**Аннотация**

*В уроке рассказывается о графическом интерфейсе и разных способах его реализации, а также начинается знакомство с библиотекой PyQT5.*

*Разбирается настройка окружения и примеры работы с основными элементами интерфейса.*

**Графический интерфейс**

Раньше большинство ваших программ запускались и выполнялись из консоли, то есть и ввод, и вывод осуществлялся с использованием интерфейса командной строки (CLI — Command Line Interface), без создания привычного пользовательского интерфейса (вы немного могли «поиграть» с графическим интерфейсом на дополнительном уроке по библиотеке tkinter).

Для программ, которые не предполагают того, что их будет использовать неподготовленный к работе с командной строкой пользователь, текстового интерфейса хватает. Существует большое количество консольных утилит, предназначенных для программистов или системных администраторов, но часто в жизни разработчика наступает момент, когда разработанную программу надо передать незнакомому с консолью пользователю. Графический интерфейс (GUI — Graphical User Interface) более «дружелюбный» к пользователям, а если в программе необходимо отображать не только текст, но и графическую или мультимедийную информацию, его использование становится необходимостью.

Рассмотрим основные понятия концепции GUI. Допустим, есть знакомый нам текстовый редактор. Когда мы нажимаем клавишу Х, возникает событие «Нажата клавиша Х». В то же время внутри программы запускается **обработчик**, который проверяет, какая клавиша нажата, какая раскладка выбрана и так далее. Затем он выполняет **действие**: выводит нужный символ на экран. Общий принцип работы можно представить в виде такой несложной схемы:.

Для языка программирования Python есть много способов создания приложений с графическим интерфейсом, в частности, уже знакомая вам библиотека tkinter. Она используется в большом числе кроссплатформенных приложений, написанных на Python. Мы абсолютно ничего не имеем против tkinter, но в этом разделе курса будем рассматривать библиотеку PyQT, так как ее возможности значительно богаче. Разобраться с другими библиотеками для построения графических интерфейсов вы можете самостоятельно, так как они имеют похожий принцип работы. Кроме того, на схеме Событие → Обработчик → Действие в том или ином виде построены почти все современные библиотеки, предназначенные для взаимодействия с пользователем. В этом вы сможете убедиться самостоятельно в следующих разделах курса.

Что же такое PyQT? Для начала разберемся, что такое QT. Это написанная на C++ библиотека с классами для создания графического интерфейса. Библиотека получилась настолько удачной, начала собирать вокруг себя большое сообщество программистов, которые разрабатывали приложения не только на C++, но и на других языках программирования. Это привело к тому, что и для других языков программирования стали появляться свои библиотеки-«обертки» для QT. Для Python это **PyQT**.

**Установка и настройка**

PyQt устанавливается так же, как и любая другая библиотека в Python:

pip install PyQT5

**Первые шаги в PyQT**

В терминологии PyQT (и достаточно большого числа других библиотек создания GUI) все графические приложения состоят из виджетов.

**Виджет**

Виджет — минимальный элемент графического интерфейса пользователя.

В библиотеке PyQT5 существует множество модулей, но чаще других используется **QtWidgets**. Именно в нем находятся классы, соответствующие различным элементам интерфейса.

Напишем простейшую программу с использованием библиотеки PyQT5:

import sys

# Импортируем из PyQt5.QtWidgets классы для создания приложения и виджета

from PyQt5.QtWidgets import QApplication, QWidget

# Унаследуем наш класс от простейшего графического примитива QWidget

class Example(QWidget):

def \_\_init\_\_(self):

# Надо не забыть вызвать инициализатор базового класса

super().\_\_init\_\_()

# В метод initUI() будем выносить всю настройку интерфейса,

# чтобы не перегружать инициализатор

self.initUI()

def initUI(self):

# Зададим размер и положение нашего виджета,

self.setGeometry(300, 300, 300, 300)

# А также его заголовок

self.setWindowTitle('Первая программа')

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

# Создадим класс приложения PyQT

app = QApplication(sys.argv)

# А теперь создадим и покажем пользователю экземпляр

# нашего виджета класса Example

ex = Example()

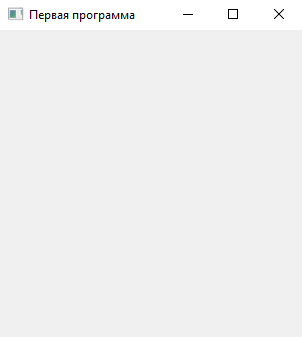
ex.show()

# Будем ждать, пока пользователь не завершил исполнение QApplication,

# а потом завершим и нашу программу

sys.exit(app.exec())

Если мы запустим эту программу, увидим такое окно:



Давайте разберемся, что происходит в этой программе. Обратите внимание на наш класс — Example. Он наследуется от базового класса QWidget, который определяет простейшее окно. От него наследуется много встроенных виджетов.

Первое, что можно увидеть в классе, — перегруженный конструктор.

super().\_\_init\_\_() — эта строка вызывает конструктор родительского класса. Потом вызывается метод класса с названием initUI.

Разумеется, инициализацию интерфейса можно реализовать и в инициализаторе класса, но хорошей практикой считается вынос этой функциональности в отдельный метод класса с названием initUI.

**PEP 8**

Обратите внимание: метод initUI называется с нарушением правил PEP 8. Как мы говорили ранее, такое отступление возможно для сохранения совместимости с используемой библиотекой, а так как библиотека QT изначально написана на C++ с именованием методов в стиле camelCase, то initUI — допустимое название метода в классе, который унаследован от QWidget.

В initUI мы определяем положение и размеры нашего окна методом setGeometry. Первые два параметра — X и У — координаты левого верхнего угла формы относительно левого верхнего угла нашего монитора. Оставшиеся — ширина и высота виджета. Методом setWindowTitle задаем заголовок нашего окна.

Мы создали класс, но пока его не используем. Чтобы начать с ним работать, необходимо куда-то разместить наш виджет.

Для этого нужно создать приложение — объект класса QApplication, строка app = QApplication(sys.argv) как раз отвечает за это. Несмотря на то, что после инициализации переменная app используется только один раз, она необходима. Все скрытое от пользователя и разработчика взаимодействие программы с операционной системой возможно только благодаря этому классу. Передавать значение sys.argv в конструктор QApplication не обязательно, можно передать любой список, даже пустой. Но приведенная запись является хорошим тоном создания PyQT-приложений, так как она позволит корректно обрабатывать запуск приложения с параметрами командной строки (о них мы поговорим позднее).

Затем мы создаем экземпляр нашего класса: ex = Example(). Все готово, можно запускать. Метод show() отображает наш виджет в приложение. Но отобразить недостаточно, надо запустить цикл обработки событий. Для этого вызываем метод app.exec().

В последней строке программы этот вызов «обернут» в sys.exit. Это сделано для корректного завершения программы.

Но просто пустое окно — это скучно, начнем добавлять туда виджеты. Первый на очереди — знакомая нам кнопка. Класс, который необходим для работы с ней, называется QPushButton.

import sys

from PyQt5.QtWidgets import QApplication, QWidget, QPushButton

class Example(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

self.setGeometry(300, 300, 300, 300)

self.setWindowTitle('Вторая программа')

# Создаем кнопку.

# Передаем 2 параметра:

# надпись и виджет, на котором будет размещена кнопка

btn = QPushButton('Кнопка', self)

# Изменяем размер кнопки. Теперь он 100 на 100 пикселей

btn.resize(100, 100)

# Размещаем кнопку на родительском виджете

# по координатам (100, 100)

btn.move(100, 100)

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

app = QApplication(sys.argv)

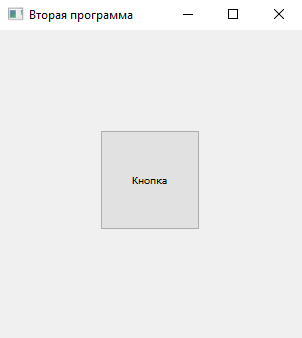
ex = Example()

ex.show()

sys.exit(app.exec())

Вспомним картинку выше: у любого виджета, кроме базового, должен быть «родитель». Когда мы добавляем кнопку, «родителем» выступает наш виджет окна. Поэтому при объявлении кнопки мы указываем не только текст, но и экземпляр класса QWidget (или, как в нашем случае, его наследника).

Метод resize позволяет изменить размеры кнопки. А с помощью метода move мы указываем расположение нашей кнопки в виджете-«родителе». Запустим программу и убедимся, что кнопка появилась.



На кнопку можно даже понажимать, но пока безрезультатно. Сделаем ее полезной — добавим функциональность. В уже рассмотренных нами терминах — добавим обработчик события «нажатие на кнопку».

import sys

from PyQt5.QtWidgets import QApplication, QWidget, QPushButton

class Example(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

self.setGeometry(300, 300, 300, 300)

self.setWindowTitle('Третья программа')

self.btn = QPushButton('Кнопка', self)

self.btn.resize(100, 100)

self.btn.move(100, 100)

# присоединим к событию нажатия на кнопку обработчик self.hello()

self.btn.clicked.connect(self.hello)

def hello(self):

# метод setText() используется для задания надписи на кпопке

self.btn.setText('Привет')

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

app = QApplication(sys.argv)

ex = Example()

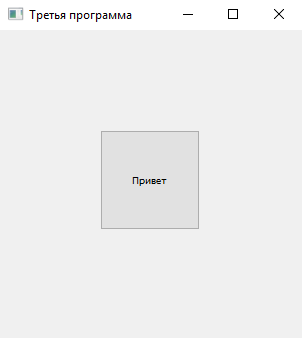
ex.show()

sys.exit(app.exec())

Что изменилось по сравнению с предыдущей программой? Во-первых, теперь btn — это поле класса, а не просто локальная переменная метода, как было в прошлый раз. Поэтому добраться до кнопки мы теперь можем не только из метода initUI, а из любого места внутри класса нашей формы (и даже из любого кода, который имеет доступ к объекту нашей формы). А во-вторых, добавилась функциональность: при нажатии на кнопку надпись на ней изменяется на строку «Привет».

self.btn.clicked.connect(self.hello) — что значит эта фраза в переводе на человеческий язык? «Если получишь **событие clicked** от **объекта self.btn**, вызови **обработчик self.hello()**». В нем с помощью метода setText мы меняем текст на кнопке.

Но эта кнопка фактически «одноразовая», поскольку текст на кнопке изменится только после первого нажатия. При последующих нажатиях он, конечно же, тоже меняется, ведь метод hello() вызывается, но мы, как пользователи, этого не видим.



Если говорить про библиотеку QT, в ней концепция «Событие → Обработчик → Действие» реализована с некоторыми особенностями, поэтому введена своя терминология — **сигналы и слоты** (хотя если вы будете использовать более общие названия, вас все равно поймут).

**Сигнал** вырабатывается, когда происходит определенное событие. **Слот** — это функция, которая **ловит** определенный сигнал. Все классы, наследуемые от QObject или его дочерних классов (например, уже знакомые нам QWidget и QPushButton) могут содержать сигналы и слоты. Сигналы вырабатываются объектами, когда они изменяют свое состояние так, что это может заинтересовать другие объекты. При этом сами объекты не знают и не заботятся о том, что у его сигнала может не быть получателя. Только класс, который определяет сигнал, (или его дочерние классы) могут вырабатывать сигнал.

**Слоты** — обычные функции (или методы класса), которые могут быть использованы для получения сигналов (хотя никто не запрещает вам вызывать эти функции и просто так). Точно так же, как объект не знает ничего о получателях своих сигналов, функция-слот ничего не знает о сигналах, которые к ней подключены.

Мы можем подключать к одному слоту сколько угодно сигналов, а один слот может быть подключен к неограниченному количеству сигналов (даже несколько раз к одному и тому же сигналу, тогда при появлении сигнала функция-слот выполнится столько раз, сколько была подключена). Кроме того, возможно подключать сигнал к другому сигналу, что вызовет выработку второго сигнала немедленно после появления первого.

В качестве аналогии можно привести радиотрансляцию с радиостанции. Сигнал — это такая трансляция, и радиостанции неизвестно, сколько радиоприемников-слотов будут настроены на волну, когда будет трансляция. Но все настроенные на нужную волну приемники получат сигнал.

Но довольно теории, давайте выводить на кнопке не один и тот же текст каждый раз, а, например, количество нажатий на нее. Никаких дополнительных атрибутов в нашем классе нам не понадобится — хранить информацию будем прямо в «кнопке».

import sys

from PyQt5.QtWidgets import QApplication, QWidget, QPushButton

class Example(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

self.setGeometry(300, 300, 300, 300)

self.setWindowTitle('Четвёртая программа')

self.btn = QPushButton('0', self)

self.btn.resize(100, 100)

self.btn.move(100, 100)

# Подпишем функцию-слот self.count() на сигнал clicked кнопки btn

self.btn.clicked.connect(self.count)

def count(self):

# Не забываем, что надпись на кнопке - это текст.

self.btn.setText(f"{int(self.btn.text()) + 1}")

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

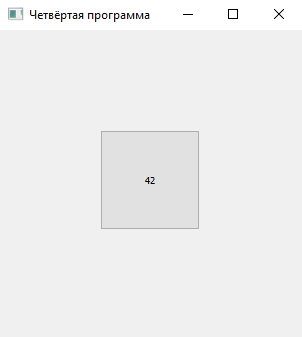
app = QApplication(sys.argv)

ex = Example()

ex.show()

sys.exit(app.exec())

Метод text() возвращает строку — текущую надпись на кнопке.



Для отображения данных в PyQT есть и более подходящие виджеты. Для текстовых данных лучше использовать QLabel, а для цифр есть красивый виджет QLCDNumber, который имитирует дисплей калькулятора.

import sys

from PyQt5.QtWidgets import QApplication, QWidget, QPushButton

from PyQt5.QtWidgets import QLCDNumber, QLabel

class Example(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

self.setGeometry(300, 300, 400, 400)

self.setWindowTitle('Пятая программа')

self.btn = QPushButton('Кнопка', self)

# Подстроим размер кнопки под надпись на ней

self.btn.resize(self.btn.sizeHint())

self.btn.move(100, 150)

# Обратите внимание: функцию не надо вызывать :)

self.btn.clicked.connect(self.inc\_click)

self.label = QLabel(self)

# Текст задается также, как и для кнопки

self.label.setText("Количество нажатий на кнопку")

self.label.move(80, 30)

self.LCD\_count = QLCDNumber(self)

self.LCD\_count.move(110, 80)

self.count = 0

def inc\_click(self):

self.count += 1

# В QLCDNumber для отображения данных используется метод display()

self.LCD\_count.display(self.count)

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

app = QApplication(sys.argv)

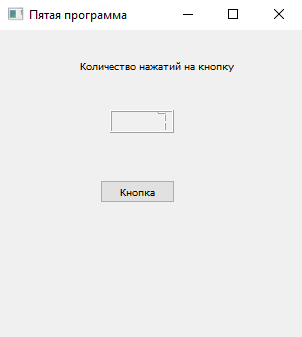
ex = Example()

ex.show()

sys.exit(app.exec())

Теперь для хранения будем использовать отдельное поле — count. Для задания значения у QLCDNumber используется метод display, а для QLabel — setText, как и у QPushButton.

Код self.btn.resize(self.btn.sizeHint()) подстраивает размеры кнопки под размер надписи на ней, чтобы не возникла ситуация, что часть надписи заходила за границу этого виджета.



QLCDNumber, кроме чисел, умеет показывать следующие символы: O (нулем), S (с помощью 5), g (с помощью 9), минус, точку, A, B, C, D, E, F, h, H, L, o, P, r, u, U, Y, кавычку, пробел. То есть вполне возможно написать на QLCDNumber какое-либо сообщение, например Error. Те символы, которые QLCDNumber отобразить не может, он просто заменяет на пробелы.

Как вы понимаете, отображать данные пользователю нашей программы — это только половина дела, не менее важно получать от него данные. Для этого существуют несколько виджетов. Если пользователь должен ввести одну строку, то для получения ее получения прекрасно подойдет QLineEdit.

import sys

from PyQt5.QtWidgets import QApplication, QWidget, QPushButton

from PyQt5.QtWidgets import QLabel, QLineEdit

class Example(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

self.setGeometry(300, 300, 400, 400)

self.setWindowTitle('Шестая программа')

self.btn = QPushButton('Кнопка', self)

self.btn.resize(self.btn.sizeHint())

self.btn.move(100, 150)

self.btn.clicked.connect(self.hello)

self.label = QLabel(self)

self.label.setText("Привет, неопознанный лев")

self.label.move(40, 30)

self.name\_label = QLabel(self)

self.name\_label.setText("Введите имя: ")

self.name\_label.move(40, 90)

self.name\_input = QLineEdit(self)

self.name\_input.move(150, 90)

def hello(self):

name = self.name\_input.text() # Получим текст из поля ввода

self.label.setText(f"Привет, {name}")

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

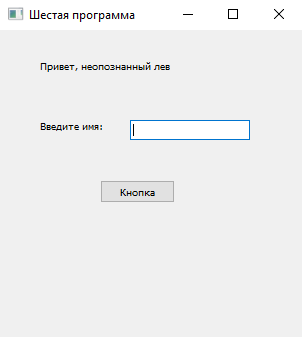
app = QApplication(sys.argv)

ex = Example()

ex.show()

sys.exit(app.exec())

Метод text() виджета QLineEdit позволяет получить введенную пользователем строчку.



**Кто отправил сигнал**

Давайте рассмотрим более детально, как происходит взаимодействие между виджетами. Когда пользователь нажимает на кнопку, возникает **сигнал**, который обрабатывается некоторой функцией-слотом. Но если у нас одна функция-обработчик для нескольких кнопок, как понять, на какую из них нажал пользователь?

Чтобы определить, кто является источником сигнала, у виджета есть метод .sender().

Рассмотрим пример такой программы.

import sys

from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QLabel

class Example(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

self.setGeometry(300, 300, 300, 200)

self.setWindowTitle('Кто отправил сигнал')

self.button\_1 = QPushButton(self)

self.button\_1.move(90, 40)

self.button\_1.setText("Кнопка 1")

self.button\_1.clicked.connect(self.run)

self.button\_2 = QPushButton(self)

self.button\_2.move(90, 80)

self.button\_2.setText("Кнопка 2")

self.button\_2.clicked.connect(self.run)

self.label = QLabel(self)

self.label.setText("Пока никто не отправлял")

self.label.move(50, 120)

self.show()

def run(self):

self.label.setText(self.sender().text())

print(self.sender().text())

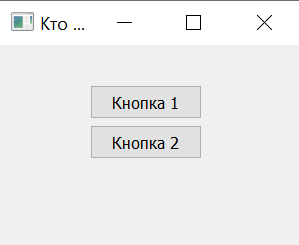
if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

ex = Example()

ex.show()

sys.exit(app.exec())



В этой программе слот run() привязан к сигналам от двух кнопок. Независимо от того, кто его вызвал, метод печатает в консоль текст нажатой кнопки, которую мы получаем с помощью sender().

**Открытие других форм**

Разумеется, далеко не всегда приложение ограничивается одной формой. В **настоящих** программах, как минимум, есть еще одна — «О программе», а также формы с настройками, диалоги открытия и сохранения файлов и так далее. Конечно, PyQT дает возможность в нашем приложении создавать формы из других форм. Обычно (но далеко не всегда) для главной формы приложения выбирают класс QMainWindow, а для дочерних форм — класс QWidget. Чтобы создать форму из другой формы, достаточно сделать объект нужного нам класса и вызвать у него метод show.

import sys

from PyQt5.QtWidgets import QApplication, QWidget, QPushButton

from PyQt5.QtWidgets import QMainWindow, QLabel

class FirstForm(QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

self.setGeometry(300, 300, 300, 300)

self.setWindowTitle('Главная форма')

self.btn = QPushButton('Другая форма', self)

self.btn.resize(self.btn.sizeHint())

self.btn.move(100, 100)

self.btn.clicked.connect(self.open\_second\_form)

def open\_second\_form(self):

self.second\_form = SecondForm(self, "Данные для второй формы")

self.second\_form.show()

class SecondForm(QWidget):

def \_\_init\_\_(self, \*args):

super().\_\_init\_\_()

self.initUI(args)

def initUI(self, args):

self.setGeometry(300, 300, 300, 300)

self.setWindowTitle('Вторая форма')

self.lbl = QLabel(args[-1], self)

self.lbl.adjustSize()

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

app = QApplication(sys.argv)

ex = FirstForm()

ex.show()

sys.exit(app.exec())

Обратите внимание на два момента:

1. Мы сохраняем созданную форму в атрибут нашей родительской формы для того, чтобы иметь возможность управлять ею из других методов. И для того, чтобы сборщик мусора Python не удалил ее случайно как объект, на который нет ссылок.
2. В инициализаторе дочерней формы, помимо self, есть еще \*args, куда мы можем помещать информацию, которую хотим передать из родительской формы в дочернюю. Тут мы передаем ссылку на объект-родитель (его обычно передают первым) и сообщение, которое мы будет отображать на второй форме в QLabel.

**Итоги**

На этом уроке мы изучили работу с основными виджетами:

* QWidget
* QPushButton
* QLCDNumber
* QLabel
* QLineEdit

Но мы рассмотрели далеко не все доступные методы. Информацию о других методах можно посмотреть на [официальном сайте QT](https://doc.qt.io/qt-5/qtwidgets-module.html). Обратите внимание: документация написана для языка C++, так что нужно обращать внимание лишь на названия методов и параметры, но не на синтаксис примеров.