Урок QT Диалоги

PyQT. Диалоги, работа с изображениями

Аннотация

На этом уроке мы рассмотрим возможности PyQT по отображению графических данных, а также изучим возможности рисования непосредственно на виджетах библиотеки.

1. QPixmap

До сих пор мы работали только с текстовой информацией, однако, как нам подсказывает здравый смысл (и материалы уроков про библиотеки в Python), в файлах может храниться не только текст, но и различные более сложные данные, например: изображения, видео или аудиозаписи. В этом уроке мы познакомимся с возможностями PyQT, позволяющими работать с графикой.

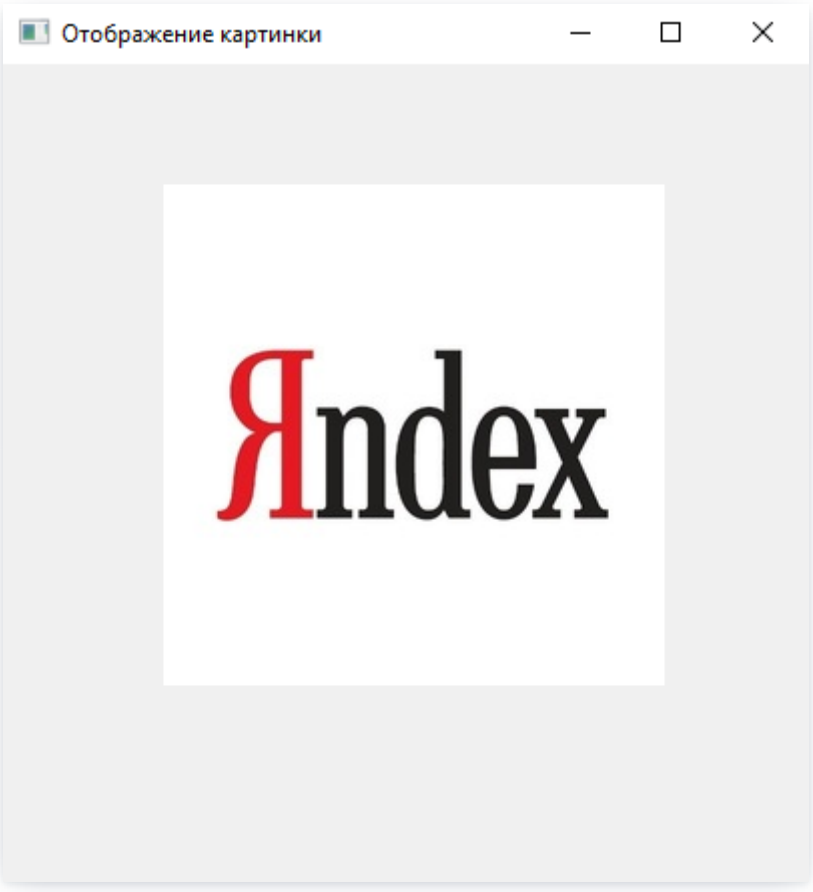
Одним из способов отображения изображения является использование **QPixmap**. Напишем простейшую программу, которая будет демонстрировать заданную картинку.

```
import sys

from PyQt5.QtGui import QPixmap
from PyQt5.QtWidgets import QApplication, QLabel, QMainWindow

SCREEN_SIZE = [400, 400]


class Example(QMainWindow):
    def __init__(self):
        super().__init__()
```

```
        self.initUI()

    def initUI(self):
        self.setGeometry(400, 400, *SCREEN_SIZE)
        self.setWindowTitle('Отображение картинки')

        ## Изображение
        self.pixmap = QPixmap('orig.jpg')
        # Если картинки нет, то QPixmap будет пустым,
        # а исключения не будет
        self.image = QLabel(self)
        self.image.move(80, 60)
        self.image.resize(250, 250)
        # Отображаем содержимое QPixmap в объекте QLabel
        self.image.setPixmap(self.pixmap)


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

Хотя раньше мы передавали размеры экрана напрямую в функцию, можно использовать список или какую-то другую структуру для хранения этих данных. Это может пригодиться в том случае, если у вас в программе много окон разного размера, или при повторном открытии окна необходимо восстановить его предыдущий размер.

Обратите ваше внимание на следующий факт: `QPixmap` не используется для «показывания» изображения. В нем только хранится объект — картинка. Для отображения используется уже знакомый нам `QLabel`, в который с помощью метода `setPixmap` загружается наш объект.



2. Диалоговые окна для выбора файла

Но что делать, если мы хотим написать программу, которая позволяет открывать несколько файлов? Менять каждый раз код программы — неоптимально. Использовать `input()` для ввода имени файла? Но мы же работаем с графическим интерфейсом. Разместить на форме `QLineEdit`, как мы делали на предыдущих занятиях? Выглядит как выход, но это неудобно по нескольким причинам:

- Путь к файлу может быть очень длинным и пользователю будет неудобно его вводить. Да и вообще пользователь может с ходу не знать, где находится нужный файл
- Пользователь может часто ошибаться при вводе, и нам придется обрабатывать эти ошибки

Чтобы повысить удобство для пользователей и минимизировать ошибки ввода, были придуманы так называемые **Диалоговые окна**, которые мы сейчас и попробуем использовать.

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.

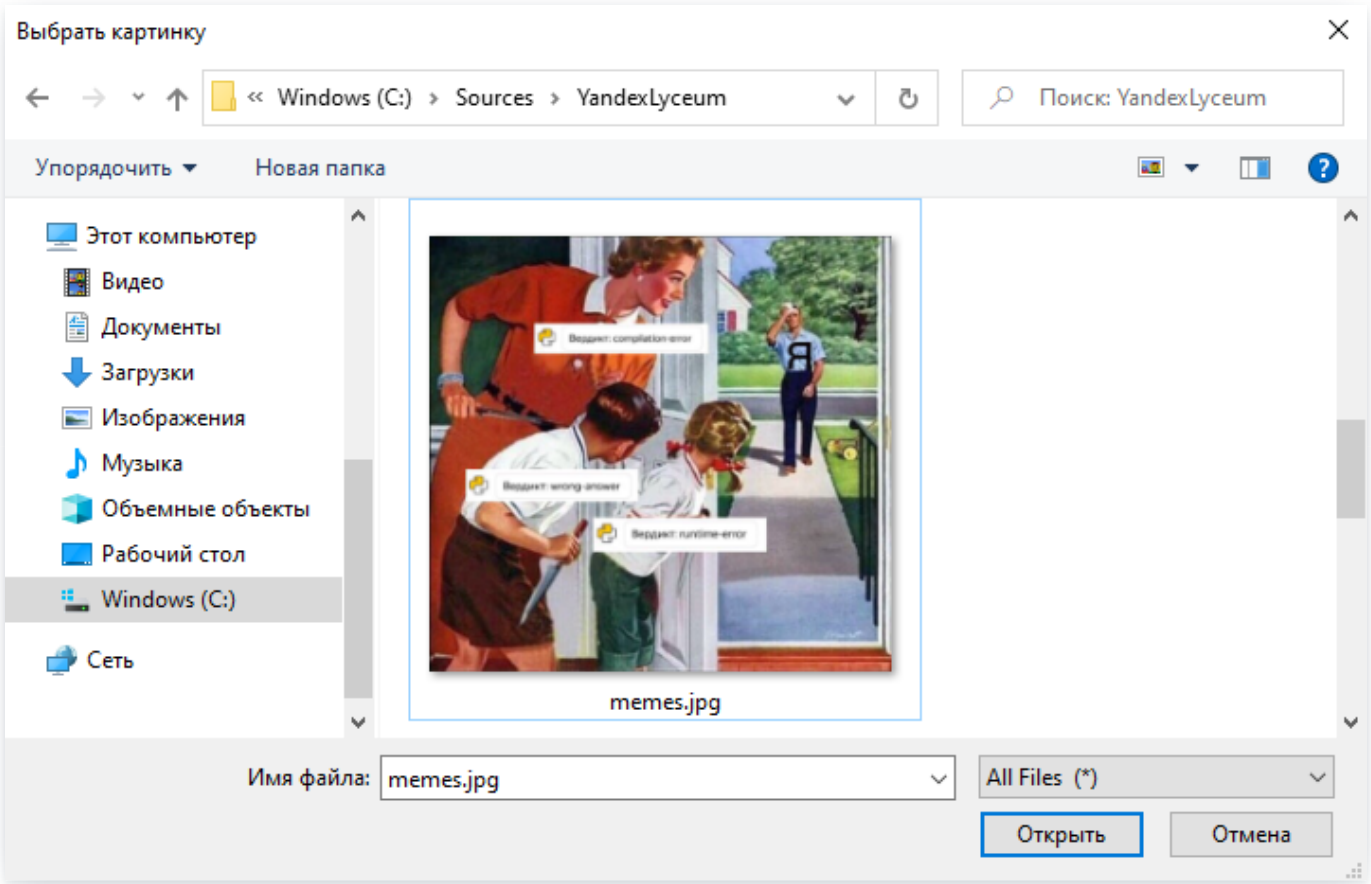
Будем получать путь к файлу с помощью **QFileDialog**. Важно не забыть импортировать этот виджет из **PyQt5.QtWidgets**.

Добавим в функцию загрузки из первой программы следующую строку:

```
fname = QFileDialog.getOpenFileName(self, 'Выбрать картинку', '')[0]
```

И будем передавать в `QPixmap` не имя файла, а переменную `fname`. Посмотрим, что будет происходить.

Перед запуском программы открывается окно выбора файла.



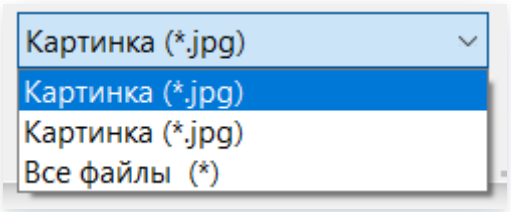
После выбора файла (после нажатия кнопки «Открыть»), диалог закроется и откроется наша форма с выбранным изображением.

Давайте разберем, какие основные параметры можно передать в метод `getOpenFileName` диалога работы с

файлами, для получения имени файла:

1. Родительский виджет (наш виджет формы)
2. Заголовок окна. В нашем случае это «Выбрать картинку». Интересный факт: этот заголовок будет отображаться только в Windows и некоторых окружениях рабочего стола в Linux, а в macOS будет игнорироваться
3. Директория открытия — папка, содержимое которой будет отображаться при открытии диалога. Можно даже дописать имя файла, который будет предлагаться к открытию. Если передать пустые кавычки, поиск начнется в той же папке, где лежит запускаемый нами скрипт
4. Набор фильтров. Если ничего не передать, будет применен фильтр 'All Files (*)', который отображает все файлы в директории. Если нужно отобразить только определенные типы файлов, можно это указать строкой вида: 'Картинка (.jpg);;Картинка (.jpg);;Все файлы (*)'. Так мы дадим пользователю выбирать, какие типы файлов он хочет отображать в папке. Разные фильтры отделяются ';;'.

```
fname = QFileDialog.getOpenFileName(
    self, 'Выбрать картинку', '',
    'Картинка (*.jpg);;Картинка (*.png);;Все файлы (*)')[0]
```



Остальные параметры метода можно посмотреть в **документации**.

Обратите внимание: эта функция возвращает кортеж, состоящий из полного пути к файлу, а также выбранный фильтр (по умолчанию — 'All Files (*)')

3. Диалоги ввода информации

Кроме диалога, позволяющего выбрать файл, существуют и другие.

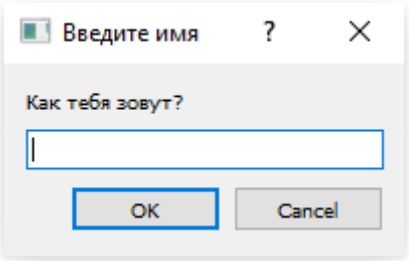
Рассмотрим виджет **QInputDialog**. Особенность этого виджета в том, что его можно настроить на получение различных типов данных: строки (одной или нескольких), числа или одного значения из списка, вызывая различные методы.

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)

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


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

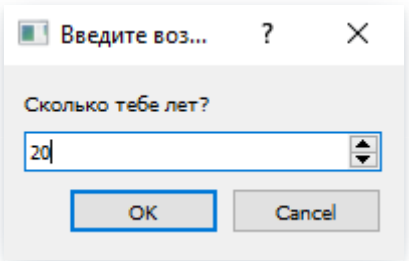
В методе, вызываемой нажатием на кнопку, создается диалоговое окно (`QInputDialog`) и указывается, какой тип оно будет возвращать (`getText`). В качестве параметров передаются родительское окно (`self`), заголовок нового окна и сообщение для пользователя. Возвращает эта функция кортеж, где на первом месте записаны введенные пользователем данные (или пустая строка, если нажата кнопка «Отмена»), а на втором — состояние кнопки «Ok».



В случае если мы хотим получить целое число, лучше воспользоваться методом `getInt()`, в котором мы можем указать значение по умолчанию, минимальное значение, максимальное значение и шаг.

```
age, ok_pressed = 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.

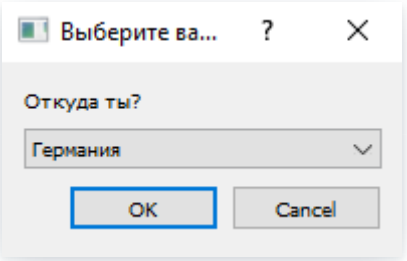


Иногда нам нужно, чтобы пользователь выбрал какое-то значение из предоставленных. Для этого есть метод `getItem`.

```
country, ok_pressed = QInputDialog.getItem(
    self, "Выберите вашу страну", "Откуда ты?",
```

```
    ("Россия", "Германия", "США"), 1, False)
```

В качестве параметров необходимо указать: родителя, заголовок окна, вопрос, итерируемый объект с вариантами ответа и индекс значения по умолчанию. А чтобы пользователь не смог что-то самостоятельно ввести, укажем следующим параметром False.

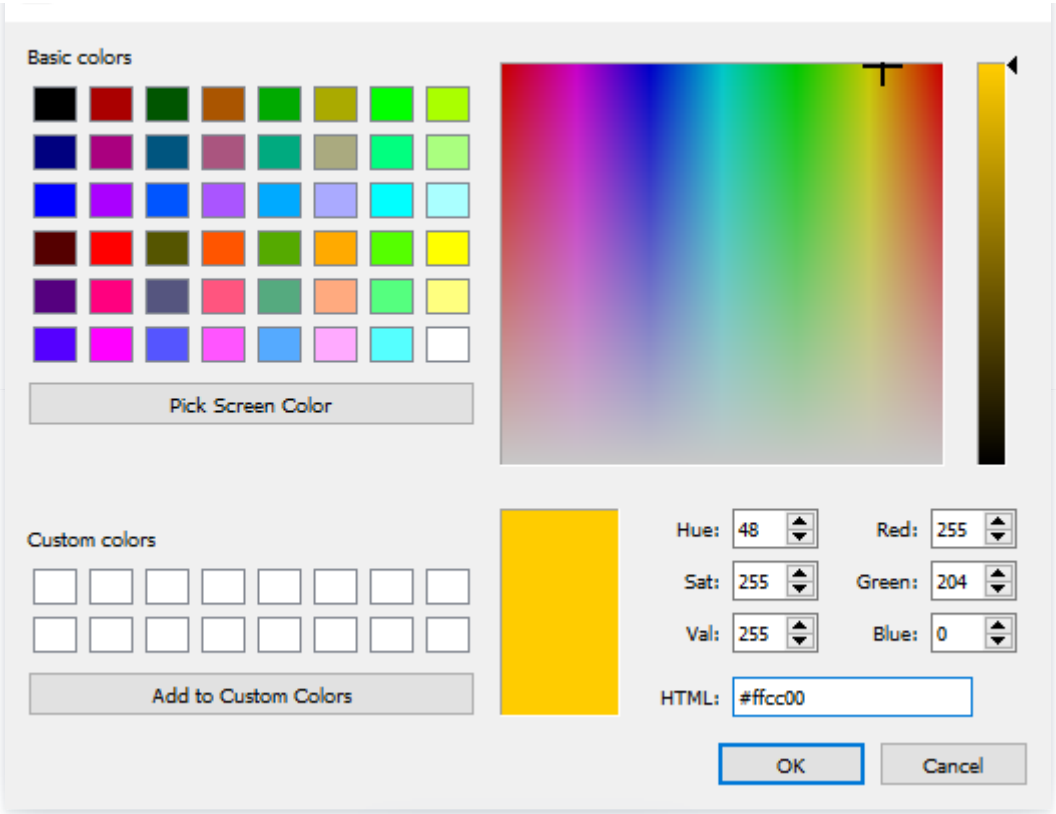


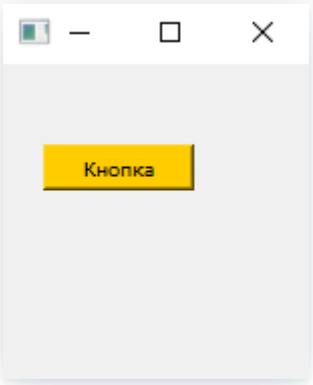
Для выбора цвета существует другой виджет — **QColorDialog**. Вот пример программы с его использованием:

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

    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()
    ex.show()
    sys.exit(app.exec())
```

Если цвет валиден, фон кнопки окрасится в него. Функция `.name()` возвращает цвет в шестнадцатитиричном формате. Управлять стилями элементов достаточно просто, синтаксис похож на CSS, с которым вы должны были познакомиться во время изучения HTML.

Примеры можно посмотреть в **документации**.



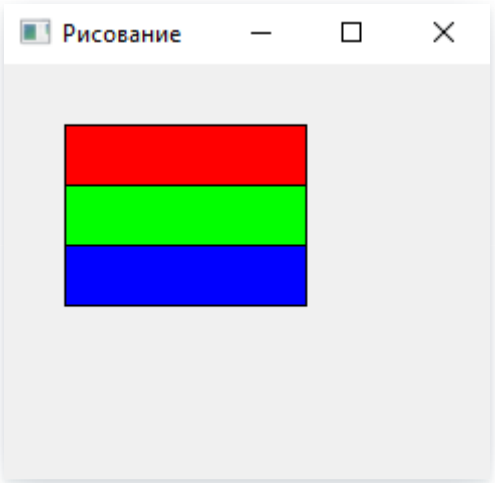


4. Рисование

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

```
import sys

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


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

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

    # Метод срабатывает, когда виджету надо
```

```
    # перерисовать свое содержимое,
    # например, при создании формы
    def paintEvent(self, event):
        # Создаем объект QPainter для рисования
        qp = QPainter()
        # Начинаем процесс рисования
        qp.begin(self)
        self.draw_flag(qp)
        # Завершаем рисование
        qp.end()

    def draw_flag(self, qp):
        # Задаем кисть
        qp.setBrush(QColor(255, 0, 0))
        # Рисуем прямоугольник заданной кистью
        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()
    ex.show()
    sys.exit(app.exec())
```



Заметьте: все работает, несмотря на то, что мы явно нигде не вызываем метод `paintEvent()`. Этот метод вызывается автоматически в те моменты, когда виджет понимает, что ему надо обновить свой внешний вид, например, при создании формы. Кроме того, в качестве параметра присутствует некий `event` — событие. Это событие, на которое подвешивается обработчик, например: движение, нажатие или отпускание кнопки мыши, нажатие или отпускание клавиши клавиатуры и многое другое.

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

В методе `drawFlag()` происходит непосредственно рисование. С помощью метода `setBrush` мы задаем цвет кисти в формате RGB. Этот метод принимает на вход не кортеж цветовых компонент, а экземпляр класса `QColor`. А он уже, в свою очередь, при создании может принимать цветовые компоненты RGB. Подробнее про возможности `QColor` можно посмотреть в **документации**.

Кроме метода `setBrush()`, у `QPainter` есть еще метод `setPen()`, который также принимает в качестве параметра объект `QColor`. Разница в том, что кисть задает цвет заливки, а ручка — цвет обводки.

Сам процесс рисования очень похож на тот, с каким мы сталкивались при изучении библиотеки Pillow. Например, чтобы нарисовать прямоугольник, применяем метод `drawRect()`. В качестве параметров ему передаются координаты левого верхнего угла, длина и высота. Как и в Pillow, существуют различные методы для рисования разных графических объектов, например:

- `drawArc()` — для рисования дуги
- `drawEllipse()` — для эллипсов
- `drawLine()` — для линий
- `drawPolygon()` — для многоугольников
- `drawText()` — для текста

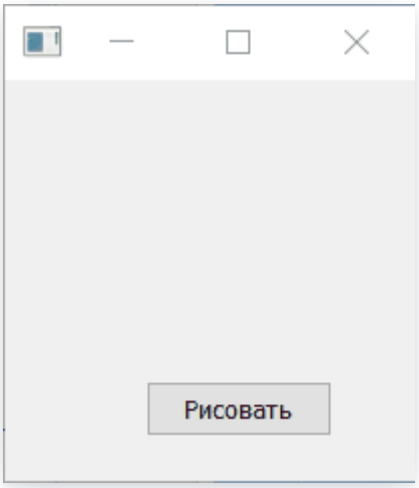
Подробнее про эти и другие методы `QPainter` можно почитать в **документации**.

Важное замечание: библиотека PyQT устроена таким образом, что рисовать можно **только** внутри `paintEvent()`, а сам `paintEvent()` вызывается не по всем событиям, например, нажатие кнопки само по себе не приводит с появлению сигнала на перерисовку. Но есть и хорошие новости: никто не запрещает вам рисовать в другом месте, например, на `QPixmap`, а потом отображать результат, как мы рассматривали в первом примере этого урока. Кроме того, вы в любой момент можете принудительно заставить форму перерисоваться, для этого надо вызвать у нее метод `update()`.

Давайте немного изменим предыдущий пример, чтобы рисование происходило только после нажатия на кнопку:

```
import sys

from PyQt5.QtGui import QPainter, QColor
from PyQt5.QtWidgets import QWidget, QApplication, QPushButton


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

    def initUI(self):
        self.setGeometry(300, 300, 200, 200)
        self.setWindowTitle('Рисование')
        self.btn = QPushButton('Рисовать', self)
        self.btn.move(70, 150)
        self.do_paint = False
        self.btn.clicked.connect(self.paint)

    def paintEvent(self, event):
        if self.do_paint:
            qp = QPainter()
            qp.begin(self)
            self.draw_flag(qp)
            qp.end()
        self.do_paint = False
```

```
    def paint(self):
        self.do_paint = True
        self.update()

    def draw_flag(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()
    ex.show()
    sys.exit(app.exec())
```



Рассмотрим еще один пример: напишем программу, которая будет рисовать пятиконечную звездочку. Для этого необходимо рассчитать точки. Пятиконечная звезда получается из правильного пятиугольника, если соединять точки через одну. Так что сначала нарисуем пятиугольник, а затем — звезду. Важно помнить, что наши координаты, в которых мы привыкли работать, могут не совпадать с экранными.

Для начала напишем функции, которые будут пересчитывать координаты в экранные.

```
def xs(self,x):
    return x + SCREEN_SIZE[0] // 2


def ys(self,y):
    return SCREEN_SIZE[1] // 2 - y
```

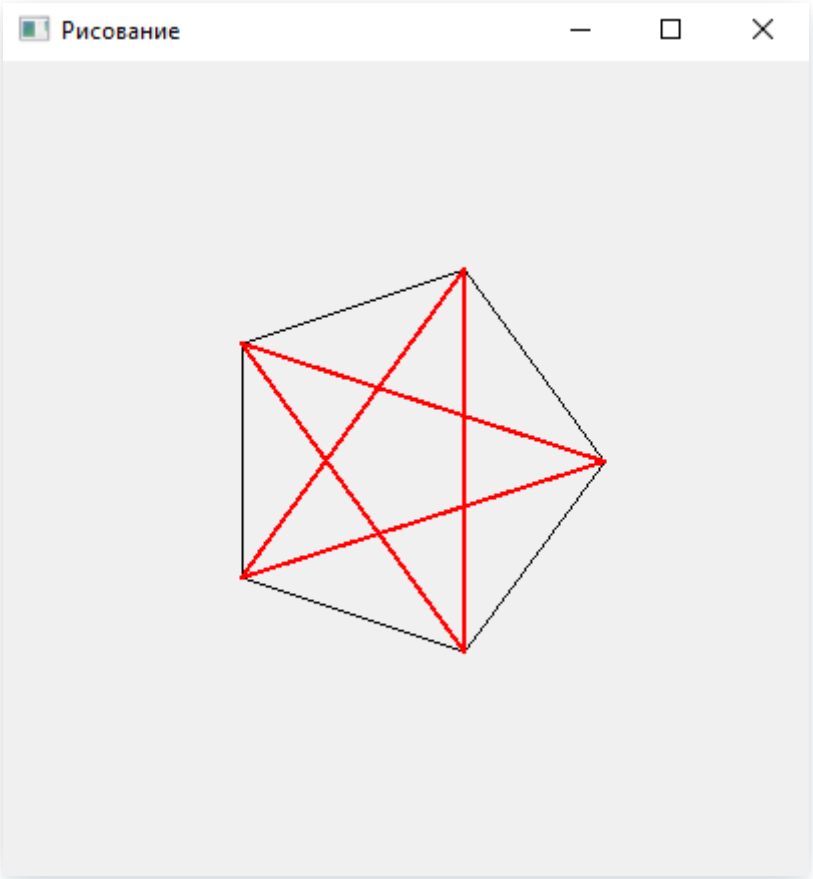
А теперь вместо метода `draw_flag()` напишем метод `draw_star()`. Для его работы, кроме тех модулей, про которые мы говорили выше, нам понадобятся объекты следующих модулей: `Qt`, `QPen`, `math`. Так что важно не забыть импортировать их.

```
import sys
```

```python
from math import cos, pi, sin

from PyQt5.QtCore import Qt
from PyQt5.QtGui import QPainter, QPen
from PyQt5.QtWidgets import QWidget, QApplication

SCREEN_SIZE = [500, 500]
# Задаём длину стороны и количество углов
SIDE_LENGTH = 200
SIDES_COUNT = 5


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

    def initUI(self):
        self.setGeometry(300, 300, *SCREEN_SIZE)
        self.setWindowTitle('Рисуем звезду')

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

    def xs(self, x):
        return x + SCREEN_SIZE[0] // 2

    def ys(self, y):
        return SCREEN_SIZE[1] // 2 - y

    def draw_star(self, qp):

        # Считаем координаты и переводим их в экранные
        nodes = [(SIDE_LENGTH * cos(i * 2 * pi / SIDES_COUNT),
                  SIDE_LENGTH * sin(i * 2 * pi / SIDES_COUNT))
                 for i in range(SIDES_COUNT)]
        nodes2 = [(int(self.xs(node[0])),
                   int(self.ys(node[1]))) for node in nodes]

        # Рисуем пятиугольник
        for i in range(-1, len(nodes2) - 1):
            qp.drawLine(*nodes2[i], *nodes2[i + 1])

        # Изменяем цвет линии
        pen = QPen(Qt.red, 2)
        qp.setPen(pen)

        # Рисуем звезду
        for i in range(-2, len(nodes2) - 2):
```

```
            qp.drawLine(*nodes2[i], *nodes2[i + 2])


if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = DrawStar()
    ex.show()
    sys.exit(app.exec())
```

Сначала мы высчитываем координаты и переводим их в экранные. Затем мы используем метод `drawLine` объекта класса `QPainter`, в него поступают координаты двух точек, между которыми строится линия. Пройдясь по всем точкам, мы получим пятиугольник. По умолчанию рисование происходит с помощью «кисти» черного цвета и с толщиной, равной единице. Создадим новую кисть `pen = QPen(Qt.red, 2)` и применим ее. А затем, пробегаясь по тем же точкам, но через одну, получим изображение звезды.



5. L-системы

В 1968 г. венгерский биолог и ботаник **Аристид Линденмайер** (Aristid Lindenmayer) предложил математическую модель для изучения развития простых многоклеточных организмов, которая позже была расширена и используется для моделирования сложных ветвящихся структур — разнообразных деревьев и цветов.

Эта модель получила название Lindenmayer System, или просто L-System. Свои идеи Линденмайер изложил в книге «Алгоритмическая красота растений» (The Algorithmic Beauty of Plants).

Идею можно объяснить так: сложный объект получается из простого с помощью перезаписи частей этого простого объекта по определенным ранее правилам. L-система является частным случаем фракталов.

Представьте, что вы — художник. Но у вас есть только карандаш, поле для творчества — лист бумаги, и вы умеете выполнять только определенный набор команд (например, «поставить карандаш на бумагу», «поднять карандаш в воздух», «нарисовать линию», «повернуться на угол» и т.д.) Для того чтобы нарисовать

на листе бумаги картину, вы получаете задание — строку символов, которую вам готовит L-автомат. Назовем эту строку **L-строкой**.

Итак, первый этап решения задачи — формирование L-строки, второй — построение по этой строке рисунка.

L-автомат не может сформировать L-строку «из неоткуда». Поэтому, у него есть **аксиома** — строка, которая определяет начало эволюции. Для дальнейшего развития L-автомат использует **теоремы** — правила, по которым и будет происходить преобразование L-строки.

Чтобы построить L-систему мы определим символы, которыми кодируется L-строка и опишем действия, которые производит художник, обрабатывая любой из символов.

Символ	Действие
F	Начертить отрезок из точки, в которой находится карандаш под текущим углом, и остаться в новой точке.
f	Перенести карандаш из точки под текущим углом и остаться в новой точке, но не чертить отрезок.
+	Повернуться на заданный угол по часовой стрелке.
-	Повернуться на заданный угол против часовой стрелки.
[	Сохранить текущее состояние карандаша.
]	Вернуться в предыдущее сохраненное состояние карандаша.
|	Повернуться на 180 градусов.

Необходимо пояснить некоторые моменты. Что такое **текущий угол** и **состояние карандаша**? Дело в том, что перед построением рисунка мы должны определиться со значением угла, на который мы будем поворачиваться в процессе работы, а под состоянием карандаша мы будем понимать набор из координат карандаша на листе и значения угла. Договоримся, что изначально угол равен нулю, а карандаш расположен в начале координат. Кроме того, надо знать длину шага для рисования.

Мы уже упомянули, что каждая фигура определяется своей аксиомой и набором теорем.

Например, аксиома может быть задана так:

```
F
```

Это означает, что наша система изначально представлена L-строкой F. То есть для того чтобы ее построить, мы возьмем из списка правил действие для строки F и выполним его, то есть построим линию.

Допустим, у нас есть только одна теорема и она выглядит так:

```
F -> F-F++F-F
```

Это означает, что на каждом шаге для получения новой эволюции L-строки мы должны каждый символ F заменить на последовательность: F-F++F-F.

Посмотрите на то, что будет представлять L-строка на первых 3-х шагах алгоритма.

Этап	L-строка
0	F
1	F-F++F-F

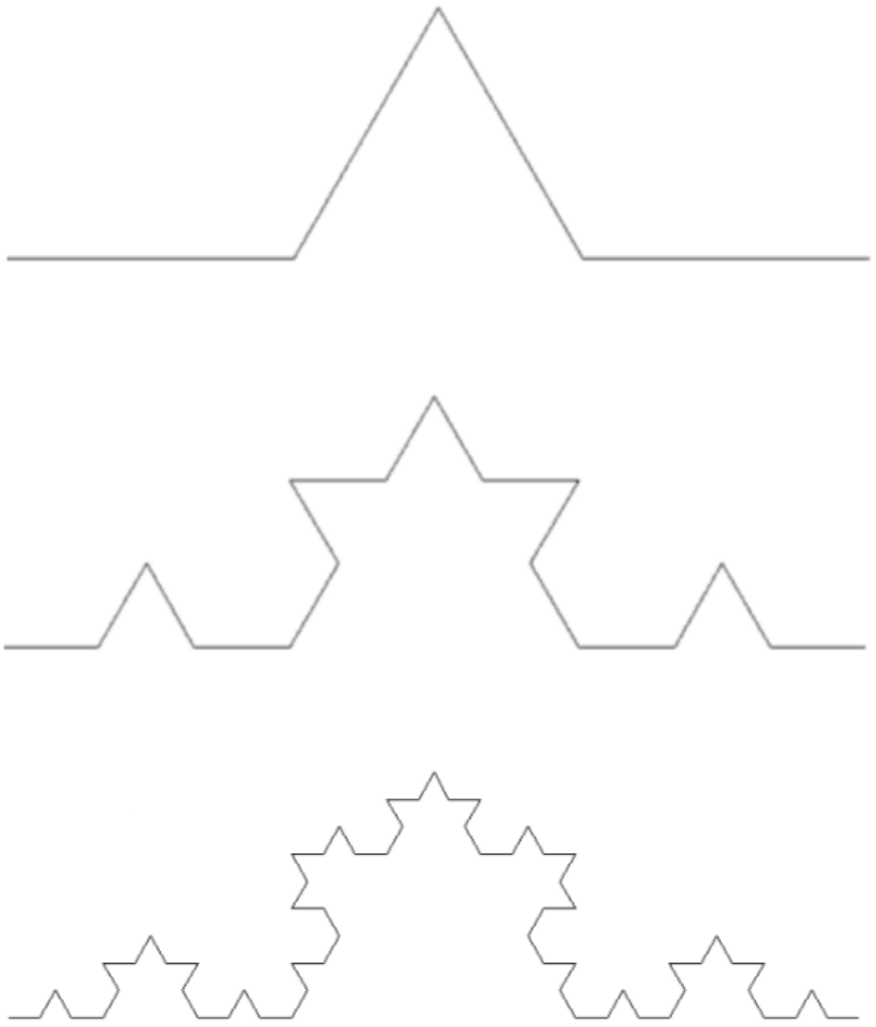
2	F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F
3	F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F++F-F-F-F++F-F++F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++F-F-F- F++F-F

В результате L-автомат будет содержать L-строку, которая описывает заданный фрактал — кривую Коха.

Вспомним про начальные условия и зададим угол поворота в 60 градусов.

Нулевой шаг алгоритма, L=F — рисуем отрезок.

Первый шаг алгоритма, L=F-F++F-F — рисуем отрезок, затем поворачиваемся против часовой стрелки на 60 градусов, затем опять рисуем отрезок, затем следует поворот на 120 градусов по часовой стрелке, рисование отрезка, поворот на 60 градусов против часовой стрелки и — ура — последний отрезок.



Рассмотрим более сложную L-систему:

Угол: 90 градусов

Аксиома: X

Теоремы:

X => -YF+XFX+FY-

Y => +XF-YFY-FX+

В этой L-системе присутствуют символы, которые не входят в перечень действий карандаша. В этом нет ничего страшного. При создании L-строки такие символы будут использованы, а при построении фрактала — нет, они

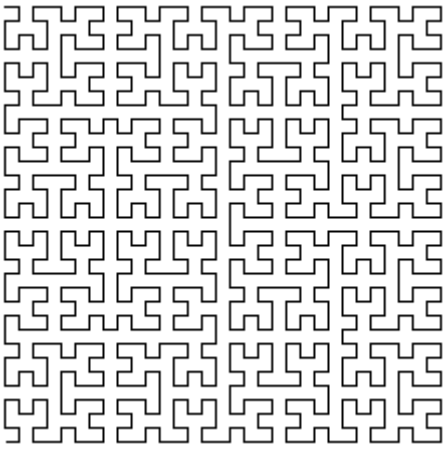
будут просто проигнорированы.

Этап	L-строка
0	X
1	-YF+XFX+FY-
2	-+XF-YFY-FX+F+-YF+XFX+FY-F-YF+XFX+FY-+F+XF-YFY-FX+-

При построении первого шага мы:

1. Повернемся на 90 градусов против часовой стрелки
2. Y – проигнорируем
3. Нарисуем отрезок
4. Повернемся на 90 градусов по часовой стрелке
5. X – проигнорируем
6. Нарисуем отрезок
7. Опять проигнорируем X
8. Повернемся на 90 градусов по часовой стрелке
9. Нарисуем отрезок
10. Y – проигнорируем
11. Повернемся на 90 градусов против часовой стрелки

И в результате получим ... кривую Гильберта. Нужно отметить, что фрактал нулевой глубины в данном случае не имеет картинки, поскольку L-строка содержит символ «X», который никак не отображается. А вот для фрактала пятой глубины получим:



Остановимся немного на поворотах. Допустим ваш карандаш находится в точке с координатами (x_0, y_0), а текущий угол — α. Пусть длина шага равна — l. Тогда координаты карандаша после окончания рисования можно вычислить по следующим формулам:

$$\begin{cases} x = x_0 + l * \cos\alpha \\ y = y_0 + l * \sin\alpha \end{cases}$$

При этом углы должны быть указаны в **радианах**. Если же мы указываем углы в градусах, то формулы приобретают следующий вид:

$$\begin{cases} x = x_0 + l * \cos \frac{\alpha * \pi}{180} \\ y = y_0 + l * \sin \frac{\alpha * \pi}{180} \end{cases}$$

Поворот на определенный угол сводится к следующему:

```
alpha = (alpha + phi) % 360
```

Все необходимые функции: синус, косинус, перевод градусов в радианы и обратно — можно найти в модуле math, который входит в стандартную библиотеку.