



Урок 4

Хранение данных в БД (продолжение) и основы Qt

SQLAlchemy (продолжение). Объект сессии. Библиотека PyQt5.
Qt Designer. Сигналы.

[Введение](#)

[SQLAlchemy \(продолжение\)](#)

[Декларативный стиль работы \(продолжение\)](#)

[Создание сессии](#)

[Добавление новых объектов](#)

[Сессия](#)

[Отслеживание состояния](#)

[Контроль транзакций](#)

[Состояния сессии](#)

[Резюме](#)

[Знакомство с Qt и PyQt](#)

[Первое приложение PyQt](#)

[Создание интерфейса](#)

[Вручную](#)

[Qt Designer](#)

[Краткое руководство](#)

[Создание формы](#)

[Изменение виджетов \(Widget Editing Mode\)](#)

[Изменение сигналов-слотов \(Signals and Slots Editing Mode\):](#)

[Изменение партнеров \(Buddy Editing Mode\)](#)

[Изменение порядка переключений \(Tab Order Editing Mode\)](#)

[Предпросмотр и сохранение](#)

[Загрузка ui-файла в программе](#)

[Преобразование ui-файла в py-файл](#)

[Модули и классы PyQt5](#)

[PyQt5.Qt](#)

[PyQt5.QtCore](#)

[PyQt5.QtGui](#)

[PyQt5.QtMultimedia](#)

[PyQt5.QtMultimediaWidgets](#)

[PyQt5.QtNetwork](#)

[PyQt5.QtOpenGL](#)

[PyQt5.QtSql](#)

[PyQt5.QtSerialPort](#)

[PyQt5.QtTest](#)

[PyQt5.QtWidgets](#)

[PyQt5.QtWinExtras](#)

[PyQt5.QtXml](#)

[PyQt5.uic](#)

[Отличия PyQt5 и PyQt4](#)

[Сигналы и обработчики](#)

[Итоги](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

[Приложение](#)

Введение

На этом уроке продолжим изучать взаимодействие Python-кода с реляционными базами данных. Научимся использовать декларативный стиль работы с ORM-библиотекой **SQLAlchemy**. Изучим основы работы с фреймворком **Qt**.

Python позволяет создавать не только консольные приложения для обработки данных, но и приложения с графическим интерфейсом пользователя (Graphical User Interface, GUI). Для этого применяют библиотеки **tkinter** (включена в стандартную библиотеку Python), **PyQt**, **wxPython**, **PyGTK**.

Библиотека **PyQt** — это набор «привязок» графического фреймворка **Qt** для Python, выполненный в виде расширения этого языка.

PyQt разработан британской компанией Riverbank Computing. **PyQt** работает на всех платформах, поддерживаемых **Qt**: Linux и других UNIX-подобных ОС, Mac OS X и Windows. Есть две версии: **PyQt5**, поддерживающий Qt 5, и **PyQt4**, поддерживающий Qt 4. **PyQt** распространяется под лицензиями GPL (2 и 3 версии) и коммерческой.

PyQt практически полностью реализует возможности **Qt**. А это более 600 классов и 6000 функций и методов, включая:

- существующий набор виджетов графического интерфейса;
- стили виджетов;
- доступ к базам данных с помощью **SQL (ODBC, MySQL, PostgreSQL, Oracle)**;
- **QScintilla**, основанный на **Scintilla** виджет текстового редактора;
- поддержку интернационализации (i18n);
- парсер **XML**;
- поддержку **SVG**;
- интеграцию с **WebKit**, движком рендеринга HTML;
- поддержку воспроизведения видео и аудио.

SQLAlchemy (продолжение)

Декларативный стиль работы (продолжение)

Создание сессии

Доступ к базе данных осуществляется через механизм сессии **Session**. При запуске приложения необходимо на одном уровне с **create_engine()** определить класс **Session**, который будет служить фабрикой объектов сессий (**Session**):

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
```

Если у приложения нет Engine-объекта базы данных, можно создать сессию:

```
Session = sessionmaker()
```

Позже, когда будет создано подключение к базе данных с помощью `create_engine()`, необходимо соединить его с сессией, используя `configure()`:

```
Session.configure(bind=engine)
```

Класс **Session** будет создавать Session-объекты, которые привязаны к базе данных. Другие транзакционные параметры тоже можно определить вызовом функции `sessionmaker()`.

Когда требуется общение с базой, надо создать объект класса **Session**:

```
session = Session()
```

Сессия здесь ассоциирована с **SQLite**, но у нее еще нет открытых соединений с этой базой. При первом использовании она получает соединение из набора, который поддерживается **engine**, и удерживает его до тех пор, пока не будут применены все изменения или (и) закрыт объект сессии.

Добавление новых объектов

Чтобы сохранить объект **User**, нужно добавить его к имеющейся сессии, вызвав ее метод `add()`:

```
admin_user = User("vasia", "Vasiliy Pypkin", "vasia2000")
session.add(admin_user)
```

Этот объект будет ожидать сохранения, SQL-запрос пока выполнен не будет. Чтобы сохранить данные пользователя, как только понадобится сессия пошлет SQL-запрос, используя процесс сброса на диск (**flush**). Если мы запросим «Васю» из базы, сначала вся ожидающая информация будет сброшена в базу, а запрос последует потом.

Создадим новый объект запроса (**Query**), который загружает User-объекты. Запрос фильтруется по атрибуту «имя=Вася», и из результата запроса методом `first()` извлекается только первый результат. Возвращается тот **User**, который был добавлен ранее:

```
q_user = session.query(User).filter_by(name="vasia").first()
print(q_user) # <User('vasia','Vasiliy Pypkin', 'vasia2000')>
```

Сессия определила, что запись из таблицы, которую она вернула, — та же, что она уже представляла во внутренней хэш-таблице объектов. Поэтому в результате был получен тот же объект, что и добавленный. Концепция **ORM**, которая работает здесь, называется картой идентичности. Благодаря ей все операции над конкретной записью внутри сессии могут работать с одним набором данных. Как только объект с заданным первичным ключом появится в сессии, все SQL-запросы на ней вернут тот же Python-объект для этого первичного ключа. Если в эту сессию попытаться поместить другой, уже сохраненный объект с тем же первичным ключом, будет выдана ошибка. Для добавления нескольких User-объектов необходимо использовать метод `add_all()`:

```
# Добавить сразу несколько записей
session.add_all([User("kolian", "Cool Kolian[S.A.]", "kolian$$$"),
                 User("zina", "Zina Korzina", "zk18")])
```

При изменении данных объекта, находящегося в сессии, сессия будет «знать», что объект был модифицирован:

```
admin_user.password = "--VP2001=="
print(session.dirty)      # IdentitySet([<User('vasia','Vasiliy Pypkin',
                             '--VP2001==')>])
```

Атрибут сессии **new** хранит объекты, ожидающие сохранения в базу данных:

```
print(session.new)
# IdentitySet([<User('kolia','Cool Kolian[S.A.]', 'kolia$$$')>,
               <User('zina','Zina Korzina', 'zk18')>])
```

Метод **commit()** фиксирует транзакцию, которая до того была в процессе, отправляя все оставшиеся изменения в базу:

```
session.commit()
```

Ресурсы подключений, что использовались в сессии, снова освобождаются и возвращаются в набор. Последовательные операции с сессией произойдут в новой транзакции, которая снова запросит себе ресурсы по первому требованию. Атрибуту **id**, который раньше был **None**, будет присвоено значение:

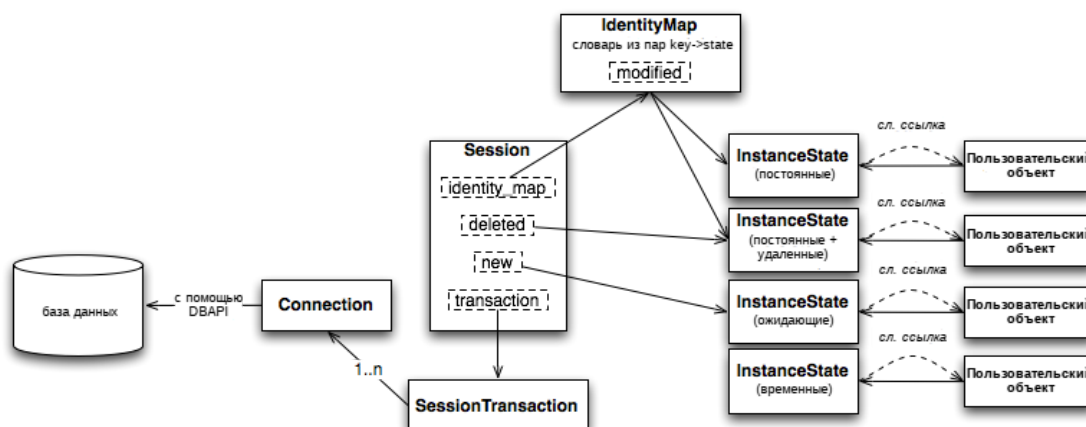
```
print('User ID:', admin_user.id)  # User ID: 1
```

Когда сессия добавит новые записи в базу, все только что созданные идентификаторы будут доступны в объекте — немедленно или по первому требованию. В данном случае при обращении к объекту была перезагружена целая запись, так как после вызова **commit()** началась новая транзакция. **SQLAlchemy** обновляет данные от предыдущей транзакции при первом обращении с новой транзакцией, так что пользователю доступно самое последнее ее состояние.

Пример работы с БД через механизм сессии представлен в **листинге 1**.

Сессия

На рисунке приведена схема взаимодействия сессии (**Session**) с базовыми структурами **SQLAlchemy**.



Общедоступные объекты здесь — это сам **Session**, а также коллекция пользовательских объектов. Каждый из них является экземпляром класса, который используется для создания отображения. Можем увидеть, что используемые для отображения объекты ссылаются на конструкцию из состава **SQLAlchemy** с именем **InstanceState**. Она отслеживает состояние отдельного объектно-реляционного отображения, включая ожидающие операции изменения атрибутов, а также не истек ли срок действия последних. Объект **InstanceState** — это инструмент для работы с атрибутами на уровне экземпляра класса. Мы рассмотрели его в разделе «Анатомия отображения». Он соответствует объекту **ClassManager** на уровне класса, позволяющего поддерживать состояние словаря, который используют для создания отображения объекта (то есть атрибута `__dict__`, описанного в рамках Python) — на стороне ассоциированных с классом объектов **AttributeImpl**.

Отслеживание состояния

Объект **IdentityMap** позволяет создавать отображение индивидуальных данных БД для объектов **InstanceState**. Они в свою очередь используются теми объектами, которым требуются эти индивидуальные, или постоянные (persistent), данные. Стандартная реализация объекта **IdentityMap** взаимодействует с объектом **InstanceState**, чтобы самостоятельно управлять объемом занятой памяти, удаляя созданные пользователем отображения, когда уничтожаются все жесткие ссылки на них. Этот объект функционирует аналогично объекту **WeakValueDictionary** из состава Python. Объект **Session** защищает набор всех объектов с пометкой «устаревший» (dirty) или «удаленный» (deleted), а также охраняет объекты с пометкой «новый» (new) от механизма сборки мусора — создает жесткие ссылки на них, если ожидаются их изменения. Все такие ссылки удаляются, когда выполняется операция сохранения данных.

Объект **InstanceState** также выполняет критически важную задачу — поддерживает «список изменений» для атрибутов определенного объекта. Используется система перемещения данных при изменении, которая сохраняет «данные предыдущего состояния» определенного атрибута в словаре с именем **committed_state** перед тем, как использовать переданное значение для изменения значения в словаре атрибутов объекта. Во время сохранения изменений содержимое словаря **committed_state**, а также ассоциированного с объектом словаря `__dict__` сравниваются, чтобы сформировать набор измененных данных для каждого из объектов.

Контроль транзакций

Объект **Session** при обычном сценарии использования поддерживает открытую транзакцию для всех операций. Она завершается в момент вызова метода **commit** или **rollback**. Объект **SessionTransaction** поддерживает набор объектов **Connection**, который может быть как пустым, так и заполненным. Причем каждый объект в нем представляет открытую транзакцию для определенной базы данных. Объект **SessionTransaction** — это объект с отложенной инициализацией, которая начинается при отсутствии данных состояния БД. Так как определенная база данных должна участвовать в выполнении запроса, соответствующий ей объект соединения **Connection** добавляется в список соединений объекта **SessionTransaction**. Обычно в каждый момент времени применяется одно соединение с базой данных, но поддерживается и сценарий использования множества соединений. Согласно ему определенное соединение задействуется для конкретной операции — в соответствии данными конфигурации, ассоциированными с объектами **Table**, **Mapper**, либо в соответствии с конструкциями языка **SQL**, применяемыми в рамках операции. При использовании множества соединений также может координироваться выполнение транзакции при применении двухфазной схемы — в случаях, когда реализация **DB-API** предоставляет ее.

Состояния сессии

Понимать состояния сессии полезно, чтобы предотвращать исключения и обрабатывать неопределенное поведение. Существует четыре состояния для объектов данных:

- временное (**Transient**) — объект вне сессии и вне базы данных;
- в ожидании (**Pending**) — объект был добавлен в сессию через **add()**, но не был сохранен в БД;
- постоянное (**Persistent**) — объект имеет соответствующую запись в БД;
- отключен (**Detached**) — объект отключен от сессии, но имеет запись в БД.

Резюме

- Вызов **sessionmaker()** необходимо выполнить только один раз, желательно в глобальном пространстве имен;
- Сессию необходимо отделять от функций и объектов (передавать ее, как параметр);
- Важно понимать, где начинается и заканчивается транзакция. Надо делать транзакции короткими (завершать их после серии операций, а не держать открытыми).

Чтобы закрепить навыки работы со связями таблиц и запросами, предлагаем самостоятельно изучить примеры: **листинг 1** и **листинг 2**. На последний обратите особое внимание — в нем демонстрируется, с какими особенностями могут реализовываться связи «многие ко многим».

Знакомство с Qt и PyQt

Установка стандартная:

```
pip install PyQt5
```

Примеры работы с библиотекой **PyQt5** можно найти на [GitHub](#) и в директории **pyqt5-examples** в файлах урока.

Документация библиотеки **PyQt5** — по адресу <http://pyqt.sourceforge.net/Docs/PyQt5/index.html> и в директории **pyqt5-doc** урока.

Первое приложение PyQt

Создадим простое приложение с небольшим окном (**листинг 3**):

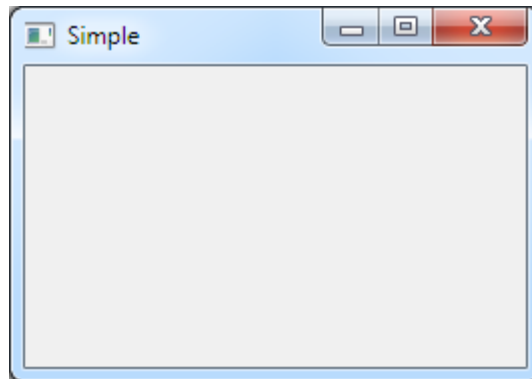
```
import sys
from PyQt5.QtWidgets import QApplication, QWidget      # [1]

if __name__ == '__main__':
    app = QApplication(sys.argv)                       # [2]

    w = QWidget()                                     # [3]
    w.resize(250, 150)                               # [4]
    w.move(300, 300)                                  # [5]
    w.setWindowTitle('Simple')                       # [6]
    w.show()                                           # [7]

    sys.exit(app.exec_())                             # [8]
```


Результат:



Пояснения к коду:

1. Базовые виджеты **Qt** располагаются в модуле **PyQt5.QtWidgets**.
2. Каждое PyQt-приложение должно создавать объект **QApplication** — основную точку входа для Qt-приложения;
3. **QWidget** — базовый класс для всех графических элементов в **PyQt5**. В примере объект создается конструктором без параметров — у него не будет родительского виджета. Виджет без родителя будет главным окном.
4. Метод **resize()** меняет размеры виджета. Ширина — 250px, высота — 150px.
5. Метод **move()** перемещает виджет в позицию на экране с координатами $x=300$, $y=300$.
6. Установка заголовка окна.
7. Метод **show()** отображает виджет на экране — после того, как он создается в памяти.
8. Вход в главный цикл приложения (**mainloop**). Обработка событий будет начинаться с этой точки. Главный цикл получает события от оконной системы и распределяет их виджетам приложения. Главный цикл завершается при вызове метода **exit()** и при уничтожении основного виджета. Функция **sys.exit()** обеспечивает безопасный выход из приложения. Текущее окружение ОС получит результат работы приложения.

Метод **exec_()** содержит в имени нижнее подчеркивание, чтобы можно было отличить его от стандартной функции **exec** из Python.

Файл приложения с графическим интерфейсом можно сохранить с расширением **.py**, но при запуске приложения под ОС Windows вместе с окном приложения будет запускаться окно консоли. Это удобно на этапе разработки/отладки — в консоль можно выводить отладочную информацию. Окно консоли не будет создаваться, если код программы сохранить в файле с расширением **.pyw**. Это удобно для конечного пользователя приложения.

Создание интерфейса

Вручную

Основной (и часто — трудоемкий) способ создать приложение с графическим интерфейсом — составить подробный Python-код с настройкой каждого виджета. Но чтобы освоить возможности библиотеки **PyQt**, важно уметь создавать виджеты и формы вручную.

Рассмотрим на примере, как создать приложение, содержащее панель инструментов ([исходный пример](#)) (**листинг 4**):

```
import sys
from PyQt5.QtWidgets import QMainWindow, QAction, QApplication
from PyQt5.QtGui import QIcon

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

    def initUI(self):
        # Действие (Action) будет использоваться при нажатии на кнопку
        exitAction = QAction(QIcon('exit.png'), 'Exit', self)
        exitAction.setShortcut('Ctrl+Q')
        exitAction.triggered.connect(QApplication.quit)

        # Создание кнопки в панели инструментов
        self.toolbar = self.addToolBar('Exit')
        self.toolbar.addAction(exitAction)

        # Установка заголовка и размеров главного окна
        self.setGeometry(300, 300, 300, 200)
        self.setWindowTitle('Toolbar')
        self.show()

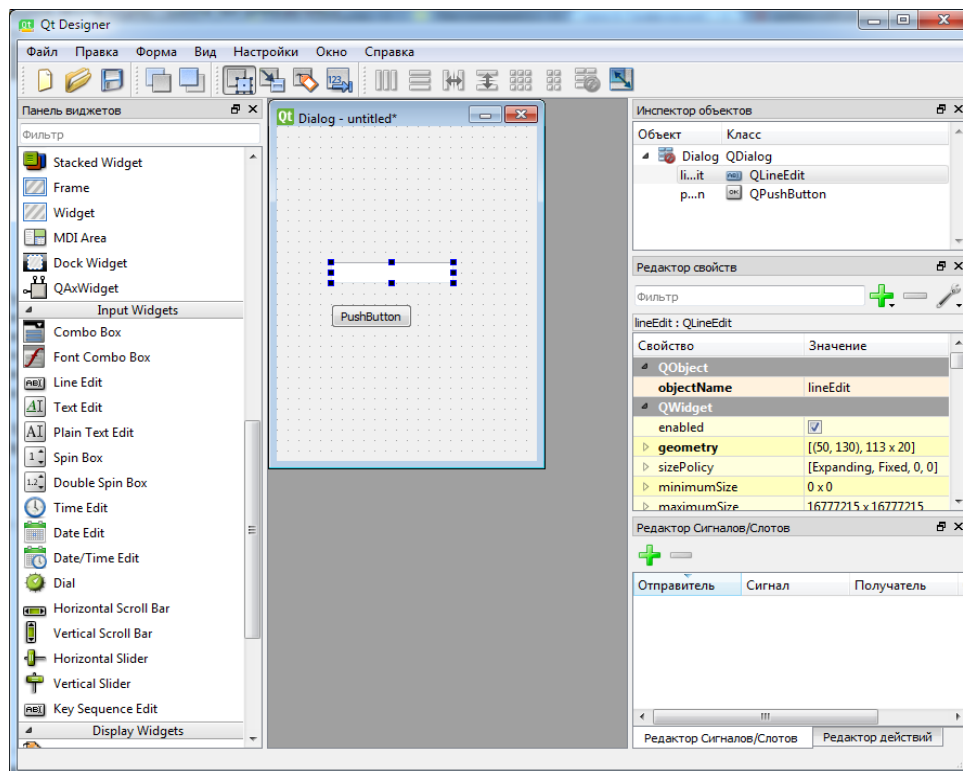
if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())
```

Qt Designer

Qt Designer — инструмент для проектирования и создания графических пользовательских интерфейсов (GUI) из компонентов **Qt**. Позволяет добавлять и настраивать виджеты или диалоги в режиме «что видишь, то и получишь» (**what-you-see-is-what-you-get**, **WYSIWYG**) и проверять их в разных стилях и разрешениях.

Установку **Qt Designer** выполняют так:

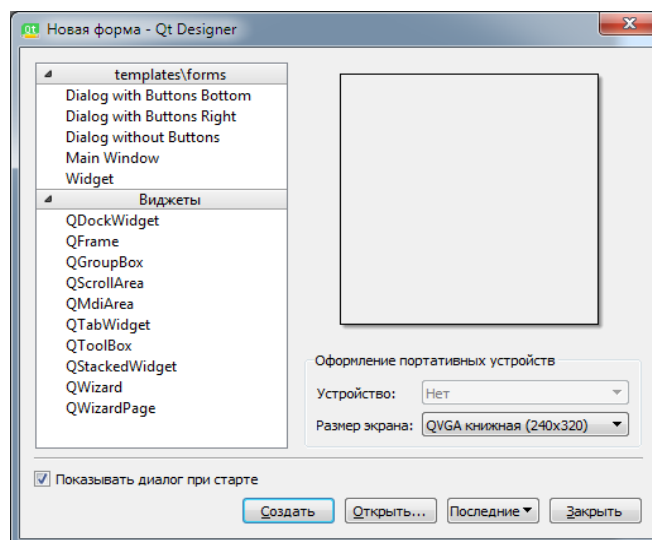
```
pip install PyQt5-tools
```



Краткое руководство

Создание формы

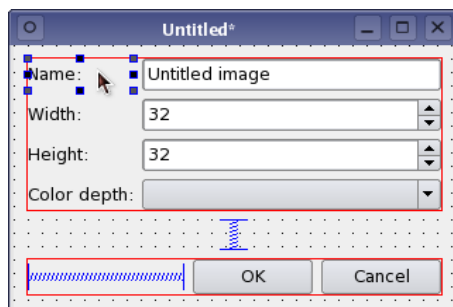
При запуске **Qt Designer** поступает предложение — создать форму на основании шаблона (**templates**) или виджета. На этом этапе можно установить размер формы по шаблонам или выбрать файл, созданный ранее.



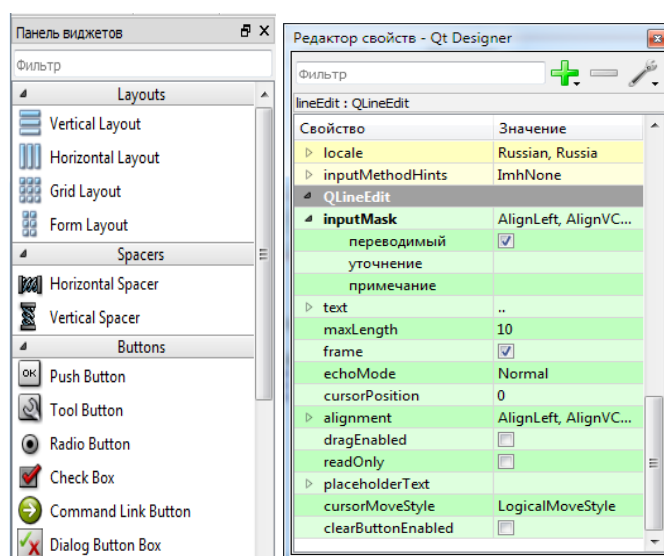
Создать новую форму или открыть сохраненную можно через меню «Файл — Новый...» или «Файл — Открыть...».

У **Qt Designer** 4 режима работы: изменения виджетов, изменения сигналов-слотов, изменения партнеров и очереди переключения.

Изменение виджетов (Widget Editing Mode)



Режим изменения виджетов — основной в **Qt Designer**. Позволяет редактировать внешний вид формы, виджетов, настраивать их взаимное расположение. Оконные формы создаются перетаскиванием необходимого компонента из панели виджетов на разрабатываемую форму.

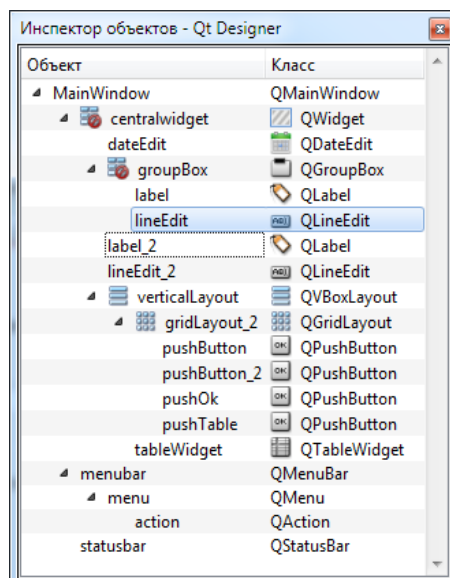


При работе с виджетами на форме доступны стандартные операции копирования, вставки и удаления (клавиши **Ctrl+C**, **Ctrl+V**, **Ctrl+X**, **Del**), а также Drag&Drop-операции — можно перетаскивать виджеты между формами.

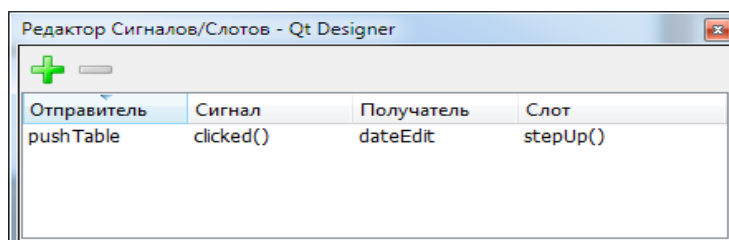
В данном режиме доступны дополнительные плавающие **сервисные окна**: «Редактор свойств», «Инспектор объектов», «Редактор сигналов/слотов».

Редактор свойств (Property Editor) позволяет настраивать каждый виджет или их группу. Также есть фильтр, который позволяет быстрее найти нужное свойство.

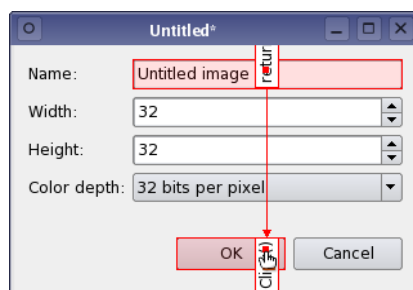
Инспектор объектов (Object Inspector) отображает иерархию расположения виджетов на форме и позволяет выбрать виджет, даже если он не виден на форме. Возможность поиска виджета активируется комбинацией **Ctrl+F**.



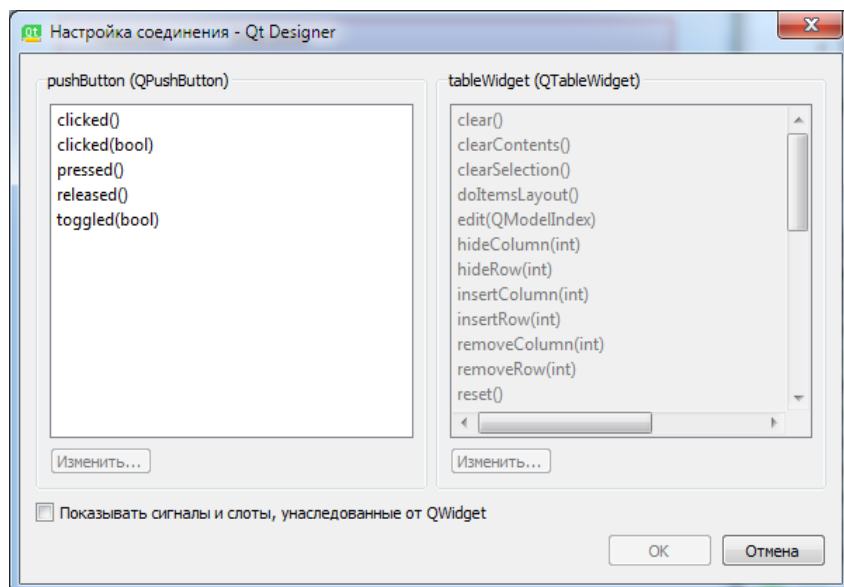
Редактор сигналов/слотов позволяет управлять сигнально-слотовыми соединениями в рамках текущей редактируемой формы (доступны только стандартные сигналы и слоты).



Изменение сигналов-слотов (Signals and Slots Editing Mode):

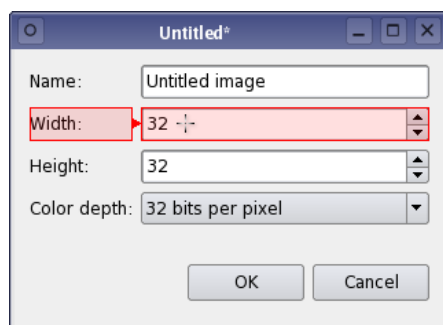


В режиме изменения сигналов-слотов (меню «Правка — Изменение сигналов-слотов» (F4)) можно интуитивно и наглядно связывать сигналы и слоты. Для связывания сигнала одного виджета нужно привести курсор мыши на этот виджет, нажать левую кнопку мыши и привести появившуюся стрелку на виджет, со слотом которого требуется выполнить соединение. Когда второй виджет выбран, выводится окно, позволяющее выбрать комбинацию сигнала и слота:



Созданное соединение «сигнал-слот» отобразится в общем редакторе сигналов/слотов. Редактировать соединение можно мышью, перетаскивая его стрелки. Также можно его удалить.

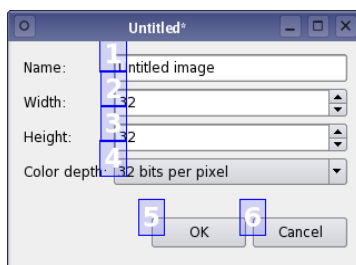
Изменение партнеров (Buddy Editing Mode)



Данный режим предназначен для управления партнерскими (**buddy**) связями **QLabel** — **QWidget**.

Если объект **QLabel** имеет атрибут **text** с приставкой в виде амперсанда (&) (например, **&Surname**), для приложения будет действовать комбинация горячих клавиш **Alt + символ** (Alt + S), передающая управление к данному объекту **QLabel**. Наличие партнера (**buddy**) у данного экземпляра **QLabel** приведет к тому, что нажатие горячих клавиш будет переводить фокус ввода на связанный виджет.

Изменение порядка переключений (Tab Order Editing Mode)



Режим изменения порядка переключений позволяет в наглядном представлении управлять очередью переключения фокуса ввода по клавишам **Tab** или **Shift+Tab**.

Установить порядок переключения можно простым нажатием мышью по виджетам в нужной последовательности. Первый по порядку элемент будет отмечен красным. Далее по клику мыши по виджетам их порядковые номера будут меняться и отмечаться зеленым.

Предпросмотр и сохранение

Чтобы проверить компоновку и настройку виджетов на форме без запуска реального приложения, можно воспользоваться **предпросмотром** (меню «Форма — Предпросмотр...»). Форма отобразится как полноценное окно в рамках **Qt Designer**.

Qt Designer позволяет выполнить **сохранение** формы в файле с расширением **.ui**. Он представляет собой XML-документ. Пример содержимого ui-файла:

```
<ui version="4.0">
<class>MainWindow</class>
<widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
    <rect>
      <x>0</x>
      <y>0</y>
      <width>506</width>
      <height>506</height>
    </rect>
  </property>
  <property name="windowTitle">
    <string>Главное окно</string>
  </property>
  ...
</widget>
<widget class="QStatusBar" name="statusbar"/>
<action name="action">
  <property name="text">
    <string>Открыть файл...</string>
  </property>
</action>
</widget>
<resources/>
<connections/>
</ui>
```

Загрузка ui-файла в программе

Ui-файл содержит текст в формате XML, поэтому подключить его с помощью команды **import** невозможно. Чтобы использовать ui-файл в программе, следует применить модуль **uic** библиотеки **PyQt**:

```
from PyQt5 import uic
```

Загрузка ui-файла выполняется функцией **loadUi()** модуля **uic**. Формат функции:

```
loadUi(<ui-файл>[, <экземпляр_класса>])
```

Если второй параметр не указан, функция возвращает ссылку на объект формы. С помощью этой ссылки можно получить доступ к виджетам формы и, например, назначить обработчики сигналов. Имена виджетов формы задаются в **Qt Designer** в свойстве **objectName**:

```
# Использование функции loadUi()
from PyQt5 import QtWidgets, uic
import sys

app = QtWidgets.QApplication(sys.argv)
window = uic.loadUi('TestForm.ui')
window.btnQuit.clicked.connect(app.quit)
window.show()
sys.exit(app.exec_())
```

Если во втором параметре указать ссылку на экземпляр класса, все компоненты формы будут доступны через ссылку **self**:

```
# Использование функции loadUi()
from PyQt5 import QtWidgets, uic
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        uic.loadUi('TestForm.ui', self)
        self.btnQuit.clicked.connect(QtWidgets.qApp.quit)

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

- **Важное замечание!** При создании пользовательского класса в качестве базового нужно указывать **тот же класс**, на основании которого создавалась форма в **Designer**'е (**QMainWindow/QWidget/QDialog**). Иначе приложение не запустится.

Загрузить ui-файл также можно при помощи функции **loadUiType()**, которая возвращает кортеж из двух элементов: ссылки на класс формы и ссылки на базовый класс. Так появляется возможность создавать множество экземпляров класса. После добавления экземпляра класса формы необходимо вызвать метод **setupUi()** и передать ему ссылку **self**:

```
# Использование функции loadUiType()

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent):
        Form, Base = uic.loadUiType('TestForm.ui', self)
        self.ui = Form()
        self.ui.setupUi(self)
        self.ui.btnQuit.clicked.connect(QtWidgets.qApp.quit)
```


Можно загрузить `ui`-файл вне класса. После этого нужно указать класс формы вторым родительским классом в списке наследования (создаваемый класс наследует все атрибуты класса формы):

```
# Использование функции loadUiType()
Form, Base = uic.loadUiType('TestForm.ui', self)

class MyWindow(QtWidgets.QWidget, Form):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.btnQuit.clicked.connect(QtWidgets.qApp.quit)
```

Преобразование `ui`-файла в `py`-файл

Вместо подключения `ui`-файла можно сгенерировать на его основе Python-код. Для этого служит утилита **pyuic5**, входящая в библиотеку **PyQt5**. Она устанавливается в поддиректорию **Scripts** — например, **C:\Python36\Scripts**.

Запуск утилиты:

```
pyuic5 ui_file.ui -o py_form.py
```

- **py_form.py** — это выходной файл с Python-кодом;
- **ui_form.ui** — входной файл с описанием виджетов, созданный в **Qt Designer**.

Созданный файл **py_form.py** подключается к коду приложения инструкцией **import**. Он содержит класс **Ui_TestForm** — его имя задается на этапе работы с **Qt Designer** — с двумя методами: **setupUi()** и **retranslateUi()**.

При использовании процедурного стиля программирования следует создать экземпляр класса формы, а затем вызвать метод **setupUi()** и передать ему ссылку на экземпляр окна:

```
import sys
from PyQt5 import QtWidgets
import py_form

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
ui = py_form.Ui_TestForm()
ui.setupUi(window)
ui.btnQuit.clicked.connect(QtWidgets.qApp.quit)
window.show()
sys.exit(app.exec_())
```

При использовании ООП-подхода можно выбрать один из способов:

- вариант 1 — создать экземпляр класса формы, а затем вызвать метод **setupUi()** и передать ему ссылку **self**;
- вариант 2 — применить множественное наследование.

```

import sys
from PyQt5 import QtWidgets
import py_form

# Вариант 1. Объект формы - атрибут класса главного окна
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.ui = py_form.Ui_TestForm()
        self.ui.setupUi(self)
        self.ui.btnQuit.clicked.connect(QtWidgets.qApp.quit)

# Вариант 2. Главное окно дополнительно наследуется от класса формы
class MyWindow_2(QtWidgets.QWidget, py_form.Ui_TestForm):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.ui.setupUi(self)
        self.ui.btnQuit.clicked.connect(QtWidgets.qApp.quit)

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())

```

Модули и классы PyQt5

С точки зрения языка Python **PyQt5** является пакетом, объединяющим модули библиотеки **PyQt5**.

Полный список модулей пакета ([PyQt5. Python Module Index](#)): Enginio, QAxContainer (Windows), Qt, QtBluetooth, QtCore, QtDBus (UNIX), QtDesigner, QtGui, QtHelp, QtLocation, QtMacExtras (OS X, iOS), QtMultimedia, QtMultimediaWidgets, QtNetwork, QtNfc, QtOpenGL, QtPositioning, QtPrintSupport, QtQml, QtQuick, QtQuickWidgets, QtSensors, QtSerialPort, QtSql, QtSvg, QtTest, QtWebChannel, QtWebEngine, QtWebEngineCore, QtWebEngineWidgets, QtWebKit, QtWebKitWidgets, QtWebSockets, QtWidgets, QtWinExtras (Windows), QtX11Extras (X11), QtXml, QtXmlPatterns, uic

Рассмотрим некоторые из них.

PyQt5.Qt

Включает классы всех модулей сразу. Это может быть удобно, чтобы не задумываться, в каком модуле находится тот или иной класс. Но при таком подходе все содержимое библиотеки **PyQt** будет загружаться в оперативную память, сильно увеличивая размер приложения в ОЗУ.

PyQt5.QtCore

Содержит классы, не связанные с реализацией графического интерфейса. От этого модуля зависят все остальные.

Список всех классов и переменных модуля: PYQT_CONFIGURATION, PYQT_VERSION, PYQT_VERSION_STR, QAbstractAnimation, QAbstractEventDispatcher, QAbstractItemModel,

QAbstractListModel, QAbstractNativeEventFilter, QAbstractProxyModel, QAbstractState, QAbstractTableModel, QAbstractTransition, QAnimationGroup, QBasicTimer, QBitArray, QBuffer, QByteArray, QByteArrayMatcher, QChildEvent, QCollator, QCollatorSortKey, QCommandLineOption, QCommandLineParser, QCoreApplication, QCryptographicHash, QDataStream, QDate, QDateTime, QDeadlineTimer, QDir, QDirIterator, QDynamicPropertyChangeEvent, QEasingCurve, QElapsedTimer, QEvent, QEventLoop, QEventLoopLocker, QEventTransition, QFile, QFileDevice, QFileInfo, QFileSelector, QFileSystemWatcher, QFinalState, QGenericArgument, QGenericReturnArgument, QHistoryState, QIODevice, QIdentityProxyModel, QItemSelection, QItemSelectionModel, QItemSelectionRange, QJsonDocument, QJsonParseError, QJsonValue, QLibrary, QLibraryInfo, QLine, QLineF, QLocale, QLockFile, QMargins, QMarginsF, QMessageAuthenticationCode, QMessageLogContext, QMessageLogger, QMetaClassInfo, QMetaEnum, QMetaMethod, QMetaObject, QMetaProperty, QMetaType, QMimeData, QMimeDatabase, QMimeType, QModelIndex, QMutex, QMutexLocker, QObject, QObjectCleanupHandler, QParallelAnimationGroup, QPauseAnimation, QPersistentModelIndex, QPluginLoader, QPoint, QPointF, QProcess, QProcessEnvironment, QPropertyAnimation, QReadLocker, QReadWriteLock, QRect, QRectF, QRegExp, QRegularExpression, QRegularExpressionMatch, QRegularExpressionMatchIterator, QResource, QRunnable, QSaveFile, QSemaphore, QSequentialAnimationGroup, QSettings, QSharedMemory, QSignalBlocker, QSignalMapper, QSignalTransition, QSize, QSizeF, QSocketNotifier, QSortFilterProxyModel, QStandardPaths, QState, QStateMachine, QStorageInfo, QStringListModel, QSysInfo, QSystemSemaphore, QT_TRANSLATE_NOOP, QT_TR_NOOP, QT_TR_NOOP_UTF8, QT_VERSION, QT_VERSION_STR, QTemporaryDir, QTemporaryFile, QTextBoundaryFinder, QTextCodec, QTextDecoder, QTextEncoder, QTextStream, QTextStreamManipulator, QThread, QThreadPool, QTime, QTimeLine, QTimeZone, QTimer, QTimerEvent, QTranslator, QUrl, QUrlQuery, QUuid, QVariant, QVariantAnimation, QVersionNumber, QWaitCondition, QWinEventNotifier, QWriteLocker, QXmlStreamAttribute, QXmlStreamAttributes, QXmlStreamEntityDeclaration, QXmlStreamEntityResolver, QXmlStreamNamespaceDeclaration, QXmlStreamNotationDeclaration, QXmlStreamReader, QXmlStreamWriter, Q_ARG, Q_CLASSINFO, Q_ENUMS, Q_FLAGS, Q_RETURN_ARG, Qt, QtCriticalMsg, QtDebugMsg, QtFatalMsg, QtInfoMsg, QtMsgType, QtSystemMsg, QtWarningMsg, bom, center, dec, endl, fixed, flush, forcepoint, forcesign, hex_, left, lowercasebase, lowercasedigits, noforcepoint, noforcesign, noshowbase, oct_, pyqtBoundSignal, pyqtPickleProtocol, pyqtProperty, pyqtRemoveInputHook, pyqtRestoreInputHook, pyqtSetPickleProtocol, pyqtSignal, pyqtSlot, qAbs, qAddPostRoutine, qAddPreRoutine, qChecksum, qCompress, qCritical, qDebug, qErrnoWarning, qFatal, qFloatDistance, qFormatLogMessage, qFuzzyCompare, qInf, qInstallMessageHandler, qIsFinite, qIsInf, qIsNaN, qIsNull, qQNaN, qRegisterResourceData, qRemovePostRoutine, qRound, qRound64, qSNaN, qSetFieldWidth, qSetMessagePattern, qSetPadChar, qSetRealNumberPrecision, qSharedBuild, qUncompress, qUnregisterResourceData, qVersion, qWarning, grand, qrand, reset, right, scientific, showbase, uppercasebase, uppercasedigits, ws.

PyQt5.QtGui

В данном модуле содержатся классы, реализующие низкоуровневую работу с оконными элементами, обработку сигналов, вывод двумерной графики и текста (и другие).

Список всех классов и переменных модуля: QAbstractOpenGLFunctions, QAbstractTextDocumentLayout, QActionEvent, QBackingStore, QBitmap, QBrush, QClipboard, QCloseEvent, QColor, QConicalGradient, QContextMenuEvent, QCursor, QDesktopServices, QDoubleValidator, QDrag, QDragEnterEvent, QDragLeaveEvent, QDragMoveEvent, QDropEvent, QEnterEvent, QExposeEvent, QFileOpenEvent, QFocusEvent, QFont, QFontDatabase, QFontInfo, QFontMetrics, QFontMetricsF, QGlyphRun, QGradient, QGuiApplication, QHelpEvent, QHideEvent, QHoverEvent, QIcon, QIconDragEvent, QIconEngine, QImage, QImageIOHandler, QImageReader, QImageWriter, QInputEvent, QInputMethod, QInputMethodEvent, QInputMethodQueryEvent, QIntValidator, QKeyEvent, QKeySequence, QLinearGradient, QMatrix2x2, QMatrix2x3, QMatrix2x4, QMatrix3x2, QMatrix3x3, QMatrix3x4, QMatrix4x2, QMatrix4x3, QMatrix4x4, QMouseEvent, QMoveEvent, QMovie, QNativeGestureEvent, QOffscreenSurface, QOpenGLBuffer, QOpenGLContext, QOpenGLContextGroup, QOpenGLDebugLogger, QOpenGLDebugMessage,

QOpenGLFramebufferObject, QOpenGLFramebufferObjectFormat, QOpenGLPaintDevice, QOpenGLPixelTransferOptions, QOpenGLShader, QOpenGLShaderProgram, QOpenGLTexture, QOpenGLTextureBlitter, QOpenGLTimeMonitor, QOpenGLTimerQuery, QOpenGLVersionProfile, QOpenGLVertexArrayObject, QOpenGLWindow, QPageLayout, QPageSize, QPagedPaintDevice, QPaintDevice, QPaintDeviceWindow, QPaintEngine, QPaintEngineState, QPaintEvent, QPainter, QPainterPath, QPainterPathStroker, QPalette, QPdfWriter, QPen, QPicture, QPictureIO, QPixelFormat, QPixmap, QPixmapCache, QPlatformSurfaceEvent, QPointingDeviceUniqueId, QPolygon, QPolygonF, QQuaternion, QRadialGradient, QRasterWindow, QRawFont, QRegExpValidator, QRegion, QRegularExpressionValidator, QResizeEvent, QRgba64, QScreen, QScrollEvent, QScrollPrepareEvent, QSessionManager, QShortcutEvent, QShowEvent, QStandardItem, QStandardItemModel, QStaticText, QStatusTipEvent, QStyleHints, QSurface, QSurfaceFormat, QSyntaxHighlighter, QTabletEvent, QTextBlock, QTextBlockFormat, QTextBlockGroup, QTextBlockUserData, QTextCharFormat, QTextCursor, QTextDocument, QTextDocumentFragment, QTextDocumentWriter, QTextFormat, QTextFragment, QTextFrame, QTextFrameFormat, QTextImageFormat, QTextInlineObject, QTextItem, QTextLayout, QTextLength, QTextLine, QTextList, QTextListFormat, QTextObject, QTextObjectInterface, QTextOption, QTextTable, QTextTableCell, QTextTableCellFormat, QTextTableFormat, QTouchEvent, QTransform, QValidator, QVector2D, QVector3D, QVector4D, QWhatsThisClickedEvent, QWheelEvent, QWindow, QWindowStateChangeEvent, qAlpha, qBlue, qFuzzyCompare, qGray, qGreen, qIsGray, qPixelFormatAlpha, qPixelFormatCmyk, qPixelFormatGrayscale, qPixelFormatHsl, qPixelFormatHsv, qPixelFormatRgba, qPixelFormatYuv, qPremultiply, qRed, qRgb, qRgba, qRgba64, qUnpremultiply, qt_set_sequence_auto_mnemonic.

PyQt5.QtMultimedia

Включает низкоуровневые классы для работы с мультимедиа.

Полный список классов модуля: QAbstractVideoBuffer, QAbstractVideoFilter, QAbstractVideoSurface, QAudio, QAudioBuffer, QAudioDecoder, QAudioDeviceInfo, QAudioEncoderSettings, QAudioFormat, QAudioInput, QAudioOutput, QAudioProbe, QAudioRecorder, QCamera, QCameraExposure, QCameraFocus, QCameraFocusZone, QCameraImageCapture, QCameraImageProcessing, QCameraInfo, QCameraViewfinderSettings, QImageEncoderSettings, QMediaBindableInterface, QMediaContent, QMediaControl, QMediaMetaData, QMediaObject, QMediaPlayer, QMediaPlaylist, QMediaRecorder, QMediaResource, QMediaService, QMediaTimeInterval, QMediaTimeRange, QMultimedia, QRadioData, QRadioTuner, QSound, QSoundEffect, QVideoEncoderSettings, QVideoFilterRunnable, QVideoFrame, QVideoProbe, QVideoSurfaceFormat.

PyQt5.QtMultimediaWidgets

Реализует высокоуровневые компоненты графического интерфейса с мультимедиа и компоненты, использующие модуль **QtMultimedia**.

Полный список классов модуля: QCameraViewfinder, QGraphicsVideoItem, QVideoWidget.

PyQt5.QtNetwork

Содержит классы для работы с сетью.

Полный список классов модуля: QAbstractNetworkCache, QAbstractSocket, QAuthenticator, QDnsDomainNameRecord, QDnsHostAddressRecord, QDnsLookup, QDnsMailExchangeRecord, QDnsServiceRecord, QDnsTextRecord, QHostAddress, QHostInfo, QHttpMultiPart, QHttpPart, QLocalServer, QLocalSocket, QNetworkAccessManager, QNetworkAddressEntry, QNetworkCacheMetaData, QNetworkConfiguration, QNetworkConfigurationManager, QNetworkCookie, QNetworkCookieJar, QNetworkDatagram, QNetworkDiskCache, QNetworkInterface, QNetworkProxy,

QNetworkProxyFactory, QNetworkProxyQuery, QNetworkReply, QNetworkRequest, QNetworkSession, QSsl, QSslCertificate, QSslCertificateExtension, QSslCipher, QSslConfiguration, QSslDiffieHellmanParameters, QSslEllipticCurve, QSslError, QSslKey, QSslPreSharedKeyAuthenticator, QSslSocket, QTcpServer, QTcpSocket, QUdpSocket.

PyQt5.QtOpenGL

Обеспечивает поддержку **OpenGL**.

PyQt5.QtSql

Обеспечивает поддержку работы с базами данных, а также реализацию **SQLite**.

Полный список классов модуля: QSql, QSqlDatabase, QSqlDriver, QSqlDriverCreatorBase, QSqlError, QSqlField, QSqlIndex, QSqlQuery, QSqlQueryModel, QSqlRecord, QSqlRelation, QSqlRelationalDelegate, QSqlRelationalTableModel, QSqlResult, QSqlTableModel.

PyQt5.QtSerialPort

Реализует классы для доступа к системным серийным портам.

Классы модуля: QSerialPort, QSerialPortInfo.

PyQt5.QtTest

Обеспечивает частичный функционал для unit-тестирования PyQt5-приложений на основе **Qt**. Подразумевается, что в Python тестирование лучше осуществлять через стандартный модуль **unittest**.

Классы модуля:

- **QSignalSpy** — обеспечивает простую интроспекцию сигналов/слотов.
- **QTest** — обеспечивает доступ к **Qt Test**.

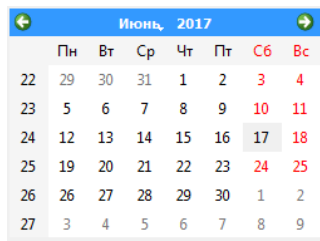
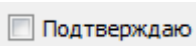
PyQt5.QtWidgets

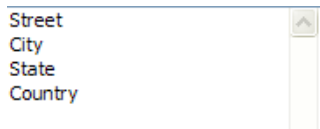
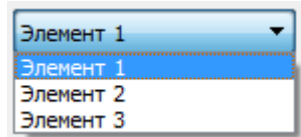

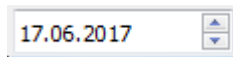
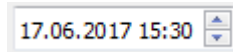
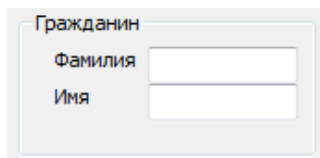
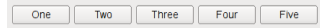
Содержит классы, реализующие компоненты графического интерфейса: окна, диалоги, надписи, кнопки, поля ввода и другие.

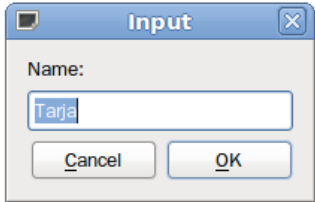
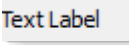
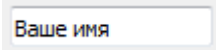
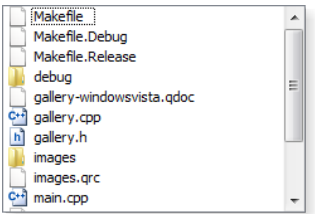
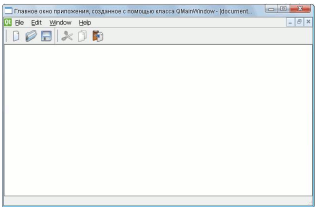
Полный список классов и переменных модуля: QAbstractButton, QAbstractGraphicsShapeltem, QAbstractItemDelegate, QAbstractItemView, QAbstractScrollArea, QAbstractSlider, QAbstractSpinBox, QAction, QActionGroup, QApplication, QBoxLayout, QButtonGroup, QCalendarWidget, QCheckBox, QColorDialog, QColumnView, QComboBox, QCommandLinkButton, QCommonStyle, QCompleter, QDataWidgetMapper, QDateEdit, QDateTimeEdit, QDesktopWidget, QDial, QDialog, QDialogButtonBox, QDirModel, QDockWidget, QDoubleSpinBox, QErrorMessage, QFileDialog, QFileIconProvider, QFileSystemModel, QFocusFrame, QFontComboBox, QFontDialog, QFormLayout, QFrame, QGesture, QGestureEvent, QGestureRecognizer, QGraphicsAnchor, QGraphicsAnchorLayout, QGraphicsBlurEffect, QGraphicsColorizeEffect, QGraphicsDropShadowEffect, QGraphicsEffect, QGraphicsEllipseItem, QGraphicsGridLayout, QGraphicsItem, QGraphicsItemGroup, QGraphicsLayout, QGraphicsLayoutItem, QGraphicsLineItem, QGraphicsLinearLayout, QGraphicsObject, QGraphicsOpacityEffect, QGraphicsPathItem, QGraphicsPixmapItem, QGraphicsPolygonItem, QGraphicsProxyWidget, QGraphicsRectItem, QGraphicsRotation, QGraphicsScale, QGraphicsScene,

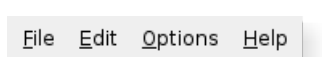
QGraphicsSceneContextMenuEvent, QGraphicsSceneDragDropEvent, QGraphicsSceneEvent,
 QGraphicsSceneHelpEvent, QGraphicsSceneHoverEvent, QGraphicsSceneMouseEvent,
 QGraphicsSceneMoveEvent, QGraphicsSceneResizeEvent, QGraphicsSceneWheelEvent,
 QGraphicsSimpleTextItem, QGraphicsTextItem, QGraphicsTransform, QGraphicsView, QGraphicsWidget,
 QGridLayout, QGroupBox, QHBoxLayout, QHeaderView, QInputDialog, QItemDelegate,
 QItemEditorCreatorBase, QItemEditorFactory, QKeyEventTransition, QKeySequenceEdit, QLCDNumber,
 QLabel, QLayout, QLayoutItem, QLineEdit, QListView, QListWidget, QListWidgetItem, QMainWindow,
 QMdiArea, QMdiSubWindow, QMenu, QMenuBar, QMessageBox, QMouseEventTransition,
 QOpenGLWidget, QPanGesture, QPinchGesture, QPlainTextDocumentLayout, QPlainTextEdit,
 QProgressBar, QProgressDialog, QProxyStyle, QPushButton, QRadioButton, QRubberBand, QScrollArea,
 QScrollBar, QScroller, QScrollerProperties, QShortcut, QSizeGrip, QSizePolicy, QSlider, QSpacerItem,
 QSpinBox, QSplashScreen, QSplitter, QSplitterHandle, QStackedLayout, QStackedWidget, QStatusBar,
 QStyle, QStyleFactory, QStyleHintReturn, QStyleHintReturnMask, QStyleHintReturnVariant, QStyleOption,
 QStyleOptionButton, QStyleOptionComboBox, QStyleOptionComplex, QStyleOptionDockWidget,
 QStyleOptionFocusRect, QStyleOptionFrame, QStyleOptionGraphicsItem, QStyleOptionGroupBox,
 QStyleOptionHeader, QStyleOptionMenuItem, QStyleOptionProgressBar, QStyleOptionRubberBand,
 QStyleOptionSizeGrip, QStyleOptionSlider, QStyleOptionSpinBox, QStyleOptionTab,
 QStyleOptionTabBarBase, QStyleOptionTabWidgetFrame, QStyleOptionTitleBar, QStyleOptionToolBar,
 QStyleOptionToolBox, QStyleOptionToolButton, QStyleOptionViewItem, QStylePainter,
 QStyledItemDelegate, QSwipeGesture, QSystemTrayIcon, QTabBar, QTabWidget, QTableView,
 QTableWidget, QTableWidgetItem, QTableWidgetSelectionRange, QTapAndHoldGesture, QTapGesture,
 QTextBrowser, QTextEdit, QTimeEdit, QToolBar, QToolBox, QToolButton, QToolTip, QTreeView,
 QTreeWidget, QTreeWidgetItem, QTreeWidgetItemIterator, QUndoCommand, QUndoGroup, QUndoStack,
 QUndoView, QVBoxLayout, QWIDGETSIZE_MAX, QWhatsThis, QWidget, QWidgetAction, QWidgetItem,
 QWizard, QWizardPage, QApplication, QDrawBorderPixmap, QDrawPlainRect, QDrawShadeLine,
 QDrawShadePanel, QDrawShadeRect, QDrawWinButton, QDrawWinPanel.

Некоторые полезные классы представлены в таблице:

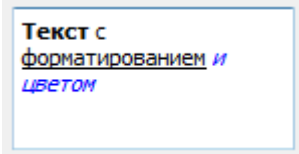
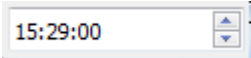

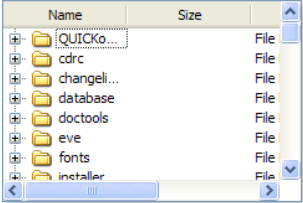
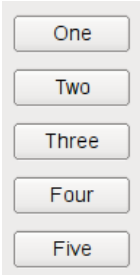
#	Класс	Описание	Графический вид
1	QApplication	Класс для запуска приложения	
2	QBoxLayout	Позволяет размещать дочерние виджеты вертикально или горизонтально	
3	QButtonGroup	Класс для группировки кнопок	
4	QCalendarWidget	Виджет календаря	
5	QCheckBox	Виджет галочки	

6	QColumnView	Виджет отображения данных шаблона «Модель/Представление» в виде столбца	
7	QComboBox	Виджет поля выбора	
8	QCommandLinkButton	Виджет командной кнопки в стиле Windows Vista соединяет возможности QPushButton и QRadioButton	
9	QDateEdit	Поле ввода даты	
10	QDateTimeEdit	Поле ввода даты и времени	
11	QDesktopWidget	Класс для доступа к экрану на системе с несколькими мониторами	
12	QFileDialog	Диалоговое окно выбора файлов	
13	QFormLayout	Класс для создания форм ввода из двух колонок: заголовки и поля ввода	
14	QFrame	Базовый класс для виджетов, которые могут содержать фреймы	
15	QGridLayout	Класс для размещения виджетов по сетке	
16	QGroupBox	Виджет-контейнер с заголовком для объединения других виджетов	
17	QHBoxLayout	Класс-контейнер для размещения дочерних виджетов горизонтально	

18	QInputDialog	Диалоговое окно ввода с полем ввода и кнопками ОК и Cancel	
19	QLabel	Виджет метки/заголовка для отображения текста/рисунка	
20	QLayout	Базовый класс-контейнер для размещения дочерних виджетов	
21	QLayoutItem	Класс абстрактного элемента, которым управляет QLayout	
22	QLineEdit	Виджет однострочного поля ввода	
23	QListView	Виджет списка с пиктограммами для шаблона «Модель/Представление»	
24	QListWidget	Виджет для списка простых элементов	
25	QListWidgetItem	Элемент списка QListWidget	
26	QMainWindow	Класс, который реализует главное окно, содержащее типовые виджеты, необходимые большинству приложений (меню, панель инструментов, рабочая область, строка состояния). Внешний вид окна уже подготовлен, и его виджеты не нуждаются в дополнительном размещении — они уже находятся в нужных местах	

27	QMdiArea	Виджет области для работы с дочерними окнами/документами (MDI)	
28	QMdiSubWindow	Виджет дочернего окна для MDI-области	
29	QMenu	Виджет меню для использования в главных и контекстных меню	
30	QMenuBar	Виджет строки меню (главного меню)	
31	QMessageBox	Диалоговое окно сообщения	
32	QPlainTextEdit	Виджет редактирования/отображения простого текста	
33	QProgressBar	Виджет горизонтальной/вертикальной строки состояния (ProgressBar)	
34	QProgressDialog	Диалоговое окно прогресса	
35	QPushButton	Обычная кнопка	
36	QRadioButton	Кнопка выбора	

37	QScrollArea	Виджет области прокрутки	
38	QScrollBar	Виджет пролистывания области прокрутки	
39	QShortcut	Класс для создания горячих клавиш/комбинаций	
40	QSizePolicy	Атрибут класса размещения (QLayout), содержащий константы для горизонтального/вертикального размещения	
41	QSlider	Виджет бегунка/слайдера	
42	QSpinBox	Виджет поля ввода целого числа со стрелками	
43	QSplashScreen	Виджет окна-предшественника — для использования при загрузке приложения	
44	QSplitter	Класс-разделитель между виджетами	
45	QStatusBar	Виджет строки состояния	
46	QStyle	Абстрактный класс для настроек стиля графического интерфейса	
47	QTabBar	Виджет строки закладок	
48	QTabWidget	Класс, реализующий страницы-закладки	
49	QTableView	Виджет табличного отображения для шаблона «Модель/Представление»	
50	QTableWidget	Виджет табличного отображения элементов	

51	QTableWidgetItem	Элемент табличного отображения (ячейка)	
52	QTableWidgetSelectionRange	Выбранные ячейки в таблице QTableWidget	
53	QTextBrowser	Виджет для реализации браузера HTML-страниц	
54	QTextEdit	Виджет для отображения/редактирования простого текста и текста с форматированием	
55	QTimeEdit	Виджет поля ввода времени	
56	QToolBar	Класс перемещаемой панели инструментов	
57	QToolButton	Виджет кнопки быстрого доступа для панелей инструментов QToolBar	
58	QToolTip	Класс для реализации подсказок для виджетов	
59	QTreeView	Виджет древовидного отображения для шаблона «Модель/Представление»	
60	QTreeWidget	Виджет древовидного отображения для заданных элементов	
61	QTreeWidgetItem	Элемент отображения QTreeWidget	
62	QVBoxLayout	Класс-контейнер для размещения дочерних виджетов вертикально	

63	QWidget	Базовый класс для всех объектов пользовательского интерфейса	
64	qApp	Глобальная ссылка на текущее приложение	

PyQt5.QtWinExtras

Включает поддержку специфических возможностей Microsoft Windows.

Полный список классов модуля: `QWinJumpList`, `QWinJumpListCategory`, `QWinJumpListItem`, `QWinTaskbarButton`, `QWinTaskbarProgress`, `QWinThumbnailToolBar`, `QWinThumbnailToolButton`, `QtWin`.

PyQt5.QtXml

Модули **PyQt5.QtXml** и **PyQt5.QtXmlPatterns** предназначены для обработки XML.

Полный список классов модуля: `QDomAttr`, `QDomCDATASection`, `QDomCharacterData`, `QDomComment`, `QDomDocument`, `QDomDocumentFragment`, `QDomDocumentType`, `QDomElement`, `QDomEntity`, `QDomEntityReference`, `QDomImplementation`, `QDomNamedNodeMap`, `QDomNode`, `QDomNodeList`, `QDomNotation`, `QDomProcessingInstruction`, `QDomText`, `QXmlAttributes`, `QXmlContentHandler`, `QXmlDTDHandler`, `QXmlDeclHandler`, `QXmlDefaultHandler`, `QXmlEntityResolver`, `QXmlErrorHandler`, `QXmlInputSource`, `QXmlLexicalHandler`, `QXmlLocator`, `QXmlNamespaceSupport`, `QXmlParseException`, `QXmlReader`, `QXmlSimpleReader`.

PyQt5.uic

Содержит классы для управления .ui-файлами, созданными в Qt Designer и реализующими графический интерфейс приложения (загрузка, рендеринг, преобразование в Python-код).

Полный список классов, функций и переменных модуля: `Compiler`, `compileUi`, `compileUiDir`, `compiler`, `exceptions`, `icon_cache`, `indenter`, `loadUi`, `loadUiType`, `objcreator`, `port_v3`, `properties`, `uiparser`, `widgetPluginPath`.

Отличия PyQt5 и PyQt4

Библиотеки **PyQt5** и **PyQt4** несовместимы — следует иметь это в виду, портируя PyQt4-приложения или используя примеры для **PyQt4**. Полный список отличий приведен на странице [Differences Between PyQt4 and PyQt5](#).

Отметим некоторые:

- Не поддерживаются версии Python младше 2.6;
- Не поддерживаются следующие вызовы:
 - **QObject.connect();**
 - **QObject.emit();**
 - **SIGNAL();**
 - **SLOT().**

- Метод **disconnect()** не принимает аргументов и отключает все подключения к **QObject**;
- Библиотека **Qt** реализует сигналы с параметрами и без них. **PyQt5** реализует сигналы, где все параметры должны быть указаны;
- **PyQt5** поддерживает определение свойств (**property**), сигналов и слотов в классах, не унаследованных от **QObject** (mixin-классы);
- **QtGui**-модуль в **PyQt5** разделен на модули **QtGui**, **QtPrintSupport** и **QtWidgets**;
- В **PyQt5** класс **QSet** реализован через **set** (множество);
- Утилита **pyuic5** не поддерживает флаг **--pyqt3-wrapper**.

Сигналы и обработчики

При взаимодействии пользователя с окном происходят **события** — извещения о том, что пользователь выполнил действие или в самой системе возникло условие. В ответ система генерирует **сигналы**, которые можно рассматривать как представления системных событий внутри библиотеки **PyQt**.

Чтобы обработать сигнал, необходимо сопоставить ему функцию или метод класса, который вызовется при наступлении события и станет его **обработчиком**.

Каждому сигналу в определенном классе соответствует атрибут этого класса. Сигнал **clicked** — свойство **clicked**. Каждый такой атрибут хранит экземпляр объекта-сигнала.

Чтобы назначить сигналу обработчик, следует использовать метод **connect()**, унаследованный от класса **QObject**. Форматы вызова этого метода:

```
<Qt_компонент>.<Сигнал>.connect (<Обработчик>[, <Тип_соединения>])

<Qt_компонент>.<Сигнал>.[<Тип>].connect (<Обработчик>[, <Тип_соединения>])
```

В качестве обработчика можно указать:

- ссылку на пользовательскую функцию;
- ссылку на метод класса;
- ссылку на экземпляр класса, в котором определен метод **__call__()**;
- lambda-функцию;
- ссылку на слот класса.

Например, такой код назначает функцию **on_clicked_button** в качестве обработчика объекта-сигнала **clicked** кнопки **button**:

```
button.clicked.connect(on_clicked_button)
```

Одному сигналу можно назначить произвольное количество обработчиков (**листинг 5**):

```

import sys
from PyQt5 import QtWidgets

def on_clicked():
    print("Кнопка нажата. Функция on_clicked")

class MyClass():
    def __init__(self, x=0):
        self.x = x
    def __call__(self):
        print("Кнопка нажата. Метод MyClass.__call__()")
        print("x = ", self.x)
    def on_clicked(self):
        print("Кнопка нажата. Метод MyClass.on_clicked()")

obj = MyClass()
app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Нажми меня")

# В качестве обработчика назначается функция
button.clicked.connect(on_clicked)

# В качестве обработчика назначается метод объекта
button.clicked.connect(obj.on_clicked)

# В качестве обработчика назначается экземпляр класса
button.clicked.connect(MyClass(10))

# В качестве обработчика назначается lambda-функция
button.clicked.connect(lambda: MyClass(5)())

button.show()
sys.exit(app.exec_())

```

Результат выполнения программы в консоли при нажатии на кнопку:

```

Кнопка нажата. Функция on_clicked
Кнопка нажата. Метод MyClass.on_clicked()
Кнопка нажата. Метод MyClass.__call__()
x = 10
Кнопка нажата. Метод MyClass.__call__()
x = 5

```

Классы **PyQt** поддерживают специальные методы — слоты, — предназначенные служить обработчиками сигналов. Например, класс **QApplication** поддерживает слот **quit()**, завершающий текущее приложение.

Любой пользовательский метод можно сделать слотом. Для этого необходимо перед его описанием добавить декоратор **pyqtSlot()**. Формат декоратора:

```
@QtCore.pyqtSlot(*<Типы_данных>, name=None, result=None)
```

В параметре **<Типы_данных>** через запятую перечисляются типы данных параметров, принимаемых слотом — например, **bool** или **int**. Если метод не принимает параметров, типы данных не указываются. Именованный параметр **name** — строка с названием слота, которая будет использоваться вместо названия метода (если параметр **name** не указан, устанавливается по названию метода). Именованный параметр **result** указывает типы данных, возвращаемых методом (если не указан, метод ничего не возвращает).

PyQt не требует обязательно превращать в слот метод, который будет использоваться как обработчик сигнала. Но вызов слота будет происходить быстрее, чем вызов обычного метода.

Итоги

В этом уроке мы продолжили знакомство с ORM-библиотекой **SQLAlchemy**: изучили особенности декларативного стиля, а также взаимодействия сессии с базовыми структурами **SQLAlchemy**.

Рассмотрели базовые возможности библиотеки **PyQt5**: создание GUI-приложения, обработку сигналов и событий, использование примеров документации библиотеки **Qt**.

Библиотека **PyQt** позволяет объединить мощь **Qt** и скорость разработки приложений на Python. Помогает создавать кроссплатформенные приложения с богатым и сложным интерфейсом пользователя.

Qt Designer, **pyuic**, **pyqtdeploy** делают **PyQt** полезным инструментом для быстрого прототипирования.

Практическое задание

1. Продолжить реализацию класса хранилища для серверной стороны.

а. Реализовать функционал работы со списком контактов по протоколу **JIM**:

Получение списка контактов

Запрос к серверу:

```
{
    "action": "get_contacts",
    "time": <unix timestamp>,
    "user_login": "login"
}
```

Положительный ответ сервера будет содержать список контактов:

```
{
    "response": "202",
    "alert": "[`nick_1`, `nick_2`,...]"
}
```

Получение списка контактов — не самая частая операция при взаимодействии с сервером. Она должна выполняться после подключения и авторизации клиента. Иницируется им же. В процессе получения списка контактов клиент не должен инициировать другие запросы.

Добавление/удаление контакта в список контактов:

Запрос к серверу:

```
{
  "action": "add_contact" | "del_contact",
  "user_id": "nickname",
  "time": <unix timestamp>,
  "user_login": "login"
}
```

Ответ сервера будет содержать одно сообщение с кодом результата и необязательной расшифровкой:

```
{
  "response": xxx,
}
```

b. Реализовать хранение информации в БД на стороне клиента:

- список_контактов;
 - история_сообщений.
2. Реализовать графический интерфейс для мессенджера, используя библиотеку **PyQt**. Реализовать графический интерфейс администратора сервера:
- a. отображение списка всех клиентов;
 - b. отображение статистики клиентов;
 - c. настройка сервера (подключение к БД, идентификация).

Дополнительные материалы

1. [ORM. Использование SQLAlchemy.](#)
2. [Вводная по сложным запросам в SQLAlchemy.](#)
3. [Вводный вебинар по PyQt.](#)
4. [Официальная документация по Qt.](#)
5. [PyQt5 для лингвистов.](#)
6. [Краткая документация по PyQt \(RiverBank\).](#)
7. [Различия PyQt4 и PyQt5 \(RiverBank\).](#)

8. [Изучение PyQt5.](#)
9. [PyQt5 tutorial.](#)
10. [Quick PyQt5 1. Signal and Slot Example in PyQt5.](#)
11. [События и сигналы в PyQt5.](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Myers Jason, Copeland Rick. Essential SQLAlchemy: Mapping Python to Databases (каталог «Дополнительные материалы»).
2. Michael Driscoll. Python 101 (Chapter 34 — SQLAlchemy) (каталог «Дополнительные материалы»).
3. [SQLAlchemy Architecture. Michael Bayer.](#)
4. [Глава 20 из второго тома книги «Архитектура приложений с открытым исходным кодом».](#)
5. Н. Прохоренок, В. Дронов. Python 3 и PyQt5 (2016) (каталог «Дополнительные материалы»).
6. Mark Summerfield. Rapid GUI Programming with Python and Qt (каталог «Дополнительные материалы»).

Приложение

Листинг 1

```
# ----- Базы данных -----  
  
# SQLAlchemy. Часть 2  
  
# Работа со связями таблиц и запросами  
  
from sqlalchemy import create_engine  
from sqlalchemy import Table, Column, Integer, Numeric, String, MetaData,  
ForeignKey  
from sqlalchemy.ext.declarative import declarative_base  
from sqlalchemy.orm import sessionmaker  
  
from sqlalchemy.orm import relationship  
  
# Воспользуемся Chinook Database  
engine = create_engine('sqlite:///Chinook_Sqlite.sqlite')  
Session = sessionmaker(bind=engine)
```

```

# Функция declarative_base создаёт базовый класс для декларативной работы
Base = declarative_base()

# ---- Структуру БД см. в файле ChinookDatabaseSchema1.1.png ----

# Определим классы, которые соответствуют таблицам в БД

class Artist(Base):
    __tablename__ = 'Artist'
    ArtistId = Column(Integer, primary_key=True)
    Name = Column(String)

    def __init__(self, name):
        self.Name = name

    def __repr__(self):
        return "<Artist ('%s')>" % self.Name

class Album(Base):
    __tablename__ = 'Album'
    AlbumId = Column(Integer, primary_key=True)
    Title = Column(String)

    # Внешний ключ указывается как аргумент для столбца таблицы
    # Указание внешнего ключа обязывает,
    # чтобы соответствующая запись была во внешней таблице
    ArtistId = Column(Integer, ForeignKey('Artist.ArtistId'))

    # Функция relationship() даёт указание ORM,
    # что класс Album связан с классом Artist через атрибут Album.Artist.
    # Параметр back_populates указывается для реализации обратной связи из
    класса Artist
    Artist = relationship("Artist", back_populates="Albums")

    def __init__(self, title, fullname, password):
        self.Title = title

    def __repr__(self):
        return "<Album ('%s')>" % self.Title

class Track(Base):
    __tablename__ = 'Track'
    TrackId = Column(Integer, primary_key=True)
    Name = Column(String)
    AlbumId = Column(Integer, ForeignKey('Album.AlbumId'))
    MediaTypeId = Column(Integer)
    GenreId = Column(Integer)
    Composer = Column(String)
    Milliseconds = Column(Integer)
    Bytes = Column(Integer)

```

```

UnitPrice = Column(Numeric(10,2))

Album = relationship("Album", back_populates="Tracks")

def __init__(self, name, album_id, media_id, genre_id, composer, msec,
bytes_, price):
    self.Name = name
    self.AlbumId = album_id
    self.MediaTypeId = media_id
    self.GenreId = genre_id
    self.Composer = composer
    self.Milliseconds = msec
    self.Bytes = bytes_
    self.UnitPrice = price

def __repr__(self):
    # При реализованных связях на уровне ORM
    return "<Track ('%s' - %s)>" % (self.Album.Artist.Name, self.Name)

# Указываем дополнительные связи для ORM,
# чтобы данные можно было получать в двух направлениях,
# т.е., например, artist.Albums будет давать список всех альбомов конкретного
исполнителя,
# а album.Artist будет предоставлять непосредственно исполнителя альбома.

# Объявление дополнительных атрибутов классов выполняется здесь (вне описания
класса),
# потому что при объявлении внутри класса привело бы к циклической
зависимости,
# когда атрибут ссылается на класс, который ещё не определён.
Artist.Albums = relationship("Album", order_by=Album.AlbumId,
back_populates="Artist")
Album.Tracks = relationship("Track", back_populates="Album")

# Создаём сессию
session = Session()

# ----- Выполним различные запросы к БД -----

print(' ---- Все доступные исполнители ----')
# q_artists = session.query(Artist).all()
# print(q_artists)

print(' ---- Все альбомы одного исполнителя ----')
artist = session.query(Artist).filter(Artist.Name=='AC/DC').one()
print('Исполнитель:', artist.Name, artist.ArtistId)
albums = artist.Albums
print(' Альбомы:', albums)

album = albums[0]

```

```

print(' ---- Все треки альбома {} ---- '.format(album.Title))
tracks = session.query(Track).filter(Track.Album==album).all()
print(tracks)

print(' -- Подсчёт количества записей в запросе --')
artist = session.query(Artist).filter(Artist.Name=='Lenny Kravitz').one()
track_count = session.query(Track).filter(Track.Album==album).count()
print(' У исполнителя {} найдено {} треков'.format(artist, track_count))

print(' --- Все треки с упоминанием слова "rock" ---')
# Можно итерироваться сразу по результату запроса без извлечения all()
for track in session.query(Track).filter(Track.Name.like('%rock%')):
    print("{} - {}".format(track.Album.Artist.Name, track.Name))

print(' --- Некоторые треки с упоминанием слова "love" ---')
# Можно задавать порядок сортировки методом order_by() (аналог ORDER BY в SQL),
# а также использовать Python-спрез (аналог параметров LIMIT и OFFSET в SQL)
for track in
session.query(Track).order_by(Track.Name).filter(Track.Name.like('%love%'))[2:10
]:
    print("{} - {}".format(track.Album.Artist.Name, track.Name))

# В случае, если результат выборки содержит несколько записей,
# применение метода one() приведёт к исключению MultipleResultsFound:

# track =
session.query(Track).order_by(Track.Name).filter(Track.Name.like('%love%')).on
e()
# >>> sqlalchemy.orm.exc.MultipleResultsFound: Multiple rows were
found for one()

print(' --- Усложнённые условия выборки ---')
from sqlalchemy import or_, and_

print(' --- Треки, содержащие love или war в названии ---')
for track in session.query(Track).filter(or_(Track.Name.like('%love%'),
Track.Name.like('%war%'))):
    print("{} - {}".format(track.Album.Artist.Name, track.Name))

print(' --- Треки указанных исполнителей размером больше 500000 байт ---')
artists = session.query(Artist).filter(Artist.Name.in_(['ABBA', 'AC/DC', 'A-NA',
'Queen']))

# Напрямую применить операцию IN для связанных таблиц не получится:
# albums = session.query(Album).filter(Album.Artist.in_(artists))

# Поэтому нужно сначала получить список id исполнителей,
# и уже для этого списка применить оператор IN:

artists = session.query(Artist.ArtistId).filter(Artist.Name.in_(['ABBA',

```

```

'AC/DC', 'A-HA', 'Queen']]))
albums = session.query(Album.AlbumId).filter(Album.ArtistId.in_(artists))

for track in
session.query(Track).filter(Track.AlbumId.in_(albums)).filter(Track.Bytes >
500000):
    print("{} - {} / {} байт /".format(track.Album.Artist.Name, track.Name,
track.Bytes))

```

Листинг 2

```

# ----- Базы данных -----

# SQLAlchemy. Часть 3

# Создание БД. Добавление, редактирование, удаление записей
# Реализация связи "Многие ко многим"

from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, Numeric, String, MetaData,
ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

from sqlalchemy.orm import relationship

# Создадим простую БД "Библиотека" - будет храниться информация о книгах и
авторах
engine = create_engine('sqlite:///Library.db')
Session = sessionmaker(bind=engine)

# Функция declarative_base создаёт базовый класс для декларативной работы
Base = declarative_base()

# Определим классы, которые соответствуют таблицам в БД

# Поскольку между сущностями Автор - Книга существует отношение "многие ко
многим"
# (один автор может написать несколько книг,
# одна книга может быть написана несколькими авторами),
# нужна таблица-связка:
association_table = Table('BookAuthor', Base.metadata,
                          Column('author_id', Integer, ForeignKey('Author.AuthorId')),
                          Column('book_id', Integer, ForeignKey('Book.BookId'))
)

# Класс Автор будет выступать первичным классом во взаимодействии Автор-Книга
class Author(Base):
    __tablename__ = 'Author'
    AuthorId = Column(Integer, primary_key=True)
    Name = Column(String)

```

```

def __init__(self, name):
    self.Name = name

def __repr__(self):
    return "<Author ('%s')>" % self.Name

# Класс Книга - дочерний по отношению к Автору
class Book(Base):
    __tablename__ = 'Book'
    BookId = Column(Integer, primary_key=True)
    Title = Column(String)

    # Создание взаимосвязи на уровне ORM через функцию relationship.
    # Параметр secondary указывает таблицу-связку.
    # Параметр backref указывает ORM использовать аргумент secondary
    #                                     для обратной связи от Автора к
Книгам
    Authors = relationship("Author",
                           secondary=association_table,
                           backref="Books")

def __init__(self, title, authors):
    self.Title = title
    self.Authors = authors

def __repr__(self):
    return "<Book ('%s')>" % self.Title

def print_books(session):
    ''' Печать книг в библиотеке
    '''
    print(' -- Книги в библиотеке --')
    for book in session.query(Book):
        print("{} {}".format(book.Title)
              ' Authors:', ', '.join(author.Name for author in book.Authors))

# Создание БД, объявленной в декларативном стиле
Base.metadata.create_all(engine)

session = Session()

# Добавление данных в БД
author_1 = Author('Erich Gamma')
author_2 = Author('Richard Helm')
author_3 = Author('Ralph Johnson')
author_4 = Author('John Vlissides')

# При создании книги авторов нужно передавать в списке -
# при этом ORM автоматически выполнит заполнение таблицы-связи.
book_1 = Book('Design Patterns', [author_1, author_2, author_3, author_4])

# Для сохранения данные нужно сначала добавить в сессию

```

```

session.add_all([author_1, author_2, author_3, author_4, book_1])

author_5 = Author('David Beazley')
author_6 = Author('Brian K. Jones')
book_2 = Book('Python Cookbook', [author_5, author_6])
book_3 = Book('Python. Essential Reference', [author_5])

session.add_all([author_5, author_6, book_2, book_3])

# После выполнения commit() все имеющиеся данные будут сохранены в БД
session.commit()

print_books(session)

print(' -- Изменим книгу в библиотеке --')
print(book_3.Title, book_3.BookId)

# Изменение объекта выполняется простым изменением атрибутов объекта и
последующим commit()
book_3.Title = 'Python. Essential Reference. Fourth Edition'
book_3.Authors = [author_5, author_1]
print('Session.dirty:', session.dirty)
session.commit()

print_books(session)

print(' -- Удалим книгу из библиотеки --')
print(book_2.Title, book_2.BookId)

# Удаление объекта из БД выполняется удалением объекта из сессии и последующим
commit()
# При удалении ORM автоматически удалит соответствующие записи из
таблицы-связки.
session.delete(book_2)
print('Session.deleted:', session.deleted)
session.commit()

print_books(session)

```

Листинг 3

```

# ----- Графический интерфейс пользователя. PyQt5
-----

# Простое приложение PyQt

import sys
from PyQt5.QtWidgets import QApplication, QWidget # [1]

if __name__ == '__main__':
    # Обязательно нужно создать объект-приложение

```

```

app = QApplication(sys.argv)                                # [2]

# Виджет основного окна приложения и его настройка
w = QWidget()                                              # [3]
w.resize(250, 150)                                         # [4]
w.move(300, 300)                                           # [5]
w.setWindowTitle('Simple')                                # [6]
# Отобразить виджет
w.show()                                                    # [7]

# Запустить приложение (цикл опроса событий)
sys.exit(app.exec_())                                     # [8]

```

Листинг 4

```

# ----- Графический интерфейс пользователя. PyQt5
#
# Демонстрация создания событий (Action)
"""
ZetCode PyQt5 tutorial

This program creates a toolbar.
The toolbar has one action, which terminates the application, if triggered.

author: Jan Bodnar
website: zetcode.com
last edited: January 2015
"""

import sys
from PyQt5.QtWidgets import QMainWindow, QAction, qApp, QApplication
from PyQt5.QtGui import QIcon

class Example(QMainWindow):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        exitAction = QAction(QIcon('exit.png'), 'Exit', self)
        exitAction.setShortcut('Ctrl+Q')
        exitAction.triggered.connect(qApp.quit)

        self.toolbar = self.addToolBar('Exit')
        self.toolbar.addAction(exitAction)

        self.setGeometry(300, 300, 300, 200)
        self.setWindowTitle('Toolbar')

```



```

        self.show()

if __name__ == '__main__':

    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())

```

Листинг 5

```

import sys
from PyQt5 import QtWidgets

def on_clicked():
    print("Кнопка нажата. Функция on_clicked")

class MyClass():
    def __init__(self, x=0):
        self.x = x
    def __call__(self):
        print("Кнопка нажата. Метод MyClass.__call__()")
        print("x = ", self.x)
    def on_clicked(self):
        print("Кнопка нажата. Метод MyClass.on_clicked()")

obj = MyClass()
app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Нажми меня")

# В качестве обработчика назначается функция
button.clicked.connect(on_clicked)

# В качестве обработчика назначается метод объекта
button.clicked.connect(obj.on_clicked)

# В качестве обработчика назначается экземпляр класса
button.clicked.connect(MyClass(10))

# В качестве обработчика назначается lambda-функция
button.clicked.connect(lambda: MyClass(5)())

button.show()
sys.exit(app.exec_())

```