**Обработка нажатий клавиатуры**

Мы уже умеем забирать данные пользовательского ввода из различных виджетов и это здорово, но иногда возникают ситуации, когда нам надо получить не введенные данные, а сам факт нажатия на ту или иную клавишу (или комбинацию клавиш). Давайте посмотрим, как это можно сделать с использованием библиотеки PyQt.

На уроке про Tkinter в прошлом году мы узнали, что у каждой клавиши на клавиатуре есть свой **уникальный** код. В PyQt также, как в Tkinter, есть достаточно удобная функциональность сопоставления кода клавиши и понятного имени. Она расположена в модуле PyQt6.QtCore.Qt. Согласитесь, что гораздо легче понимать смысл вот такой строки: print(Qt.Key.Key\_F), нежели такой: print(70).

Подключить модуль Qt можно, например, таким способом:

from PyQt6.QtCore import Qt

Все имена кнопок и их коды можно найти [здесь](https://doc.qt.io/qt-6/qt.html).

Для обработки нажатий на клавиши в классе создаваемого виджета необходимо переопределить (создать) метод keyPressEvent(self, event):

def keyPressEvent(self, event):

if event.key() == Qt.Key.Key\_F:

# code

Если во время работы приложения будет нажата клавиша **F**, условие выполнится.

Для обработки комбинаций клавиш можно использовать понятие модификатора клавиши, например: Qt.AltModifier или Qt.ShiftModifier. Для обработки «горячего сочетания» (**Hotkey**) ALT + SHIFT + Q код должен быть таким:

def keyPressEvent(self, event):

if event.modifiers() == (Qt.KeyboardModifier.AltModifier | Qt.KeyboardModifier.ShiftModifier):

if event.key() == Qt.Key.Key\_Q:

# code

Параметр event в этом методе будет иметь класс QKeyEvent. Почитать про этот класс можно [тут](https://doc.qt.io/qt-5/qkeyevent.html).

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

**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 PyQt6.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)

def mouseMoveEvent(self, event):

self.coords.setText(f"Координаты: {event.pos().x()}, {event.pos().y()}")

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

app = QApplication(sys.argv)

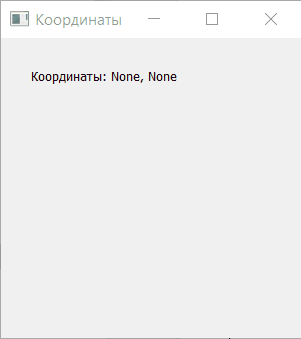
ex = Example()

ex.show()

sys.exit(app.exec())

Метод mouseMoveEvent() срабатывает при изменении положения мышки. Функция pos()у параметра event класса QMouseEvent ([ссылка на документацию](https://doc.qt.io/qt-6/qmouseevent.html)) возвращает координаты текущего положения курсора относительно левого верхнего угла формы в виде объекта класса QPoint . Функции x() и y() объекта класса QPoint возвращают конкретные значения координат по осям. Но, если вы запустите эту программу и попробуете просто поводить мышкой, ничего не произойдет. По умолчанию это событие активируется только при зажатой клавише мыши (причем неважно какой, хоть при нажатии колесика). Зажмите любую клавишу — и все заработает. А чтобы отлавливать положение мыши, даже когда клавиша не зажата, в инициализатор нашего виджета надо добавить следующую строку:

self.setMouseTracking(True)



Кроме метода mouseMoveEvent(), есть и другие. Полный список событий виджета, реакции на которые можно переопределить, можно посмотреть [тут](https://doc.qt.io/qt-6/qwidget.html#protected-functions). Применительно к работе с мышкой, помимо рассмотренного метода mouseMoveEvent(), есть еще:

* mousePressEvent, отвечающий за обработку нажатия на какую-либо из кнопок мыши
* mouseReleaseEvent, отвечающий за отпускания нажатия на кнопку мыши
* mouseDoubleClickEvent, обрабатывающий двойной клик

Все они в качестве параметра принимают объект класса QMouseEvent, который может сообщить не только координаты, но и какая из кнопок была нажата. Для этого существует метод button(). Сравнить результат вызова можно со встроенными значениями из модуля PyQt5.QtCore.Qt: MouseButton.LeftButton, MouseButton.RightButton и даже MouseButton.MiddleButton.

import sys

from PyQt6.QtCore import Qt

from PyQt6.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.btn = QLabel(self)

self.btn.setText("Никакая")

self.btn.move(30, 50)

def mousePressEvent(self, event):

self.coords.setText(f"Координаты:{event.pos().x()}, {event.pos().y()}")

if event.button() == Qt.MouseButton.LeftButton:

self.btn.setText("Левая")

elif event.button() == Qt.MouseButton.RightButton:

self.btn.setText("Правая")

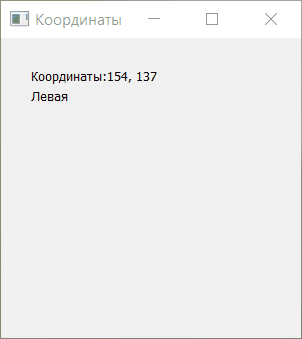
if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

ex = Example()

ex.show()

sys.exit(app.exec())



Если есть необходимость обработать одновременное нажатие на несколько кнопок мыши, то можно воспользоваться не методом button(), а методом buttons().

**Создание standalone приложений**

В отличие от компилируемых языков программирования вроде C++, Python — язык интерпретируемый. А языки подобного плана не совсем предназначены для создания привычных нам исполняемых exe-файлов (если говорить про ОС Windows), поэтому 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 ОС (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

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

pyinstaller --onefile --noconsole main.py

Где main.py — главный файл вашего приложения.

Что делают модификаторы:

* **onefile** собирает программу в один exe-файл. Это удобно, если программа небольшая, у нее мало графических, аудио и прочих файлов, не относящихся к программному коду
* **noconsole**. Если не писать этот модификатор, то, кроме окна программы будет открываться Python-консоль. Иногда, например, для отладки, удобно оставлять командную строку для вывода системных сообщений, при передаче же программы клиенту консоль обычно скрывают

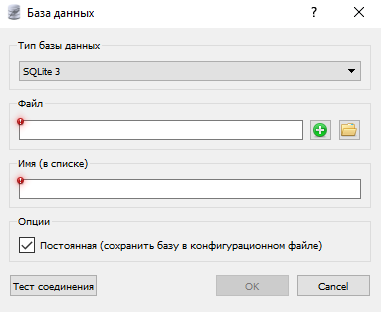
Есть и другие модификаторы. Например, **icon**. В нем можно указать путь до картинки в формате .ico, и она будет отображаться при просмотре программы в проводнике. Документацию на библиотеку можно почитать [тут](https://pyinstaller.readthedocs.io/en/stable/).

После выполнения команды упаковки в исполняемый файл 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.

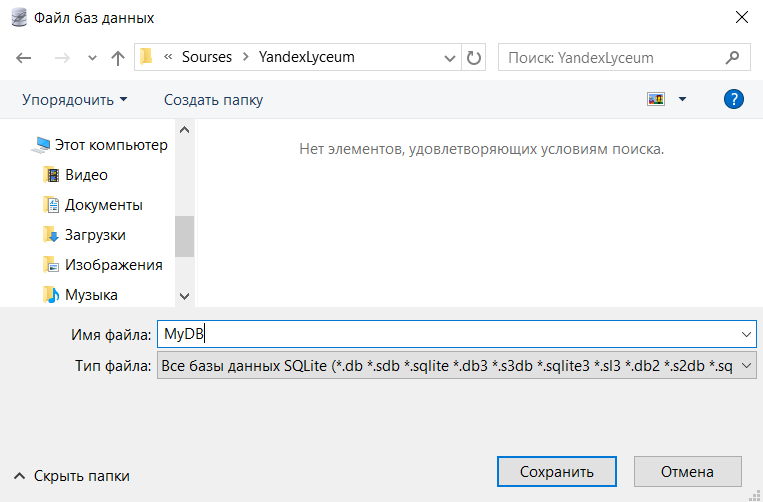
**Создание БД**

С чего начинается любой проект по созданию приложения для работы с базой данных? Конечно же, с проектирования базы данных. Как мы уже отмечали ранее, процесс проектирования сложных баз данных — это достаточно серьезная область знаний, и мы к ней только прикоснемся. Есть большое количество специализированных инструментов, которые помогают в создании масштабных баз данных, которые насчитывают сотни или тысячи таблиц и связей, но для простых баз данных этап проектирования достаточно часто объединяется с непосредственно созданием с помощью менеджеров баз данных. Мы уже работали с SQLiteStudio, давайте воспользуемся им для создания базы данных, которая будет хранить информацию о любимых книгах.

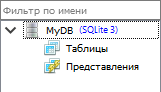
Открываем в верхнем меню вкладку **Database** и выбираем пункт **Add a database**.



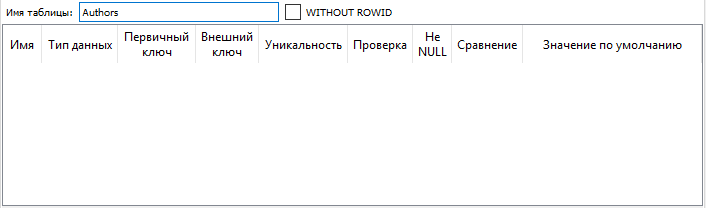
Нажимаем на **плюс** и вводим желаемое имя для нашей базы.



Сохраняем. В боковом меню должна появиться новая база данных.

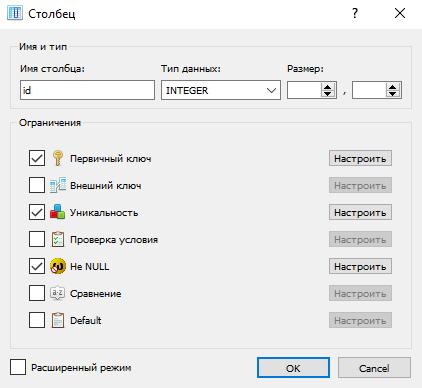


Теперь в нашей БД необходимо создать таблицы, в которых будет храниться информация. Для этого необходимо навести курсор на базу, нажать правую кнопку и выбрать пункт **Create a table**. Откроется окно для создания таблицы. Введем ее имя.

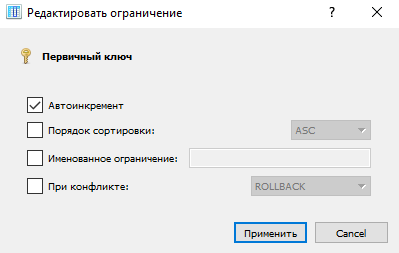


Двойным нажатием на белое поле откроем редактор столбца. Для того чтобы идентифицировать книги, создадим специальный столбец, назовем его id. Это будет число (то есть тип INTEGER), которое будет являться т. н. **первичным ключом**, то есть оно должно быть обязательно заполнено (не может быть NULL, аналог None в Python), уникально и для удобства он будет автоматически увеличиваться при добавлении новых значений (это называется **Автоинкремент**).

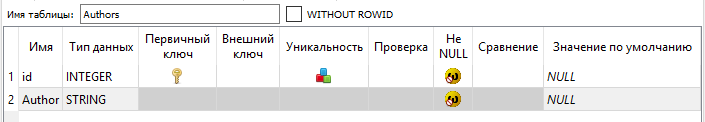
Настроим все эти значения:



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



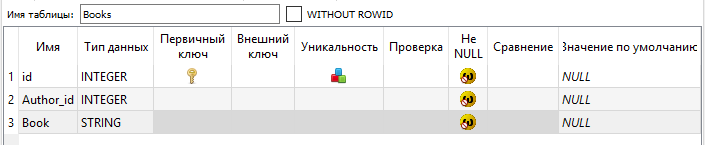
Теперь в нашей таблице есть первое поле — id. Добавим еще одно поле — author, в котором будем хранить фамилию автора и его инициалы. Для него подойдет тип данных STRING.



После добавления всех полей необходимо применить все изменения. 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.

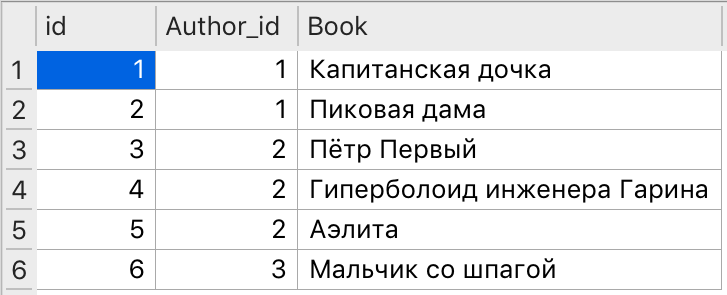
Теперь аналогичным образом создаем таблицу для хранения книг. Но будем указывать не ФИО автора, а его идентификатор из первой таблицы.



Теперь заполним наши таблицы. Таблицу авторов:



И таблицу книг:



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

1. Чтобы избежать опечаток и последующих проблем с выборками. Ведь ФИО можно записать очень по-разному.
2. 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](http://www.sqlitetutorial.net/).

**Управление зависимостями**

Относительно большие проекты, вы в этом сможете убедиться уже очень скоро, 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

Если не указать версию пакета, будет установлена последняя стабильная версия. Кроме равенства версий, можно указывать версии, которые должны быть больше/меньше/не больше/не меньше указанных с помощью соответствующих операторов сравнения. Например, запись pkg>=1.0,<=2.0 означает, что должна быть установлена версия пакета не меньше 1.0, но не больше 2.0.

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.

