Руководство по работе с библиотекой Qt с использованием Python 3

Установка необходимых компонентов для работы

1. Установить *Python* версии не менее 3.5.

   В операционных системах семейства *Windows* процесс установки сводится к скачиванию установщика с сайта *https://www.python.org* и его запуска.

   В операционных система семейства *Unix* процесс установки сводится к запуску пакетного менеджера с выбором необходимого пакета.

   Пример установки Python 3 в Ubuntu:

   ```
   sudo apt-get install python3
   ```

   По умолчанию, путь к интерпретатору *Python 3* при установке на любой операционной системе добавляется к переменной среды *PATH*. В *Windows* для запуска интепретатора достаточно набрать в консоли:

   ```
   python путь_до_скрипта
   ```

   В *Unix* системах, как правило, может быть уже установлена более ранняя версия интерпретатора, поэтому для корректного запуска *3-ей ветки Python* рекомендуется выполнить в терминале команду:

   ```
   python3 путь_до_скрипта
   ```

2. В *Python* установка сторонних библиотек (или пакетов) происходит с помощью пакетного менеджера **pip**.

   По умолчанию, в *Windows* при установке самого интерпретатора устанвливается и пакетный менеджер.

   В *Unix* системах установка пакетного менеджера *pip* происходит отдельно.

   Пример установки pip в Ubuntu:

   ```
   sudo apt-get install python3-pip
   ```

   Установка сторонних пакетов (библиотек) сводится к следующей команде в терминале *Unix*/консоли *Windows*:

   ```
   pip3 install название_стороннего_пакета
   ```

   pip автоматически найдет, скачает и установит необходимый пакет из сети интернет.

3. Для установки библиотеки Qt следует выполнить следующую команду:

   ```
   pip3 install PyQt5
   ```

Основы работы с Qt

Библиотека *Qt* является кросплатформенной и предназначена для написания графических приложений. Она инкапсулирует в себе все основные понятия любой из операционных графических систем: окна, фреймы, модальные диалоги, кнопки, текстовые поля, панели и т.д. Ядро библиотеки и всех ее компонентов написаны на языке программирования *C++*. Библиотека не является монолитной и разбита на несколько основных модулей, которые содержат классы для работы с графикой, сетью, файлами, аудио/видео, потоками ОС и т.д.

Пакет *PyQt5* является портом для работы с *Qt* на языке *Python*. Ввиду этого, пакет наследует основные названия модулей библиотеки и их классов. Однако, в отличие от процесса работы на *C++* под *Qt*, работа на языке *Python* с использованием пакета *PyQt5* избавляет программиста от низкоуровневых особенностей *C++* и ускоряет процесс написания программного кода. В пользу *PyQt5* стоит сказать, что скорость понимания абстракций графического интерфейса, написанных на языке *Python*, бывает более высокой, нежели на *C++*.

Как правило, процесс ознакомления с *PyQt5* тесно связан с чтением документации по тому или иному классу. Однако, стоит заметить, что все описания классов модулей *Qt* представлены на языке *C++*. Но такие понятия, как классы, его атрибуты и методы в языке *C++* интуитивно легко перекладываются на язык *Python*. Тем самым, любые описания и примеры использования того или иного атрибута/метода на языке *C++* в документации *Qt* справедливы и для *Python*, где любые упоминания про указатели и ссылки просто опускаются при работы с *PyQt5*.

На данный момент библиотека *Qt* развивается в двух направлениях: **Qt Widgets** и **Qt Quick**. *Qt Widgets* является фундаментальным и базовым направлением библиотеки. Данный модуль существует с момента существования платформы и направлен на создания графических приложений в стиле объектно-ориентированного подхода, где любой компонент интерфейса представлен объектом класса, тем самым организуя весть графический интерфейс пользователя в иерархию объектов. *Qt Quick* является более современным ответвлением платформы *Qt*. Данное направление вводит новые высокоуровневые абстракции вроде *машины конечного автомата* и вносит принципы *реактивного программирования*. Также, *Qt Quick* предлагает идею декларативного описания пользовательского интерфейса средствами *JavaScript подобного языка QML*.

В данном руководстве описываются базовые принципы работы с *Qt Widgets* совместно с пакетом *PyQt5*.

Ссылка на документацию Qt 5: *http://doc.qt.io/qt-5/qtwidgets-index.html*

Ссылка на документацию PyQt5: *http://pyqt.sourceforge.net/Docs/PyQt5*

1. Основной цикл обработки событий

Приложение *Qt* построено на основе бесконечного цикла, который обратывает события: события операционной системы, события пользоваля приложения (поведение мыши, использование клавиатуры). Для запуска минимального приложения, следует передать циклу контекст, в котором начнут обрабатываться события. Базовым контекстом является понятие **Виджет**, которое представлено классом **QWidget**.

Виджет - это аналог *окна*, которое как может иметь рамку, кнопки сворачивания и закрытия, а может их и не иметь. С точки зрения операционных систем, это именно контекст, где операционная система может реагировать на события пользователя. В *Qt* виджеты могут содержать иерархию из других виджетов. При этом, разработчик никак не ограничен тем, как будет выглядеть виджет. Им могут быть как стандартное текстовое поле, кнопка, текстовая метка, так и сложный графический объект со своим стилем прорисовки.

Минимальное приложение *Qt* состоит из определения класса наследника *QWidget*, создания его экземпляра и запуска бесконечного цикла обработки событий.

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


class MyWidget(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self._initUI()

    def _initUI(self):
        self.label = QLabel('Hello', self)

        self.layout = QVBoxLayout(self)
        self.layout.addWidget(self.label)

        self.setLayout(self.layout)
        self.setGeometry(0, 0, 100, 100)
        self.show()

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

2. Виджеты

Как было сказано, виджет представлен классом *QWidget*. Для создания виджета следует определить класс-наследник *QWidget*. В конструкторе класса следует добавить последний аргумент по умолчанию *parent* и вызвать инструкцию:

```
super().__init__(parent)
```

Параметр *parent* указывает на родительский виджет описываемого. Если описываемый является корневым, *parent = None*. Стоит сказать, что создание любого другого дочернего виджета, разметки должно просходить с передачей последним аргументом родительского виджета *parent*. Наиболее часто, родителем оказывается описываемый виджет *self*.

Часто, виджетам устанавливают ту или иную разметку, по правилам которой внутри располагаются другие виджеты. Наиболее используемыми классами разметки являются *QVBoxLayout* (вертикальная разметка) и *QHBoxLayout* (горизонтальная разметка). Для

добавления дочерних виджетов в разметку предназначен метод *addWidget(QWidget)*. Чтобы установить виджету ту или иную разметку используется метод *setLayout(QLayout)*.

Хорошим стилем считается создание всех дочерних разметок/виджетов описываемого в теле контструктора класса. Как правило, создается *приватный метод*, например *_initUI*, и вызывается в конструкторе *__init__*.

Qt предоставляет набор удобных стандартных виджетов, поведение которых также можно изменить определением нового класса-наследника.

2.1 Текстовая метка: QLabel

Пример использования:

```
from PyQt5.QtWidgets import QLabel

...

self.label = QLabel('Text', self)
```

Часто используемые методы:

1. *text() → string*: возвращает текст метки.
2. *setText(string)*: устанавливает текст метки.

2.2 Текстовое поле: QLineEdit

Пример использования

```
from PyQt5.QtWidgets import QLineEdit

...

self.edit = QLineEdit('текст в поле ввода', self)
```

Часто используемые методы:

1. *text() → string*: возвращает текст поля ввода.
2. *setText(string)*: устанавливает текст поля ввода.

2.3 Кнопка: QPushButton

Пример использования

```
from PyQt5.QtWidgets import QPushButton

...

self.button = QPushButton('Текст на кнопке', self)
```

Часто используемые методы:

1. *setText(string)*: устанавливает текст на кнопке.

2.4 Многострочное поле ввода: QTextEdit

Пример использования

```
from PyQt5.QtWidgets import QTextEdit

...

self.muli_edit = QTextEdit(self)
```

Часто используемые методы:

1. *setText(string)*: устанавливает текст в поле ввода.
2. *toPlainText() → string*: возвращает обычный текст.
3. *toHtml() → string*: возвращает текст в формате *HTML*.

2.5 Слайдер: QSlider

Пример использования

```
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QSlider

...

self.slider = QSlider(Qt.Horizontal, self)
```

Часто используемые методы:

1. *setTickInterval(int)*: устанавливает шаг.
2. *tickInterval() → int*: возвращает шаг.
3. *setMinimum(int)*: устанавливает минимальное значение.
4. *setMaximum(int)*: устанавливает максимальное значение.

2.6 Чек-бокс: QCheckBox

Пример использования

```
from PyQt5.QtWidgets import QCheckBox

...

self.checkbox = QCheckBox('Метка чек-бокса', self)
```

Часто используемые методы:

1. *toogle()*: ставит/снимает чек-бокс.

3. Взаимодействие с пользователем: введение в сигналы/слоты

В библиотеке *Qt* любые из классов, которые являются наследниками класса *QObject* могут участвовать в механизме сигналов/слотов. Практически все классы *Qt* являются наследниками *QObject*, в том числе и виджеты.

Механизм сигналов/слотов - является базовым понятием в обработке событий. Событиями могут выступать: действия пользователя графического интерфейса, события операционной системы и т.д. Само понятие события в *Qt* носит название **Сигнал**. Реакция на *сигнал* это всегда какая-либо простая функция, которая носит название **Слот**.

Как правило, дочерние компоненты описываемого виджета генерируют какие-либо сигналы. Например, сигнал клика по кнопке. Для реакции на данное событие создается метод внтури описываемого виджета. Стоит упомянуть, что сигнал может передавать любую порцию информации: *число, строка, список и т.д.*. Если сигнал отправляет какие-либо данные, то в методе на реакцию данного сигнала должен передаваться аргумент(ы) для обработки передаваемой информации.

Пример использования механизма сигналов/слотов:

```
import sys
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import (QWidget, QLCDNumber, QSlider,
                             QVBoxLayout, QApplication)


class Example(QWidget):

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

        self.initUI()

    def initUI(self):

        lcd = QLCDNumber(self)
        sld = QSlider(Qt.Horizontal, self)

        vbox = QVBoxLayout()
        vbox.addWidget(lcd)
        vbox.addWidget(sld)

        self.setLayout(vbox)
        sld.valueChanged.connect(lcd.display)

        self.setGeometry(300, 300, 250, 150)
        self.show()


if __name__ == '__main__':

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

Как видно из примера, виджет *sld* генерирует сигнал *valueChanged*, информирующий об изменении позиции слайдера. В свою очередь, данный сигнал связан с методом *display* виджета *lcd*. В данном случае, *valueChanged* является сигналом и отсылает значение

типа *int*, а *display* является методом-сигналом, в который передается значение типа *int*. Связывание сигнала и слота происходит с помощью метода сигнала *connect*, который имеется у любого сигнала *Qt*.

Для определения слота, следует создать метод у класса описываемого виджета и привязать нужный сигнал к новому слоту с помощью метода сигнала *connect*.

Пример определения слота:

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


class MyWidget(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self._initUI()

    def _initUI(self):
        self.label = QLabel('Push the Button', self)
        self.button = QPushButton('Push', self)
        self.button.clicked.connect(self._handleClickButton)

        self.layout = QVBoxLayout(self)
        self.layout.addWidget(self.label)
        self.layout.addWidget(self.button)

        self.setLayout(self.layout)
        self.show()

    def _handleClickButton(self):
        self.label.setText('Push Done.')

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

Чтобы определить слот, который реагирует на сигналы, отправляющие какую-либо информацию, следует лишь добавить аргумент(ы).

Пример определения слота на сигнал, передающий значение типа int:

```
import sys
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import (QWidget, QApplication, QVBoxLayout,
                             QLabel, QSlider)


class MyWidget(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self._initUI()

    def _initUI(self):
        self.label = QLabel('Change Slider', self)
        self.slider = QSlider(Qt.Horizontal, self)
        self.slider.valueChanged.connect(self._handleChangeSlider)

        self.layout = QVBoxLayout(self)
        self.layout.addWidget(self.label)
        self.layout.addWidget(self.slider)

        self.setLayout(self.layout)
        self.show()

    def _handleChangeSlider(self, value):
        self.label.setText(str(value))

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

Виджет *slider* генерирует сигнал *valueChanged* при изменении слайдера. В свою очередь, данный сигнал связан с слотом/методом *_handleChangeSlider*, который принимает аргумент *value* типа *int*. При любом изменении слайдера вызывается метод *_handleChangeSlider*, который устанавливает текст метке *label* на значение ползунка слайдера. Стоит сказать, что метод метки *label.setText* принимает строковое значение, поэтому значение, отправляемое сигналом, числового типа *int* явно приводится к строковому типу *str*.

В документации библиотеки *Qt* к тому или иному классу виджета все сигналы находятся в секции *Signals*. Особое внимание стоит обращать на типы данных, которые возвращают сигналы.