчанию, если программист не задает текст заголовков, это просто номера. Если в функцию передано недостаточное количество заголовков, после того как они «закончатся», следующие заголовки будут цифровыми. Например:

```
self.tableWidget.setVerticalHeaderLabels(['A', 'B', 'C', 'D'])
```



	1	2	3
A	Адамович А., Гранин Д.	Блокадная книга	322
B	Айтманов Ч.	И дольше века длится день	168
C	Аксенов В.	Звездный билет	609
D	Алексин А.	Мой брат играет на кларнете	558
5	Арсеньев В.	ДерсуУзала	153
6	Астафьев В.	Пастух и пастушка	408
7	Бабель И.	Одесские рассказы	230
8	Бажов П.	Уральские рассказы	459
9	Белых Л., Пантелеев Л.	Республика ШКИД	175
10	Богомолов В.	Момент истины	548

Рисунок 1.4

Зная количество заголовков, можно установить количество столбцов в нашей таблице, используя метод `setColumnCount`. Но мы ведь не знаем, сколько у нас элементов в нашем итераторе, поэтому установим изначальное количество, равное 0, а затем будем на каждом шаге увеличивать его на единицу.

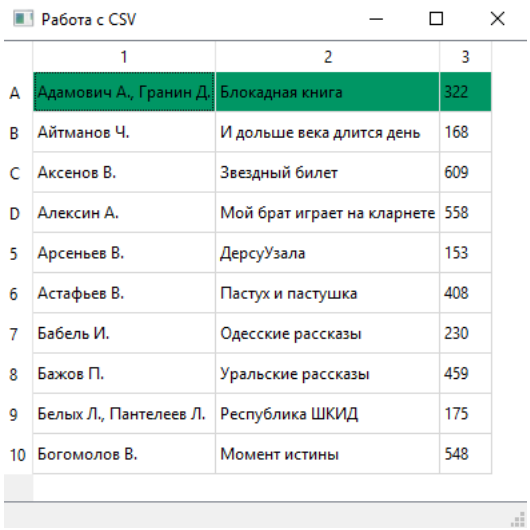
Основной функцией при работе с TableWidget является `tableWidget.setItem(i, j, QTableWidgetItem(elem))`. Эта функция помещает в заданную с помощью координат ячейку соответствующее значение. Важно не забыть, что в таблице хранятся не текстовые, не числовые и не какие-либо другие данные, а данные типа `QTableWidgetItem`, поэтому не забывайте приводить значения к этому типу.

Очень часто непонятно заранее, какого размера будет содержимое той или иной ячейки в таблице. Для того чтобы размеры ячеек соответствовали длине текста, используем функцию `resizeColumnsToContents()`. Она растянет или сожмет при необходимости размеры ячейки, чтобы пользователю был виден весь текст.

Согласитесь, что сейчас таблица выглядит блекло и неинтересно. Давайте раскрасим ее. Для этого нам понадобится импортировать модуль, отвечающий за работу с цветом: `from PyQt5.QtGui import QColor`. К сожалению, по умолчанию в PyQT нет функции, чтобы изменить цвет целого ряда, а есть только для одной ячейки. Так что придется написать такую функцию самостоятельно.

```
def colorRow(self, row, color):
    for i in range(self.tableWidget.columnCount()):
        self.tableWidget.item(row, i).setBackground(color)
```

И теперь, если мы захотим выделить нулевой ряд цветом, нам достаточно лишь вызвать эту функцию: `self.colorRow(0, QColor(0, 150, 100))`. Получим такой результат:



	1	2	3
A	Адамович А., Гранин Д.	Блокадная книга	322
B	Айтманов Ч.	И дольше века длится день	168
C	Аксенов В.	Звездный билет	609
D	Алексин А.	Мой брат играет на кларнете	558
5	Арсеньев В.	ДерсуУзала	153
6	Астафьев В.	Пастух и пастушка	408
7	Бабель И.	Одесские рассказы	230
8	Бажов П.	Уральские рассказы	459
9	Белых Л., Пантелеев Л.	Республика ШКИД	175
10	Богомолов В.	Момент истины	548

Рисунок 1.5

1.6. PyQT. Запись из таблицы в .csv файл

Мы уже умеем читать данные из файла и отображать их с помощью таблицы. Теперь попробуем обратный процесс. Для этого напишем программу, которая может использоваться при выставлении баллов команде за различные этапы. Известно, что и команд, и испытаний — фиксированное число. Так что можно создать сетку для нашей таблицы, используя QtDesigner. Для этого

необходимо навести курсор на виджет, нажать правую кнопку мыши, выбрать пункт меню **Edit Items...**, а затем указать данные.

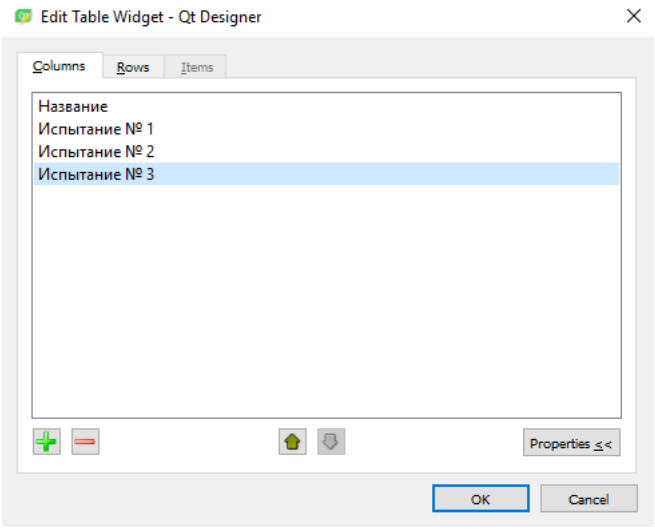


Рисунок 1.6

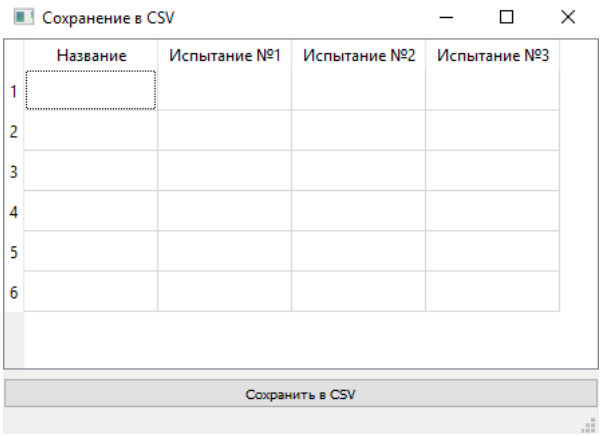


Рисунок 1.7

Для сохранения данных была написана функция `saveToCsv`:

```
def saveToCsv(self):
    with open('results.csv', 'w', newline='') as csvfile:
        writer = csv.writer(
            csvfile, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)
        # Получение списка заголовков
        writer.writerow([self.tableWidget.horizontalHeaderItem(i).text()
                         for i in range(self.tableWidget.columnCount())])
        for i in range(self.tableWidget.rowCount()):
```

```
row = []
for j in range(self.tableWidget.columnCount()):
    item = self.tableWidget.item(i, j)
    if item is not None:
        row.append(item.text())
writer.writerow(row)
```

К сожалению, в PyQT нет встроенной функции, которая возвращает список заголовков, так что пришлось использовать генератор и встроенную функцию `horizontalHeaderItem(i)`, которая возвращает `i`-й заголовок. А затем каждая строка переносится в список, после чего этот список записывается в файл.

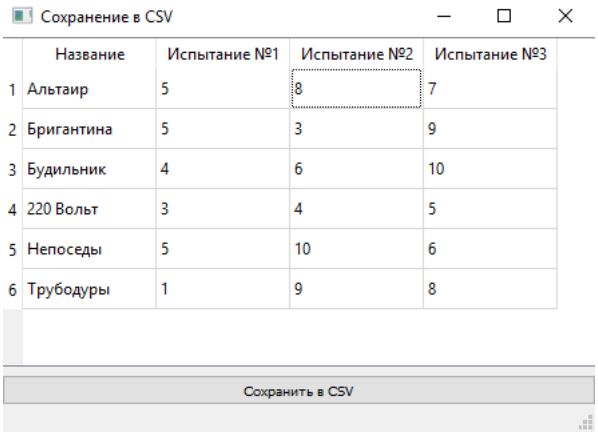


Рисунок 1.8

Результатом будет такой .csv-файлик:

Название	Испытание №1	Испытание №2	Испытание №3
Альтаир	5	8	7
Бригантина	5	3	9
Будильник	4	6	10
220 Вольт	3	4	5
Непоседы	5	10	6
Трубодуры	1	9	8

Таблица 11

2. Введение в БД, работа с SQL-таблицами и отображение данных в PyQT.

В этой главе мы познакомимся с базами данных и языком SQL [5].

2.1. Введение в базы данных

На прошлых лекциях вы уже сталкивались с хранением данных во внешних источниках: простых текстовых документах, документах с особым форматированием, например, csv-таблицах. Но зачастую такие хранилища данных неудобны, потому что дублируются несколько раз, занимают место на жестком диске, а поиск в них занимает много времени. Для решения этих проблем был придуман такой способ хранения информации, как база данных (БД).

Существуют различные системы управления базами данных, так называемые СУБД. База данных — непосредственное хранилище информации, а СУБД предоставляют разработчикам интерфейс для общения с базой. В наших проектах мы будем использовать СУБД **SQLite.** [6] Основным достоинством этой СУБД является возможность упаковки всей базы в один файл и встроенная в Python по умолчанию библиотека для работы с ней.

Данные в БД хранятся с помощью таблиц и связей между этими таблицами. Мы будем работать с БД, в которой хранится информация о фильмах. Их название, год выпуска, жанр и продолжительность в минутах. Каждое поле занимает определенный объем, в зависимости от типа данных: целочисленный, строковый или используемый для хранения времени.

Если бы мы хранили информацию, относящуюся к каждому фильму, нам для каждой записи приходилось бы занимать место для названия жанра. Но, поскольку число жанров ограничено и они наверняка будут повторяться для разных фильмов, мы можем создать дополнительную таблицу, в которой будем хранить пару формата **Ключ: Значение**. В качестве ключа используется

уникальный идентификатор (id), а в качестве значения — название жанра. А в первой таблице просто хранить ссылку на вторую.

Очень часто для наглядного представления таблиц и связей используют такую визуализацию:

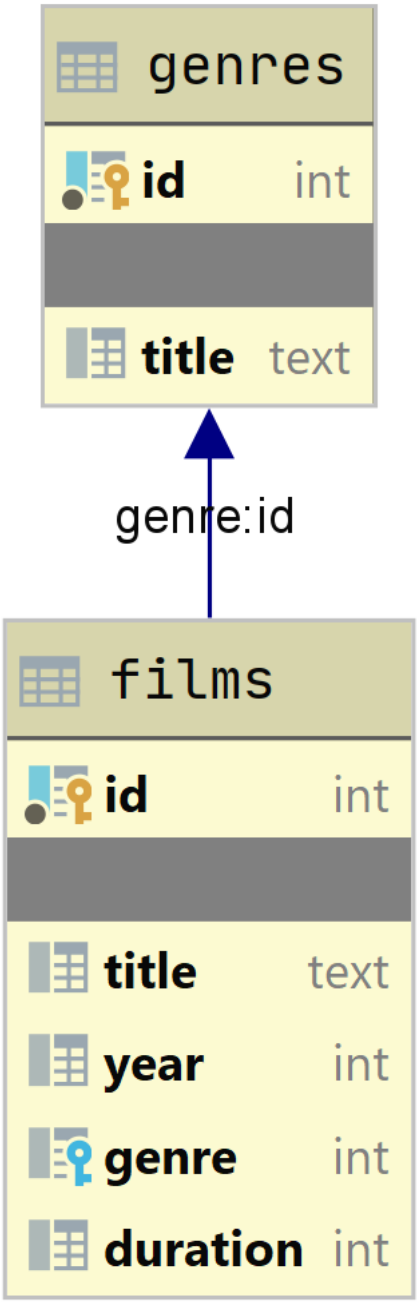


Рисунок 2.1

Именно так выглядит база данных, с которой мы будем работать. Скачать ее можно здесь.

2.2. Основы SQL

Для работы с базами данных был придуман специальный язык — SQL (structured query language — «язык структурированных запросов»). Основной командой для получения какой-либо информации из БД является команда SELECT. Ее синтаксис выглядит так:

```
SELECT ЧТО FROM ИМЯ_ТАБЛИЦЫ
       WHERE УСЛОВИЕ
```

Кроме этого, есть и различные модификаторы этой команды. Например, `ORDER BY ПОЛЕ` — тогда результаты будут выведены в отсортированном виде по заданному полю. В условии может быть и вложенный запрос.

Перед тем как начать работать с БД с помощью Python, поработаем с ней с помощью официальной программы SQLiteStudio. Первое, что необходимо сделать после установки, — добавить нашу базу фильмов.

Для этого в основном меню необходимо выбрать пункт Add a Database и в открывшемся окне указать путь к файлу нашей БД.

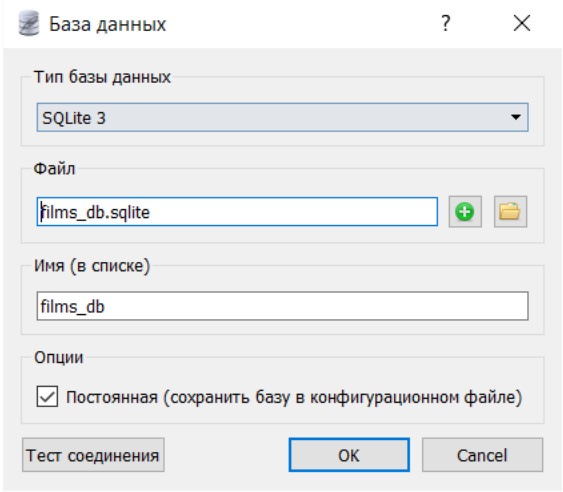


Рисунок 2.2

Если все прошло успешно, можно писать запросы. Для этого необходимо открыть редактор SQL. Его логотип выглядит как свиток бумаги с карандашом.

Интерфейс редактора состоит из двух частей: первая, где пишется сам запрос, и вторая, где отображаются результаты.

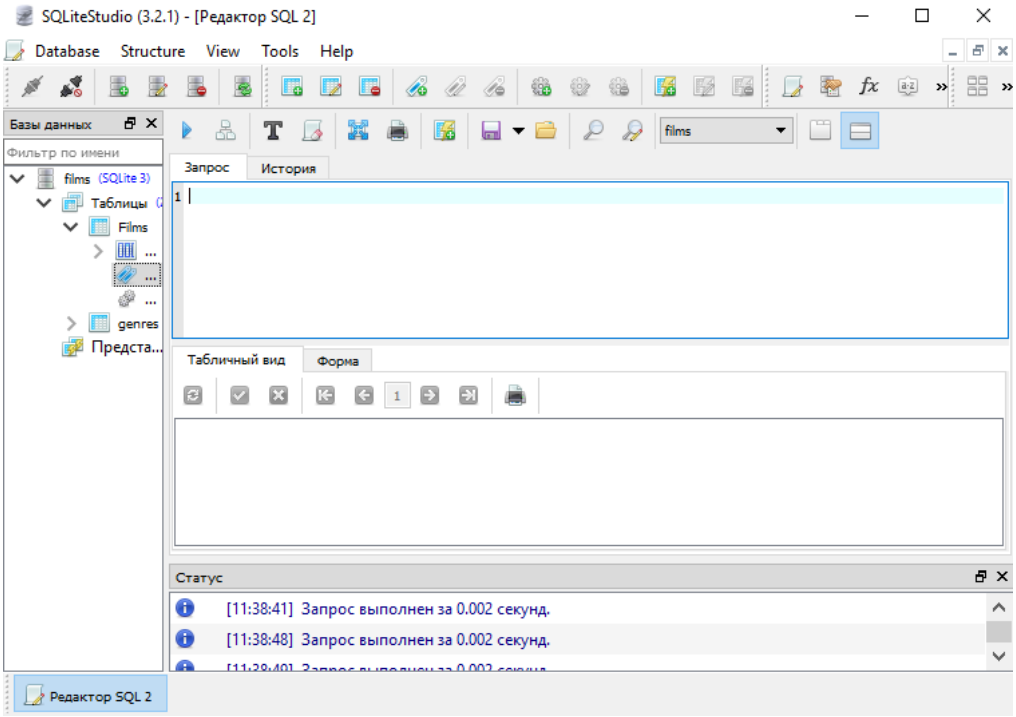


Рисунок 2.3

Напишем наш первый запрос. Получим все фильмы, выпущенные в 2010 году.

```
SELECT * FROM Films

    WHERE year = 2010
```

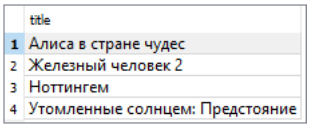
	id	title	year	genre	duration
1	248	Алиса в стране чудес	2010	13	
2	4382	Железный человек 2	2010	11	
3	9138	Ноттингем	2010	11	
4	15495	Утомленные солнцем: Предстояние	2010	2	

Рисунок 2.4

Символ * обозначает, что нам необходимо получить все поля. Однако очень часто нам нужно получить только одно или два поля. Модифицируем запрос так, чтобы выводилось только название.

```
SELECT title FROM Films

    WHERE year = 2010
```



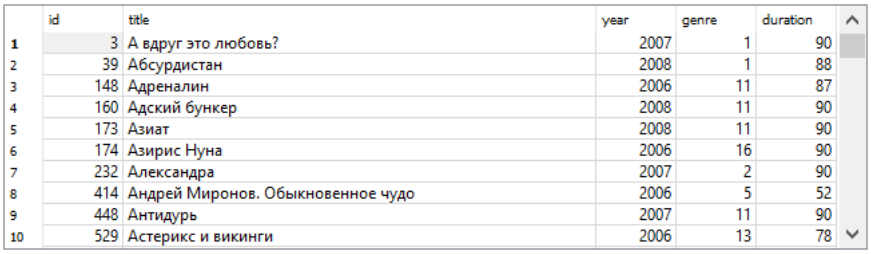
	title
1	Алиса в стране чудес
2	Железный человек 2
3	Ноттингем
4	Утомленные солнцем: Предстояние

Рисунок 2.5

Условий может быть и несколько: выберем фильмы, выпущенные после 2005 года с продолжительностью от 40 минут до 1,5 часов:

```
SELECT * FROM Films

    WHERE (year > 2005) AND (duration >= 45) AND (duration <= 90)
```



	id	title	year	genre	duration
1	3	А вдруг это любовь?	2007	1	90
2	39	Абсурдистан	2008	1	88
3	148	Адреналин	2006	11	87
4	160	Адский бункер	2008	11	90
5	173	Азиат	2008	11	90
6	174	Азирис Нуна	2006	16	90
7	232	Александра	2007	2	90
8	414	Андрей Миронов. Обыкновенное чудо	2006	5	52
9	448	Антидурь	2007	11	90
10	529	Астерикс и викинги	2006	13	78

Рисунок 2.6

А как вывести все фильмы определенного жанра, например, фантастику?

```
SELECT title FROM Films

    WHERE genre=(

SELECT id FROM genres

    WHERE title = 'фантастика')
```

Сначала выполнится внутренний запрос: из таблицы `genres` будет получен id категории «Фантастика», а затем будет выполнено сравнение и выведен результат.

Помимо того, может быть выполнено сравнение не с одним элементом, а проверка на попадание в список. Это делается с помощью специального слова `IN`. Например:

```
SELECT title, duration FROM Films

WHERE duration IN (45,90)
```

Кроме строго сравнения (больше, меньше, равно), существуют специальные операторы, облегчающие жизнь. В этой главе мы рассмотрим два.

Первый — BETWEEN — проверяет, попадает ли заданное значение в диапазон.

```
SELECT * FROM Films

    WHERE (year > 2005) AND duration BETWEEN 45 AND 60
```

И второй — оператор LIKE, с помощью которого можно проверить, насколько похожа та или иная строка на заданный шаблон. Для шаблона используются специальные символы:

% — обозначает любое количество, в том числе нулевое, любых символов

_ — обозначает один любой символ

Рассмотрим пример с нашей базой. Получим список фильмов, у которых первая буква в названии — **А** и третья — **к.**

```
SELECT * FROM Films

    WHERE title like 'А_к%'
```

	id	title	year	genre	duration
1	9	А к нам цирк приехал	1978	2	23
2	10	А как же Боб?	1991	1	97
3	11	А кто-то все видит	1994	4	88
4	196	Аккаттоне!	1961	2	120
5	197	Аккомпаниаторша	1993	2	111
6	198	Аккумулятор	1994	8	102
7	423	Анкор, еще анкор!	1992	1	101
8	491	Арктическая тоска	1993	11	91
9	556	Аукцион	1983	13	89

Рисунок 2.7

Из любого условия достаточно просто сделать обратное. Для этого перед ним необходимо добавить специальное слово — `NOT`.

Кроме этого, есть возможность избавиться от повторов строк, используя в запросе специальное слово — `DISTINCT`. Например, вот так можно получить список по годам, в которые выходили фильмы в нашей базе данных, без повторений.

```
SELECT DISTINCT year FROM Films
```

2.3. SQL + Python

Для работы с SQLite из Python существует специальная библиотека — `sqlite3`. Очень часто подход при работе с БД из различных языков программирования стандартизирован. Существуют два основных понятия: 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 и производится работа с БД

В принципе с SQLite базой данных можно работать и напрямую через подключение, но так делать плохо, потому что это не отвечает принципам работы с базами данных из Python, и, если вы будете портировать свое приложение под другую СУБД, вы можете столкнуться с тем, что весь код по работе с базой данных придется переписывать через курсоры.

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")


# 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 кортеж 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: год, продолжительность и т. д. Для этого существует удобный синтаксис. Вместо значения в запросе указывается вопросительный знак, а затем в итерируемом объекте (чаще всего в кортеже) указываются параметры.

```
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()
```

Но не будем забывать и о графическом интерфейсе. Напишем программу, которая будет отображать результаты введенного запроса в таблице.

С помощью QtDesigner создадим интерфейс: поле для ввода запроса, таблица для отображения результатов и кнопка для запуска выполнения запроса. Работа с **QtableWidget** разбиралась, когда мы работали с csv-файлами. В данном случае принцип абсолютно такой же.

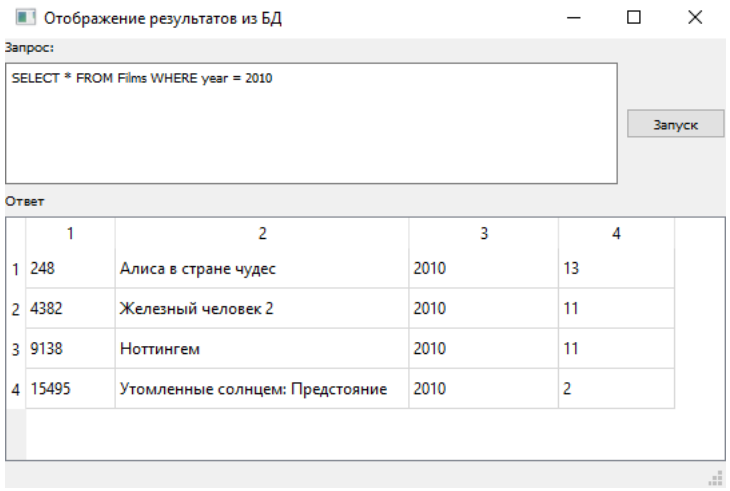


Рисунок 2.8

Больше узнать про синтаксис команды SELECT можно здесь.

Перед тем, как получать информацию из базы, необходимо ее туда поместить. Для этого существует команда INSERT. Ее 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.

21	23	исторический
22	24	нуар
23	42	Фантасмагория
24	43	Лекции

Рисунок 2.9

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 команду INSERT несколько раз, а достаточно через запятую перечислить все значения, которые необходимо добавить. Например, так:

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

Часто случается, что информация в базе нуждается в изменении. Для этого используется команда UPDATE.

```
UPDATE имя_таблицы

SET название_колонки = новое_значение

WHERE условие
```

Предположим, что мы посмотрели фильм «Аватар», и теперь хотим указать его продолжительность. 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'
```

	id	title	year	genre	duration
1	54	Аватар	2009	11	162

Рисунок 2.10

Но бывает и так, что информация уже совершенно точно не нужна. Тогда ее необходимо удалить из БД, чтобы она не занимала лишнее пространство в памяти.

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

```
DELETE from films

where year < 1985
```

2.4. Модификация из Python

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.

```
import sqlite3
import sys

from PyQt5.QtWidgets import QApplication, QMainWindow, QTableWidgetItem
from ui_3 import Ui_Form



class MyWidget(QMainWindow, Ui_Form):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.con = sqlite3.connect("films.db")
        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):
        cur = self.con.cursor()
        # Получили результат запроса, который ввели в текстовое поле
        result = cur.execute("Select * from films WHERE id=?",
                             (self.spinBox.text(),)).fetchall()
        # Заполнили размеры таблицы
        self.tableWidget.setRowCount(len(result))
        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 = {}

    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()

    def save_results(self):
        if self.modified:
            cur = self.con.cursor()
            que = "UPDATE films SET\n"
            for key in self.modified.keys():
                que += "{}='{}'\n".format(key, self.modified.get(key))
            que += "WHERE id = ?"
            cur.execute(que, (self.spinBox.text(),))
            self.con.commit()


app = QApplication(sys.argv)
ex = MyWidget()
ex.show()
sys.exit(app.exec_())
```

Для того чтобы обработать изменение содержимого ячейки, воспользуемся сигналом `self.tableWidget.itemChanged`. Будем записывать в словарь пару, состоящую из наименования поля и нового значения. Это происходит в функции `item_changed`. Затем нам необходимо сохранить полученные результаты в БД. Так как заранее мы не знаем, какие поля были модифицированы, мы не можем использовать конструкцию с вопросительным знаком. Так что просто «склеим» необходимый нам запрос, используя обыкновенную конкатенацию строк. Это описывается в функции `save_results`.

Изначальное отображение:

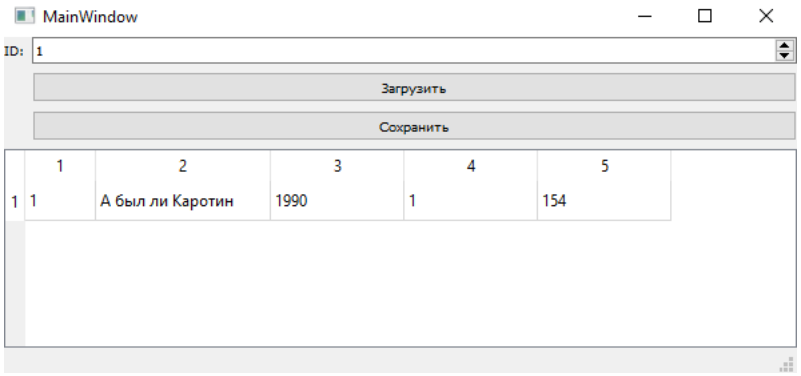


Рисунок 2.11

После изменения и сохранения:

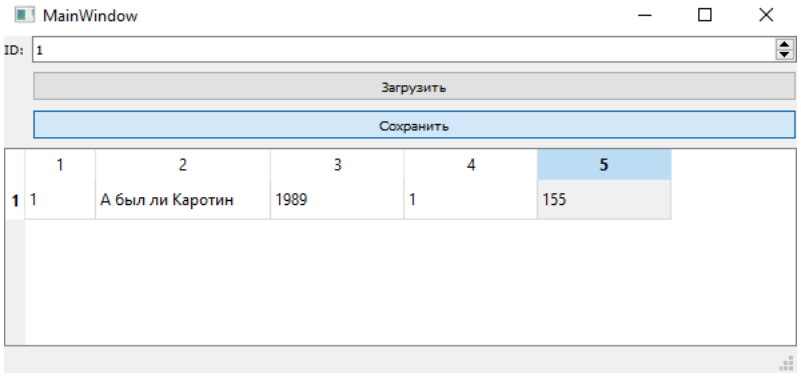


Рисунок 2.12

Очень часто для редактирования или создания нового элемента не используют основное окно программы, а открывают дополнительное. Такой подход позволяет оптимизировать ресурсы и не «пробегаться» по множеству элементов. 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 «Фильмы». И теперь в поле для ввода будем вводить только условие. А сам запрос будет формироваться так:

```
que = "SELECT * FROM Films WHERE " + self.textEdit.toPlainText()
```

И теперь интерфейс нашей программы, выглядит вот так:

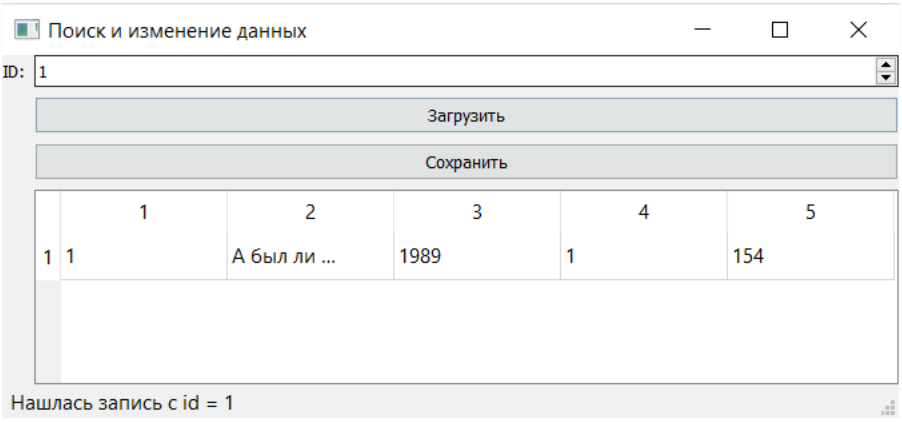


Рисунок 2.13

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

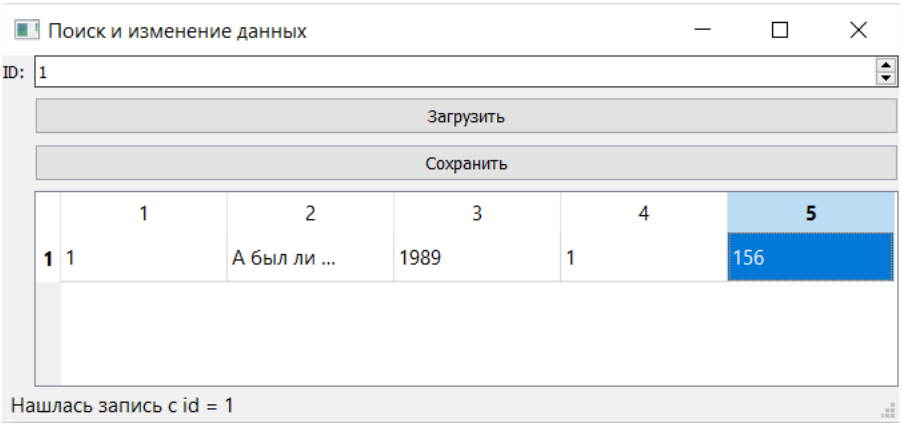


Рисунок 2.14

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

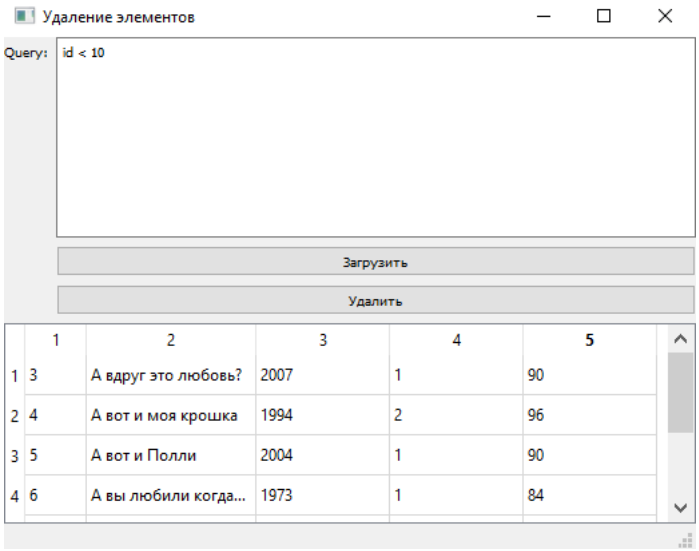


Рисунок 2.15

Основная функция, отличающая данную программу от предыдущих, — функция удаления выделенных элементов. Рассмотрим ее поподробнее.

```
def delete_elem(self):
    rows = list(set([i.row() for i in self.tableWidget.selectedItems()]))
    ids = [self.tableWidget.item(i, 0).text() for i in rows]
    valid = QMessageBox.question(self, '',
                                 "Действительно удалить элементы с id " +
                                 ",".join(ids), QMessageBox.Yes, QMessageBox.No)
    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, 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.

В следующей строке мы создаем объект «Вопрос» класса **QMessageBox** (важно не забыть импортировать его из PyQt5.QtWidgets). В качестве параметров ему передаются:

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

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 впервые столкнулись с оператором `self.con.commit()`. Этот оператор очень важен при изменении БД: он сохраняет в текущем файле все изменения. Если забыть его вызвать, при следующем запуске программы все, что было исправлено, не сохранится.

3. Обработка событий. Сборка независимого приложения

В этой главе мы научимся взаимодействовать с пользователем на новом уровне: добавим в программу работу с клавиатурой и мышкой. И рассмотрим, как построить собрать нашу программу в exe-файл.

3.1. Обработка нажатий клавиатуры

У каждой клавиши на клавиатуре есть свой уникальный код. Однако работать с числами не всегда удобно, поэтому в PyQt в модуле Qt с кодами сопоставлены и понятные имена. Согласитесь, что гораздо легче понимать смысл вот такой строки: `print(Qt.Key_F)`, нежели такой: `print(70)`.
Подключить модуль `Qt` можно, например, таким способом:

```
from PyQt5.QtCore import Qt
```

Все имена кнопок и их коды можно найти здесь.
Для обработки нажатий на клавиши в вашем классе необходимо переопределить (создать) метод `keyPressEvent(self, event)`:

```
def keyPressEvent(self, event):
    if event.key() == Qt.Key_F:
        # code
```

Если во время работы приложения будет нажата клавиша **F**, условие выполнится.
Чтобы обработать комбинацию клавиш (Hotkey), код должен быть таким:

```
def keyPressEvent(self, event):
    if int(event.modifiers()) == (Qt.AltModifier + Qt.ShiftModifier):
        if event.key() == Qt.Key_Q:
            # code
```

Теперь код выполнится, если нажмут клавишу **Q** при зажатых кнопках **Shift** и **Alt**.

Важно, что кнопки обработаются только в том случае, если окно активно. Активным называется окно, с которым пользователь работает в данный момент.

3.2. Работа с мышью

Для работы с мышью точно так же, как и в случае с клавиатурой, необходимо переопределять встроенные методы. Рассмотрим пример, в котором мы будем выводить координаты курсора на экран.

```
import sys
from PyQt5.QtWidgets import QWidget, QApplication, QLabel



class Example(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.setGeometry(300, 300, 300, 300)
        self.setWindowTitle('Координаты')

        self.coords = QLabel(self)
        self.coords.setText("Координаты:None, None")
        self.coords.move(30, 30)
        self.show()

    def mouseMoveEvent(self, event):
```

```
        self.coords.setText("Координаты:{}, {}".format(
            event.x(), event.y()))


if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())
```

Метод `mouseMoveEvent` срабатывает при изменении положения мышки. Функции `.x()` и `.y()` возвращают координаты текущего положения курсора в экранных координатах. Но, если вы запустите эту программу и попробуете просто поводить мышкой, ничего не произойдет. По умолчанию это событие активируется только при зажатой левой клавише мыши. Зажмите ее — и все заработает. А чтобы отлавливать положение мыши, даже когда клавиша не зажата, в метод `init(self)` надо добавить следующую строку:

```
self.setMouseTracking(True)
```

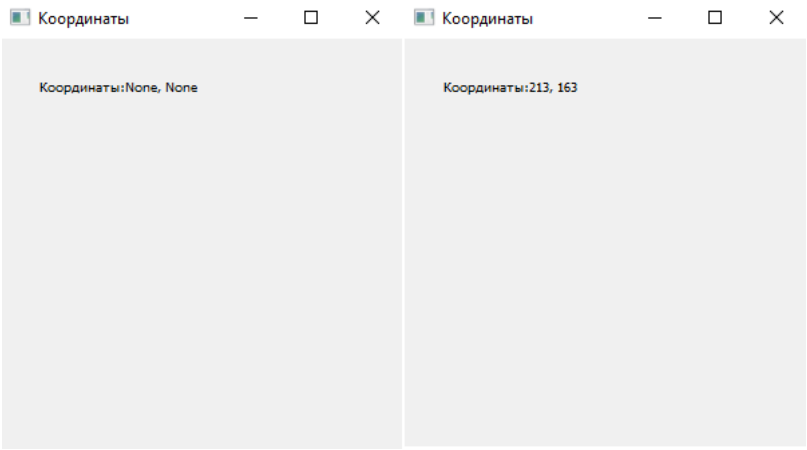


Рисунок 3.1

Кроме метода `mouseMoveEvent`, есть и другие. Например, метод `mousePressEvent`, который отвечает за обработку нажатия мыши. Кроме координат, полезно узнать, какая из кнопок была нажата, левая или правая. Для

этого существует метод `.button()`. Сравнить можно со встроенными значениями из модуля `PyQt5.QtCore.Qt`: `LeftButton`, `RightButton` и даже `MiddleButton`.

```
import sys
from PyQt5.QtWidgets import QWidget, QApplication, QLabel
from PyQt5.QtCore import Qt



class Example(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.setGeometry(300, 300, 300, 300)
        self.setWindowTitle('Координаты')

        self.coords = QLabel(self)
        self.coords.setText("Координаты:None, None")
        self.coords.move(30, 30)

        self.btn = QLabel(self)
        self.btn.setText("Никакая")
        self.btn.move(30, 50)
        self.show()

    def mousePressEvent(self, event):
        self.coords.setText("Координаты:{}, {}".format(
```

```
            event.x(), event.y()))
        if (event.button() == Qt.LeftButton):
            self.btn.setText("Левая")
        elif (event.button() == Qt.RightButton):
            self.btn.setText("Правая")


if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())
```

3.3. Создание standalone-приложений

В отличие от компилируемых языков программирования вроде C++, Python — язык интерпретируемый [7]. И создание standalone-приложения — по своей сути «упаковка» виртуальной машины Python, заточенной под выполнение одной программы.

Мы будем рассматривать работу в операционной системе Windows, отличия в других OS (Linux и MacOS) не являются существенными.

Для создания .exe приложений из программ на Python есть несколько библиотек. Мы используем одну из самых популярных. Она называется `pyinstaller`.

Сначала необходимо ее установить:

```
pip install pyinstaller
```

Или так, если надо установить последнюю, еще не опубликованную версию:

```
pip install git+https://github.com/pyinstaller/pyinstaller
```

Затем открываем командную строку, переходим в папку с нашим проектом и выполняем следующую команду:

```
pyinstaller --onefile --noconsole main.py
```

Что делают модификаторы:

- **onefile** собирает программу в один exe-файл. Это удобно, если программа небольшая, у нее мало аудиофайлов, графических файлов и пр., не относящихся к программному коду
- **noconsole**. Если не писать этот модификатор, то, кроме окна программы, будет открываться python-консоль. Иногда, например, для отладки, это удобно

Есть и другие модификаторы. Например, **icon**. В нем можно указать путь до картинки в формате .ico, и она будет отображаться при просмотре программы в проводнике.

После выполнения этой команды в директории проекта появится две папки — build и dist. В папке dist будет лежать наша программа, или, если мы не указали модификатор **onefile**, кроме .exe, там будет еще множество файлов и папок. В нее следует добавить папки с графическими и другими файлами, которые используются в программе.

3.4. Создание БД

Чтобы создать простую БД, можно воспользоваться SQLiteStudio [8]. Создадим базу данных для хранения любимых книг.

Открываем в верхнем меню вкладку **Database** и выбираем пункт **Add a database**.

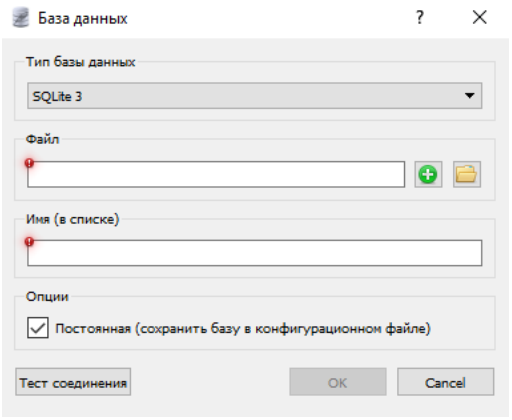


Рисунок 3.2

Нажимаем на **плюс** и вводим желаемое имя для нашей базы.

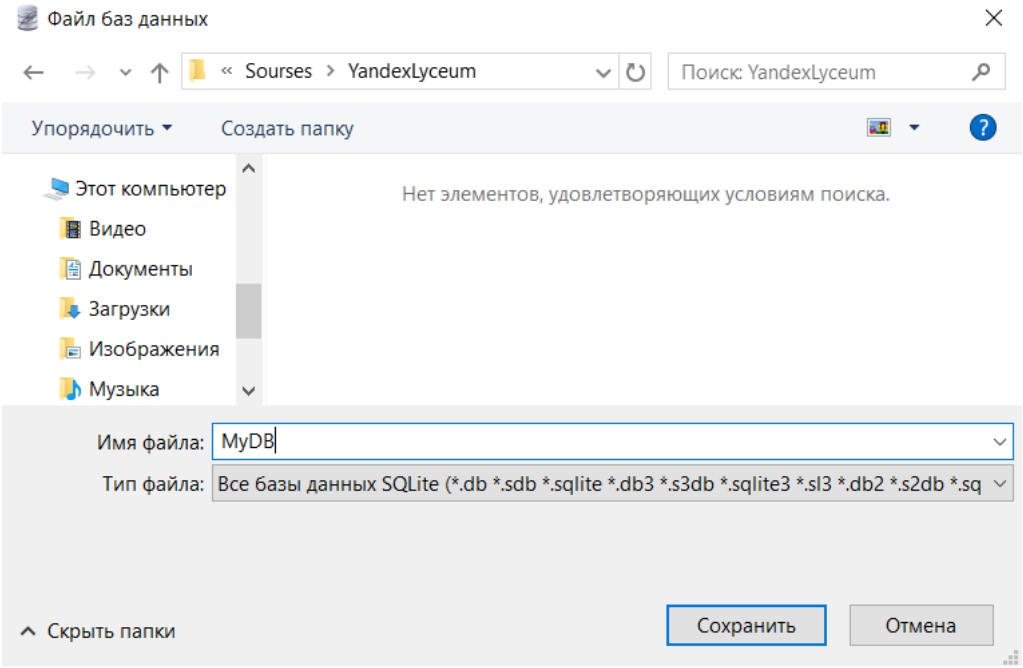


Рисунок 3.3

Сохраняем. В боковом меню должна появиться новая база данных.

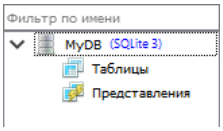


Рисунок 3.4

Теперь в нашей БД необходимо создать таблицы, в которых будет храниться информация. Для этого необходимо навести курсор на базу, нажать правую кнопку и выбрать пункт **Create a table**. Откроется окно для создания таблицы. Введем ее имя.

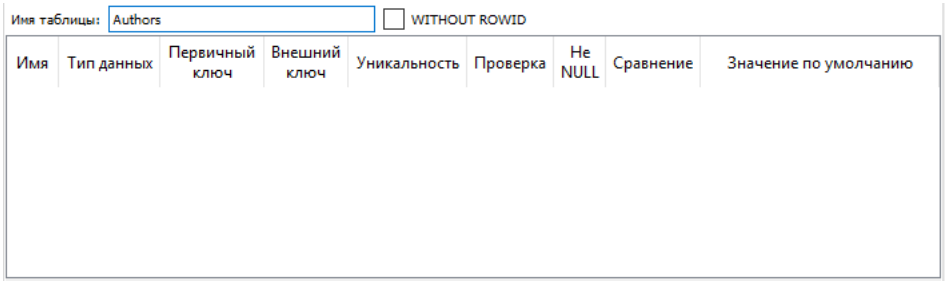


Рисунок 3.5

Двойным нажатием на белое поле откроем редактор столбца. Для того чтобы идентифицировать книги, создадим специальный столбец, назовем его id. Это будет число (то есть тип INTEGER), которое будет являться т. н. **первичным ключом**, то есть оно должно быть обязательно заполнено (не может быть NULL), уникально и для удобства он будет автоматически увеличиваться при добавлении новых значений (это называется автоинкремент).

Настроим все эти значения:

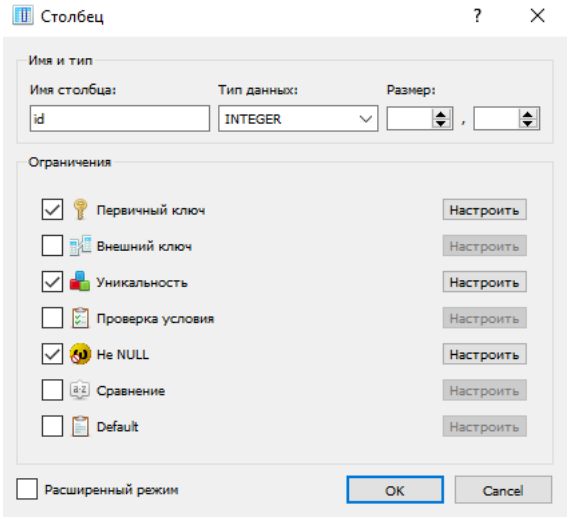


Рисунок 3.6

Чтобы включить функцию автоинкремента, необходимо нажать на кнопку **Настроить** напротив **Первичного ключа**:

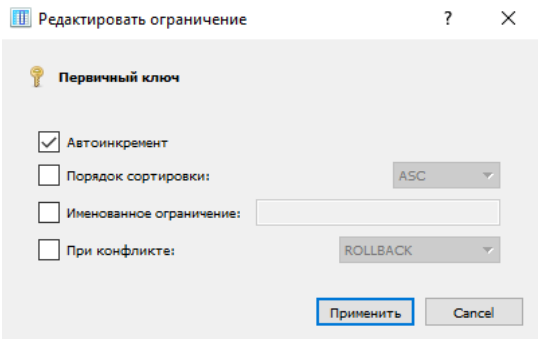


Рисунок 3.7

Теперь в нашей таблице есть первое поле — id. Добавим еще одно поле — Author, в котором будем хранить фамилию автора и его инициалы. Для него подойдет тип данных STRING.

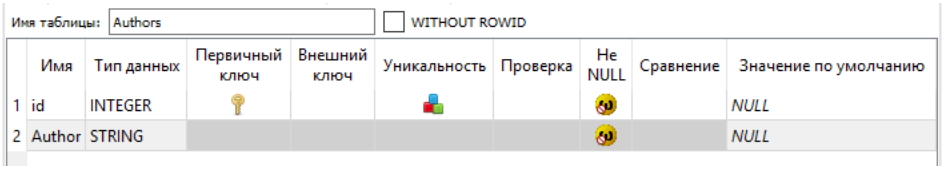


Рисунок 3.8

После добавления всех полей необходимо применить все изменения. Затем выполнить commit. При нажатии на зеленую кнопку с галочкой будет показан запрос для создания таблицы, которую мы создали с помощью графического интерфейса.

Смело нажимаем ОК.

Теперь аналогичным образом создаем таблицу для хранения книг. Но будем указывать не ФИО автора, а его идентификатор из первой таблицы.

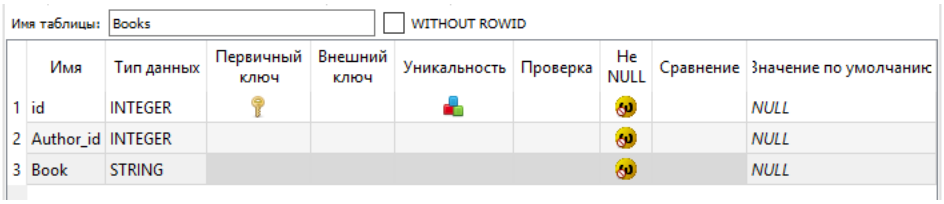


Рисунок 3.9

Теперь заполним наши таблицы. Таблицу авторов:

	id	Author
1	1	Пушкин А.С
2	2	Толстой А.Н.
3	3	Крапивин В.П.

Рисунок 3.10

И таблицу книг:

	id	Author_id	Book
1	1	1	Капитанская дочка
2	2	1	Пиковая дама
3	3	2	Пётр Первый
4	4	2	Гиперболоид инженера Гарина
5	5	2	Аэлита
6	6	3	Мальчик со шпагой

Рисунок 3.11

Зачем мы использовали идентификаторы вместо полных ФИО? На это есть несколько причин:

– Чтобы избежать опечаток и последующих проблем с выборками. Ведь ФИО можно записать очень по-разному.
– Для экономии памяти. Одна ячейка типа INTEGER занимает во много раз меньше, нежели строковая ячейка.

Поздравляем! Вы создали вашу первую базу данных и первые две таблицы в ней. Подробнее про создание БД и таблиц можно прочитать здесь.

3.5. requirements.txt

Очень часто большие проекты содержат массу зависимостей. Под зависимостями подразумеваются библиотеки, необходимые для корректной работы программы. Если программа распространяется не как standalone-приложение, а в исходном коде, хорошим тоном считается создавать специальный файл, в котором указываются зависимости. Принято называть этот файл **requirements.txt**. Как он выглядит? На каждой новой строке написано название пакета и необходимая версия в таком формате:

```
название_пакета==его.версия
```

Например:

```
PyQt5==5.11
```

```
matplotlib==2.2.2
```

И чтобы установить все зависимости, достаточно в консоли (терминале) вызвать команду

```
pip install -r requirements.txt
```

Благодаря ключу `-r` pip автоматически установит внутренние зависимости. То есть, если в файле есть указание пакета X, которому для работы требуется пакет Y, который не указан в этом файле и не является стандартным, он все равно будет установлен.

Чтобы узнать версию уже установленных библиотек, достаточно воспользоваться командой `pip list`. Эта команда отображает список установленных пакетов и их версии.

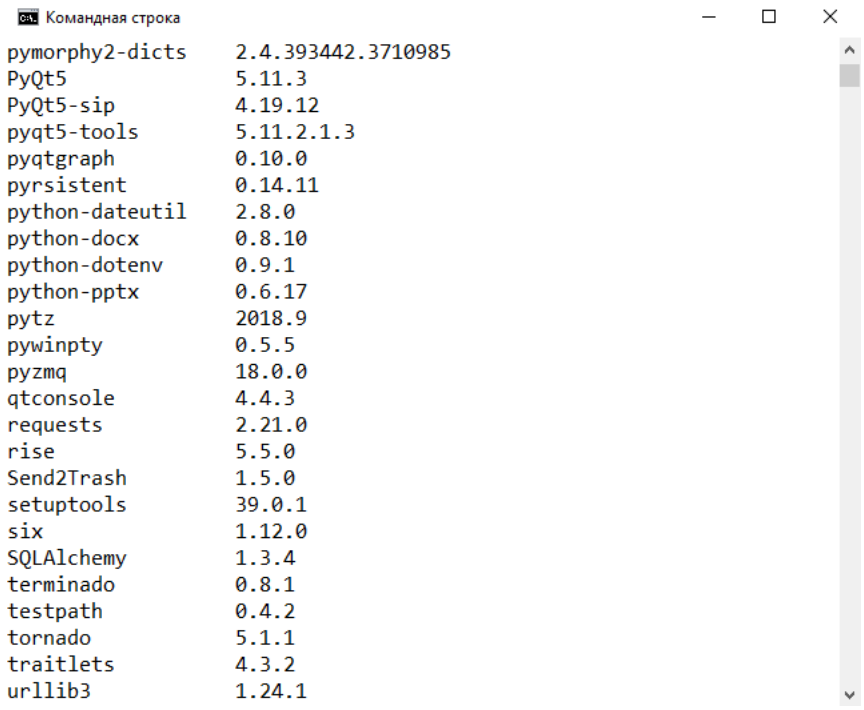


Рисунок 3.12

4. QT. Установка дополнительных компонентов. PyQTGraph

В этой главе мы научимся пользоваться дополнительными компонентами PyQT на примере PyQTGraph — библиотеки для построения графиков. А еще научимся проигрывать музыку в своих приложениях.

4.1. PyQTGraph

Несмотря на большое количество встроенных виджетов, их не всегда хватает для реализации всех задумок. Хорошая новость заключается в том, что PyQT позволяет создавать собственные виджеты со своими функциями. Некоторые разработчики выкладывают эти библиотеки в Интернет, чтобы их могли использовать другие люди. В этой главе мы познакомимся с библиотекой PyQTGraph, которая включает в себя виджеты для работы с графиками.

Для установки библиотеки выполните стандартную команду в командной строке:

```
pip install pyqtgraph
```

Недавно выяснилось, что версия pyqtgraph, которая опубликована в pip, плохо «упаковывается» в исполняемый файл. Если такое происходит, то необходимо установить последнюю версию из github’a. Вот таким образом:

```
pip install git+https://github.com/pyqtgraph/pyqtgraph
```

Конечно, это сработает только в случае, когда на компьютере установлен git.

Затем советуем вам настроить QtDesigner для работы с этой библиотекой. Это — необязательное условие, работать с ней можно и «вручную».

Открываем новый виджет в QtDesigner и добавляем **Graphics View**.

Нажимаем на него правой кнопкой мыши и выбираем вкладку **Преобразовать в...** (Promote to...).

Заполняем поля, как показано на картинке:

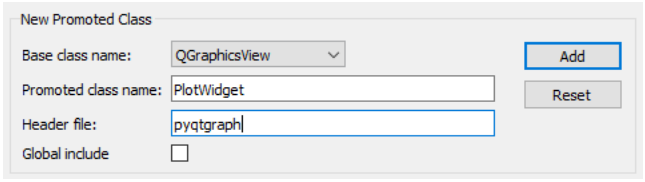


Рисунок 4.1

Нажимаем на кнопку «Добавить», а затем — на кнопку «Преобразовать».

Если нам в следующий раз понадобится добавить такой же виджет, то мы сможем выбрать его из выпадающего меню **Преобразовать в...**.

Напишем программу, которая будет строить графики одной из трех функций в зависимости от выбранного Radio Button.

Сначала создадим интерфейс в QtDesigner, не забыв при этом выполнить преобразование Graphics View в наш Plot Widget.

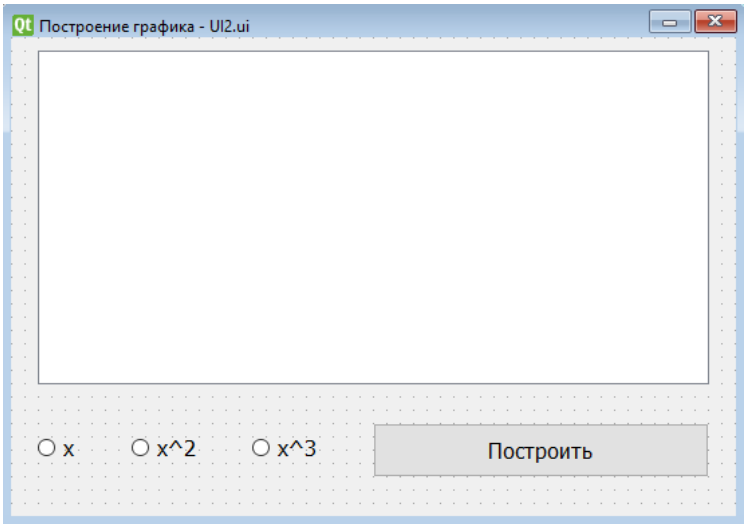


Рисунок 4.2

Преобразуем его с помощью **pyuic5** и начнем выполнять построение.

```
import sys

from PyQt5.QtWidgets import QApplication, QWidget, QMainWindow

from ui import Ui_Form



class MyWidget(QMainWindow,Ui_Form):

    def __init__(self):
```

```
        super().__init__()
        self.setupUi(self)
        self.pushButton.clicked.connect(self.run)


    def run(self):
        if self.radioButton.isChecked():
            self.widget.clear()
            self.widget.plot([i for i in range(10)], [i for i in range(10)],
pen='r')
        elif self.radioButton_2.isChecked():
            self.widget.clear()
            self.widget.plot([i for i in range(10)], [i ** 2 for i in range(10)],
pen='g')
        elif self.radioButton_3.isChecked():
            self.widget.clear()
            self.widget.plot([i for i in range(10)], [i ** 3 for i in range(10)],
pen='b')



app = QApplication(sys.argv)
ex = MyWidget()
ex.show()
sys.exit(app.exec_())
```

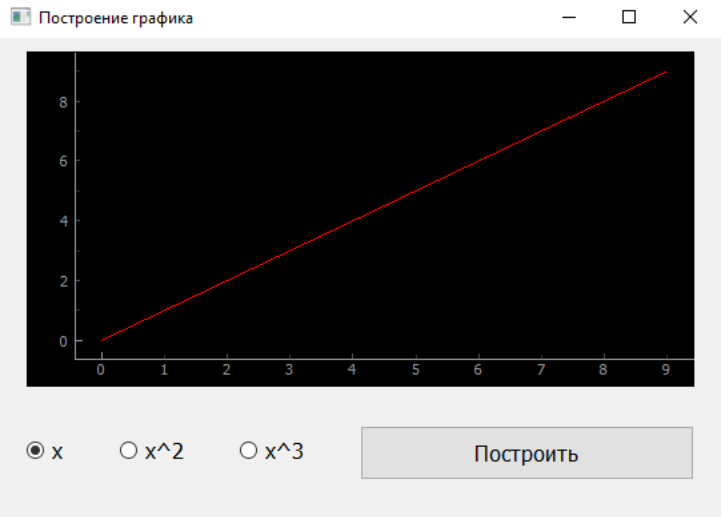


Рисунок 4.3

Подробнее про модуль `pyqtgraph` можно почитать в его официальной документации.

4.2. Проигрывание музыки

Часто в программах хочется добавить какую-либо аудио-информацию. Злоупотреблять этим не следует, но работать со звуками надо уметь. Напишем простейшее приложение-плеер, который будет проигрывать одну песню. Создадим интерфейс нашего приложения в QtDesigner.

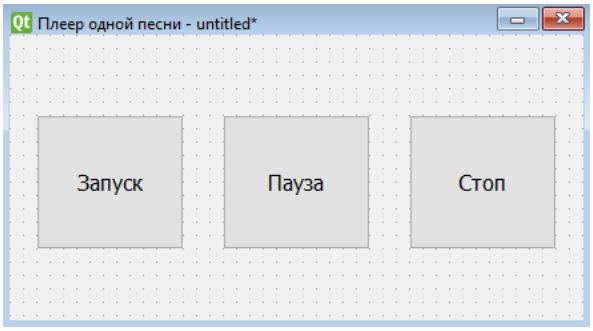


Рисунок 4.4

Для работы с музыкальными файлами нам понадобится импортировать виджет **QtMultimedia**, а чтобы получить полный путь к файлу — **QtCore**.

```
from PyQt5 import QtCore, QtMultimedia
```

Посмотрим, как выглядит наше приложение:

```
from player import Ui_Form

from PyQt5 import QtCore, QtMultimedia

from PyQt5.QtWidgets import QApplication, QMainWindow


class MyWidget(QMainWindow, Ui_Form):

    def __init__(self):

        super().__init__()

        self.setupUi(self)

        self.load_mp3('/Users/user/Documents/Yandex/FileLesson/test.mp3')

        self.playBtn.clicked.connect(self.player.play)

        self.pauseBtn.clicked.connect(self.player.pause)

        self.stopBtn.clicked.connect(self.player.stop)


    def load_mp3(self,filename):

        media = QtCore.QUrl.fromLocalFile(filename)

        content = QtMultimedia.QMediaContent(media)

        self.player = QtMultimedia.QMediaPlayer()

        self.player.setMedia(content)


app = QApplication(sys.argv)

ex = MyWidget()

ex.show()

sys.exit(app.exec_())
```

В нем всего две функции. Начнем с конца, с функции `load_mp3`. Она отвечает за загрузку музыкального файла и создание объекта **плеера**, в котором эта музыка будет содержаться.

Важно, что сначала мы передаем в эту функцию не название файла, а полный путь, после чего происходит «оборачивание» в формат `QUrl`. Затем создается медиа-объект, который помещается в плеер.

Во время инициализации, после того, как мы подключили наш интерфейс, мы вызываем эту функцию, и в нашем классе появится поле `player`. Затем необходимо лишь настроить кнопки, привязав к каждой из них соответствующую функцию нашего объекта. Функции объекта `player` достаточно очевидны по их названиям.

5. QT. Многопоточное приложение с помощью QThread.

Стандартное приложение GUI на PyQT работает в одном основном потоке, который запускает цикл событий и графический интерфейс. Если вы запустите длительную задачу в этом потоке, ваш графический интерфейс зависнет, пока задача не завершится. В течение этого времени пользователь не сможет взаимодействовать с приложением. К счастью, Qthread класс PyQT позволяет обойти эту проблему.

В этой главе мы научимся использовать PyQT QThread для предотвращения зависания графических интерфейсов.

5.1. Блокировка графического интерфейса пользователя длительными задачами

Длительные задачи, занимающие основной поток приложения с графическим интерфейсом пользователя и вызывающие зависание приложения, являются распространенной проблемой в программировании графического интерфейса, которая почти всегда приводит к плохому взаимодействию с пользователем. Например, рассмотрим следующее приложение с графическим интерфейсом пользователя:

Рисунок 5.1

Допустим, вам нужен label *Counting*, чтобы отразить общее количество кликов по кнопке *Click me!*. Щелчок по кнопке *Long-Running Task!* запустит задачу, выполнение которой требует много времени. Вашей длительной задачей может быть загрузка файла, запрос к большой базе данных или любая другая ресурсоемкая операция.

Вот первый подход к написанию кода этого приложения с использованием PyQT и одного потока выполнения:

```
import sys
from time import sleep

from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)

class Window(QMainWindow):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.clicksCount = 0
        self.setupUi()

    def setupUi(self):
        self.setWindowTitle("Freezing GUI")
        self.resize(300, 150)
        self.centralWidget = QWidget()
        self.setCentralWidget(self.centralWidget)
        # Create and connect widgets
        self.clicksLabel = QLabel("Counting: 0 clicks", self)
        self.clicksLabel.setAlignment(Qt.AlignHCenter | Qt.AlignVCenter)
        self.stepLabel = QLabel("Long-Running Step: 0")
        self.stepLabel.setAlignment(Qt.AlignHCenter | Qt.AlignVCenter)
        self.countBtn = QPushButton("Click me!", self)
        self.countBtn.clicked.connect(self.countClicks)
        self.longRunningBtn = QPushButton("Long-Running Task!", self)
        self.longRunningBtn.clicked.connect(self.runLongTask)
        # Set the layout
        layout = QVBoxLayout()
        layout.addWidget(self.clicksLabel)
        layout.addWidget(self.countBtn)
        layout.addStretch()
        layout.addWidget(self.stepLabel)
        layout.addWidget(self.longRunningBtn)
        self.centralWidget.setLayout(layout)

    def countClicks(self):
        self.clicksCount += 1
        self.clicksLabel.setText(f"Counting: {self.clicksCount} clicks")
```

```
    def reportProgress(self, n):
        self.stepLabel.setText(f"Long-Running Step: {n}")

    def runLongTask(self):
        """Long-running task in 5 steps."""
        for i in range(5):
            sleep(1)
            self.reportProgress(i + 1)

app = QApplication(sys.argv)
win = Window()
win.show()
sys.exit(app.exec())
```

В этом приложении `Freezing GUI` `.setupUi()` создает все необходимые графические компоненты для GUI. При нажатии левой кнопкой мышы на кнопку *Click me!* вызывается функция `.countClicks()`, что заставляет текст счетчика отражать количество нажатий кнопки.

Примечание. PyQT был впервые разработан для Python 2, в котором есть `exec` *ключевое слово. Чтобы избежать конфликта имен в этих более ранних версиях PyQT, в конец файла* `.exec_()` *ставился символ подчеркивания.*

Несмотря на то, что PyQt5 нацелен только на Python 3, в котором нет `exec` *ключевого слова, библиотека предоставляет два метода для запуска цикла обработки событий приложения:*

```
.exec_()
.exec()
```

Оба варианта метода работают одинаково, поэтому вы можете использовать любой из них в своих приложениях.

При клике по кнопке *Long-Running Task!* вызывается функция `.runLongTask()`, которая выполняет задачу, выполнение которой занимает 5 секунды. Это гипотетическая задача , которую вы кодируются с помощью `time.sleep(secs)`, [9] которая приостанавливает выполнение вызывающего потока для заданного количества секунд, `secs`.

В `.runLongTask()`, вы также вызываете, `.reportProgress()` чтобы label *Long-Running Step* отражала ход операции.

Это приложение работает так, как вы предполагаете? Запустите приложение и проверьте его поведение.

Когда вы нажимаете кнопку *Click me!*, label показывает количество нажатий. Однако, если вы щелкните по кнопке *Long-Running Task!*, приложение зависает и не отвечает. Кнопки больше не реагируют на щелчки, а labels не отражают состояние приложения.

Через пять секунд графический интерфейс приложения снова обновится. Label *Counting* показывает десять щелчков, отражающих пять щелчков, которые произошли, когда графический интерфейс был заморожен. Label *Long-Running Task!* не отражает прогресс вашей длительной операции. Он прыгает с нуля до пяти, не показывая промежуточных шагов.

Примечание. Даже если графический интерфейс вашего приложения зависает во время длительной задачи, приложение все равно регистрирует такие события, как щелчки и нажатия клавиш. Их просто невозможно обработать, пока не освободится основной поток.

Графический интерфейс приложения зависает в результате блокировки основного потока. Основной поток занят обработкой длительной задачи и не сразу реагирует на действия пользователя. Это раздражающее поведение, потому что пользователь не знает наверняка, правильно ли работает приложение или произошел сбой.

К счастью, есть несколько приемов, которые можно использовать для решения этой проблемы. Обычно используемое решение - запустить вашу длительную задачу вне основного потока приложения с помощью рабочего потока.

В разделах ниже вы узнаете, как использовать встроенную поддержку потоков PyQT для решения проблемы зависания или зависания графических

интерфейсов пользователя и обеспечения наилучшего взаимодействия с пользователем в ваших приложениях.

5.2. Многопоточность: Основы

Иногда вы можете разделить свои программы на несколько более мелких подпрограмм или задач, которые можно запускать в нескольких потоках. Это может ускорить ваши программы или может помочь вам улучшить взаимодействие с пользователем, предотвратив зависание ваших программ при выполнении длительных задач.

Thread представляет собой отдельный поток выполнения. В большинстве операционных систем поток является компонентом процесса [10], и процессы могут иметь несколько потоков, выполняющихся одновременно [11]. Каждый процесс представляет собой экземпляр программы или приложения, которое в данный момент выполняется в данной компьютерной системе.

У вас может быть столько потоков, сколько вам нужно. Задача состоит в том, чтобы определить правильное количество используемых потоков. Если вы работаете с потоками, привязанными к вводу-выводу [12], то количество потоков будет ограничено доступными системными ресурсами. С другой стороны, если вы работаете с потоками, привязанными к центральному процессору (ЦП) [13], то вы выиграете от наличия количества потоков, равного или меньшего, чем количество ядер ЦП в вашей системе.

Создание программ, способных выполнять несколько задач с использованием разных потоков - это метод программирования, известный как многопоточное программирование. В идеале при использовании этого метода несколько задач выполняются независимо одновременно. Однако это не всегда возможно. Есть как минимум два элемента, которые могут помешать программе запускать несколько потоков параллельно:

1. Центральный процессор (ЦП)
2. Язык программирования

Например, если у вас есть машина с одноядерным процессором , вы не можете запускать несколько потоков одновременно. Однако некоторые одноядерные процессоры могут имитировать выполнение параллельных потоков, позволяя операционной системе планировать время обработки между несколькими потоками. Это заставляет ваши потоки работать параллельно, даже если они действительно работают по одному.

С другой стороны, если у вас многоядерный процессор или компьютерный кластер, вы можете одновременно запускать несколько потоков. В этом случае ваш язык программирования становится важным фактором.

Некоторые языки программирования имеют внутренние компоненты, которые фактически запрещают реальное выполнение нескольких потоков параллельно. В этих случаях кажется, что потоки выполняются параллельно, потому что они используют преимущества системы планирования задач.

Многопоточные программы обычно сложнее писать, поддерживать и отлаживать, чем однопоточные программы, из-за сложности, связанной с разделением ресурсов между потоками, синхронизацией доступа к данным и координацией выполнения потоков. Это может вызвать несколько проблем:

- Race condition [14] - это когда поведение приложения становится недетерминированным из-за непредсказуемого порядка событий. Часто это результат того, что два или более потока обращаются к общему ресурсу без надлежащей синхронизации. Например, чтение и запись в память из разных потоков может привести к состоянию гонки, если операции чтения и записи выполняются в неправильном порядке.
- Deadlock [15] возникает, когда потоки бесконечно ждут освобождения заблокированного ресурса. Например, если поток блокирует ресурс и не разблокирует его после использования, тогда другие потоки не смогут использовать этот ресурс и будут ждать бесконечно. Взаимоблокировки также могут возникать, если поток A ожидает, пока поток B разблокирует

ресурс, а поток B ожидает, пока поток A разблокирует другой ресурс. Оба потока будут ждать вечно.

- Livelock [16] - это ситуация, в которой два или более потока постоянно действуют в ответ на действия друг друга. Потоки с динамической блокировкой не могут продолжить выполнение своей конкретной задачи, потому что они слишком заняты, отвечая друг другу. Однако они не заблокированы и не мертвы.
- Starvation [17] происходит, когда процесс никогда не получает доступа к ресурсам, необходимым для завершения своей работы. Например, если у вас есть процесс, который не может получить доступ к процессорному времени, значит, этому процессу не хватает процессорного времени, и он не может выполнять свою работу.

При создании многопоточных приложений нужно быть осторожным, чтобы защитить свои ресурсы от одновременной записи или доступа для изменения состояния. Другими словами, вам нужно предотвратить одновременный доступ нескольких потоков к данному ресурсу.

Широкий спектр приложений может извлечь выгоду из использования многопоточного программирования как минимум тремя способами:

1. Ускорение ваших приложений за счет использования преимуществ многоядерных процессоров;
2. Упрощение структуры приложения путем разделения его на более мелкие подзадачи;
3. Поддержание отзывчивости и актуальности приложения за счет передачи длительно выполняемых задач рабочим потокам.

В реализации Python на языке C , также известной как CPython , потоки не работают параллельно. CPython имеет global interpreter lock (GIL) [18], который

представляет собой замок, который в основном позволяет выполнять только один Python поток в конкретный момент времени.

Это может негативно повлиять на производительность многопоточных приложений Python из-за накладных расходов, связанных с переключением контекста между потоками. Однако многопоточность в Python может помочь вам решить проблему зависания или зависания приложений при обработке длительно выполняемых задач.

5.3. Многопоточность в PyQt с Qthread

Qt и, следовательно, PyQT , предоставляют собственную инфраструктуру для создания многопоточных приложений с использованием QThread. Приложения PyQT могут иметь два разных типа потоков:

1. Основной поток
2. Рабочие потоки

Основной поток приложения существует всегда. Здесь запускается приложение и его графический интерфейс. С другой стороны, наличие рабочих потоков зависит от потребностей приложения в обработке. Например, если ваше приложение обычно выполняет тяжелые задачи, выполнение которых требует много времени, вам может потребоваться, чтобы рабочие потоки выполняли эти задачи и избегали зависания графического интерфейса приложения.

Основной поток

В приложениях PyQT основной поток выполнения также известен как поток GUI, поскольку он обрабатывает все виджеты и другие компоненты GUI. Вы запускаете этот поток, вызывая `.exec()` своего `QApplication` объекта. Основной поток запускает цикл событий приложения, а также ваш код Python. Он также обрабатывает ваши окна, диалоги и связь с операционной системой.

По умолчанию любое событие или задача, которые происходят в основном потоке приложения, включая события пользователя в самом графическом

интерфейсе, будут выполняться синхронно или одна задача за другой. Итак, если вы запускаете длительную задачу в основном потоке, тогда приложению необходимо дождаться завершения этой задачи, и графический интерфейс перестает отвечать.

Важно отметить, что вы должны создавать и обновлять все свои виджеты в потоке графического интерфейса. Однако вы можете выполнять другие длительные задачи в рабочих потоках и использовать их результаты для загрузки компонентов графического интерфейса вашего приложения. Это означает, что компоненты графического интерфейса будут действовать как потребители, которые получают информацию от потоков, выполняющих фактическую работу.

Рабочие потоки

Вы можете создать столько рабочих потоков, сколько вам нужно в ваших приложениях PyQT. Рабочие потоки - это вторичные потоки выполнения, которые можно использовать для выгрузки длительно выполняемых задач из основного потока и предотвращения зависания графического интерфейса.

Вы можете создавать рабочие потоки, используя QThread. Каждый рабочий поток может иметь собственный цикл событий и поддерживать механизм сигналов и слотов PyQT для связи с основным потоком. Если вы создаете объект из любого класса, который наследуется от `QObject` определенного потока, то говорят, что этот объект принадлежит этому потоку или имеет с ним сходство. Его дочерние элементы также должны принадлежать к тому же потоку.

`QThread` сам по себе не поток. Это оболочка для потока операционной системы. Настоящий объект потока создается при вызове `QThread.start()`.

`QThread` предоставляет интерфейс прикладного программирования высокого уровня (API) для управления потоками. Этот API включает в себя сигналы, такие как `.started()` и `.finished()`, которые генерируются при

запуске и завершении потока. Он также включает в себя методы и слоты, такие как `.start()`, `.wait()`, `.exit()`, `.quit()`, `.isFinished()`, и `.isRunning()`.

Как и в случае с любыми другими решениями для потоковой `QThread` передачи, вы должны защищать свои данные и ресурсы от одновременного доступа. В противном случае вы столкнетесь с множеством проблем, включая взаимоблокировки, повреждение данных и так далее.

5.4. Использование QThread vs Python’s threading

Когда дело доходит до работы с потоками в Python, вы обнаружите, что стандартная библиотека Python предлагает согласованное и надежное решение с `threading` модулем. Этот модуль предоставляет высокоуровневый API для многопоточного программирования на Python.

Обычно вы будете использовать `threading` в своих приложениях Python. Однако, если вы используете PyQT для создания приложений с графическим интерфейсом на Python, у вас есть другой вариант. PyQT предоставляет полный, полностью интегрированный высокоуровневый API для многопоточности.

Вам может быть интересно, что мне использовать в своих приложениях PyQT, поддержку потоков Python или поддержку потоков PyQT? Ответ в том, что это зависит от обстоятельств.

Например, если вы создаете приложение с графическим интерфейсом пользователя, которое также будет иметь веб-версию, потоки Python могут иметь больше смысла, потому что ваш сервер вообще не будет зависеть от PyQT. Однако, если вы создаете простые приложения PyQT, потоки PyQT для вас.

Использование поддержки потоков PyQT дает следующие преимущества:

– Классы, связанные с потоками, полностью интегрированы с остальной инфраструктурой PyQT;

- Рабочие потоки могут иметь свой собственный цикл событий, который позволяет обрабатывать события;
- Межпоточная связь возможна с использованием сигналов и слотов.

Практическим правилом может быть использование поддержки потоков PyQT, если вы собираетесь взаимодействовать с остальной частью библиотеки, и использование поддержки потоков Python в противном случае.

5.5. Использование QThread для предотвращения зависания графических интерфейсов

Обычно потоки в приложении с графическим интерфейсом используются для разгрузки длительно выполняемых задач рабочими потоками, чтобы графический интерфейс оставался отзывчивым на действия пользователя. В PyQT вы используете `QThread` для создания рабочих потоков и управления ими.

Согласно документации Qt, есть два основных способа создания рабочих потоков с помощью `QThread`:

1. Создайте экземпляр `QThread` напрямую и создайте worker `QObject`, а затем вызовите `.moveToThread()`, используя поток в качестве аргумента. Worker должен содержать все необходимые функции для выполнения конкретной задачи.
2. Подкласс `QThread` и переопределение `.run()`. Реализация `.run()` должна содержать все необходимые функции для выполнения конкретной задачи.

Создание экземпляра `QThread` обеспечивает параллельный цикл событий. Цикл событий позволяет объектам, принадлежащим потоку, получать сигналы в свои слоты, и эти слоты будут выполняться внутри потока. Создание подклассов `QThread` позволяет приложению выполнять параллельный код без цикла обработки событий.

В сообществе Qt ведутся споры о том, какой из этих подходов лучше всего подходит для создания рабочих потоков. Однако сообщество Qt и сопровождающие его рекомендуют первый подход.

Первый подход к созданию рабочих потоков требует следующих шагов:

1. Подготовьте worker объект путем создания подкласса `QObject` и поместите в него свою длительную задачу.
2. Создайте новый экземпляр worker класса.
3. Создайте новый `QThread` экземпляр.
4. Переместите worker объект во вновь созданный поток, вызвав `.moveToThread(thread)`.
5. Подключите необходимые сигналы и слоты, чтобы гарантировать межпотоковую связь.
6. Вызовите `.start()` у `QThread`.

Вы можете превратить ваше приложение Freezing GUI в приложение с адаптивным графическим интерфейсом, выполнив следующие действия:

```
from PyQt5.QtCore import QObject, QThread, pyqtSignal
# Snip...

# Step 1: Create a worker class
class Worker(QObject):
    finished = pyqtSignal()
    progress = pyqtSignal(int)

    def run(self):
        """Long-running task."""
        for i in range(5):
            sleep(1)
            self.progress.emit(i + 1)
        self.finished.emit()

class Window(QMainWindow):
    # Snip...
    def runLongTask(self):
        # Step 2: Create a QThread object
        self.thread = QThread()
        # Step 3: Create a worker object
        self.worker = Worker()
        # Step 4: Move worker to the thread
        self.worker.moveToThread(self.thread)
        # Step 5: Connect signals and slots
        self.thread.started.connect(self.worker.run)
        self.worker.finished.connect(self.thread.quit)
        self.worker.finished.connect(self.worker.deleteLater)
```

```
        self.thread.finished.connect(self.thread.deleteLater)
        self.worker.progress.connect(self.reportProgress)
        # Step 6: Start the thread
        self.thread.start()

        # Final resets
        self.longRunningBtn.setEnabled(False)
        self.thread.finished.connect(
            lambda: self.longRunningBtn.setEnabled(True)
        )
        self.thread.finished.connect(
            lambda: self.stepLabel.setText("Long-Running Step: 0")
        )
```

Во-первых, вы выполняете необходимый импорт. Затем вы выполняете шаги, которые видели раньше.

На шаге 1 вы создаете `Worker` подкласс `QObject`. В `Worker`, вы создаете два сигнала, `finished` и `progress`. Обратите внимание, что вы должны создавать сигналы как атрибуты класса.

Вы также создаете метод с именем `.runLongTask()`, в который помещаете весь необходимый код для выполнения вашей длительной задачи. В этом примере вы моделируете длительную задачу, используя цикл `for`, который повторяется 5 раз, с задержкой в одну секунду на каждой итерации. Цикл также излучает `progress` сигнал, указывающий на прогресс операции. Наконец, `.runLongTask()` издает `finished` сигнал, указывающий на то, что обработка завершена.

На шагах 2–4 вы создаете экземпляр `QThread`, который предоставит пространство для выполнения этой задачи, а также экземпляр `Worker`. Вы переместите `Worker` объект в поток путем вызова `.moveToThread()` у `worker`, используя в `thread` качестве аргумента.

На шаге 5 вы подключаете следующие сигналы и слоты:

– Сигнал потока `started` в слот `worker`, `.runLongTask()` чтобы гарантировать, что при запуске потока `.runLongTask()` он будет вызываться автоматически;
– Сигнал `worker finished` к `.quit()` слоту потока, для выхода из `thread` когда `worker` завершает свою работу;

– `finished` сигнал в `.deleteLater()` слот в обоих объектов для удаления объектов `worker` и `thread`, когда работа завершается.

Наконец, на шаге 6 вы запускаете поток, используя `.start()`.

После запуска потока вы делаете некоторые сбросы, чтобы приложение работало согласованно. Вы отключаете кнопку *Long-Running Task!*, чтобы пользователь не нажал на нее во время выполнения задачи. Вы также соединяете `finished` сигнал потока с lambda функцией, которая включает кнопку *Long-Running Task!*, когда выполнения задания в потоке закончится. Ваше последнее соединение сбрасывает текст label *Long-Running Step*.

Если вы запустите это приложение, то на вашем экране появится следующее окно:

Рисунок 5.2

ЗАКЛЮЧЕНИЕ

Данное учебно-методическое пособие даст базовые знания по работе «прослойки» PyQT библиотеки QT, в области: форматов с фиксированной и с произвольной длиной записи (DSV, TSV, CSV); форматов с произвольной длиной записи, методов работы с ними при помощи строковых функций и специализированной библиотеки csv; базы данных и языка SQL; взаимодействия с пользователем – работой с клавиатурой и мышкой; сборки программы в exe-файл; дополнительных компонентов PyQT на примере PyQTGraph — библиотеки для построения графиков; проигрывание музыки в своих приложениях; работы с многопоточными приложениями с помощью QThread.

ЛИТЕРАТУРА

1. https://en.wikipedia.org/wiki/Delimiter-separated_values.

2. https://en.wikipedia.org/wiki/Tab-separated_values.

3. https://ru.wikipedia.org/wiki/CSV.

4. Тюменцев Е. А. О формализации процесса разработки программного обеспечения //Математические структуры и моделирование. – 2017. – №. 3 (43).

5. Beaulieu A. Learning SQL: Generate, Manipulate, and Retrieve Data. – O'Reilly Media, 2020.

6. Крамаренко Т. А., Деменков И. А., Михеев А. И. Выбор клиент-серверной СУБД для реализации информационной системы //Современные информационные технологии. – 2016. – Т. 24. – С. 11.

7. Guzdial M., Ericson B. Introduction to computing and programming in python. – Pearson, 2016.

8. Nield T. Getting Started with SQL: A Hands-on Approach for Beginners. – " O'Reilly Media, Inc.", 2016.

9. https://realpython.com/python-sleep/.

10. https://en.wikipedia.org/wiki/Process_(computing).

11. https://en.wikipedia.org/wiki/Concurrent_computing.

12. https://en.wikipedia.org/wiki/I/O_bound.

13. https://en.wikipedia.org/wiki/CPU-bound.

14. https://en.wikipedia.org/wiki/Race_condition.

15. https://en.wikipedia.org/wiki/Deadlock.

16. https://en.wikipedia.org/wiki/Deadlock#Livelock.

17. https://en.wikipedia.org/wiki/Starvation_(computer_science).

18. Hungilo G. G., Emmanuel G., Pranowo. Performance comparison in simulation of Mandelbrot set fractals using Numba //AIP Conference Proceedings. – AIP Publishing LLC, 2020. – Т. 2217. – №. 1. – С. 030007.

СВЕДЕНИЯ ОБ АВТОРЕ

Тарланов Арслан Тарланович, преподаватель кафедры КБ-4 «Интеллектуальные системы информационной безопасности» Института комплексной безопасности и специального приборостроения РТУ МИРЭА («МИРЭА – Российский технологический университет»)
(119454, Россия, Москва, пр-т Вернадского, д. 78).