Qt (продолжение), Qt и потоки

Сигналы и обработчики (продолжение). PyQt и взаимодействие с базами данных. Шаблон «Модель-представление». PyQt и потоки.

[Введение](#_87tlti4ngqx)

[Сигналы и обработчики (продолжение)](#_tyjcwt)

[Блокировка и удаление обработчика](#_3dy6vkm)

[Генерация сигналов](#_1t3h5sf)

[Передача данных в обработчик](#_4d34og8)

[Пример работы с сигналами-слотами](#_2s8eyo1)

[Перехват различных событий](#_17dp8vu)

[Как пользоваться документацией Qt](#_3rdcrjn)

[Взаимодействие с БД](#_26in1rg)

[Соединение с базой данных](#_lnxbz9)

[SQLAlchemy + PyQt](#_4i7ojhp)

[PyQt и потоки](#_2xcytpi)

[Итоги](#_1ci93xb)

[Практическое задание](#_3whwml4)

[Дополнительные материалы](#_2bn6wsx)

[Используемая литература](#_qsh70q)

[Приложение](#_r93gc1w9mvzp)

# 

# 

# Введение

На этом уроке продолжим знакомство с библиотекой **PyQt5**: изучим сигналы и обработчики, проанализируем особенности взаимодействия библиотеки **PyQt** и баз данных, реализацию поточных приложений в привязке к **PyQt**.

# Сигналы и обработчики (продолжение)

## Блокировка и удаление обработчика

Для блокировки и удаления обработчиков предназначены следующие методы класса **QObject**:

* **blockSignals(<Флаг>)** — блокировка приема сигналов, если <Флаг> равен True, и отмена блокировки, если False;
* **signalsBlocked()** — проверка, установлена ли блокировка сигналов (True — установлена, False — нет);
* **disconnect()** — удаление обработчика сигнала. Форматы:

| <Компонент>.<Сигнал>.disconnect([Обработчик]) <Компонент>.<Сигнал>[<Тип>].disconnect([Обработчик]) |
| --- |

## Генерация сигналов

В некоторых случаях необходимо сгенерировать сигнал программно. Например, при заполнении последнего текстового поля и нажатии клавиши **<Enter>** можно имитировать нажатие кнопки **OK** — так подтверждается ввод пользователя. Сгенерировать сигнал из программы позволяет метод **emit()** класса **QObject**. Форматы метода:

| <Компонент>.<Сигнал>.emit([Данные]) <Компонент>.<Сигнал>[<Тип>].emit([Данные]) |
| --- |

Метод **emit()** всегда вызывается у объекта, которому посылается сигнал.

Сигналу и его обработчику можно передать данные, указав их в вызове метода **emit()**:

| button.clicked[bool].emit(False) button.clicked["bool"].emit(False) |
| --- |

Чтобы создавать собственные сигналы, разработчику необходимо в классе определить атрибут, имя которого будет именем сигнала. Атрибут сигнал создается функцией **pyqtSignal()** модуля **QtCore**. Формат функции:

| <Объект\_сигнала> = pyqtSignal(\*<Типы\_данных>[, name=<Имя\_сигнала>]) |
| --- |

* <Типы\_данных> — перечисление через запятую типов данных, передаваемых сигналу. При использовании типа данных C++ его название нужно указывать в виде строки. Если сигнал не принимает параметров, <Типы\_данных> не указываются:

| mysignal\_1 = QtCore.pyqtSignal(int) mysignal\_2 = QtCore.pyqtSignal(int, str) mysignal\_3 = QtCore.pyqtSignal("QDate") |
| --- |

## Передача данных в обработчик

При назначении обработчика в метод **connect()** передается ссылка на функцию или метод (без аргументов). Передать данные из сигнала в обработчик можно несколькими способами:

* создать «обертку» в виде lambda—функции для вызова обработчика с аргументами:

| self.button.clicked.connect(lambda: self.on\_clicked\_button1(10)) |
| --- |

* передать ссылку на экземпляр класса, внутри которого определен метод **\_\_call()\_\_**. Передаваемое значение указывается в качестве параметра конструктора этого класса:

| class MyClass():  def \_\_init\_\_(self, x=0):  self.x = x   def \_\_call\_\_(self):  print("x = ", self.x) ...  self.button1.clicked.connect(MyClass(10)) |
| --- |

* передать ссылку на обработчик и данные в функцию **partial** из модуля **functools**:

| from functools import partial self.button.clicked.connect(partial(self.on\_clicked\_button1, 10)) |
| --- |

## Пример работы с сигналами-слотами

Рассмотрим пример взаимодействия двух диалоговых окон одного приложения через механизм сигналов/слотов (полный код — в **листинге 1**).

Создадим два класса диалоговых окон: в одном есть ползунок (Slider) — модуль **slider.py** (**листинг 2**), в другом — строка прогресса (**ProgressBar**) — модуль **progress.py** (**листинг 3**). Диалоговые окна можно создать в приложении **QtDesigner** или набрать для этого полный код виджетов.

Нужно сделать так, чтобы движение ползунка в одном диалоговом окне изменяло строку прогресса в другом. Для этого в классе диалога с ползунком должен быть определен PyQt-сигнал, передающий значение прогресса. А в классе диалога со строкой прогресса необходимо определить PyQt-слот (метод объекта данного класса), принимающий значение прогресса:

| import sys from PyQt5 import QtGui, QtWidgets from PyQt5.QtCore import Qt, pyqtSignal, pyqtSlot import slider import progress  class SliderDialog(QtWidgets.QDialog):  *# Добавляем Qt—сигнал как атрибут класса*  changedValue = pyqtSignal(int)   def \_\_init\_\_(self, parent=None):  QtWidgets.QWidget.\_\_init\_\_(self, parent)  self.ui = slider.Ui\_SliderDialog()  self.ui.setupUi(self)  *# Связываем оригинальный сигнал слайдера с функцией данного класса*  self.ui.horizontalSlider.valueChanged.connect(self.on\_changed\_value)   def on\_changed\_value(self, value):  *# Активируем сигнал*  self.changedValue.emit(value)  class ProgressDialog(QtWidgets.QDialog):  def \_\_init\_\_(self, parent=None):  QtWidgets.QWidget.\_\_init\_\_(self, parent)  self.ui = progress.Ui\_ProgressDialog()  self.ui.setupUi(self)   *# Создаем Qt—слот*  @pyqtSlot(int)  def get\_slider\_value(self, val):  self.ui.progressBar.setValue(val)   def make\_connection(self, slider\_object):  *# Связываем "свой" сигнал со "своим" слотом*  slider\_object.changedValue.connect(self.get\_slider\_value) |
| --- |

Объединение данных диалоговых окон происходит в основном коде программы:

| import sys from PyQt5 import QtGui, QtWidgets  if \_\_name\_\_ == '\_\_main\_\_':  app = QtWidgets.QApplication(sys.argv)  slider = SliderDialog()  progress = ProgressDialog()   *# Непосредственное связывание ProgressBar'а и Sliser'а*  progress.make\_connection(slider)  progress.show()  slider.show()  sys.exit(app.exec\_()) |
| --- |

## Перехват различных событий

Обработка внешних событий осуществляется иначе, чем обмен сигналами между компонентами. Чтобы обработать событие, нужно определить в классе специальный метод — например, чтобы обработать нажатие клавиши, следует определить метод **keyPressEvent()**. Такие специальные методы принимают объект, содержащий детальную информацию о событии, — например, код нажатой клавиши. Эти объекты наследуют от класса **QEvent** следующие методы:

* **accept()** — устанавливает флаг, разрешающий дальнейшую обработку события. Данный флаг обычно установлен по умолчанию;
* **ignore()** — сбрасывает флаг, разрешающий дальнейшую обработку события;
* **setAccepted(<Флаг>)** — если <Флаг> равен True, работает как **accept()**; если False — как **ignore()**;
* **isAccepted()** — возвращает текущее состояние флага, разрешающего дальнейшую обработку события;
* **spontaneous()** — возвращает **True**, если событие сгенерировано системой, и **False** — если внутри программы;
* **type()** — возвращает тип события. Перечислим некоторые типы (полный список — в документации по классу [QEvent](http://doc.qt.io/qt-5/qevent.html)):
* 0 — нет события;
* 1 — **Timer** — событие таймера;
* 2 — **MouseButtonPress** — нажата кнопка мыши;
* 3 — **MouseButtonRelease** — отпущена кнопка мыши;
* 4 — **MouseButtonDblClick** — двойной щелчок мышью;
* 5 — **MouseMove** — перемещение мыши;
* 6 — **KeyPress** — клавиша на клавиатуре нажата;
* 7 — **KeyRelease** — клавиша на клавиатуре отпущена;
* 8 — **FocusIn** — получен фокус ввода с клавиатуры;
* 9 — **FocusOut** — потерян фокус ввода с клавиатуры;
* 10 — **Enter** — указатель мыши входит в область компонента;
* 11 — **Leave** — указатель мыши покидает область компонента;
* 19 — **Close** — окно закрыто;
* 31 — **Wheel** — прокручено колесико мыши;
* 82 — **ContextMenu** — событие контекстного меню;
* 1000 — **User** — пользовательское событие;
* 65535 — **MaxUser** — максимальный идентификатор пользовательского события.

Чтобы зарегистрировать пользовательский тип события (в пределах от **QEvent.User (1000)** до **QEvent.MaxUser (65535)**), необходимо воспользоваться статическим методом **registerEventType([<Число>])**.

Перехват всех событий осуществляется с помощью метода **event(self, <event>)**. Через параметр **<event>** доступен объект с дополнительной информацией о событии. Этот объект отличается для разных типов событий — например, для события **MouseButtonPress** объект будет экземпляром класса **QMouseEvent**, а для события **KeyPress** — экземпляром класса **QKeyEvent**.

Стандартные методы событий и их соответствующие типы представлены в таблице.

| **Метод события** | **Тип объекта event** | **Описание** |
| --- | --- | --- |
| paintEvent() | QPaintEvent | Вызывается, когда виджет надо перерисовать. Если он отображает нестандартный контент, в нем должен быть реализован данный метод. Использование **QPainter** возможно только через **paintEvent()** |
| resizeEvent() | QResizeEvent | Непрерывно вызывается при изменении размеров окна |
| mousePressEvent() | QMouseEvent | Вызывается при нажатии кнопки мыши, когда ее указатель находится в пределах виджета |
| mouseReleaseEvent() | QMouseEvent | Вызывается при отпускании кнопки мыши. Виджет получает это событие, если эта кнопка была нажата над ним |
| mouseDoubleClickEvent() | QMouseEvent | Вызывается при двойном клике на виджете. В этом случае он получает следующую последовательность событий: **MousePress**, **MouseRelease**, **MousePress**, **mouseDoubleClick**, **MouseRelease**. Невозможно отличить простой одинарный клик от двойного до второго клика |
| keyPressEvent() | QKeyEvent | Вызывается, когда нажимается клавиша на клавиатуре, и повторно при длительном нажатии (если клавиша имеет возможность самоповторения — например, **Enter**). Клавиши **Tab** и **Shift+Tab** передаются виджету, только если они не задействованы в механизме установки фокуса. Чтобы перехватить нажатия этих клавиш, нужно реализовать метод **QWidget.event()** |
| keyReleaseEvent() | QKeyEvent | Вызывается, когда клавиша клавиатуры отпускается или удерживается. В случае с самоповторяющейся клавишей виджет будет получать пары событий **KeyRelease** и **KeyPress** на каждом повторе. Обработка клавиш **Tab** и **Shift+Tab** аналогична **keyPressEvent** |
| focusInEvent() | QFocusEvent | Вызывается, когда виджет получает фокус ввода |
| focusOutEvent() | QFocusEvent | Вызывается, когда виджет теряет фокус ввода |
| mouseMoveEvent() | QMouseEvent | Вызывается при движении курсора мыши с ее же зажатой клавишей. Может быть полезно при обработке drag&drop-операций |
| wheelEvent() | QWheelEvent | Вызывается при движении колесика мыши, когда фокус ввода установлен на виджете |
| enterEvent() | QEvent | Вызывается, когда курсор мыши входит в визуальную площадь виджета (кроме площади дочерних виджетов) |
| leaveEvent() | QEvent | Вызывается, когда курсор выходит из экранного пространства виджета (кроме случаев выхода из пространства дочерних виджетов) |
| moveEvent() | QMoveEvent | Вызывается, когда виджет был перемещен относительно родителя |
| closeEvent() | QCloseEvent | Вызывается, когда пользователь закрывает виджет (также при вызове метода **close()** |
| changeEvent() | QWindowStateChange | Вызывается при изменении состояния окна, приложения или виджета, заголовка окна, его палитры, статуса активности окна верхнего уровня, языка, локали и другого |
| showEvent() | QShowEvent | Вызывается при отображении виджета |
| hideEvent() | QHideEvent | Вызывается при скрытии виджета |
| moveEvent() | QMoveEvent | Непрерывно вызывается при перемещении окна |

Из метода **event()** следует вернуть в качестве результата значение **True**, если событие было обработано, и **False** — если нет. Если возвращается значение **True**, родительский компонент не получает событие. Чтобы продолжить распространение события, необходимо вызвать метод **event()** базового класса и передать ему текущий объект события, например:

| *# Для случая с родительским классом QWidget* return QtWidgets.QWidget.event(self, e)  *# Для случая с родительским классом QLabel* return QtWidgets.QLabel.event(self, e) |
| --- |

Рассмотрим пример перехвата событий нажатия клавиши, клика мышью и закрытия окна (**листинг 4**):

| import sys from PyQt5 import QtCore, QtWidgets  class MyWindow(QtWidgets.QWidget):  def \_\_init\_\_(self, parent=None):  QtWidgets.QWidget.\_\_init\_\_(self, parent)  self.resize(300, 100)   def event(self, e):  if e.type() == QtCore.QEvent.KeyPress:  print("Нажата клавиша на клавиатуре")  print("Код:", e.key(), ", текст:", e.text())  elif e.type() == QtCore.QEvent.Close:  print("Окно закрыто")  elif e.type() == QtCore.QEvent.MouseButtonPress:  print("Клик мышью. Координаты:", e.x(), e.y())   *# Событие отправляется дальше*  return QtWidgets.QWidget.event(self, e)  if \_\_name\_\_ == '\_\_main\_\_':  app = QtWidgets.QApplication(sys.argv)  window = MyWindow()  window.show()  sys.exit(app.exec\_()) |
| --- |

# Как пользоваться документацией Qt

Поскольку библиотека **PyQt** — это надстройка над **Qt**, могут потребоваться [официальная документация](http://doc.qt.io/qt-5/reference-overview.html) и [примеры библиотеки](http://doc.qt.io/qt-5/qtexamplesandtutorials.html) **Qt**. У программиста, не работавшего с языком C++, это может вызвать затруднения.

Рассмотрим подходы, позволяющие использовать примеры и документацию библиотеки **Qt** без знаний в C++. Для наглядности выполним преобразование C++-кода примера [Calculator Example](http://doc.qt.io/qt-5/qtwidgets-widgets-calculator-example.html) из документации **Qt** в Python-код (полный код — в файле **листинг 5** к уроку):

1. Конструктор класса в С++ имеет такое же имя, как и сам класс:

| Button::Button(const QString &text, QWidget \*parent) : QToolButton(parent) {  setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Preferred);  setText(text); } |
| --- |

В Python конструктором будет метод \_\_init\_\_:

| class Button(QToolButton):  def \_\_init\_\_ (self, text, parent):  super().\_\_init\_\_(parent)  self.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)  self.setText(text) |
| --- |

1. В языке С++ классы объявляются в файлах с расширением **.h**, реализация методов — в файлах с расширением **.cpp**. В Python и объявление класса, и его реализация находятся в одном файле.
2. Указатель **this** в C++ — это указатель на текущий объект, то есть в Python — это **self**. Однако в C++ внутри методов класса доступ к атрибутам объекта осуществляется просто по имени атрибута:

| Calculator::Calculator(QWidget \*parent) : QWidget(parent) {  sumInMemory = 0.0;  sumSoFar = 0.0;  waitingForOperand = true; } |
| --- |

В Python доступ к атрибуту объекта осуществляется через имя объекта **self**:

| class Calculator(QWidget):  def \_\_init\_\_(self, parent=None):  super().\_\_init\_\_(parent)  self.sumInMemory = 0.0  self.sumSoFar = 0.0  self.waitingForOperand = True |
| --- |

1. Доступ к пространству имени класса в С++ осуществляется через оператор **::** (два двоеточия):

| mainLayout—>setSizeConstraint(QLayout::SetFixedSize); |
| --- |

В Python доступ к пространству имени любого объекта осуществляется через оператор **.** (точка):

| self.mainLayout.setSizeConstraint(QLayout.SetFixedSize) |
| --- |

1. В языке C++ есть понятия «указатель» (символ \* перед именем переменной), «ссылка» (символ амперсанда (&) перед именем переменной):

| *// Атрибут pointButton — указатель на объект типа Button* Button \*pointButton = createButton(tr("."), SLOT(pointClicked())); *// Аргумент text — ссылка на объект типа QString* Button \*Calculator::createButton(const QString &text, const char \*member) { ... *// любой код* } |
| --- |

В Python существует только понятие ссылки на объект. Доступ к атрибутам объекта осуществляется через оператор . (точка):

| *# Атрибут pointButton — ссылка на объект кнопки* self.pointButton = self.createButton(".", self.pointClicked) *# Аргумент text — изначально ссылка на объект* def createButton(self, text, member): |
| --- |

1. Если в С++ указатель хранит адрес памяти, в которой содержится сложный объект, доступ к атрибутам объекта через указатель осуществляется через оператор —> (стрелка), например:

| *// mainLayout — указатель на объект типа QGridLayout* *//* ***new*** *выделяет память под объект QGridLayout и возвращает указатель на этот объект* QGridLayout \*mainLayout = new QGridLayout; *// setSizeConstraint — метод объекта типа QGridLayout* mainLayout—>setSizeConstraint(QLayout::SetFixedSize); |
| --- |

В Python доступ к атрибутам объекта через ссылку осуществляется через оператор . (точка):

| *# mainLayout — ссылка на объект типа QGridLayout* self.mainLayout = QGridLayout() *# setSizeConstraint — метод объекта типа QGridLayout* self.mainLayout.setSizeConstraint(QLayout.SetFixedSize) |
| --- |

1. Некоторые базовые типы **Qt** транслируются в базовые типы Python. Например, **QString** из **Qt** преобразуется в **str** в **PyQt**. Поэтому часть методов из кода C++ нужно заменить на стандартные операторы и функции Python (см. таблицу).

Соответствие кода С++ и Python

| **Код на C++** | **Код на Python** |
| --- | --- |
| QString::number(sumSoFar) | str(sumSoFar) |
| display—>text().toDouble() | float(display.text()) |
| text.isEmpty() | text == ‘’ |
| text.clear() | text = ‘’ |

1. Библиотека **Qt** упрощает локализацию приложений — помогает переводить строки на другой национальный язык. Для этого чаще используется функция **tr**, но с **PyQt** следует применять только метод **translate**, принадлежащий классу **QCoreApplication** (см. [документацию по локализации](http://pyqt.sourceforge.net/Docs/PyQt5/i18n.html) PyQt5-приложений):

| QtCore.QCoreApplication.translate("Calculator", "Sqrt") |
| --- |

# Взаимодействие с БД

## Соединение с базой данных

Библиотека **PyQt5** включает встроенные средства для работы с базами данных форматов **SQLite**, **MySQL**, **Oracle**, **PostgreSQL** и других. С их помощью можно выполнять любые SQL-запросы и обрабатывать их результаты, получать доступ к отдельным таблицам базы, работать с транзакциями, использовать модели для вывода содержимого таблиц или запросов в любом виджете-представлении (**QListView**, **QTableView**, **QTreeView**).

Все классы, обеспечивающие работу с базами данных, определены в модуле **QtSqlDatabase**.

Чтобы установить соединение с базой, следует вызвать статический метод **addDatabase()** этого класса:

| addDatabase(<Формат\_базы\_данных>[, connectionName='qt\_sql\_default\_connection']) |
| --- |

Первым параметром указывается строка, обозначающая формат открываемой базы данных. Поддерживаются следующие форматы: QMYSQL и QMYSQL3 (MySQL), QODBC и QODBC3 (ODBC), QPSQL и QPSQL7 (PostgreSQL) и QSQLITE (SQLite версии 3).

Вторым параметром можно задать имя соединения — на случай, если приложение работает сразу с несколькими БД.

Метод **addDatabase()** возвращает экземпляр класса **QSqlDatabase** (база данных, с которой установлено соединение). Далее можно задать параметры базы данных:

* **setHostName(<Хост>)** — задает хост, на котором расположена СУБД;
* **setPort(<Номер\_порта>)** — задает номер TCP-порта для подключения к хосту;
* **setDatabaseName(<Имя\_или\_путь\_к\_БД>)** — задает имя базы данных (для **MySQL**, **PostgreSQL**), путь к ней (для **SQLite**) или полный набор параметров подключения (**ODBC**);
* **setUserName(<Имя>)** — задает имя пользователя для подключения к БД;
* **setPassword(<Пароль>)** — задает пароль для подключения к БД;
* **setConnectionOptions(<Параметры>)** — задает набор дополнительных параметров для подключения к БД в виде строки.

Для работы с базой данных предназначены следующие методы класса [QSqlDatabase](http://doc.qt.io/qt-5/qsqldatabase.html):

* **open()** — открывает базу данных (возвращает **True/False**);
  + перед созданием соединения с базой данных следует обязательно добавить объект приложения (**QApplication**). Иначе **PyQt** не загрузит драйвер указанного формата БД и соединение не будет создано.
* **open(<Имя>, <Пароль>)** — открывает базу данных с указанным именем и паролем (возвращает **True/False**);
* **isOpen()** — проверяет, открыта ли база данных (возвращает True/False);
* **isOpenError()** — проверяет наличие ошибок при открытии БД (возвращает True/False);
* **transaction()** — запускает транзакцию (возвращает True/False);
* **commit()** — завершает транзакцию (возвращает True/False);
* **rollback()** — отменяет транзакцию (возвращает True/False);
* **lastError()** — возвращает сведения о последней возникшей ошибке при работе с БД (**QSqlError**);
* **tables([type=Tables])** — возвращает список таблиц, хранящихся в базе. В параметре **type** можно указать тип или комбинацию типов таблиц (через оператор |):
  + **Tables = 1** — обычные таблицы;
  + **SystemTables = 2** — служебные таблицы;
  + **Views = 4** — представления;
  + **AllTables = 255** — все вышеперечисленное;
* **record(<Имя\_таблицы>)** — возвращает сведения о структуре заданной таблицы, представленные объектом класса **QSqlRecord**, или пустой экземпляр этого класса, если таблицы с таким именем нет;
* **primaryIndex(<Имя\_таблицы>)** — возвращает сведения о ключевом индексе заданной таблицы, представленные объектом класса **QSqlIndex**, или пустой экземпляр этого класса, если таблицы с таким именем нет;
* **close()** — закрывает базу данных;
* **drivers()** — возвращает список всех поддерживаемых **PyQt** форматов баз данных.

Примеры соединений с базами данных разных форматов:

| import sys from PyQt5 import QtWidgets, QtSql  app = QtWidgets.QApplication(sys.argv) *# Открытие базы данных SQLite* con1 = QtSql.QSqlDatabase.addDatabase('QSQLITE') con1.setDatabaseName('C://work//data.sqlite') con1.open() con1.close() *# Открытие базы данных MySQL* con2 = QtSql.QSqlDatabase.addDatabase('QMYSQL') con2.setHostName('localhost') con2.setDatabaseName('TestDB') con2.setUserName('Mario') con2.setPassword('karabuka') con2.open() con2.close() |
| --- |

## SQLAlchemy + PyQt

При создании приложения с графическим интерфейсом может возникнуть ситуация, когда GUI необходимо реализовать в дополнение к уже написанному коду взаимодействия с БД. В этом случае нет необходимости переносить весь код взаимодействия с БД на SQL-провайдеры **PyQt**. Достаточно реализовать класс модели данных **PyQt**, унаследовавшись от **QAbstractTableModel** или **QSqlTableModel**.

В соответствии с [документацией по QAbstractTableModel](http://doc.qt.io/Qt-5/qabstracttablemodel.html#details) при описании своей табличной модели требуется определить методы:

* **rowCount()** — возвращает количество строк в выборке;
* **columnCount()** — возвращает количество столбцов в выборке;
* **data()** — получение данных с указанной ролью по указанному индексу;
* **headerData()** — получение данных из заголовка таблицы.

Для редактируемых таблиц следует определить методы:

* **setData()** — установка значения указанной ячейки в выборке;
* **flags()** — должен возвращать значение, содержащее **Qt.ItemIsEditable**.

Если таблица может быть изменена (добавлены/удалены строки или столбцы) следует определить методы:

* **insertRows()** — добавление строк. При реализации нужно вызвать **beginInsertRows()** перед добавлением новых строк и **endInsertRows()** — сразу после добавления;
* **removeRows()** — удаление строк. При реализации нужно вызвать **beginRemoveRows()** перед удалением строк и **endRemoveRows()** — сразу после удаления;
* **insertColumns()** — добавление столбцов. При реализации нужно вызвать **beginInsertColumns()** перед добавлением столбцов и **endInsertColumns()** — сразу после;
* **removeColumns()** — удаление столбцов. При реализации нужно вызвать **beginRemoveColumns()** до удаления столбцов и **endRemoveColumns()** — сразу после.

Подробности создания моделей можно найти в разделе [Model Subclassing Reference](http://doc.qt.io/Qt-5/model-view-programming.html#model-subclassing-reference) документации **Qt**.

Примеры, как совмещать библиотеки **SQLAlchemy** и **PyQt5**, приведены в **листинге 6** и **листинге 7** из учебных материалов урока.

Объединение возможностей двух больших библиотек (**PyQt** и **SQL** **Alchemy**) позволяет расширить функционал приложения, сохраняя лаконичность программного кода.

# PyQt и потоки

Одна из самых частых задач, которая решается при помощи многопоточного программирования — написание графического интерфейса пользователя. Чтобы он не «замораживался» при выполнении некоторых функций программы, их выносят в отдельные потоки. В таком случае пользователю можно предоставить способы прекратить выполняющиеся функции.

Если для реализации интерфейса пользователя рассматривать библиотеку **PyQt**, стоит отметить следующие особенности:

* наличие класса **QThread**, выполняющего роль интерфейса для стандартных потоков ОС;
* наличие сигнально-слотового взаимодействия, обеспечивающего удобное безопасное «общение» потока и графического интерфейса.

Рассмотрим пример GUI-приложения для поиска текста на разных URL интернете (см. **листинг 8** в Приложении).

Класс **Finder** — «поисковый движок»:

| from urllib.request import urlopen from threading import Thread from queue import Queue import re  class Finder:  ''' Класс "поискового движка" '''  def \_\_init\_\_(self, text, res\_queue=None):  self.text = text  self.res\_queue = res\_queue  self.is\_alive = False   def search\_in\_url(self, url):  ''' Искать text по указанному URL '''  f = urlopen(url)  data = f.read().decode('utf—8')  pattern = "<p>.\*?{}.\*?</p>".format(self.text)  res = re.findall(pattern, data)  return res   def \_in\_urls(self, urls):  ''' Поиск текста по списку urls '''  self.is\_alive = True  for url in urls:  *# На каждом шаге выполняется проверка — не был ли остановлен поиск*  if not self.is\_alive:  break  data = self.search\_in\_url(url)  *# Если есть очередь для результатов, он передается в нее* if self.res\_queue is not None:  self.res\_queue.put((url, data))   self.is\_alive = False  if self.res\_queue is not None:  *# None — флаг для сигнализации окончания поиска*  self.res\_queue.put(None)  self.res\_queue.join()   def search\_in\_urls(self, urls):  ''' Запуск поиска в отдельном потоке '''  t = Thread(target=self.\_in\_urls, args=(urls, ))  t.daemon = True  t.start()   def stop\_search(self):  ''' Остановка поиска'''  self.is\_alive = False |
| --- |

Реализация данного класса достаточно проста. Использует очередь, которая рассмотрена в разделе «Модуль queue». Класс **Finder** реализован как GUI-независимый класс, выполняющий запросы в потоке **threading.Thread** и возвращающий результаты поиска через очередь **queue.Queue**. Это позволяет абстрагироваться от конкретной GUI-библиотеки.

Особое внимание стоит обратить на реализацию графического интерфейса. Форма главного окна была реализована через **Qt Designer** и преобразована в py-файл утилитой **pyuic5** (см. **листинг 9** и **листинг 10**). GUI-часть данного приложения использует класс **QThread** для выполнения отдельных функций в потоке. Применение класса **QThread** вместо **threading.Thread** объясняется философией использования «родных» классов фреймворка, если они есть.

Важная деталь: класс **QThread** предназначен для использования в качестве интерфейса к потокам операционной системы, но не для того, чтобы помещать в него код, предназначенный для выполнения в отдельном потоке. В ООП наследование класса выполняется, чтобы расширить или углубить функциональность базового класса. Смысл наследования **QThread** — добавление такой функциональности, которой в **QThread** не существует. Например, передача указателя на область памяти, которую поток может использовать для своего стека. Или добавление поддержки интерфейсов реального времени. Но загрузка файлов, работа с базами данных и подобные функции не должны присутствовать в наследуемых классах **QThread** — они должны реализовываться в других объектах.

Объекты классов-наследников **QObject** имеют метод **moveToThread**, помещающий объект в указанный поток. В таком случае необходимо настроить связь сигналов/слотов потока, объекта в потоке и графического интерфейса.

Класс, объекты которого будут помещаться в поток, имеет простую реализацию (**листинг 11**):

| from queue import Queue from finder import Finder  class FinderMonitor(QObject):  ''' Класс—монитор, принимающий результаты поиска из очереди результатов  '''  gotData = pyqtSignal(tuple)  finished = pyqtSignal(int)   def \_\_init\_\_(self, parent, urls, text):  super().\_\_init\_\_()  self.parent = parent  self.urls = urls  self.text = text  self.res\_queue = Queue()  self.finder = Finder(self.text, self.res\_queue)   def search\_text(self):  ''' Запуск функции поиска (в потоке) и опрос очереди '''  self.finder.search\_in\_urls(self.urls)  while True:  data = self.res\_queue.get()  if data is None:  break  self.gotData.emit(data)  self.res\_queue.task\_done()   self.res\_queue.task\_done()  self.finished.emit(0)   def stop(self):  self.finder.stop\_search() |
| --- |

Класс **FinderMonitor** — «переходник» от поисковика к GUI-интерфейсу (принимает данные из очереди результатов, формирует сигналы для графического интерфейса). Он будет выполняться в отдельном потоке, но уже класса **QThread**. Основную нагрузку несет метод **search\_text**, который запускает функцию поиска, а далее находится в бесконечном цикле опроса очереди результатов. Как только из очереди приходит результат поиска, функция активирует Qt-сигнал **gotData** для передачи данных в графический интерфейс. При завершении поиска активируется сигнал **finished**, чтобы обозначить окончание поиска.

Далее создается класс основного окна GUI-приложения (приводится в сокращении, полный код смотрите в **листинге 11**):

| import sys from PyQt5 import QtGui, QtWidgets from PyQt5.QtCore import Qt, QObject, QThread, pyqtSignal, pyqtSlot  from search\_form import Ui\_FinderForm  class ProgressDialog(QtWidgets.QDialog):  ''' Класс GUI—формы "Поисковика" '''  def \_\_init\_\_(self, parent=None):  QtWidgets.QDialog.\_\_init\_\_(self, parent)  self.ui = Ui\_FinderForm()  self.ui.setupUi(self)  ...   @pyqtSlot(tuple)  def update\_results(self, data):  ''' Отображение результатов поиска '''  for text in data[1]:  self.ui.plainTextEdit.appendPlainText(text)   @pyqtSlot()  def update\_progress(self):  ''' Изменение строки прогресса '''  self.progress += self.prog\_val  self.ui.progressBar.setValue(self.progress)   def start\_search(self):  ''' Запуск поиска '''  if not self.is\_active:  self.is\_active = True  *# Получение списка URL и текста для поиска*  urls = self.ui.plainTextEdit\_2.toPlainText().split('\n')  text = self.ui.lineEdit.text()  self.progress = 0  self.prog\_val = 100 / len(urls)   *# Создание объекта-монитора результатов*  self.monitor = FinderMonitor(self, urls, text)  self.monitor.gotData.connect(self.update\_results)  self.monitor.gotProgress.connect(self.update\_progress)   *# Создание потока и помещение объекта-монитора в этот поток*  self.thread = QThread()  self.monitor.moveToThread(self.thread)   *# —————————— Важная часть — связывание сигналов и слотов ——————————*  *# При запуске потока будет вызван метод search\_text*  self.thread.started.connect(self.monitor.search\_text)   *# При завершении поиска необходимо завершить поток и изменить GUI*  self.monitor.finished.connect(self.thread.quit)  self.monitor.finished.connect(self.finished)   *# Завершение процесса поиска по кнопке "Остановить"*  self.ui.pushButton\_2.clicked.connect(self.monitor.stop)   *# Запуск потока, который запустит self.monitor.search\_text*  self.thread.start()  if \_\_name\_\_ == '\_\_main\_\_':  app = QtWidgets.QApplication(sys.argv)  progress = ProgressDialog()  progress.show()  sys.exit(app.exec\_()) |
| --- |

Важная часть данного кода — помещение объекта в поток **self.monitor.moveToThread(self.thread)** и связывание сигналов и слотов. Запуск метода **search\_text** в потоке происходит посредством активации сигнала **thread.started** при старте потока **thread.start()**.

# Итоги

На этом занятии мы продолжили изучать особенности использования библиотеки **PyQt5** для создания GUI-приложений. Рассмотрели взаимосвязь библиотеки **PyQt** и потоков.

Возможности библиотеки **Qt** и **PyQt** широки, и дальнейшее знакомство с ними выносим на самостоятельное изучение.

# Практическое задание

Продолжаем работать над мессенджером:

1. Реализовать графический интерфейс пользователя на стороне клиента:
   1. Отображение списка контактов;
   2. Выбор чата двойным кликом на элементе списка контактов;
   3. Добавление нового контакта в локальный список контактов;
   4. Отображение сообщений в окне чата;
   5. Набор сообщения в окне ввода сообщения;
   6. Отправка введенного сообщения.

# Дополнительные материалы

1. [Вводный вебинар по PyQt](https://geekbrains.ru/events/79).
2. [Официальная документация по Qt](http://doc.qt.io/qt-5/index.html).
3. [PyQt5 для лингвистов](http://pyqtforlinguists.appspot.com/ebook/index.html).
4. [Краткая документация по PyQt (RiverBank)](http://pyqt.sourceforge.net/Docs/PyQt5/introduction.html).
5. [Различия PyQt4 и PyQt5 (RiverBank)](http://pyqt.sourceforge.net/Docs/PyQt5/pyqt4_differences.html).
6. [Изучение PyQt5](http://python-3.ru/category/pyqt).
7. [PyQt5 tutorial](http://zetcode.com/gui/pyqt5/).
8. [Quick PyQt5 1. Signal and Slot Example in PyQt5](https://blog.manash.me/quick-pyqt5-1-signal-and-slot-example-in-pyqt5-bf502ccaf11d).
9. [События и сигналы в PyQt5](https://pythonworld.ru/gui/pyqt5-eventssignals.html).

# Используемая литература

1. David Beazley, Brian K. Jones. Python Cookbook, Third Edition (каталог «Дополнительные материалы»).
2. Giancarlo Zaccone. Python Parallel Programming Cookbook (каталог «Дополнительные материалы»).
3. Jan Palach. Parallel Programming with Python (каталог «Дополнительные материалы»).
4. Бизли Дэвид. Python. Подробный справочник. 4-е издание (каталог «Дополнительные материалы»).
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**

| *# ---------------- Графический интерфейс пользователя. PyQt5 ------------------* *# Демонстрация взаимодействия "сигнал-слот". Основное приложение*  import sys from PyQt5 import QtGui, QtWidgets from PyQt5.QtCore import Qt, pyqtSignal, pyqtSlot  from slider import slider from progress import progress   class SliderDialog(QtWidgets.QDialog):  ''' Диалог со слайдером '''   *# Добавляем Qt-сигнал*  changedValue = pyqtSignal(int)   def \_\_init\_\_(self, parent=None):  QtWidgets.QWidget.\_\_init\_\_(self, parent)  self.ui = slider.Ui\_SliderDialog()  self.ui.setupUi(self)  *# Связываем оригинальный сигнал слайдера с функцией данного класса*  self.ui.horizontalSlider.valueChanged.connect(self.on\_changed\_value)    def on\_changed\_value(self, value):  *# Активируем сигнал*  self.changedValue.emit(value)   class ProgressDialog(QtWidgets.QDialog):  ''' Диалог с прогресс-строкой '''    def \_\_init\_\_(self, parent=None):  QtWidgets.QWidget.\_\_init\_\_(self, parent)  self.ui = progress.Ui\_ProgressDialog()  self.ui.setupUi(self)   *# Создаём Qt-слот*  @pyqtSlot(int)  def get\_slider\_value(self, val):  self.ui.progressBar.setValue(val)   def make\_connection(self, slider\_object):  *# Связываем "свой" сигнал со "своим" слотом*  slider\_object.changedValue.connect(self.get\_slider\_value)    if \_\_name\_\_ == '\_\_main\_\_':  app = QtWidgets.QApplication(sys.argv)  slider = SliderDialog()  progress = ProgressDialog()    progress.make\_connection(slider)   progress.show()  slider.show()  sys.exit(app.exec\_()) |
| --- |

**Листинг 2**

| *# -\*- coding: utf-8 -\*-*  *# Form implementation generated from reading ui file 'slider.ui'* *#* *# Created by: PyQt5 UI code generator 5.8.1* *#* *# WARNING! All changes made in this file will be lost!*  from PyQt5 import QtCore, QtGui, QtWidgets  class Ui\_SliderDialog(object):  def setupUi(self, SliderDialog):  SliderDialog.setObjectName("SliderDialog")  SliderDialog.resize(240, 45)  self.horizontalSlider = QtWidgets.QSlider(SliderDialog)  self.horizontalSlider.setGeometry(QtCore.QRect(20, 10, 160, 19))  self.horizontalSlider.setOrientation(QtCore.Qt.Horizontal)  self.horizontalSlider.setObjectName("horizontalSlider")   self.retranslateUi(SliderDialog)  QtCore.QMetaObject.connectSlotsByName(SliderDialog)   def retranslateUi(self, SliderDialog):  \_translate = QtCore.QCoreApplication.translate  SliderDialog.setWindowTitle(\_translate("SliderDialog", "Slider")) |
| --- |

**Листинг 3**

| *# -\*- coding: utf-8 -\*-*  *# Form implementation generated from reading ui file 'progress.ui'* *#* *# Created by: PyQt5 UI code generator 5.8.1* *#* *# WARNING! All changes made in this file will be lost!*  from PyQt5 import QtCore, QtGui, QtWidgets  class Ui\_ProgressDialog(object):  def setupUi(self, ProgressDialog):  ProgressDialog.setObjectName("ProgressDialog")  ProgressDialog.resize(237, 44)  self.progressBar = QtWidgets.QProgressBar(ProgressDialog)  self.progressBar.setGeometry(QtCore.QRect(10, 10, 211, 23))  self.progressBar.setProperty("value", 0)  self.progressBar.setObjectName("progressBar")   self.retranslateUi(ProgressDialog)  QtCore.QMetaObject.connectSlotsByName(ProgressDialog)   def retranslateUi(self, ProgressDialog):  \_translate = QtCore.QCoreApplication.translate  ProgressDialog.setWindowTitle(\_translate("ProgressDialog", "Progress")) |
| --- |

**Листинг 4**

| import sys from PyQt5 import QtCore, QtWidgets  class MyWindow(QtWidgets.QWidget):  def \_\_init\_\_(self, parent=None):  QtWidgets.QWidget.\_\_init\_\_(self, parent)  self.resize(300, 100)   def event(self, e):  if e.type() == QtCore.QEvent.KeyPress:  print("Нажата клавиша на клавиатуре")  print("Код:", e.key(), ", текст:", e.text())  elif e.type() == QtCore.QEvent.Close:  print("Окно закрыто")  elif e.type() == QtCore.QEvent.MouseButtonPress:  print("Клик мышью. Координаты:", e.x(), e.y())   *# Событие отправляется дальше*  return QtWidgets.QWidget.event(self, e)  if \_\_name\_\_ == '\_\_main\_\_':  app = QtWidgets.QApplication(sys.argv)  window = MyWindow()  window.show()  sys.exit(app.exec\_()) |
| --- |

**Листинг 5**

| *# ---------------- Графический интерфейс пользователя. PyQt5 ------------------* *# Приложение "Калькулятор". Класс основного окна калькулятора*  import sys import math   from PyQt5.QtWidgets import QWidget, QLineEdit, QGridLayout, QLayout from PyQt5 import QtCore, QtGui from PyQt5.QtCore import QCoreApplication  from button import Button   class Calculator(QWidget):   NUMDIGITBUTTONS = 10   def \_\_init\_\_(self, parent=None):  super().\_\_init\_\_(parent)  self.sumInMemory = 0.0  self.sumSoFar = 0.0  self.factorSoFar = 0.0  self.waitingForOperand = True  self.pendingAdditiveOperator = ''  self.pendingMultiplicativeOperator = ''   self.display = QLineEdit("0")  self.display.setReadOnly(True)  self.display.setAlignment(QtCore.Qt.AlignRight)  self.display.setMaxLength(15)   font = self.display.font()  font.setPointSize(font.pointSize() + 8)  self.display.setFont(font)   self.digitButtons = []  for i in range(self.NUMDIGITBUTTONS):  self.digitButtons.append(self.createButton(str(i), self.digitClicked))   self.pointButton = self.createButton(".", self.pointClicked)  self.changeSignButton = self.createButton(QCoreApplication.translate("Calculator","±"),  self.changeSignClicked)   self.backspaceButton = self.createButton("Backspace", self.backspaceClicked)  self.clearButton = self.createButton("Clear", self.clear)  self.clearAllButton = self.createButton("Clear All", self.clearAll)   self.clearMemoryButton = self.createButton("MC", self.clearMemory)  self.readMemoryButton = self.createButton("MR", self.readMemory)  self.setMemoryButton = self.createButton("MS", self.setMemory)  self.addToMemoryButton = self.createButton("M+", self.addToMemory)   self.divisionButton = self.createButton("÷", self.multiplicativeOperatorClicked)  self.timesButton = self.createButton("×", self.multiplicativeOperatorClicked)  self.minusButton = self.createButton("-", self.additiveOperatorClicked)  self.plusButton = self.createButton("+", self.additiveOperatorClicked)   self.squareRootButton = self.createButton("Sqrt", self.unaryOperatorClicked)  self.powerButton = self.createButton("x²", self.unaryOperatorClicked)  self.reciprocalButton = self.createButton("1/x", self.unaryOperatorClicked)  self.equalButton = self.createButton("=", self.equalClicked)   self.mainLayout = QGridLayout()   self.mainLayout.setSizeConstraint(QLayout.SetFixedSize)  self.mainLayout.addWidget(self.display, 0, 0, 1, 6)  self.mainLayout.addWidget(self.backspaceButton, 1, 0, 1, 2)  self.mainLayout.addWidget(self.clearButton, 1, 2, 1, 2)  self.mainLayout.addWidget(self.clearAllButton, 1, 4, 1, 2)   self.mainLayout.addWidget(self.clearMemoryButton, 2, 0)  self.mainLayout.addWidget(self.readMemoryButton, 3, 0)  self.mainLayout.addWidget(self.setMemoryButton, 4, 0)  self.mainLayout.addWidget(self.addToMemoryButton, 5, 0)   for i in range(1, self.NUMDIGITBUTTONS):  row = ((9 - i) / 3) + 2  column = ((i - 1) % 3) + 1  self.mainLayout.addWidget(self.digitButtons[i], row, column)   self.mainLayout.addWidget(self.digitButtons[0], 5, 1)   self.mainLayout.addWidget(self.pointButton, 5, 2)  self.mainLayout.addWidget(self.changeSignButton, 5, 3)   self.mainLayout.addWidget(self.divisionButton, 2, 4)  self.mainLayout.addWidget(self.timesButton, 3, 4)  self.mainLayout.addWidget(self.minusButton, 4, 4)  self.mainLayout.addWidget(self.plusButton, 5, 4)   self.mainLayout.addWidget(self.squareRootButton, 2, 5)  self.mainLayout.addWidget(self.powerButton, 3, 5)  self.mainLayout.addWidget(self.reciprocalButton, 4, 5)  self.mainLayout.addWidget(self.equalButton, 5, 5)   self.setLayout(self.mainLayout);  self.setWindowTitle("Calculator")   def digitClicked(self):  clickedButton = self.sender()  digitValue = int(clickedButton.text())  if self.display.text() == "0" and digitValue == 0.0:  return   if (self.waitingForOperand):  self.display.clear()  self.waitingForOperand = False   self.display.setText(self.display.text() + str(digitValue))   def unaryOperatorClicked(self):  clickedButton = self.sender()  clickedOperator = clickedButton.text()  operand = float(self.display.text())  result = 0.0   if clickedOperator == "Sqrt":  if operand < 0.0:  self.abortOperation()  return  result = math.sqrt(operand)  elif clickedOperator == "x²":  result = operand \*\* 2  elif clickedOperator == "1/x":  if (operand == 0.0):  self.abortOperation()  return  result = 1.0 / operand  self.display.setText(str(result))  self.waitingForOperand = True   def createButton(self, text, member):  button = Button(text, self)  button.clicked.connect(member)  return button   def additiveOperatorClicked(self):  clickedButton = self.sender()  clickedOperator = clickedButton.text()  operand = float(self.display.text())   if self.pendingMultiplicativeOperator:  if not self.calculate(operand, self.pendingMultiplicativeOperator):  self.abortOperation()  return  self.display.setText(str(self.factorSoFar))  operand = self.factorSoFar  self.factorSoFar = 0.0  self.pendingMultiplicativeOperator = ''   if self.pendingAdditiveOperator:  if not self.calculate(operand, self.pendingAdditiveOperator):  self.abortOperation()  return  self.display.setText(str(self.sumSoFar))  else:  self.sumSoFar = operand   self.pendingAdditiveOperator = clickedOperator  self.waitingForOperand = True   def multiplicativeOperatorClicked(self):  clickedButton = self.sender()  clickedOperator = clickedButton.text()  operand = float(self.display.text())   if self.pendingMultiplicativeOperator:  if not self.calculate(operand, self.pendingMultiplicativeOperator):  self.abortOperation()  return   self.display.setText(str(self.factorSoFar))  else:  self.factorSoFar = operand   self.pendingMultiplicativeOperator = clickedOperator  self.waitingForOperand = True   def equalClicked(self):  operand = float(self.display.text())   if self.pendingMultiplicativeOperator:  if not self.calculate(operand, self.pendingMultiplicativeOperator):  self.abortOperation()  return  operand = self.factorSoFar  self.factorSoFar = 0.0  self.pendingMultiplicativeOperator = ''    if self.pendingAdditiveOperator:  if not self.calculate(operand, self.pendingAdditiveOperator):  self.abortOperation()  return   self.pendingAdditiveOperator = ''  else:  self.sumSoFar = operand   self.display.setText(str(self.sumSoFar))  self.sumSoFar = 0.0  self.waitingForOperand = True   def pointClicked(self):  if self.waitingForOperand:  self.display.setText("0")  if '.' not in self.display.text():  self.display.setText(self.display.text() + ".")  self.waitingForOperand = False   def changeSignClicked(self):  text = self.display.text()  value = float(text)   if value > 0:  text = "-" + text  elif value < 0:  text = text[1:]  self.display.setText(text)   def backspaceClicked(self):  if self.waitingForOperand:  return  text = self.display.text()[:-1]  if not text:  text = "0"  self.waitingForOperand = True  self.display.setText(text)   def clear(self):  if self.waitingForOperand:  return  self.display.setText("0")  self.waitingForOperand = True   def clearAll(self):  self.sumSoFar = 0.0  self.factorSoFar = 0.0  self.pendingAdditiveOperator = ''  self.pendingMultiplicativeOperator = ''  self.display.setText("0")  self.waitingForOperand = True   def clearMemory(self):  self.sumInMemory = 0.0   def readMemory(self):  self.display.setText(str(self.sumInMemory))  self.waitingForOperand = True   def setMemory(self):  self.equalClicked()  self.sumInMemory = float(self.display.text())   def addToMemory(self):  self.equalClicked()  self.sumInMemory += float(self.display.text())   def abortOperation(self):  self.clearAll()  self.display.setText("####")   def calculate(self, rightOperand, pendingOperator):  if pendingOperator == "+":  self.sumSoFar += rightOperand  elif pendingOperator == "-":  self.sumSoFar -= rightOperand  elif pendingOperator == "×":  self.factorSoFar \*= rightOperand;  elif pendingOperator == "÷":  if rightOperand == 0.0:  return False  self.factorSoFar /= rightOperand   return True |
| --- |

**Листинг 6**

| *# ---------------- Графический интерфейс пользователя. PyQt5 ------------------* *# Взаимодействие SQLAlchemy и PyQt.*  *# Класс модели данных, связываемой с выборкой SQLAlchemy*   *# Original code:* *# (c) 2013 Mark Harviston, BSD License*  """ Qt data models that bind to SQLAlchemy queries """  from PyQt5.QtWidgets import QMessageBox from PyQt5.QtCore import QAbstractTableModel, QVariant, Qt   class AlchemicalTableModel(QAbstractTableModel):  """  A Qt Table Model that binds to a SQL Alchemy query   Example:  >>> model = AlchemicalTableModel(Session, [('Name', Entity.name)])  >>> table = QTableView(parent)  >>> table.setModel(model)  """   def \_\_init\_\_(self, session, query, columns):  super(AlchemicalTableModel, self).\_\_init\_\_()  self.session = session  self.fields = columns  self.query = query   self.results = None  self.count = None  self.sort = None  self.filter = None   self.refresh()   def headerData(self, col, orientation, role):  ''' Данные заголовков для указанной роли role и столбца col '''   if orientation == Qt.Horizontal and role == Qt.DisplayRole:  return QVariant(self.fields[col][0])  return QVariant()   def setFilter(self, filter):  """ Установка/очистка фильтра данных (для очистки filter=None). """  self.filter = filter  self.refresh()   def refresh(self):  """ Пересчет атрибутов self.results и self.count """  self.layoutAboutToBeChanged.emit()   q = self.query  if self.sort is not None:  order, col = self.sort  col = self.fields[col][1]  if order == Qt.DescendingOrder:  col = col.desc()  else:  col = None   if self.filter is not None:  q = q.filter(self.filter)   q = q.order\_by(col)   self.results = q.all()  self.count = q.count()  self.layoutChanged.emit()   def flags(self, index):  ''' Набор флагов для элемента с указанным индексом index '''  \_flags = Qt.ItemIsEnabled | Qt.ItemIsSelectable   if self.sort is not None:  order, col = self.sort  if self.fields[col][3].get('dnd', False) and index.column() == col:  \_flags |= Qt.ItemIsDragEnabled | Qt.ItemIsDropEnabled   if self.fields[index.column()][3].get('editable', False):  \_flags |= Qt.ItemIsEditable   return \_flags   def supportedDropActions(self):  ''' Поддерживаемые drop-действия при drag&drop операциях '''  return Qt.MoveAction   def dropMimeData(self, data, action, row, col, parent):  ''' Управляет данными data в drag&drop операциях при событии action '''  if action != Qt.MoveAction:  return  return False   def rowCount(self, parent):  ''' Количество строк '''  return self.count or 0   def columnCount(self, parent):  ''' Количество количество столбцов/полей '''  return len(self.fields)   def data(self, index, role):  ''' Получение данных из запроса '''  if not index.isValid():  return QVariant()  elif role not in (Qt.DisplayRole, Qt.EditRole):  return QVariant()  row = self.results[index.row()]  name = self.fields[index.column()][2]  return str(getattr(row, name))   def setData(self, index, value, role=None):  ''' Установка данных в связанном запросе '''  row = self.results[index.row()]  name = self.fields[index.column()][2]   try:  setattr(row, name, value)  self.session.commit()  except Exception as ex:  QMessageBox.critical(None, 'SQL Error', str(ex))  return False  else:  self.dataChanged.emit(index, index)  return True   def sort(self, col, order):  """ Сортировка таблицы по указанному номеру столбца. """  self.sort = order, col  self.refresh() |
| --- |

**Листинг 7**

| *# ---------------- Графический интерфейс пользователя. PyQt5 ------------------* *# Взаимодействие SQLAlchemy и PyQt.*  *# Пример использования SQLAlchemy + PyQt*  import sys  from sqlalchemy import create\_engine from sqlalchemy.ext.declarative import declarative\_base from sqlalchemy import Column, Integer, String from sqlalchemy.orm import sessionmaker  from PyQt5.QtWidgets import QTableView, QWidget, QApplication, QPushButton from PyQt5 import QtCore  from alchemical\_model import AlchemicalTableModel   *# ------------- Создание и настройка SQLAlchemy-механизмов ------------------* eng = create\_engine('sqlite:///:memory:')  Base = declarative\_base()   class Car(Base):  \_\_tablename\_\_ = "Cars"    Id = Column(Integer, primary\_key=True)  Name = Column(String)   Price = Column(Integer)    Base.metadata.bind = eng  Base.metadata.create\_all()    Session = sessionmaker(bind=eng) ses = Session()   ses.add\_all(  [Car(Id=1, Name='Audi', Price=52642),   Car(Id=2, Name='Mercedes', Price=57127),  Car(Id=3, Name='Skoda', Price=9000),  Car(Id=4, Name='Volvo', Price=29000),  Car(Id=5, Name='Bentley', Price=350000),  Car(Id=6, Name='Citroen', Price=21000),  Car(Id=7, Name='Hummer', Price=41400),  Car(Id=8, Name='Volkswagen', Price=21600)]) ses.commit()  rs = ses.query(Car).all()  *# Простая печать выборки для демонстрации* for car in rs:  print(car.Name, car.Price)   *# ---------------------- Передача данных в PyQt-классы -----------------------*  *# Создание QTable Model/View* model = AlchemicalTableModel (  ses,   ses.query(Car),   [ *# Список кортежей, описывающих столбцы:*  *# (заголовок, sqlalchemy-столбец, имя столбца, словарь доп. параметров).*  *# Если sqlalchemy-столбец имеет имя, например, Car.Name,*   *# тогда имя столбца в кортеже нужно указывать 'Name'.*  *# Car.Name будет использоваться для установки и сортировки данных,*  *# 'Name' - для получения данных.*  ('Car Name', Car.Name, 'Name', {'editable': True}),  ('Price', Car.Price, 'Price', {'editable': True}),  ])   def print\_data():  rs = ses.query(Car).all()  for car in rs:  print(car.Name, car.Price)   app = QApplication(sys.argv)  widget = QWidget() widget.setMinimumSize(QtCore.QSize(328, 228)) table = QTableView(widget)  *# Назначение модели данных виджету таблицы* table.setModel(model)  button = QPushButton(widget) button.clicked.connect(print\_data)  widget.show()  sys.exit(app.exec\_()) |
| --- |

**Листинг 8**

| *# ======================= Потоки и многозадачность ============================* *# ----------------------------- GUI и потоки ----------------------------------*  *# ----------------------- Класс "поискового движка" ---------------------------*   from urllib.request import urlopen from threading import Thread from queue import Queue import re  class Finder:  ''' Класс "поискового движка" '''   def \_\_init\_\_(self, text, res\_queue=None):  self.text = text  self.res\_queue = res\_queue  self.is\_alive = False   def search\_in\_url(self, url):  ''' Искать text по указанному URL  '''  print("-->>", url)  f = urlopen(url)  data = f.read().decode('utf-8')  pattern = "<p>.\*?{}.\*?</p>".format(self.text)  res = re.findall(pattern, data)  return res   def \_in\_urls(self, urls):  ''' Поиск текста по списку urls  '''  self.is\_alive = True  for url in urls:  *# На каждом шаге выполняется проверка - не был ли остановлен поиск*  if not self.is\_alive:  break  data = self.search\_in\_url(url)  *# Если есть очередь для результатов, они передаются в эту очередь*  if self.res\_queue is not None:  self.res\_queue.put((url, data))   self.is\_alive = False   if self.res\_queue is not None:  *# В результирующую очередь помещается None-флаг для сигнализации окончания поиска*  self.res\_queue.put(None)  self.res\_queue.join()   def search\_in\_urls(self, urls):  ''' Запуск поиска в отдельном потоке  '''  t = Thread(target=self.\_in\_urls, args=(urls, ))  t.daemon = True  t.start()   def stop\_search(self):  ''' Остановка поиска'''  self.is\_alive = False    if \_\_name\_\_ == '\_\_main\_\_':  urls = ['http://www.python.org/', 'https://habrahabr.ru/search/?q=%5Bpython%5D', 'https://python-scripts.com/']  res\_queue = Queue()  finder = Finder('python', res\_queue=res\_queue)  finder.search\_in\_urls(urls)  while True:  data = res\_queue.get()  if data is None:  break  print(data) |
| --- |

**Листинг 9**

| <?xml version="1.0" encoding="UTF-8"?> <ui version="4.0">  <class>FinderForm</class>  <widget class="QWidget" name="FinderForm">  <property name="geometry">  <rect>  <x>0</x>  <y>0</y>  <width>634</width>  <height>482</height>  </rect>  </property>  <property name="windowTitle">  <string>Поисковик</string>  </property>  <widget class="QPushButton" name="pushButton">  <property name="geometry">  <rect>  <x>10</x>  <y>220</y>  <width>111</width>  <height>23</height>  </rect>  </property>  <property name="text">  <string>Запустить поиск</string>  </property>  </widget>  <widget class="QPushButton" name="pushButton\_2">  <property name="enabled">  <bool>false</bool>  </property>  <property name="geometry">  <rect>  <x>130</x>  <y>220</y>  <width>111</width>  <height>23</height>  </rect>  </property>  <property name="text">  <string>Остановить поиск</string>  </property>  </widget>  <widget class="QLineEdit" name="lineEdit">  <property name="geometry">  <rect>  <x>10</x>  <y>30</y>  <width>611</width>  <height>20</height>  </rect>  </property>  <property name="text">  <string>python</string>  </property>  </widget>  <widget class="QLabel" name="label">  <property name="geometry">  <rect>  <x>10</x>  <y>10</y>  <width>111</width>  <height>16</height>  </rect>  </property>  <property name="text">  <string>Текст для поиска</string>  </property>  </widget>  <widget class="QLabel" name="label\_2">  <property name="geometry">  <rect>  <x>10</x>  <y>60</y>  <width>281</width>  <height>16</height>  </rect>  </property>  <property name="text">  <string>Список URL для просмотра (один URL на строке)</string>  </property>  </widget>  <widget class="QLabel" name="label\_3">  <property name="geometry">  <rect>  <x>10</x>  <y>280</y>  <width>281</width>  <height>16</height>  </rect>  </property>  <property name="text">  <string>Результаты поиска</string>  </property>  </widget>  <widget class="QProgressBar" name="progressBar">  <property name="geometry">  <rect>  <x>10</x>  <y>250</y>  <width>611</width>  <height>16</height>  </rect>  </property>  <property name="value">  <number>0</number>  </property>  </widget>  <widget class="QPlainTextEdit" name="plainTextEdit">  <property name="geometry">  <rect>  <x>10</x>  <y>300</y>  <width>611</width>  <height>171</height>  </rect>  </property>  </widget>  <widget class="QPlainTextEdit" name="plainTextEdit\_2">  <property name="geometry">  <rect>  <x>10</x>  <y>80</y>  <width>611</width>  <height>131</height>  </rect>  </property>  <property name="plainText">  <string>http://www.python.org/ https://habrahabr.ru/search/?q=%5Bpython%5D https://python-scripts.com/</string>  </property>  </widget>  </widget>  <resources/>  <connections/> </ui> |
| --- |

**Листинг 10**

| *# -\*- coding: utf-8 -\*-*  *# Form implementation generated from reading ui file 'search\_form.ui'* *#* *# Created by: PyQt5 UI code generator 5.8.1* *#* *# WARNING! All changes made in this file will be lost!*  from PyQt5 import QtCore, QtGui, QtWidgets  class Ui\_FinderForm(object):  def setupUi(self, FinderForm):  FinderForm.setObjectName("FinderForm")  FinderForm.resize(634, 482)  self.pushButton = QtWidgets.QPushButton(FinderForm)  self.pushButton.setGeometry(QtCore.QRect(10, 220, 111, 23))  self.pushButton.setObjectName("pushButton")  self.pushButton\_2 = QtWidgets.QPushButton(FinderForm)  self.pushButton\_2.setEnabled(False)  self.pushButton\_2.setGeometry(QtCore.QRect(130, 220, 111, 23))  self.pushButton\_2.setObjectName("pushButton\_2")  self.lineEdit = QtWidgets.QLineEdit(FinderForm)  self.lineEdit.setGeometry(QtCore.QRect(10, 30, 611, 20))  self.lineEdit.setObjectName("lineEdit")  self.label = QtWidgets.QLabel(FinderForm)  self.label.setGeometry(QtCore.QRect(10, 10, 111, 16))  self.label.setObjectName("label")  self.label\_2 = QtWidgets.QLabel(FinderForm)  self.label\_2.setGeometry(QtCore.QRect(10, 60, 281, 16))  self.label\_2.setObjectName("label\_2")  self.label\_3 = QtWidgets.QLabel(FinderForm)  self.label\_3.setGeometry(QtCore.QRect(10, 280, 281, 16))  self.label\_3.setObjectName("label\_3")  self.progressBar = QtWidgets.QProgressBar(FinderForm)  self.progressBar.setGeometry(QtCore.QRect(10, 250, 611, 16))  self.progressBar.setProperty("value", 0)  self.progressBar.setObjectName("progressBar")  self.plainTextEdit = QtWidgets.QPlainTextEdit(FinderForm)  self.plainTextEdit.setGeometry(QtCore.QRect(10, 300, 611, 171))  self.plainTextEdit.setObjectName("plainTextEdit")  self.plainTextEdit\_2 = QtWidgets.QPlainTextEdit(FinderForm)  self.plainTextEdit\_2.setGeometry(QtCore.QRect(10, 80, 611, 131))  self.plainTextEdit\_2.setObjectName("plainTextEdit\_2")   self.retranslateUi(FinderForm)  QtCore.QMetaObject.connectSlotsByName(FinderForm)   def retranslateUi(self, FinderForm):  \_translate = QtCore.QCoreApplication.translate  FinderForm.setWindowTitle(\_translate("FinderForm", "Поисковик"))  self.pushButton.setText(\_translate("FinderForm", "Запустить поиск"))  self.pushButton\_2.setText(\_translate("FinderForm", "Остановить поиск"))  self.lineEdit.setText(\_translate("FinderForm", "python"))  self.label.setText(\_translate("FinderForm", "Текст для поиска"))  self.label\_2.setText(\_translate("FinderForm", "Список URL для просмотра (один URL на строке)"))  self.label\_3.setText(\_translate("FinderForm", "Результаты поиска"))  self.plainTextEdit\_2.setPlainText(\_translate("FinderForm", "http://www.python.org/\n" "https://habrahabr.ru/search/?q=%5Bpython%5D\n" "https://python-scripts.com/")) |
| --- |

**Листинг 11**

| *# ======================= Потоки и многозадачность ============================* *# ----------------------------- GUI и потоки ----------------------------------*  *# ------------------- Основное приложение "Поисковика" ------------------------*  *# Библиотека Qt имеет специальный класс QThread, представляющий собой "обёртку"* *# над потоками, специфичными для конкретной платформы.*  *# При использовании QThread возможны два варианта:* *# - создать класс-наследник QObject со всеми необходимыми функциями, а затем*  *# выполнить метод moveToThread(), чтобы поместить экземпляра класса в поток* *# (предпочтительное решение)* *# - создать класс-наследник QThread и реализовать метод run (не универсальное решение)*  import sys  from PyQt5 import QtGui, QtWidgets from PyQt5.QtCore import Qt, QObject, QThread, pyqtSignal, pyqtSlot  from queue import Queue  from finder import Finder from search\_form import Ui\_FinderForm  class FinderMonitor(QObject):  ''' Класс-монитор, принимающий результаты поиска из очереди результатов  Данный класс будет помещён в отдельный поток QThread  '''  gotData = pyqtSignal(tuple)  finished = pyqtSignal(int)   def \_\_init\_\_(self, parent, urls, text):  super().\_\_init\_\_()  self.parent = parent  self.urls = urls  self.text = text  self.res\_queue = Queue()  self.finder = Finder(self.text, self.res\_queue)   def search\_text(self):  ''' Запуск поиска.  Поиск будет выполняться в отдельном потоке  '''  self.finder.search\_in\_urls(self.urls)  *# Текущая функция будет:*   *# - принимать результаты из очереди;*  *# - создавать сигналы для взаимодействия с GUI*  while True:  data = self.res\_queue.get()  if data is None:  break  self.gotData.emit(data)  self.res\_queue.task\_done()   self.res\_queue.task\_done()  self.finished.emit(0)   def stop(self):  self.finder.stop\_search()   class ProgressDialog(QtWidgets.QDialog):  ''' Класс GUI-формы "Поисковика"  '''  def \_\_init\_\_(self, parent=None):  QtWidgets.QDialog.\_\_init\_\_(self, parent)  self.ui = Ui\_FinderForm()  self.ui.setupUi(self)  self.ui.pushButton.clicked.connect(self.start\_search)  self.ui.pushButton\_2.clicked.connect(self.stop\_search)  self.monitor = None  self.is\_active = False  self.progress = 0  self.prog\_val = 1   @pyqtSlot(tuple)  def update\_results(self, data):  ''' Отображение результатов поиска  '''  self.ui.plainTextEdit.appendPlainText("++ {} ++".format(data[0]))  for text in data[1]:  self.ui.plainTextEdit.appendPlainText(" " + text)  self.ui.plainTextEdit.appendPlainText("")   @pyqtSlot()  def update\_progress(self):  ''' Изменение строки прогресса  '''  self.progress += self.prog\_val  self.ui.progressBar.setValue(self.progress)   def stop\_search(self):  ''' Остановка поиска  '''  if self.monitor is not None:  self.is\_active = False  self.monitor.stop()   def finished(self):  ''' Действия при завершении поиска  '''  self.is\_active = False  self.ui.pushButton\_2.setEnabled(False)  self.ui.pushButton.setEnabled(True)    def start\_search(self):  ''' Запуск поиска  '''  if not self.is\_active:  self.ui.plainTextEdit.clear()  self.is\_active = True  urls = self.ui.plainTextEdit\_2.toPlainText().split('\n')  text = self.ui.lineEdit.text()  *# Сброс значения прогресса и вычисление единицы прогресса*  self.progress = 0  self.prog\_val = 100 / len(urls)   self.monitor = FinderMonitor(self, urls, text)  self.monitor.gotData.connect(self.update\_results)  self.monitor.gotData.connect(self.update\_progress)   *# Создание потока и помещение объекта-монитора в этот поток*  self.thread = QThread()  self.monitor.moveToThread(self.thread)  self.ui.pushButton\_2.setEnabled(True)  self.ui.pushButton.setEnabled(False)   *# ---------- Важная часть - связывание сигналов и слотов ----------*  *# При запуске потока будет вызван метод search\_text*  self.thread.started.connect(self.monitor.search\_text)   *# При завершении поиска необходимо завершить поток и изменить GUI*   self.monitor.finished.connect(self.thread.quit)  self.monitor.finished.connect(self.finished)   *# Завершение процесса поиска по кнопке "Остановить"*  self.ui.pushButton\_2.clicked.connect(self.monitor.stop)   *# Запуск потока, который запустит self.monitor.search\_text*  self.thread.start()   if \_\_name\_\_ == '\_\_main\_\_':  app = QtWidgets.QApplication(sys.argv)  progress = ProgressDialog()  progress.show()  sys.exit(app.exec\_()) |
| --- |