Проект QT. Работаем с pyqtgraph, собираем независимое приложение

План урока

Аннотация

В этом уроке мы научимся взаимодействовать с пользователем на новом уровне: добавим в программу работу с клавиатурой и мышкой, диалоговые окна, новые виджеты. Также рассмотрим, как построить график функции и собрать нашу программу в exe-файл.

1. Обработка нажатий клавиатуры

У каждой клавиши на клавиатуре есть свой **уникальный** код. Однако работать с числами не всегда удобно, поэтому в PyQt в модуле **Qt** с кодами сопоставлены и понятные имена. Согласитесь, что гораздо легче понимать смысл вот такой строки: print(Qt.Key_F), нежели такой: print(70).

Подключить модуль **Qt** можно, например, таким способом:

```
from PyQt5.QtCore import Qt
```

Все имена кнопок и их коды можно найти здесь.

Для обработки нажатий на клавиши в вашем классе необходимо переопределить (создать) метод **keyPressEvent(self, event)**:

```
def keyPressEvent(self, event):
    if event.key() == Qt.Key_F:
        # code
```

Если во время работы приложения будет нажата клавиша **F**, то условие выполнится.

Чтобы обработать комбинацию клавиш (Hotkey), код должен быть таким:

```
def keyPressEvent(self, event):
    if int(event.modifiers()) == (Qt.AltModifier + Qt.ShiftModifier):
        if event.key() == Qt.Key_Q:
            # code
```

Теперь код выполнится, если нажмут на клавишу **Q** при зажатых кнопках **Shift** и **Alt**.

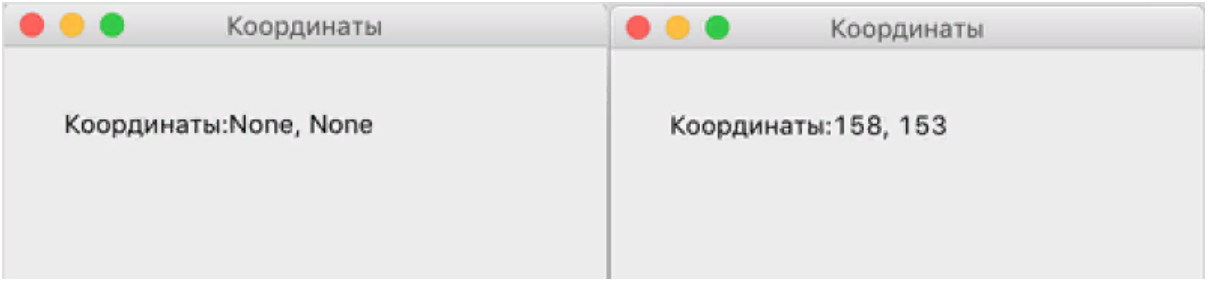
Важно, что кнопки обработаются только в том случае, если окно **активно**. Активным называется окно, с которым пользователь работает в данный момент.

2. Работа с мышью

Для работы с мышью, точно так же, как и в случае с клавиатурой, необходимо переопределять встроенные методы. Рассмотрим пример, в котором мы будем выводить координаты курсора на экран.

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


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

    def initUI(self):
        self.setGeometry(300, 300, 300, 300)
        self.setWindowTitle('Координаты')

        self.coords = QLabel(self)
        self.coords.setText("Координаты:None, None")
        self.coords.move(30, 30)
        self.show()

    def mouseMoveEvent(self, event):
        self.coords.setText("Координаты:{}, {}".format(event.x(),
                                                       event.y()))


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

Метод **mouseMoveEvent** срабатывает при изменении положения мышки. Функции **.x()** и **.y()** возвращают координаты текущего положения курсора в экранных координатах. Но если вы запустите эту программу и попробуете просто поводить мышкой, то ничего не произойдёт. По умолчанию это событие активируется только при зажатой левой клавише мыши. Зажмите её — и всё заработает. А чтобы отлавливать положение мыши, даже когда клавиша не зажата, в метод **init(self)** надо добавить следующую строку: self.setMouseTracking(True).



Кроме метода **mouseMoveEvent**, есть и другие. Например, метод **mousePressEvent**, который отвечает за обработку нажатия мыши. Кроме координат, полезно узнать, какая из кнопок была нажата, левая или правая. Для этого существует метод **.button()**. Сравнить можно

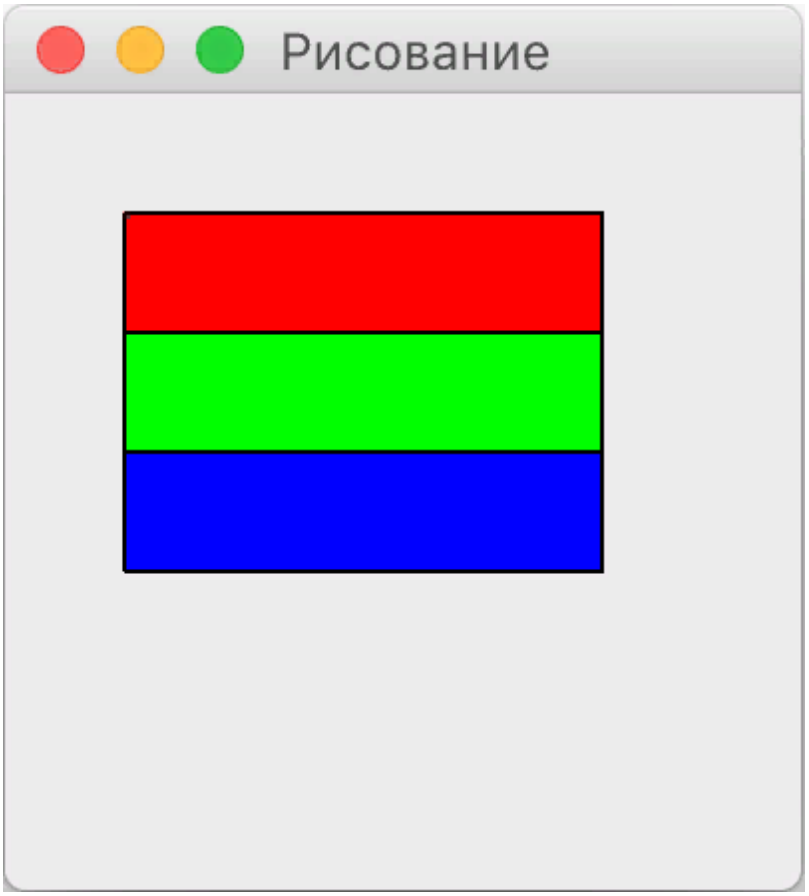
со встроенными значениями из модуля **PyQt5.QtCore.Qt**: LeftButton, RightButton и даже MiddleButton.

```
import sys
from PyQt5.QtWidgets import QWidget, QApplication, QLabel
from PyQt5.QtCore import Qt


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

    def initUI(self):
        self.setGeometry(300, 300, 300, 300)
        self.setWindowTitle('Координаты')

        self.coords = QLabel(self)
        self.coords.setText("Координаты:None, None")
        self.coords.move(30, 30)

        self.btn = QLabel(self)
        self.btn.setText("Никакая")
        self.btn.move(30, 50)
        self.show()

    def mousePressEvent(self, event):
        self.coords.setText("Координаты:{}, {}".format(event.x(),
                                                       event.y()))
        if (event.button() == Qt.LeftButton):
            self.btn.setText("Левая")
        elif(event.button() == Qt.RightButton):
            self.btn.setText("Правая")


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

3. Рисование

До этого момента, чтобы создать и вывести какое-либо изображение, надо было создавать картинку с помощью PIL, а затем выводить её, используя виджет **QPixmap**. Но в PyQT есть модули, которые позволяют рисовать прямо на поле. Посмотрим, как с ними работать.

```
import sys
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QColor


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

    def initUI(self):
        self.setGeometry(300, 300, 200, 200)
        self.setWindowTitle('Рисование')
        self.show()

    def paintEvent(self, event):
        qp = QPainter()
        qp.begin(self)
        self.drawFlag(qp)
        qp.end()

    def drawFlag(self,qp):
        qp.setBrush(QColor(255, 0, 0))
        qp.drawRect(30, 30, 120, 30)
        qp.setBrush(QColor(0, 255, 0))
        qp.drawRect(30, 60, 120, 30)
        qp.setBrush(QColor(0, 0, 255))
        qp.drawRect(30, 90, 120, 30)


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



В методе **paintEvent** происходит инициализация экземпляра класса **QPainter**, который отвечает за рисование на виджетах. Рисовать нужно между вызовами методов begin и end.

В функции **drawFlag** происходит непосредственно рисование. С помощью метода **setBrush** мы задаём цвет кисти в формате RGB. Для того, чтобы преобразовать цвет в нужный нам тип, используем метод **QColor**. Чтобы нарисовать прямоугольник, применяем метод **drawRect**. В качестве параметров ему передаются координаты левого верхнего угла, длина и высота.

Кроме прямоугольников, получится нарисовать ещё массу всего. Узнать о других методах можно в документации.

Во всех функциях в качестве параметра присутствует некий **event** — событие. Это событие, на которое подвешивается обработчик. Это может быть движение, нажатие или отпускание кнопки мыши, нажатие или отпускание клавиши и многое другое.

4. Диалоговые окна

Диалоговые окна нужны для того, чтобы получить какую-либо информацию от пользователя. Это может быть текстовая информация, цвет, настройки шрифта и даже файлы. В PyQT уже есть встроенные виджеты, реализующие различные диалоговые окна.

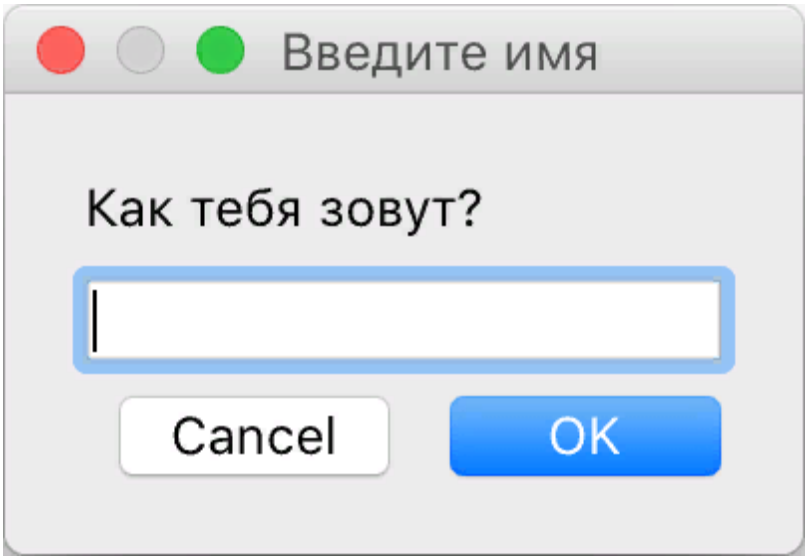
Рассмотрим виджет **QInputDialog**. Особенность этого виджета в том, что его можно настроить на получение различных типов данных: строки (одной или нескольких), числа или одного

значения из списка.

Напишем программу, которая получает имя пользователя с помощью диалогового окна.

```
import sys
from PyQt5.QtWidgets import QWidget, QApplication, QPushButton
from PyQt5.QtWidgets import QInputDialog


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

    def initUI(self):
        self.setGeometry(300, 300, 150, 150)
        self.setWindowTitle('Диалоговые окна')

        self.button_1 = QPushButton(self)
        self.button_1.move(20, 40)
        self.button_1.setText("Кнопка")
        self.button_1.clicked.connect(self.run)

        self.show()

    def run(self):
        i, okBtnPressed = QInputDialog.getText(
            self, "Введите имя", "Как тебя зовут?"
        )
        if okBtnPressed:
            self.button_1.setText(i)


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

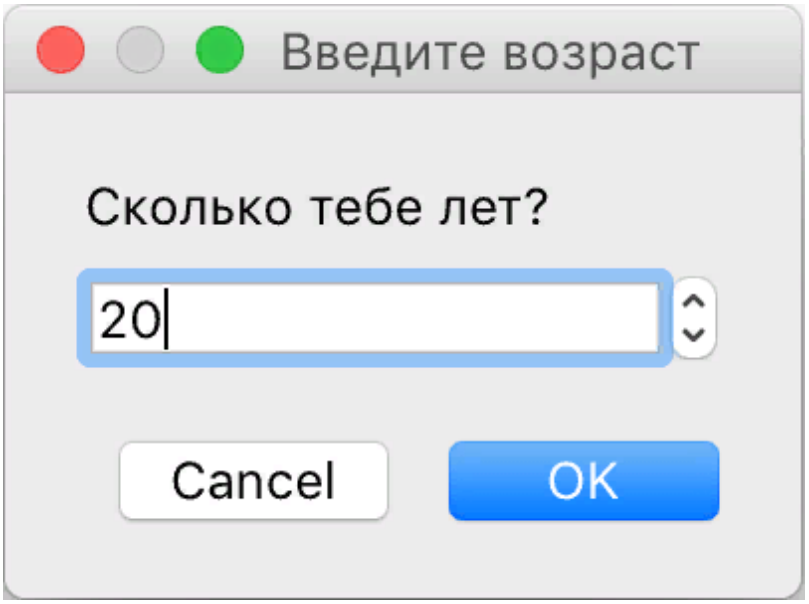
В функции, вызываемой нажатием на кнопку, создаётся диалоговое окно (**QInputDialog**) и указывается, какой тип оно будет возвращать (**getText**). В качестве параметров передаются родительское окно (**self**), заголовок нового окна и сообщение для пользователя. Возвращает эта функция кортеж данных, где на первом месте записано состояние кнопки «Ok», а на втором, если кнопка была нажата — пользовательские данные.



В случае, когда нам нужно получить целое число, мы указываем текущее значение, минимальное и максимальное значение, а также шаг.

```
i, okBtnPressed = QInputDialog.getInt(
    self, "Введите возраст", "Сколько тебе лет?", 20, 18, 27, 1
)
```

В этом примере значение по умолчанию — 20, минимальное значение — 18, максимальное — 27, а шаг — 1.

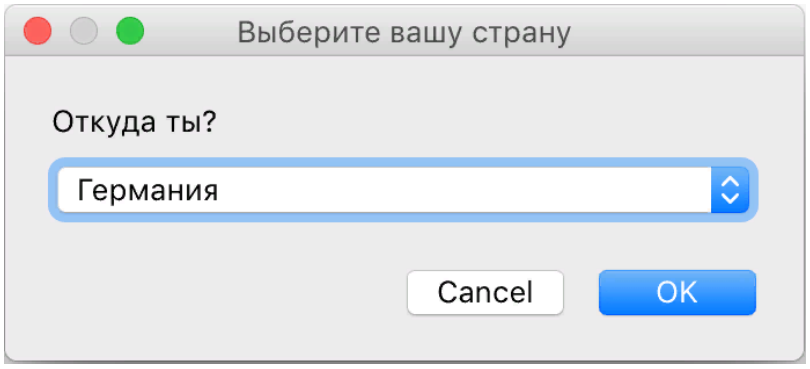


Иногда нам надо, чтобы пользователь выбрал какое-то значение из предоставленных ему. Для этого есть метод **getItem**.

```
i, okBtnPressed = QInputDialog.getItem(
    self,
    "Выберите вашу страну",
    "Откуда ты?",
    ("Россия", "Германия", "США"),
```

```
    1,
    False
)
```

В качестве параметров ему необходимо указать итерируемый объект и индекс значения по умолчанию. А чтобы пользователь не смог что-то самостоятельно ввести, надо указать следующим параметром False.

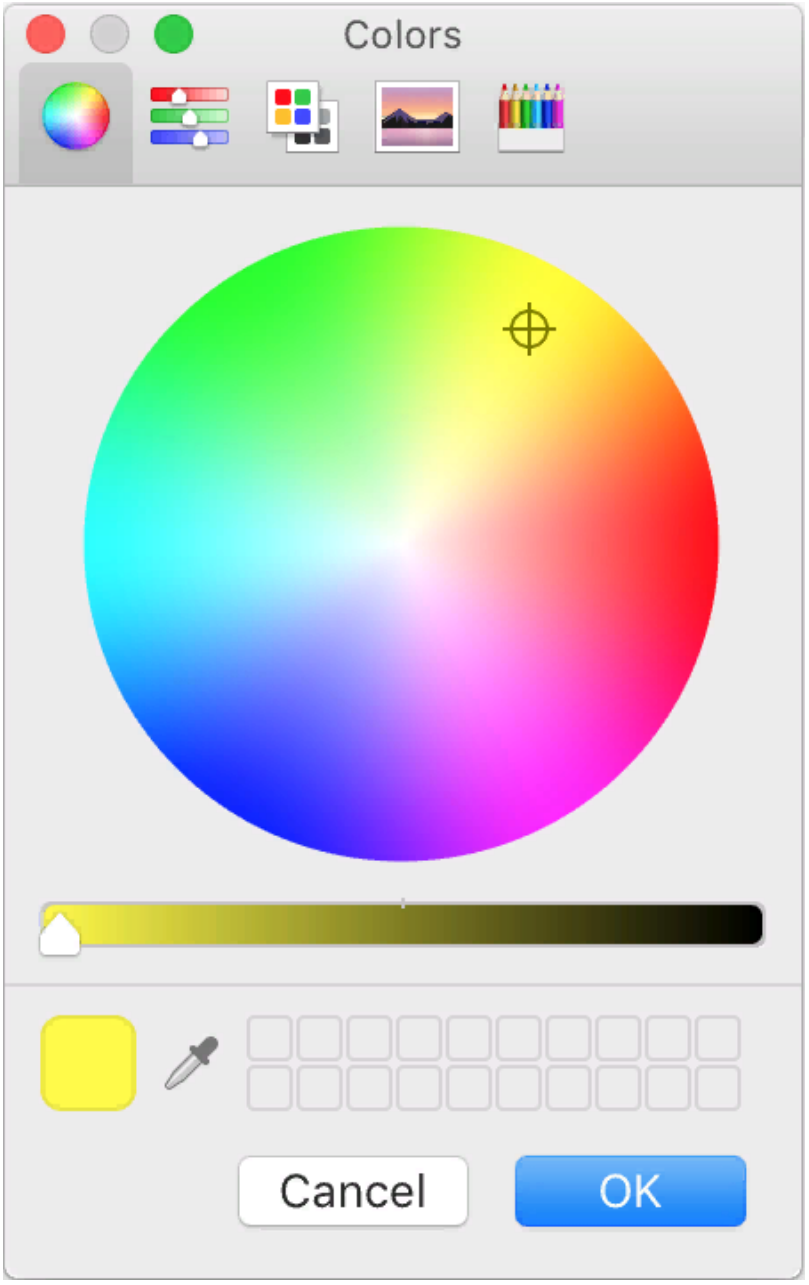


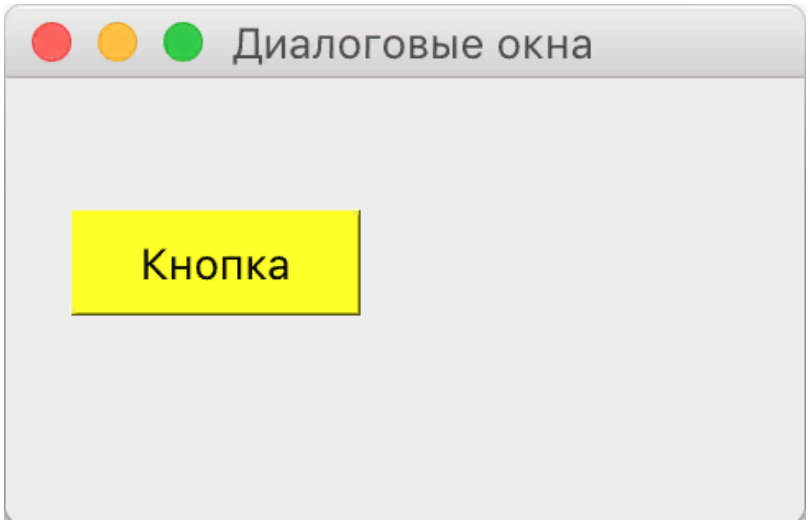
Для выбора цвета существует другой виджет: **QColorDialog**. Вот пример программы с его использованием:

```
import sys
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtWidgets import QPushButton, QColorDialog


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

    def initUI(self):
        self.setGeometry(300, 300, 150, 150)
        self.setWindowTitle('Диалоговые окна')

        self.button_1 = QPushButton(self)
        self.button_1.move(20, 40)
        self.button_1.setText("Кнопка")
        self.button_1.clicked.connect(self.run)

        self.show()

    def run(self):
        color = QColorDialog.getColor()
        if color.isValid():
            self.button_1.setStyleSheet(
                "background-color: {}".format(color.name())
```

```
            )


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

Если цвет валиден, то фон кнопки окрасится в него. Функция **.name()** возвращает цвет в 16-тиричном формате.





5. PyQtGraph

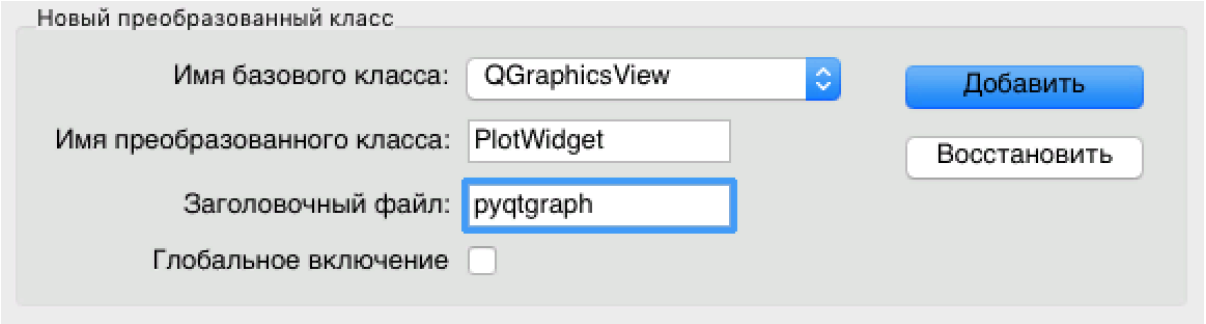
Несмотря на большое количество встроенных виджетов, их не всегда хватает. Но PyQT позволяет создавать собственные виджеты со своим функционалом. Некоторые разработчики выкладывают эти библиотеки в Интернет, чтобы их могли использовать другие люди. В этом уроке мы познакомимся с библиотекой PyQTGraph, которая включает в себя виджеты для работы с графиками.

Для установки библиотеки выполните стандартную команду в командной строке:

```
pip install pyqtgraph
```

Затем лучше настроить QtDesigner для работы с этой библиотекой. Это необязательное условие, ведь работать с ней можно и «вручную», как мы делали в самом первом уроке — но это не столь удобно.

1. Открываем новый виджет в QtDesigner и добавляем **Graphics View**.
2. Нажимаем на него правой кнопкой мыши и выбираем вкладку **Преобразовать в...**.
3. Заполняем поля, как на картинке:

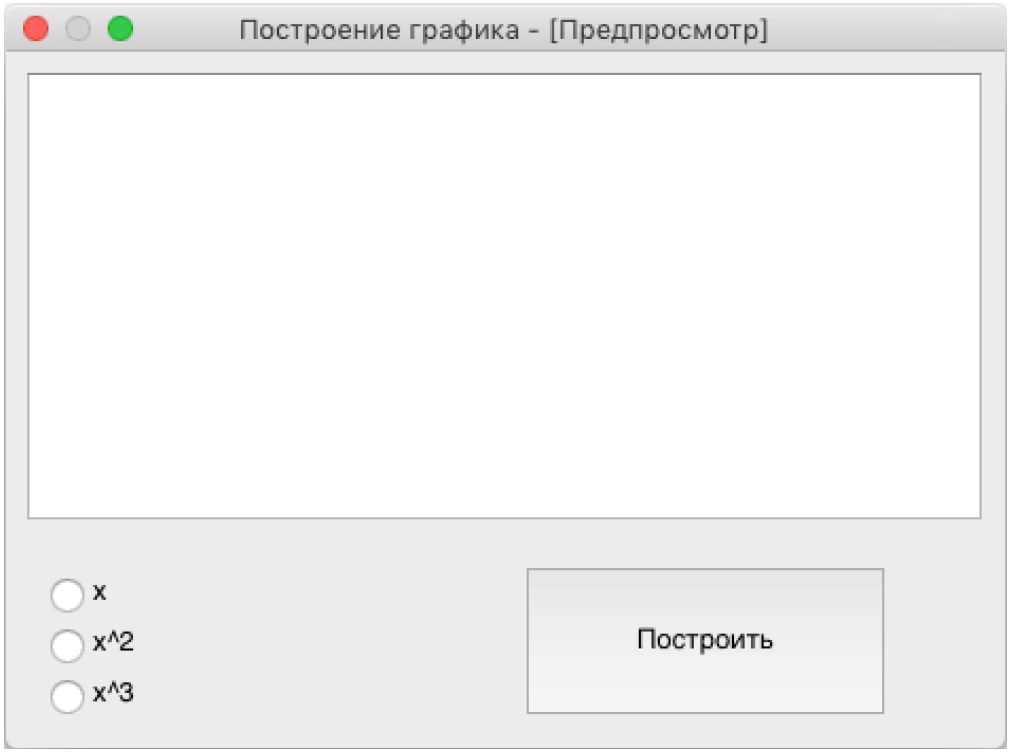


4. Нажимаем на «Добавить», а затем на «Преобразовать».

Если нам в следующий раз понадобиться добавить такой же виджет, то мы сможем выбрать его из выпадающего меню **Преобразовать в...**.

Напишем программу, которая будет строить графики одной из трёх функций в зависимости от выбранного Radio Button.

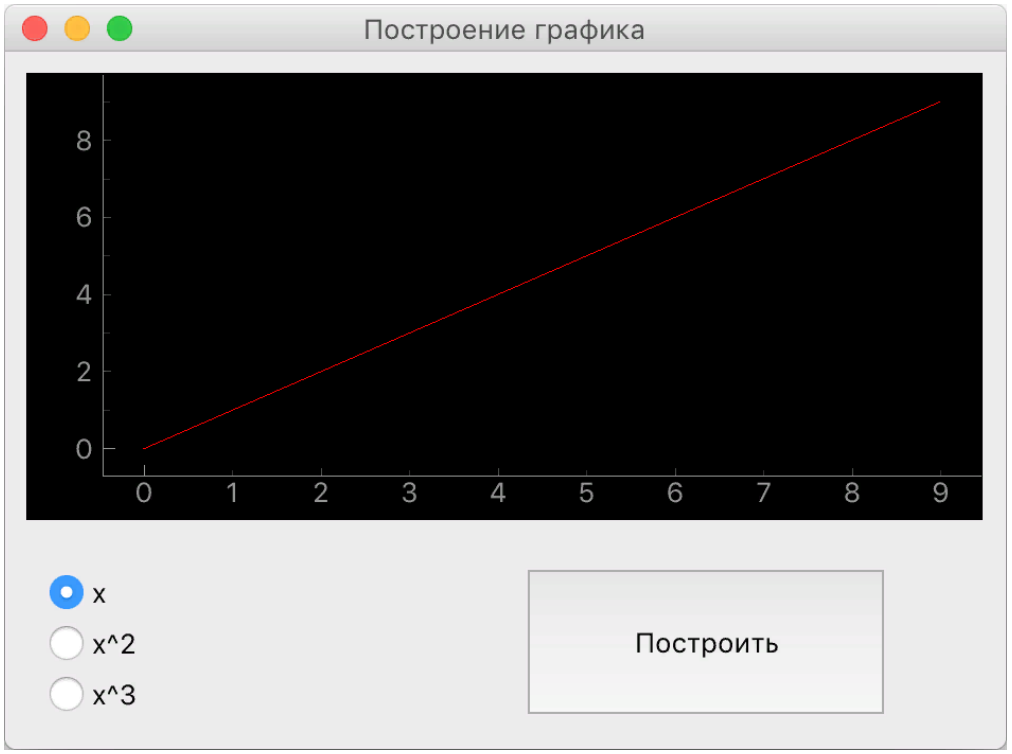
Сначала создаём интерфейс в QtDesigner, не забывая выполнить преобразование Graphics View в наш Plot Widget.



Преобразуем его с помощью **pyuic5** и начнём выполнять построение.

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget
from PyQt5.QtWidgets import QMainWindow
from ui import Ui_Form


class MyWidget(QMainWindow,Ui_Form):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.pushButton.clicked.connect(self.run)

    def run(self):
        if self.radioButton.isChecked():
            self.widget.clear()
            self.widget.plot([i for i in range(10)],
                             [i for i in range(10)],
```

```
                                pen='r')
        elif self.radioButton_2.isChecked():
            self.widget.clear()
            self.widget.plot([i for i in range(10)],
                                [i**2 for i in range(10)],
                                pen='g')
        elif self.radioButton_3.isChecked():
            self.widget.clear()
            self.widget.plot([i for i in range(10)],
                                [i**3 for i in range(10)],
                                pen='b')


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



Подробнее про pyqtgraph можно почитать в его официальной документации.

6. Создание standalone приложений

В отличие от компилируемых языков программирования вроде C++, Python — язык интерпретируемый. И создание standalone приложения — по своей сути, «упаковка» виртуальной машины Python, заточенной под выполнение одной программы.

Мы будем рассматривать работу в операционной системе Windows, отличия в других OS (Linux и MacOS) не являются существенными.

Для создания .exe приложений из программ на Python есть несколько библиотек.
Мы используем одну из самых популярных. Она называется **pyinstaller**.

Сначала необходимо её установить:

```
pip install pyinstaller
```

Затем открываем командную строку, переходим в папку с нашим проектом и выполняем следующую команду:

```
pyinstaller --onefile --noconsole main.py
```

Что делают модификаторы:

— **onefile** собирает программу в один exe-файл. Это удобно, если программа небольшая, у неё мало графических, аудио и прочих файлов, не относящихся к программному коду;

— **noconsole**. Если не писать этот модификатор, то, кроме окна программы, будет открываться python-консоль. Иногда, например для отладки, это удобно.

Есть и другие модификаторы. Например, **icon**. В нём можно указать путь до картинки в формате .ico, и она будет отображаться при просмотре программы в проводнике.

После выполнения этой команды в директории проекта появится две папки, **build** и **dist**. В папке **dist** будет лежать наша программа, или, если мы не указали модификатор "onefile", то, кроме .exe, там будет ещё множество файлов и папок. В неё следует добавить папки с графическими и другими файлами, которые используются в программе.