Урок 7

Фреймворк PyQt

Особенности использования фреймворка PyQt.

PyQt, ее назначение и установка

- ***Что такое PyQt? Для чего используется и как устанавливается?***

Согласно официальной терминологии, PyQt представляет собой набор Python-связей для фреймворка Qt. Более понятное определение: это библиотека, которая предлагает разработчику инструменты, с помощью которых реализуются графические интерфейсы пользователя для приложений на Python. Речь идет именно о традиционных оконных интерфейсах. Библиотека PyQt реализована в качестве комплекта Python-модулей, который включает более 600 классов, а также 6 тысяч функций и методов. Это кроссплатформенный инструмент, поддерживаемый большинством ОС, в том числе Windows, Unix, MacOS.

По состоянию на 2006 год в мире насчитывалось более 200 коммерческих пользователей библиотеки PyQt, среди которых Sony Pictures, Pixar, Disney, Dreamworks. Сам фреймворк Qt демонстрирует устойчивый рост популярности, поскольку позволяет проектировать интерфейсы практически любого уровня сложности. С Qt работают бренды Adobe, Amazon, Cannon, Cisco Systems, Disney, Intel, Panasonic, Pioneer, Philips, Oracle, Nasa, Nokia, Samsung, Siemens, Sony, Xerox, Yamaha.

Для Windows

Официальный дистрибутив библиотеки доступен по ссылке https://riverbankcomputing.com в разделе **Download**. Для скачивания предлагается выбрать одну из двух поддерживаемых версий PyQt (4 или 5). Интерфейсы, реализованные на PyQt4 и PyQt5, будут отличаться в части программного кода. На представленном ресурсе доступны версии для 32- и 64-разрядных ОС. При установке библиотеки также необходимо учитывать версию Python. Для Python 3.4 (и более ранних) реализована возможность работы только с PyQt4. Чтобы использовать PyQt5, необходимо установить на вычислительном устройстве интерпретатор Python 3.5 (или более поздних версий).

Для Unix

Пользователям Unix-подобных ОС для установки библиотеки необходимо в терминале ввести следующую команду:

```
sudo apt-get install python3-pyqt4 pyqt4-dev-tools
```

Или:

```
sudo apt-get install python3-pyqt5 pyqt5-dev-tools
```

Способы создания приложений на PyQt

- ***Какие существуют способы создания приложений на PyQt и в чем их особенности?***

В PyQt реализован весь комплекс средств для разработки традиционных оконных графических оболочек — с панелями инструментов, кнопками, текстовыми полями, флажками, выпадающими списками и т.д. При разработке интерфейса применяются два основных подхода: написание кода вручную и использование свободно-распространяемой среды разработки **QtDesigner**.

Вручную

Этот подход позволяет «прощупать изнутри» принципы построения интерфейса. Писать код вручную необходимо, когда структура графической оболочки не определена заранее и может изменяться в зависимости от действий пользователя. Подход трудоемкий, поскольку надо знать особенности добавления элементов интерфейса в окна. Разработчику приходится вручную размещать каждый элемент управления (виджет) и настраивать его местоположение, размеры, стилизацию и другое. Но чтобы разбираться в PyQt-классах, надо уметь проектировать интерфейсы без помощи программных средств.

Рассмотрим простейший пример проектирования главного окна интерфейса (с кнопкой на панели инструментов) без применения программных средств:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# Импорт модулей
import sys
from PyQt5.QtWidgets import QMainWindow, QAction, QApplication

class CatalogMainWindow(QMainWindow):
    def __init__(self, parent=None):
        QMainWindow.__init__(self, parent)

        # Структура главного окна
        # Создаем меню
        self.menu_bar = self.menuBar()

        # Создаем блоки меню
        # Блок меню 'Учет движения товаров'
        self.gma_menu = self.menu_bar.addMenu('Учет движения товаров')
        self.ro_open_btn = QAction(self)
        self.ro_open_btn.setText('Приходный ордер')
        self.wo_open_btn = QAction(self)
        self.wo_open_btn.setText('Расходный ордер')
        self.gma_menu.addAction(self.ro_open_btn)
        self.gma_menu.addAction(self.wo_open_btn)

# Отобразить главное окно
if __name__ == "__main__":
    app = QApplication(sys.argv)
    CMW = CatalogMainWindow()
    CMW.setWindowTitle('Складской учет')
    CMW.setFixedSize(1200, 800)
    CMW.show()
    sys.exit(app.exec_())
```

Использование QtDesigner

Во всех остальных случаях, в том числе при проектировании сложных интерфейсов, на первый план выходит среда разработки **QtDesigner**. Это программное средство, позволяющее проектировать графические интерфейсы в режиме «что видишь, то и получаешь» (what-you-see-is-what-you-get, **WYSIWYG**). То есть разработчик выбирает виджеты и размещает их в рабочей области. **QtDesigner** поставляется вместе с PyQt и не требует отдельной установки.

Созданный с помощью **QtDesigner** интерфейс сохраняется в файле с расширением **.ui**, который содержит описание структуры интерфейса в XML-формате. Поэтому подключить этот файл с

помощью привычного импорта (команда **import**) невозможно — необходимо использовать модуль **uic** библиотеки PyQt. Импорт модуля:

```
from PyQt5 import uic
```

Для подключения файла необходимо воспользоваться функцией **loadUi()**, формат которой выглядит следующим образом:

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

Второй параметр функции содержит указание на экземпляр окна. С помощью этого указателя обеспечивается доступ к его виджетам и реализуется возможность назначить обработчики сигналов.

```
from PyQt5.QtWidgets import QWidget, qApp, QApplication
from PyQt5 import uic
import sys

class Window(QWidget):
    def __init__(self, parent=None):
        QWidget.__init__(self, parent)
        uic.loadUi('window_form.ui', self)
        self.btn_quit.clicked.connect(qApp.quit)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = Window()
    win.show()
    sys.exit(app.exec_())
```

Вместо подключения ui-файла можно генерировать на его основе файл с Python-кодом и импортировать его в программу. Эта задача решается с помощью утилиты **pyuic5**, которая входит в библиотеку PyQt5. Для запуска утилиты необходимо выполнить следующую команду:

```
pyuic5 имя_исходного_ui_файла -o имя_конечного_py_файла
например,
pyuic5 ui_file.ui -o py_form.py
```

Конечный файл **py_form.py**, который импортируется в приложение с помощью инструкции **import**, содержит класс **Ui_WindowForm**. Это имя задается разработчиком в процессе работы с программой **QtDesigner**. Метод **setupUi()** этого класса позволяет выполнить привязку созданного интерфейса к окну.

Пример:

```
import sys
from PyQt5.QtWidgets import QWidget, qApp
# Импортируем сгенерированный файл py_form
import py_form

class Window(QWidget):
    def __init__(self, parent=None):
        QWidget.__init__(self, parent)
        # Создаем экземпляр класса формы (экземпляр интерфейса)
        # из файла py_form
        self.ui = py_form.Ui_WindowForm()
        # Связываем экземпляр интерфейса с данным окном
        self.ui.setupUi(self)
        # Устанавливаем для кнопки обработчик
        self.ui.btn_quit.clicked.connect(qApp.quit)
```

Преимущества PyQt

- ***В чем преимущества библиотеки PyQt?***

Помимо PyQt для разработки графических интерфейсов может применяться библиотека **Tkinter**, входящая в стандартную библиотеку Python. PyQt имеет ряд преимуществ перед аналогом:

- Возможность не только реализовывать графические оболочки, но и обеспечивать взаимодействия с базами данных.
- Мощная библиотека классов-компонентов интерфейса, возможность разработки пользовательских компонентов.
- Структурированная документация, охватывающая весь функционал библиотеки.
- Собственная среда разработки **QtDesigner**, упрощающая создание интерфейсов различной сложности.

Назначение основных PyQt-классов

- ***Для чего используется каждый из следующих PyQt-модулей: QtCore, QtGui, QtMultimedia, QtNetwork, QtSql, QtTest, QtWidgets, QPrintSupport, uic?***

QtCore. Этот модуль содержит классы, которые не связаны непосредственно с реализацией элементов графического интерфейса. Они реализуют систему сигналов и слотов, кроссплатформенные абстракции для Unicode, потоки, регулярные выражения и т.д.

QtGui. Модуль, предоставляющий базовые классы компонентов графической оболочки (элементов управления).

QtMultimedia. Предоставляет низкоуровневую функциональность для реализации мультимедийных возможностей.

QtNetwork. Модуль, предоставляющий классы для упрощения сетевого программирования — например, для реализации клиент-серверного взаимодействия с помощью UDP и TCP.

QtSql. Модуль, содержащий классы для реализации взаимодействия с реляционными базами данных.

QtTest. Модуль, предоставляющий классы для реализации unit-тестирования приложений и библиотек.

QtWidgets. Дополняет класс **QtGui** «строительным материалом» в виде виджетов для реализации компонентов интерфейса.

QPrintSupport. Содержит классы, обеспечивающие взаимодействие с принтером через библиотеку PyQt.

uic. Преобразует XML-файлы, созданные в QtDesigner, в файлы с Python-кодом при проектировании графических интерфейсов.

Различия между PyQt4 и PyQt5

- ***В чем различия между PyQt4 и PyQt5?***

При разработке приложения с графическим интерфейсом необходимо принимать во внимание, что версии 4 и 5 библиотеки PyQt несовместимы. Поэтому программные средства, реализованные с помощью PyQt4, требуют существенной переработки для использования с PyQt5.

Перечислим особенности версии PyQt5:

- ***Реорганизованы модули.*** Часть модулей была исключена (например, QtScript), а часть — разделена на подмодули. В частности, модуль **QtGui** разделен на **QtGui**, **QtPrintSupport**, **QtWidgets**.
- ***Реализован новый стиль обработки сигналов и слотов.*** Такие вызовы, как **SIGNAL()** и **SLOT()**, больше не поддерживаются.
- ***Реализована поддержка нескольких новых модулей***: **QtBluetooth** (взаимодействие с bluetooth-устройствами средствами Qt), **QtPositioning** (определение позиций с использованием различных источников, включая спутник или wi-fi).
- ***Прекращена поддержка версий Python младше 2.6***.

Назначение обработчика для сигнала

- ***Как назначить обработчик для сигнала и передать в него данные?***

Когда пользователь взаимодействует с окном программы, происходят события — например, действия самого пользователя или возникновение условия в самой системе. В ответ на событие генерируется определенный сигнал, который можно рассматривать как представление системного события в библиотеке PyQt.

Для обработки сигнала необходимо связать его с определенной функцией или методом класса, который будет вызван при наступлении события и станет его обработчиком. Каждый сигнал имеет обозначение: например, для сигнала нажатия кнопки — это оператор **clicked**, а для сигнала установки флажка — оператор **checked**.

Шаблоны привязки обработчика к сигналу:

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

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

Обработчиком может быть:

- ссылка на пользовательскую функцию,
- ссылка на метод класса,
- ссылка на экземпляр класса, где определен метод **__call__()**,
- lambda-функция,
- ссылка на слот класса — например, приведенный ниже фрагмент назначает функцию-обработчик **on_checked_chck_box** для сигнала **checked** элемента управления «флажок»:

```
chck_box.checked.connect(on_checked_chck_box)
```

Пример использования различных типов обработчиков:

```
import sys
from PyQt5 import QtWidgets

def on_checked():
    print("Флажок установлен. Функция on_checked")

class MyClass():
    def __init__(self, y=0):
        self.y = y

    def __call__(self):
        print("Флажок установлен. Метод MyClass.__call__()")
        print("x = ", self.x)

    def on_checked(self):
        print("Флажок установлен. Метод MyClass.on_checked()")

obj = MyClass()
app = QtWidgets.QApplication(sys.argv)
chck_box = QtWidgets.QCheckBox("Установите флажок")

# В качестве обработчика назначается функция
chck_box.checked.connect(on_checked)

# В качестве обработчика назначается метод объекта
chck_box.checked.connect(obj.on_checked)

# В качестве обработчика назначается класс
chck_box.checked.connect(MyClass(10))

# В качестве обработчика назначается lambda-функция
chck_box.checked.connect(lambda: MyClass(5)())

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

Результат выполнения программы в консоли при нажатии на кнопку:

```
Флажок установлен. Функция on_checked
Флажок установлен. Метод MyClass.on_checked()
Флажок установлен. Метод MyClass.__call__()
x = 10
Флажок установлен. Метод MyClass.__call__()
x = 5
```

Создание пользовательского сигнала

- ***Как создать пользовательский сигнал?***

При разработке программы может потребоваться программно генерировать сигнал. Создадим сигнал с помощью функции **pyqtSignal()** модуля **QtCore**:

```
my_signal = pyqtSignal(str, str)
```

В этот сигнал передаются строковые данные. Теперь определим для него обработчик:

```
my_signal.connect(self.on_my_signal)
```

Создадим обработчик сигнала, который будет взаимодействовать с переданными в сигнал текстовые данные:

```
def on_my_signal(self, str_1, str_2):
    print('Обработан пользовательский сигнал btn_signal')
    print('x = ' + str_1 + ', ' + 'y = ' + str_2
```

Теперь необходимо генерировать сигнал. Для этого в нужном месте программного кода приложения нужно разместить инструкцию:

```
mysignal.emit('text_1', 'text_2')
```

Передача данных в обработчик

- ***Как передать данные в обработчик?***

При назначении обработчика с помощью метода **connect()** в качестве его аргумента передается ссылка на функцию — без параметров самой функции. Чтобы передать параметры в обработчик, можно воспользоваться одним из следующих подходов:

1. Создать «обертку» в виде lambda-функции для реализации вызова обработчика с набором параметров:

```
self.chck_box.checked.connect(lambda: self.on_checked_chck_box(5))
```

2. Передать ссылку на экземпляр класса, содержащий определение метода **call()**. При этом передаваемое значение указывается как параметр конструктора класса:

```
class MyClass():
    def __init__(self, z=0):
        self.z = z

    def __call__(self):
        print("z = ", self.z)
...

self.chck_box.checked.connect(MyClass(10))
```

3. Передать ссылку на функцию-обработчик и необходимые параметры в функцию **partial** из модуля **functools**:

```
from functools import partial
self.chck_box.checked.connect(partial(self.on_checked_chck_box, 5))
```

Взаимодействие с базами данных из PyQt

- ***Можно ли взаимодействовать с базами данных из PyQt?***

В библиотеку PyQt входят средства для работы с различными СУБД: SQLite, MySQL, PostgreSQL, Oracle. При не надо устанавливать дополнительные Python-библиотеки. В PyQt реализован механизм выполнения любых SQL-запросов и обработки результатов. Можно использовать особые модели для вывода содержимого таблиц и результатов запросов в любой из компонентов-представлений — например, **QTableView()**.

Особенности работы с различными СУБД средствами PyQt

- ***Каковы особенности работы с различными СУБД через PyQt?***

Класс QSqlDatabase

Для работы с базами данных в PyQt используются классы модуля **QtSql**. В частности, класс **QSqlDatabase** служит для установки соединения с базой данных. Шаблон инструкции для подключения к базе данных из PyQt:

```
имя_соединения = QtSql.QSqlDatabase.addDatabase('формат_базы_данных')
```

Реализована поддержка следующих форматов баз данных:

- **QMYSQL** и **QMYSQL3** (**MySQL**);
- **QODBC** и **QODBC3** (**QODBC**);
- **QPSQL** и **QPSQL7** (**PostgreSQL**);
- **QSQLITE** (**SQLite3**).

Пример:

```
conn = QtSql.QSqlDatabase.addDatabase('QSQLITE')
```

Для файловых баз данных предусмотрен метод **setDatabaseName** объекта соединения с указанием пути до файла:

```
conn.setDatabaseName('C://programs//my_db.sqlite')
```

Для клиент-серверных баз данных есть методы для подключения к серверу:

```
conn.setDatabaseName("имя_базы_данных")
conn.setHostName("имя_хоста")
conn.setUserName("имя_пользователя")
conn.setPassword("пароль")
conn.setPort("порт")
```

Пример:

```
conn.setDatabaseName("my_db")
conn.setHostName("127.0.0.1")
conn.setUserName("postgres")
conn.setPassword("12345")
conn.setPort("5432")
```

Для открытия и закрытия базы данных используются методы **open()** и **close()** объекта соединения. При открытии БД можно реализовать структуру ветвления:

```
if con.open():
    # Выполняем операции с базой данных
else:
    # Выводим текст ошибки
    print(con.lastError().text())
```

Класс QSqlQuery

QSqlQuery является конструктором для формирования SQL-запроса. Рассмотрим пример создания экземпляра данного класса с последующим запуском выполнения запроса и представлением результата:

```
query = QtSql.QSqlQuery()
query.exec("select * from my_table")
results = []
if query.isActive():
    query.first()
    while query.isValid():
        results.append(query.value('item_name'))
        query.next()
    for el in results:
        print(el)
```

В этом фрагменте кода в первом условии проверяется, находится ли запрос в активном состоянии, после чего с помощью метода **first()** устанавливается указатель на первую запись результата. Пока запрос является валидным, то есть не «пройдены» все записи результата, в список заносится

значение, соответствующее полю **item_name** очередной «пройденной» записи. Далее выполняется перебор элементов списка с выводом их содержимого.

Класс QSqlQueryModel

Позволяет сформировать модель на основе данных, извлеченных с помощью SQL-запроса. Применяется, когда необходимо реализовать вывод данных без редактирования. Пример создания экземпляра класса **QSqlQueryModel**:

```
query_model = QtSql.QSqlQueryModel()
```

Для привязки запроса к модели применяется метод **setQuery()**, аргументом которого является текст SQL-запроса:

```
# query_model.setQuery('Код запроса')
```

Чтобы выводить содержимое модели, может применяться компонент-представление **QTableView**, предназначенный для отображения данных моделей в табличной форме:

```
query_table = QtWidgets.QTableView()
query_model = QtSql.QSqlQueryModel(parent=query_table)
query_model.setQuery('select * from users_table')
query_table.setModel(query_model)
```

Класс QSqlTableModel

Позволяет связать модель с таблицей, обеспечивает возможность редактирования данных таблицы базы данных:

```
table_model = QtSql.QSqlTableModel()
table_model.setModel('vendors_table')
```

Для считывания данных из таблицы в модель используется метод **select()**. Он возвращает значение **True** при успешном считывании данных:

```
table_model.select()
```

Практическое задание

Продолжаем работу над десктопным приложением с графическим интерфейсом пользователя, которое помогает вести складской учет.

1. Создать главное окно программы, реализовать для него меню с шестью пунктами, верхний и центральный виджеты. Каждый из пунктов должен соответствовать одной из шести таблиц. SQL-запросы их создания были написаны в практическом задании к предыдущему уроку. Например, это могут быть пункты меню «Категории товаров», «Единицы измерения товаров»,

«Должности» и так далее. Сделать так, чтобы пользователь не мог выбирать пункты меню (метод **setEnabled()**).

2. Создать верхний виджет программы с фреймом, в котором расположить три виджета:
    a. надпись — путь к базе данных;
    b. текстовое поле для отображения пути к БД — сделать его недоступным для редактирования;
    c. кнопка, открывающая диалог выбора файла sqlite-базы данных.
3. В центральном виджете программы реализовать виджет с табличным компонентом-представлением (**QTableView**) и двумя кнопками (для добавления и удаления записи таблицы БД). В компоненте-представлении будут отображаться модели данных, соответствующие каждой из таблиц. Пользователь сможет добавлять и удалять записи.
4. К кнопке, открывающей диалог выбора файла sqlite-базы данных, привязать обработчик нажатия. Он должен открывать окно диалога для выбора файла БД (класс **QFileDialog**). При этом полный путь до базы данных должен сохраняться в текстовом поле верхнего виджета программы. Должно устанавливаться соединение с базой данных — для этого используйте фрагмент кода из практического задания с предыдущего урока. Сделать доступными меню главного окна программы.
5. Для каждого из пунктов меню реализовать обработчики нажатия. Их код реализовать в отдельном модуле, который должен импортироваться в главное окно программы. В каждый из обработчиков поместить фрагменты программного кода (из практического задания предыдущего урока), отвечающие за создание соответствующих таблиц БД. В обработчике должны создаваться соответствующие модели-таблицы (модели на основе таблиц БД). Для этого применяется PyQt-класс **QSqlTableModel**.
6. К каждой из кнопок добавления и удаления записей привязать обработчики этих событий. В компоненте-представлении **QTableView** должны отображаться изменения — таблицы с добавленными записями или без удаленных.

Дополнительные материалы

1. python, pyqt, sql.
2. QtSql Module.
3. Qt Documentation.

Используемая литература

1. Введение в PyQt5.
2. Python PyQt5.
3. Н. Прохоренок, В. Дронов. Python 3 и PyQt5. — 2016.