Программирование

на языке

Python

Блок 2:

Не самое необходимое, но очень важное.

В. Б. Пикулев,
доцент КФТТ ПетрГУ.

Scilink.ru, 2024

Проблемы процедурного программирования

- Любую функцию, которая использует и/или изменяет глобальные данные, сложно повторно использовать в другой программе. Иными словами, нельзя взять функцию, которая использует инструкции **global**, и использовать ее в другой программе без изменения кода этой функции.
- Процедурная парадигма позволяет писать программы с использованием самых разнообразных приемов, что хорошо только для понимания языка. Однако если над проектом работает несколько программистов, и проект длится несколько лет, то требуется создать и беспрекословно соблюдать определённый **кодекс программирования**, а это слишком сложно реализовать внутри процедурной парадигмы.
- Все данные процедуры доступны только внутри нее. Их нельзя вызвать из другого места программы и при необходимости придется писать аналогичный код, что противоречит одному из основополагающих принципов программирования: **Don’t Repeat Yourself**.

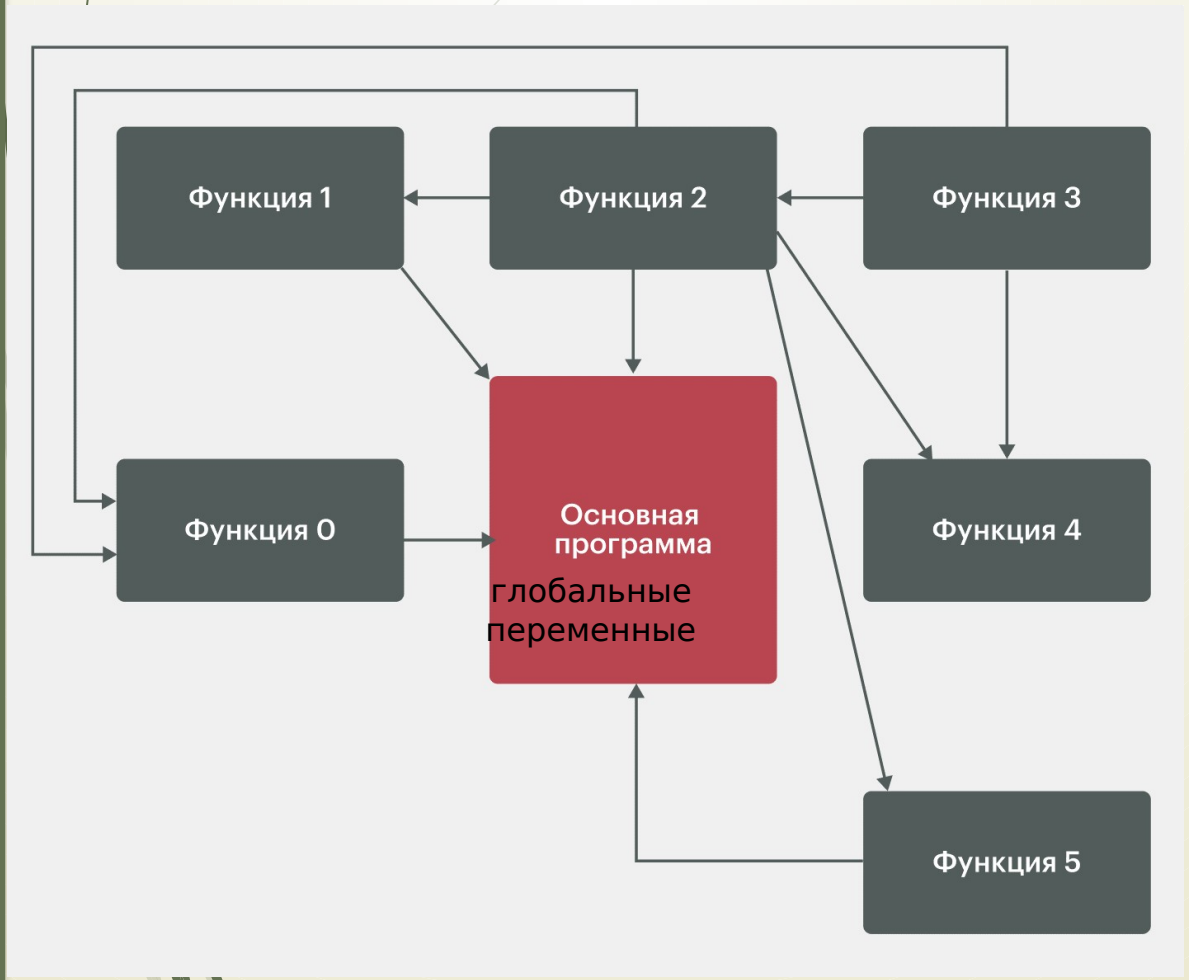
```
# Процедурный выключатель света
def turnOn():
    global switchIsOn
    # включаем свет
    switchIsOn = True
def turnOff():
    global switchIsOn
    # выключаем свет
    switchIsOn = False

# Основной код
# глобальная логическая переменная
switchIsOn = False
# код теста
print(switchIsOn)
turnOn()
print(switchIsOn)
turnOff()
print(switchIsOn)
turnOn()
print(switchIsOn)
```

Процедурное программирование можно сравнить с постройкой маленького домика - нет необходимости тратить время и ресурсы на продумывание архитектуры. ООП же похоже на создание сложного архитектурного сооружения, где очень важно сначала продумать и описать все конструктивные элементы, и только потом приступать к реализации.

Процедурный подход vs ООП

3

Каждый объект в ООП создаётся по определённому классу — абстрактной модели, описывающей, из чего должен состоять объект и что с ним можно делать.



ООП: Класс и объект

Класс – это определяемый пользователем тип, который предназначен для создания своих экземпляров (объектов).

- **Класс** – это код, который определяет, что созданных на его основе объект запомнит (данные и состояние) и что он сможет сделать (т.е. его функции и поведение).
- **Класс** – это образец, на основе которого создаются объекты одного типа. Класс определяет набор атрибутов, методов и свойств, поддерживаемых соответствующими объектами.
- **Объект класса** – это сложная структура, хранящая множество каких-либо значений и содержащая инструменты для их обработки.
- **Объект класса** можно вызвать так, как если бы это была функция. Такой вызов (инстанциализация) возвращает объект, называемый экземпляром класса.
- Объекты независимы друг от друга и **самодостаточны**, так что, если что-то меняется в одном объекте, это никак не отражается на других.
- **Все** значения, обрабатываемые Python, представляют собой объекты. Стандартная библиотека Python содержит множество классов, называемых *встроенными*. На их основе можно создавать свои, *пользовательские* классы.

```
# ОО выключатель света
# описываем класс
class LightSwitch():
    def __init__(self):
        self.switchIsOn = False
    def turnOn(self):
        # включаем свет
        self.switchIsOn = True
    def turnOff(self):
        # выключаем свет
        self.switchIsOn = False
    def show(self):
        # добавлено для самотестирования
        print(self.switchIsOn)

# создаём объект
oLightSwitch = LightSwitch()
# тестируем объект
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
oLightSwitch.turnOff()
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
```

Имя класса должно удовлетворять тем же требованиям, что и имена переменных. Имена классов принято писать с прописной буквы, а объектов — со строчной. Self можно (но нежелательно) заменить на любое другое слово.

Атрибуты и методы

Атрибут – это дескрипторы или данные внутри класса. Очень часто говорят, что атрибут – это любая переменная, определённая внутри класса, значение которой принадлежит только своему объекту.

- **Метод** – это функция, определённая внутри класса, которая может взаимодействовать с атрибутами своего объекта. Методы могут использовать переменные экземпляра класса, вызываемые в виде self.<variableName>.
- Первым обязательным параметром метода является ключевое слово **self**, которое символизирует ссылку на родной объект, чтобы в коде метода можно было обратиться к атрибутам и методам текущего объекта. Self отличает *метод класса* от обычной функции. Вызывая метод, self всегда игнорируем.

```
# ОО выключатель света
# описываем класс
class LightSwitch():
    def __init__(self):
        self.switchIsOn = False
    def turnOn(self):
        # включаем свет
        self.switchIsOn = True
    def turnOff(self):
        # выключаем свет
        self.switchIsOn = False
    def show(self):
        # добавлено для самотестирования
        print(self.switchIsOn)

# создаём объекты
oLightSwitch1 = LightSwitch()
oLightSwitch2 = LightSwitch()
# тестируем объекты
oLightSwitch1.turnOn()
oLightSwitch1.show() # True
oLightSwitch2.turnOff()
oLightSwitch2.show() # False
oLightSwitch1.show() # True
```

```
class Самый_простой_класс():
    A = 1

самый_простой_объект = Самый_простой_класс()
print(самый_простой_объект.A) # 1
самый_простой_объект.A = 2
print(самый_простой_объект.A) # 2
```

В теле класса можно определить любое количество методов. Все методы оформляются как функции и считаются частью класса, поэтому код, который их определяет, должен начинаться отступом. Каждая функция представляет определенное поведение, которое может выполнять объект, созданный из класса.

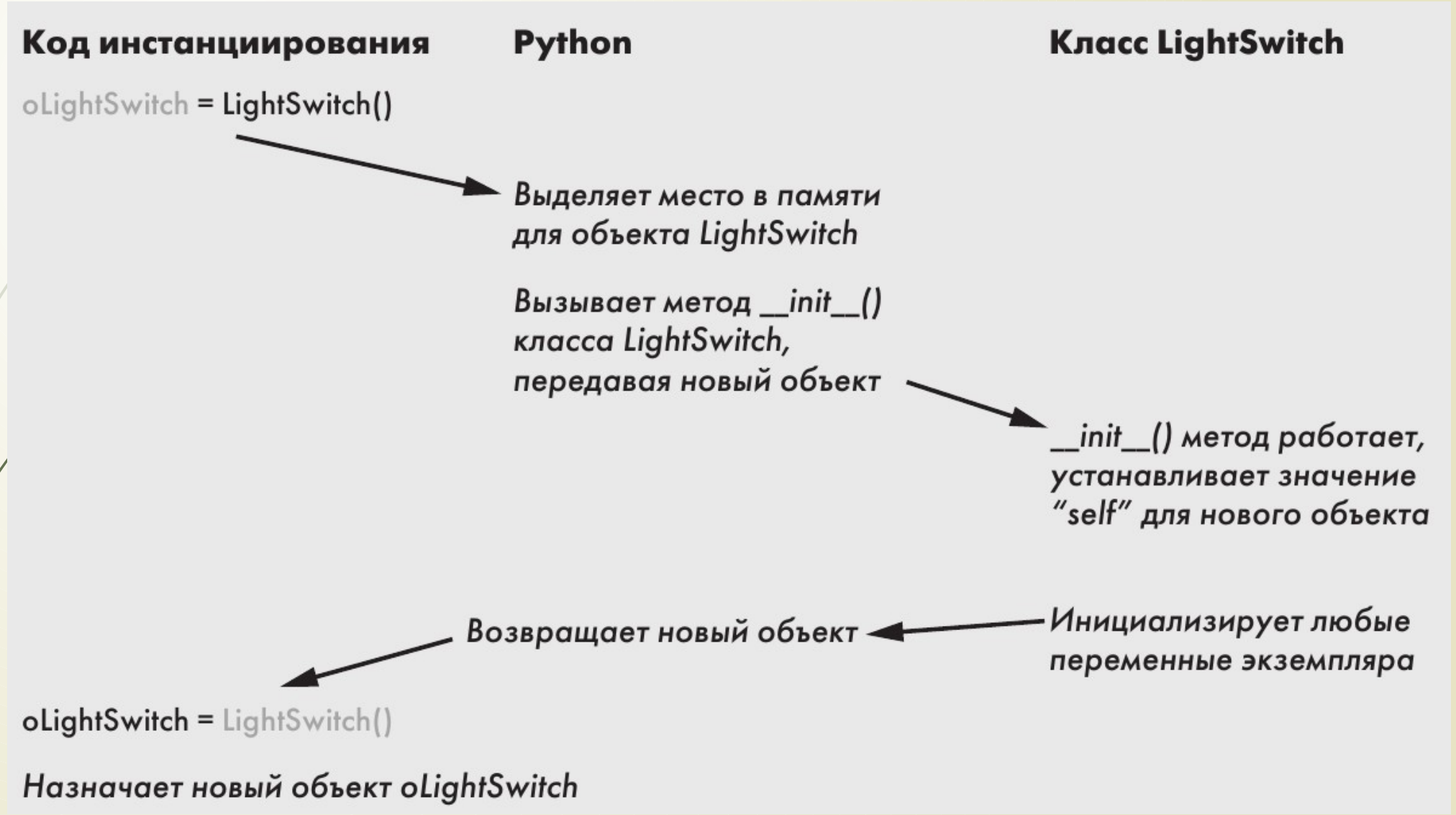
Конструктор и деструктор

- Конструктор __init__ (двойное подчёркивание до и после имени) – метод, который автоматически вызывается при создании любого нового объекта данного класса. Метод обычно используется для создания всех переменных объекта и присвоения им начальных значений. Первым параметром конструктору, как и другим методам, передаётся ссылка на текущий объект. Остальные параметры обычно служат для инициализации начальных значений атрибутов объекта. Метод __init__ не должен возвращать значение, отличное от None.
- Деструктор __del__ – метод, который автоматически вызывается перед уничтожением объекта данного класса, когда перестаёт существовать последняя ссылка на этот объект. У него только один параметр – self.
- Метод __new__ непосредственно создает новый экземпляр класса и выполняется перед __init__, однако дополнять этот метод своим содержанием обычно не требуется.

```
# ОО выключатель света
# описываем класс
class LightSwitch():
    def __init__(self):
        self.switchIsOn = False
    def turnOn(self):
        # включаем свет
        self.switchIsOn = True
    def turnOff(self):
        # выключаем свет
        self.switchIsOn = False
    def show(self):
        # добавлено для самотестирования
        print(self.switchIsOn)

# создаём объект
oLightSwitch = LightSwitch()
# тестируем объект
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
oLightSwitch.turnOff()
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
```

```
class Самый_простой_класс():
    A = 1
    def __init__(self, forA):
        self.A = forA
```

Особенности создания объектов



Инкапсуляция и свойства

Инкапсуляцией называют присутствие в объекте данных и методов для их обработки, при этом доступ к данным возможен исключительно через эти методы.

В соответствии с общепринятым соглашением идентификаторы, начинающиеся с _ одиночного символа подчеркивания, рассматриваются как частные (открытые только для методов своего класса) переменные. Однако реального скрытия при этом не происходит. Декоратор `@property` (**свойство**) делает метод по обращению похожим на переменную, при этом доступную только для чтения; для изменения значения внутренней переменной требуется написать соответствующий метод-"сеттер". Атрибуты и методы, чьи имена начинаются с __ двойного подчёркивания – доступны только внутри класса.

```
class Cat():
    def __init__(self,
                 breed='Неизвестная',
                 color='Неизвестный', age=0):
        self._breed = breed
        self._color = color
        self._age = age
    @property
    def breed(self):
        return self._breed
    @property
    def color(self):
        return self._color
    @property
    def age(self):
        return self._age
    @age.setter
    def age(self, new_age):
        if new_age > self._age:
            self._age = new_age
        return self._age

cat1 = Cat('Абиссинская', 'Рыжая', 3)
cat2 = Cat('Беспородная', 'Белая', 10)
cat3 = Cat()

print(cat1.breed, cat1.color, cat1.age)
print(cat2.breed, cat2.color, cat2.age)
print(cat3.breed, cat3.color, cat3.age)
cat1.age = 4 # изменим возраст
print(cat1.breed, cat1.color, cat1.age)
```

Разрешаем изменять только атрибут «возраст», но и то только в большую сторону, а атрибуты «порода» и «цвет» лучше открыть только для чтения, поскольку они не меняются.

Наследование

9

Наследование – это создание производного класса (дочернего) на основе другого (родительского, или суперкласса). Производный класс получает все атрибуты и методы родительского.

Наследование в Python может быть как одиночным, так и множественным.

```
class Horse():
    isHorse = True
class Donkey():
    isDonkey = True
class Mule(Horse, Donkey):
    isMule = True

donkey = Donkey()
mule = Mule()

print(donkey.isDonkey) # True
print(donkey.isHorse) # ошибка
print(mule.isHorse) # True
print(mule.isDonkey) # True
```

```
class HomeCat(Cat):
    def __init__(self, breed, color, age, owner, name):
        super().__init__(breed, color, age)
        self._owner = owner
        self._name = name

    @property
    def owner(self):
        return self._owner

    @property
    def name(self):
        return self._name

cat1 = HomeCat('Сибирская', 'Чёрно-белая', 8, 'Виталий',
'Мурка')
for i in vars(cat1):
    print(i, vars(cat1)[i])
```

Если в родительском и в дочернем классах присутствуют методы с одинаковыми именами, при вызове такого метода из объекта дочернего класса будет также вызываться метод дочернего класса. Иными словами, осуществляется **перекрытие** методов.
Если вызывается метод дочернего класса, в котором, в свою очередь, предусмотрен вызов метода родительского класса (т.е. функциональность родительского класса дополняется инструкциями дочернего), то говорят о **переопределении** метода.

Не канонично, но удобно для ООП

10

Pygame — фреймворк языка Python для поддержки программирования игр.

```
import pygame
import random
WIDTH = 800
HEIGHT = 600
WHITE = (255, 255, 255)
BLUE = (0, 0, 255)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE_BLOBS = 10
RED_BLOBS = 6
GREEN_BLOBS = 8
```

```
class Blob:
    # некий двумерный объект
    def __init__(self, color, size):
        # определяем его местоположение и цвет
        self.width = size[0]
        self.height = size[1]
        self.x = random.randrange(0, self.width)
        self.y = random.randrange(0, self.height)
        self.size = random.randrange(4, 8)
        self.color = color
    def move(self):
```

```
def draw_environment(blob_list):
    game_display.fill(WHITE)
    # залить кадр фоновым цветом
    # это кружок!
    for blob_dict in blob_list:
        for blob_id in blob_dict:
            blob = blob_dict[blob_id]
            pygame.draw.circle(game_display,
blob.color, [blob.x, blob.y], blob.size)
            blob.move()

    pygame.display.update()
    # вывести кадр на экран
```

```
def main():
    blue_blobs = dict(enumerate([Blob(BLUE, (WIDTH, HEIGHT)) for i in range(BLUE_BLOBS)]))
    red_blobs = dict(enumerate([Blob(RED, (WIDTH, HEIGHT)) for i in range(RED_BLOBS)]))
    green_blobs = dict(enumerate([Blob(GREEN, (WIDTH, HEIGHT)) for i in range(GREEN_BLOBS)]))
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit() # реакция на попытку закрытия окна
                quit()
        draw_environment([blue_blobs, red_blobs, green_blobs])
        clock.tick(60) # удержание 60 кадров в секунду
if __name__ == '__main__':
    main()
```

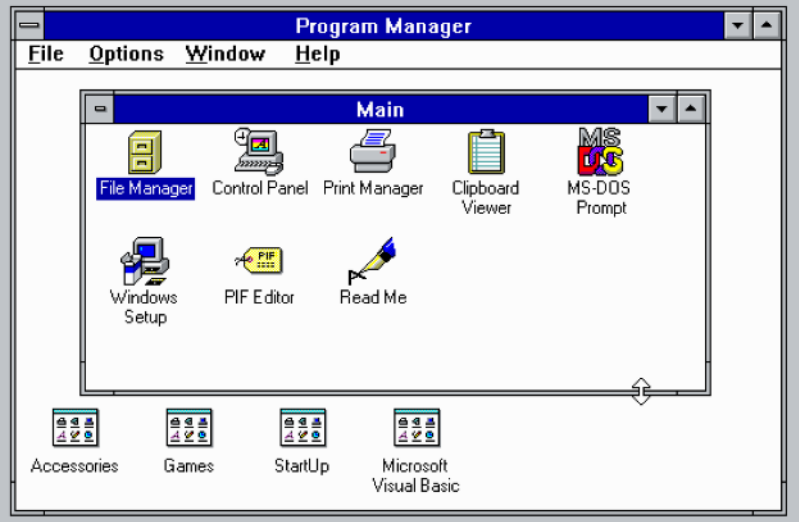
```
game_display = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption('Игровое поле')
clock = pygame.time.Clock()
```

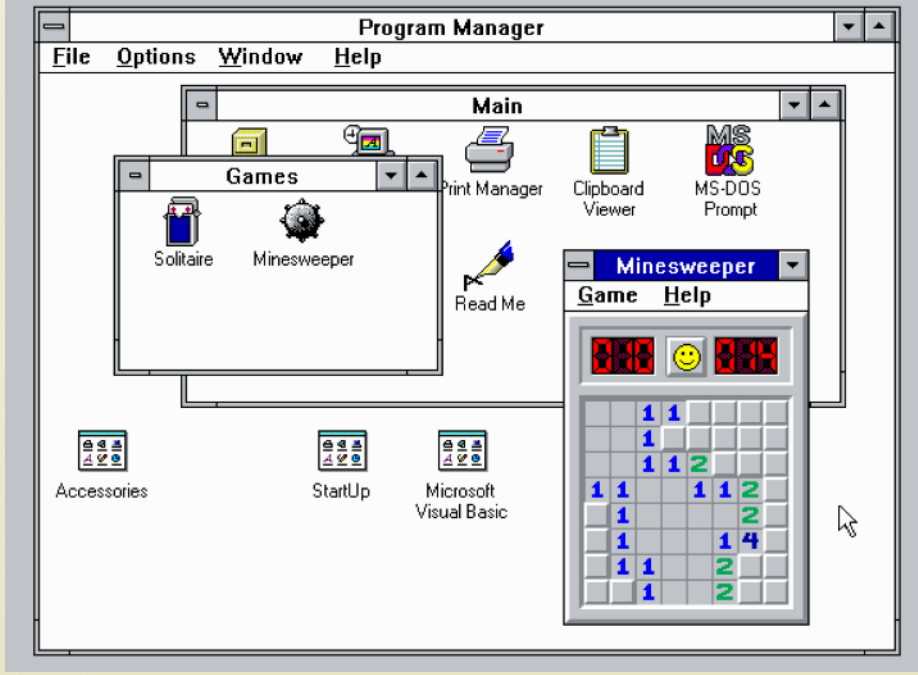
Программирование сложно назвать наукой.
По сути, это такое же ремесло, как, к примеру, кузнечное дело.

scilink.ru

Окна. Как это было.

Рождение графического интерфейса пользователя (GUI) не стоит ассоциировать с Биллом Гейтсом и **Microsoft Windows**, хотя для многих пользователей именно она явилась первой операционной системой с "графическими окнами". Однако ещё в 1968 году **Stanford's NLS** публично представила концепцию мыши и окон. За этим последовала система **Xerox PARC Smalltalk GUI** (1973), которая является основой большинства современных универсальных графических интерфейсов. Эти ранние системы уже имели многие из функций, которые считаются само собой разумеющимися в современных настольных графических интерфейсах: это окна, меню, переключатели, флажки, значки (windows, icons, menus, pointing device — **WIMP**). В 1979 году была выпущена первая коммерческая система с графическим интерфейсом — рабочая станция **PERQ**. Это подстегнуло ряд других производителей компьютеров (в том числе — персональных) по созданию графического интерфейса, включая, безусловно, **Apple Lisa** (1983), которая добавила концепцию панели меню и элементов управления окнами. На ОС UNIX графическая X Window System появилась в 1984 году, а первая версия Microsoft Windows для персональных компьютеров была выпущена всего лишь в 1985 году.





https://www.pcjs.org/software/pcx86/sys/windows/3.10/

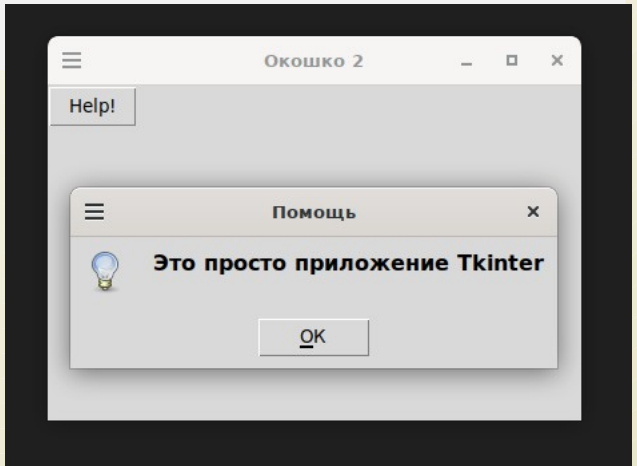
GUI. Как это в Python?

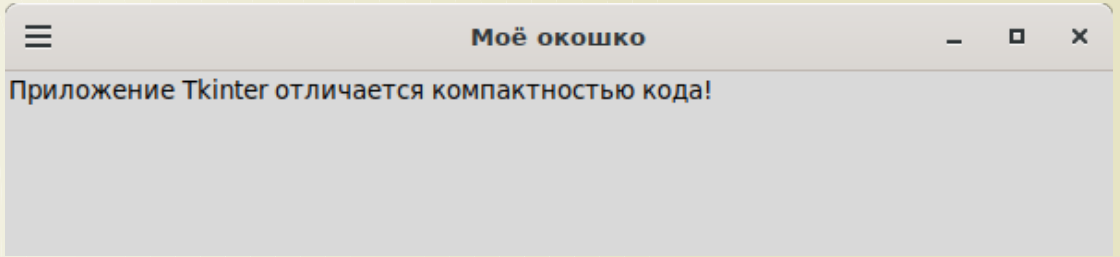
13

1. Tkinter

Tkinter – это свободно распространяемая библиотека для Python, предназначенная для работы с компонентами более общей библиотеки Tk, содержащей компоненты графического интерфейса пользователя. Фактически модуль Tkinter можно представить как переводчик с Python на язык Tcl. Графический интерфейс, разрабатываемый с помощью Tkinter – платформонезависимый (Windows, Linux)

Tkinter удобно применять для написания относительно простых оконных приложений с небольшим набором визуальных элементов. Серьёзные приложения будет сложно проектировать даже с точки зрения дизайна виджетов. Библиотека Tkinter установлена в Python в качестве стандартного модуля, т.е. её не нужно устанавливать, на неё распространяется общая Python-лицензия.






```
from tkinter import *

window = Tk()
window.title("Моё окошко")
window.geometry('600x100')
lbl = Label(window,
    text="Приложение Tkinter отличается компактностью кода!")
lbl.grid()
window.mainloop()
```

```
from tkinter import *
from tkinter import messagebox

def clicked():
    messagebox.showinfo('Помощь',
     'Это просто приложение Tkinter')


window = Tk()
window.title("Окошко 2")
window.geometry('400x250')
btn = Button(window, text='Help!', command=clicked)
btn.grid(column=0, row=0)
window.mainloop()
```

GUI. Как это в Python?

2. PyQt

Qt — это бесплатная и открытая библиотека кодов виджет-инструментария для языка C++ для создания кроссплатформенных приложений с графическим интерфейсом, позволяющий приложениям одинаково успешно запускаться на любых графических платформах (Windows, macOS, Linux, Android), при этом никаких изменений в код создаваемой программы вносить не нужно. Существуют вариации Qt для других языков: **PyQt** для Python, QtRuby для Ruby, Qt Jambi для Java. С использованием Qt написаны, в частности, мессенджер Telegram, продукты Autodesk, окружение рабочего стола для многих систем под Linux. Конкретно на PyQt написаны, например, Cura, Arduino IDE, Electrum, Calibre.

Qt — это гораздо больше, чем просто набор виджет-инструментов для реализации встроенных функций поддержки мультимедиа, баз данных, векторной графики и интерфейсов MVC. В первую очередь это объектно-ориентированный фреймворк для разработки графических приложений, определяющий свой стиль и методологию разработки программ.

Qt был создан Eirik Chambe-Eng и Haavard Nord в 1991 году, основавшими первую компанию Trolltech в 1994 году. В настоящее время Qt разрабатывается компанией The Qt Company и продолжает регулярно обновляться, добавляя функции и расширяя мобильную и кроссплатформенную поддержку.

Поскольку Qt в первую очередь предназначен для написания GUI на **C++**, это один из самых быстрых и надёжных графических интерфейсов. Для Python-разработок это существенное достоинство.

У официального Qt тройное лицензирование. Существуют три варианта библиотеки, каждый из них — под своей лицензией. Один предназначен для коммерческой разработки, второй — для проектов с открытым исходным кодом, третий — для персональных проектов. Для коммерческих проектов нельзя использовать бесплатную версию.

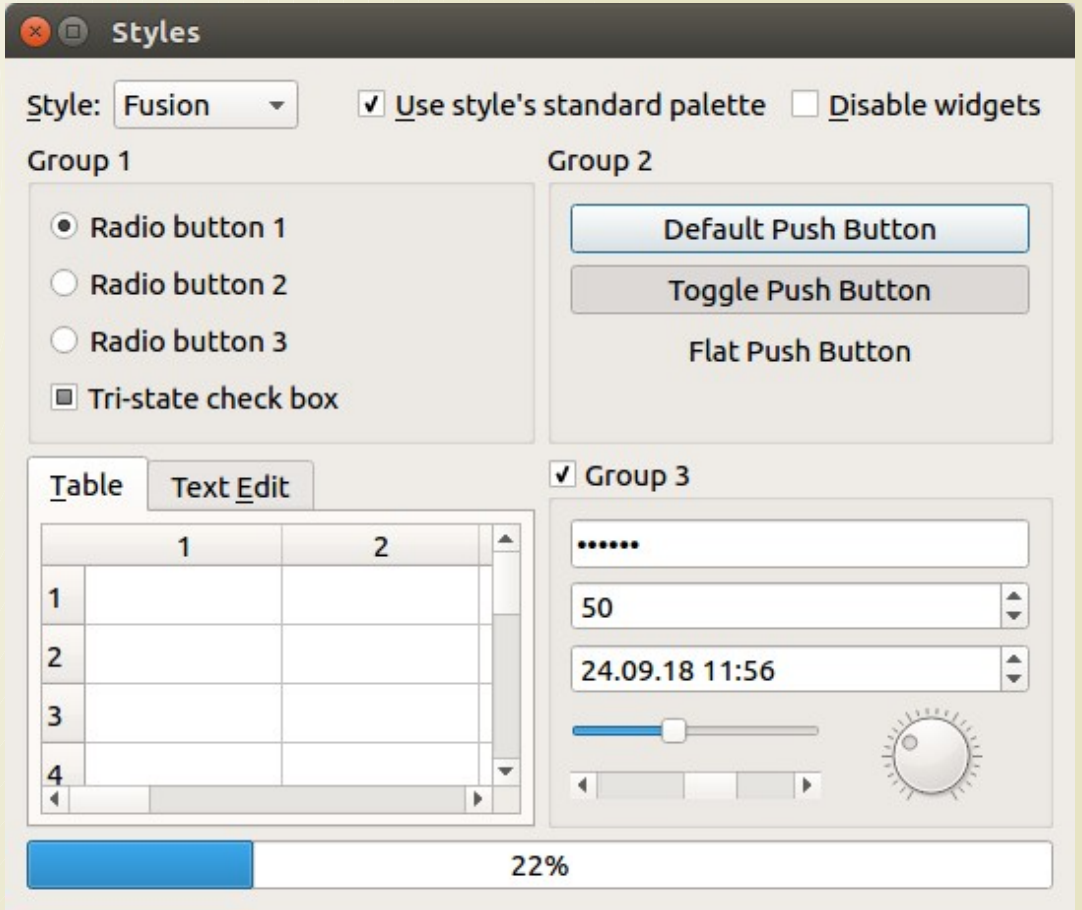
Библиотека **PySide** является вторым вариантом привязки инструментария Qt к языку Python. Основное отличие между ней и PyQt заключается в более простой и удобной лицензии LGPL. В частности, PySide разрешается свободно использовать в коммерческих проектах.

PyQt. Общая информация.


```
> pip install pyqt5, pyqt5designer, pyqt5-tools
```

Чтобы написать GUI-программу, надо выполнить следующие действия:

- ✔ Описать главное окно.
- ✔ Описать виджеты и выполнить конфигурацию их свойств (опций).
- ✔ Определить события, на которые будут откликаться виджеты и окно программы.
- ✔ Описать обработчики соответствующих событий.
- ✔ Подключить виджеты к главному окну.
- ✔ Запустить цикл обработки событий, при этом ранее спроектированное окно появляется на экране и начинает реагировать на события, происходящие в его зоне ответственности.



Взаимодействие между различными компонентами пользовательского интерфейса в PyQt5 осуществляется с помощью:

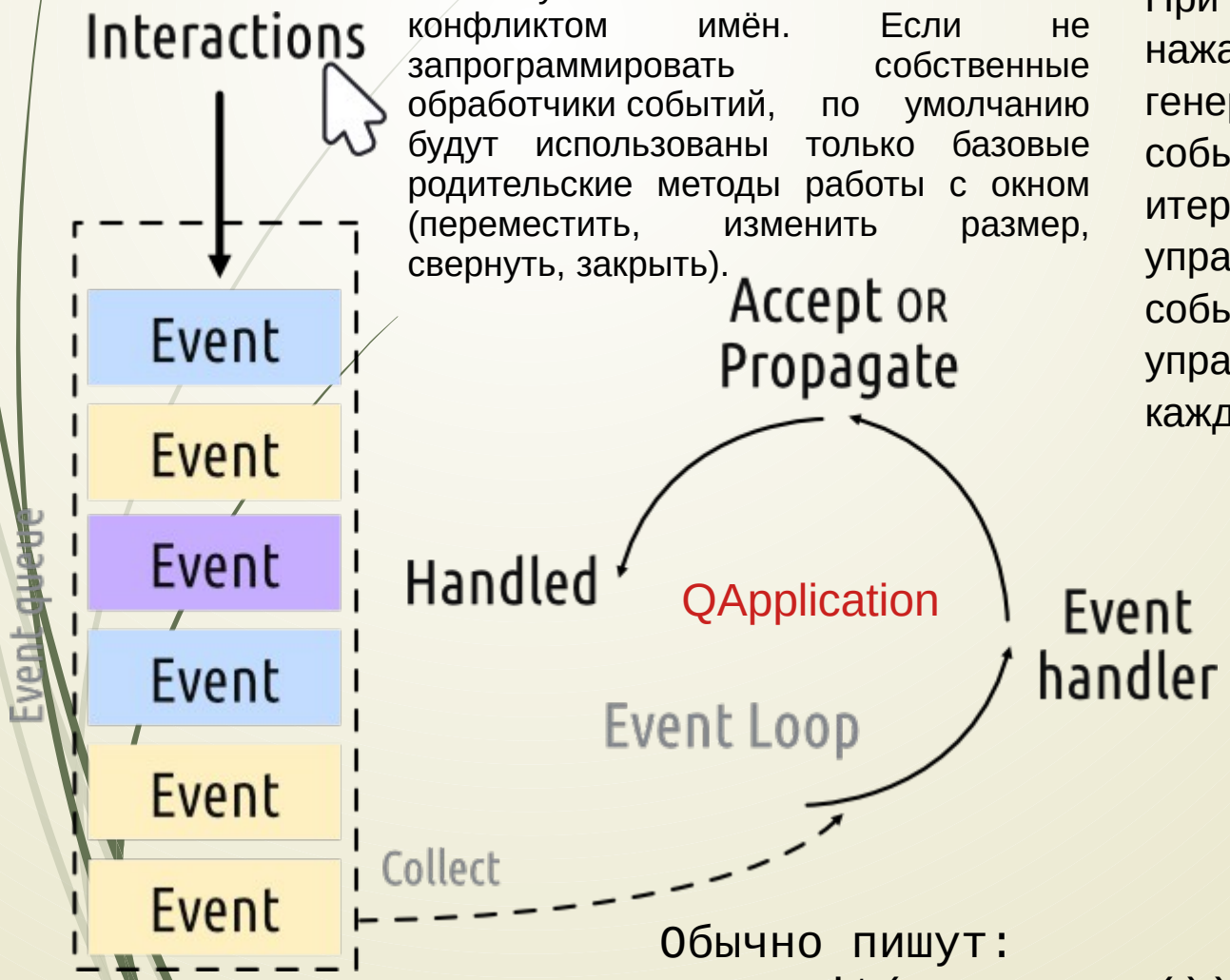
- ✔ сигналов — событий, возникающих при определенных действиях пользователя, таких как нажатие кнопки или изменение значения поля ввода;
- ✔ слотов — функций, которые выполняются в ответ на сигналы.

наиболее распространенные виджеты Qt

QLabel, QComboBox, QCheckBox, QRadioButton,QPushButton, QTableWidget, QLineEdit, QSlider,QProgressBar

PyQt5. Обработка событий.

app.exec_() передает управление модулю Qt, который закончит работу только тогда, когда пользователь закроет его из GUI. Подчёркивание используется в связи с возможным конфликтом имён. Если не запрограммировать собственные обработчики событий, по умолчанию будут использованы только базовые родительские методы работы с окном (переместить, изменить размер, свернуть, закрыть).

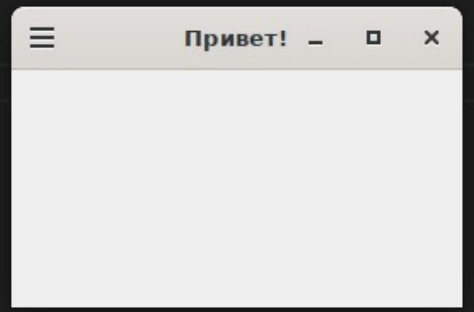


Обычно пишут:
`sys.exit(app.exec_())`

Основной элемент всех приложений в Qt — класс **QApplication**. Для работы каждому приложению нужен один (и только один) объект QApplication, который содержит цикл событий приложения. Это основной цикл, управляющий всем взаимодействием пользователя с графическим интерфейсом. При каждом взаимодействии с приложением — будь то нажатие клавиши, щелчок или движение мыши — генерируется событие, которое помещается в очередь событий. В цикле событий очередь проверяется на каждой итерации: если найдено ожидающее событие, оно вместе с управлением передаётся определённому обработчику этого события. Последний обрабатывает его, затем возвращает управление в цикл событий и ждёт новых событий. Для каждого приложения выполняется только один цикл событий.

qt6_0.py > ...

```
1  from PyQt6.QtWidgets import QApplication, QWidget
2  app = QApplication([])
3  w = QWidget()
4  w.setWindowTitle('Привет!')
5  w.show()
6  app.exec()
7
```



PyQt5. Базовые классы.

В Qt все построено на объектах, главным базовым классом является **QObject**. Все классы, имеющие сигналы и слоты в Qt унаследованы от него. QObject содержит в себе поддержку сигналов и слотов. Сигнал можно соединять с различным количеством слотов, при этом высылаемый сигнал поступит ко всем подключенным слотам.

Qt5 включает в себя около 620 классов и ~ 6000 функций и методов.

- **QtCore** - содержит классы, не связанные с реализацией графического интерфейса. Используется для работы со временем, файлами, потоками, URL. От этого модуля зависят все остальные модули
- **QtGui** - содержит классы, реализующие низкоуровневую работу с оконными элементами, обработку сигналов, вывод двухмерной графики и текста и др.
- **QtWidgets** - содержит классы, реализующие компоненты графического интерфейса: окна, диалоговые окна, надписи, кнопки, текстовые поля и др.
- **QtWebKit** - включает низкоуровневые классы для отображения Web-страниц
- **QtWebKitWidgets** - реализует высокоуровневые компоненты графического интерфейса, предназначенные для вывода Web-страниц и использующие модуль QtWebKit
- **QtMultimedia** - включает низкоуровневые классы для работы с мультимедиа
- **QtMultimediaWidgets** - реализует высокоуровневые компоненты графического интерфейса с мультимедиа, использующие модуль QtMultimedia
- **QtSql** - включает поддержку работы с базами данных, а также реализацию SQLite
- **QtSvg** - позволяет работать с векторной графикой (SVG)
- **QtOpenGL** - обеспечивает поддержку OpenGL
- **QtNetwork** - содержит классы, предназначенные для работы с сетью
- **QtXml** и **QtXmlPatterns** - предназначены для обработки XML
- **QtHelp** - содержат инструменты для создания интерактивных справочных систем

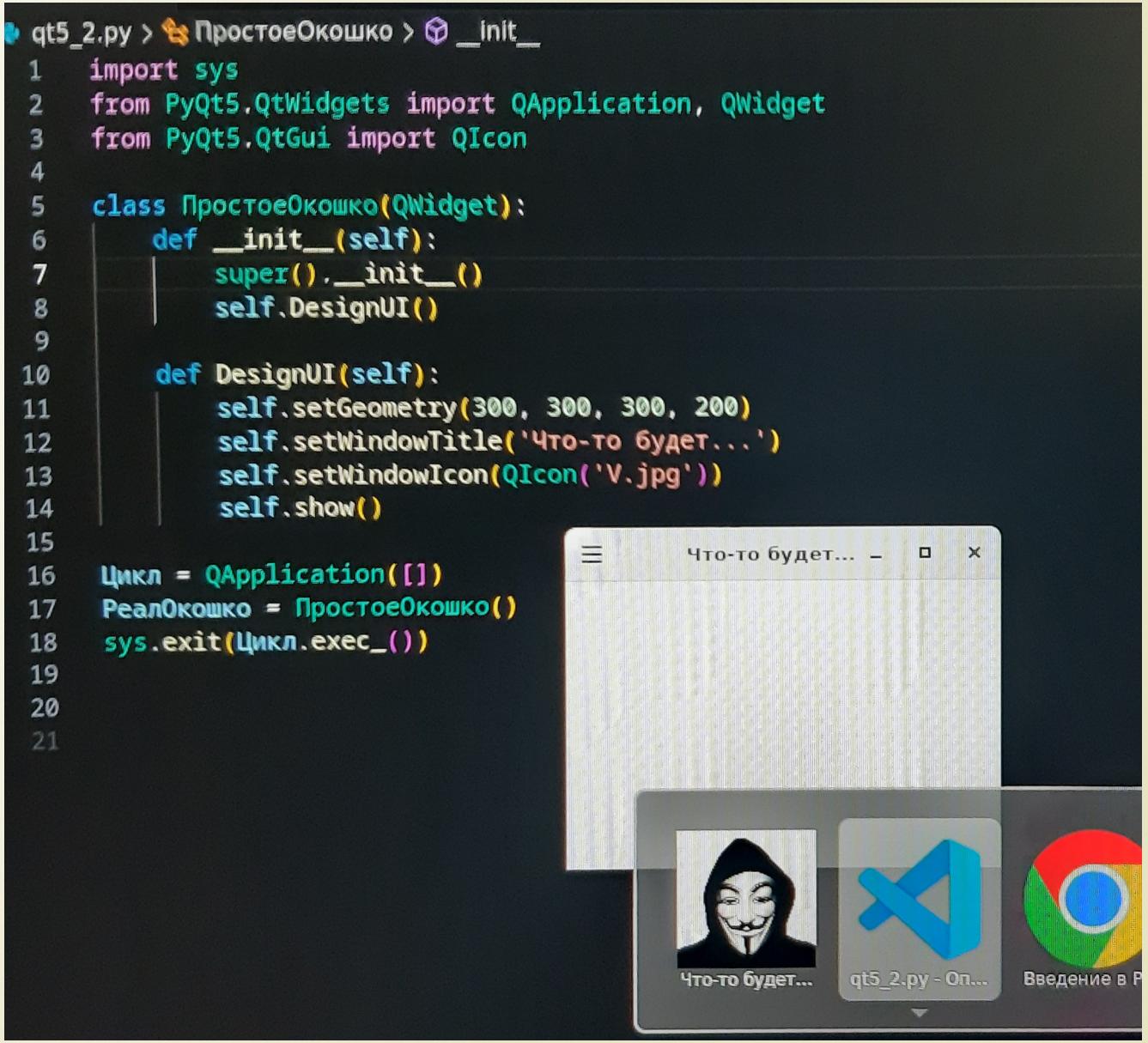
Окно с иконкой.

18

Методы, унаследованные от QWidget:

- **setGeometry** — определяет место окна на экране и устанавливает его размер. Вместо него можно использовать resize() и move().
- **setWindowTitle** — устанавливает заголовок окна.
- **setWindowIcon** — устанавливает вид изображения свёрнутого окна, картинка которого готовится методом QIcon.

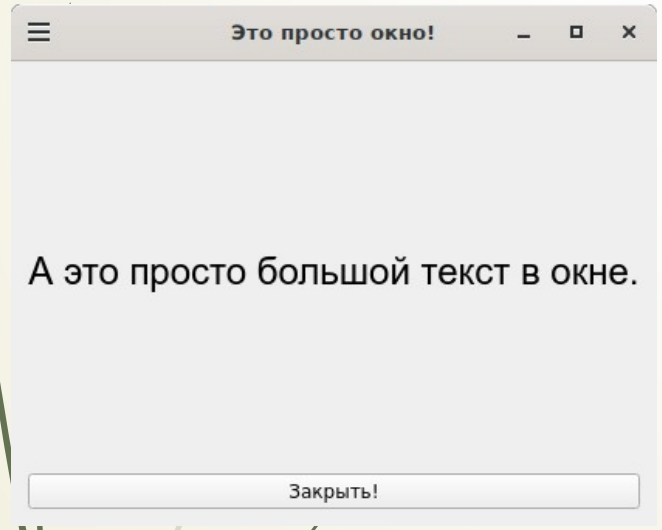
Если приложению не требуется доступ к командной строке, расширение файла для GUI-приложений — .pyw.
Библиотека **fbs** позволяет создавать автономные исполняемые файлы (.exe для Windows) для приложений PyQt.



Окно с кнопкой и текстом.

Окно — это виджет без родителя.

19




```python
import sys
from PyQt5 import QtWidgets
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QFont

class Окошечко (QtWidgets.QWidget):
    # конструктор
    def __init__(self, parent = None):
        super().__init__(parent)
        self.setWindowTitle("Это просто окно!")
        self.resize(400, 300) # ширина и высота окна в пикселях
        self.текстик = QtWidgets.QLabel("А это просто большой текст в окне.")
        self.текстик.setFont(QFont('Arial',18))
        # создаём визуальные элементы окна
        self.текстик.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.кнопочка = QtWidgets.QPushButton("Закрыть!")
        # помещаем элементы в контейнер окна
        self.лужайка = QtWidgets.QVBoxLayout()
        self.лужайка.addWidget(self.текстик)
        self.лужайка.addWidget(self.кнопочка)
        # подключаем контейнер к окну
        self.setLayout(self.лужайка)
        # определяем обработчик события нажатия на кнопку
        self.кнопочка.clicked.connect(QtWidgets.QApplication.instance().quit)
        self.show()

# основная программа
if __name__ == "__main__":
    приложение = QtWidgets.QApplication(sys.argv)
    реальное_окно = Окошечко() # создаём объект окна
    sys.exit(приложение.exec())
```

Если написать:

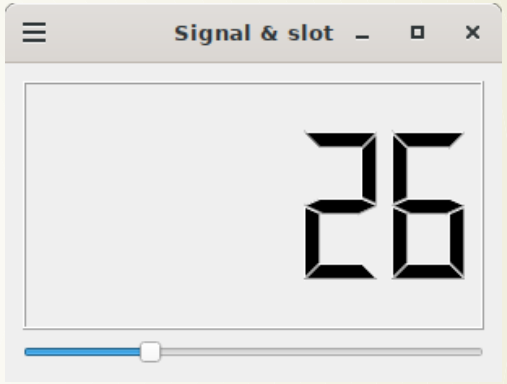
button.clicked.connect(click) — то имя click определяет метод:

```
def click(self):
    print("Кнопка нажата!")
```

Использование сигналов и слотов.

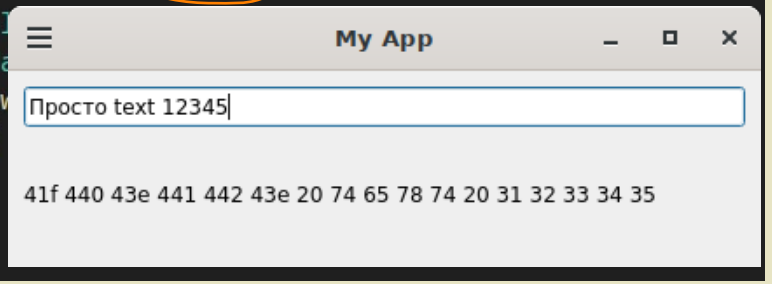
20

```
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.designUI()


    def designUI(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.setWindowTitle('Signal & slot')
        self.show()

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



- **clicked** — сигнал испускается при активизации кнопки
- **pressed** — генерируется при нажатии кнопки мыши
- **released** — генерируется при отпускании ранее нажатой кнопки мыши
- **textChanged** — в случае изменения текста (само изменение текста — это событие)
- **valueChanged** — в случае изменения значения виджета

```
from PyQt5.QtWidgets import (QApplication, QMainWindow,
    QLabel, QLineEdit, QVBoxLayout, QWidget)
import sys


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        self.label = QLabel()

        self.input = QLineEdit()
        self.input.textChanged.connect(self.toHex)

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

        container = QWidget()
        container.setLayout(layout)
        self.setCentralWidget(container)

    def toHex(self):
        A = self.input.text()
        S = ' '.join(f'{ord(s):x}' for s in A)
        self.label.setText(S)

app = QAppl
window = Ma
window.show
app.exec()
```

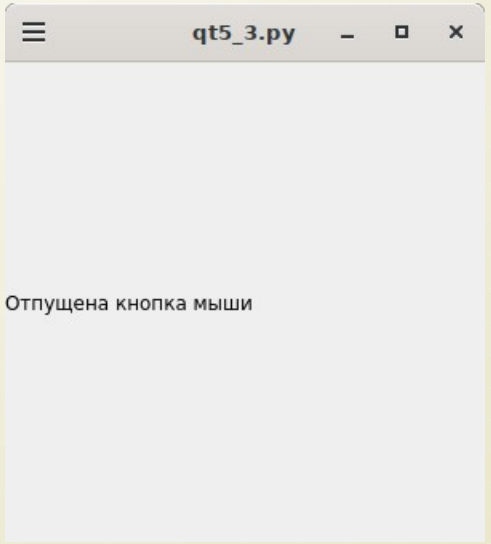


PyQt5. Пример реакции на события.

21 Обработка событий от мышки

В Qt события представлены объектами событий, в которые упакована информация о произошедшем. Обработчики событий определяются так же, как и любой другой метод, но название обработчика зависит от типа обрабатываемого события. Например, **QMouseEvent** — одно из основных событий, получаемых виджетами. События QMouseEvent создаются для каждого отдельного нажатия кнопки мыши и её перемещения в виджете.

Обработчик	*Событие*
mouseMoveEvent	Мышь переместилась
mousePressEvent	Кнопка мыши нажата
mouseReleaseEvent	Кнопка мыши отпущена
mouseDoubleClickEvent	Двойной клик мыши




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


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.myLabel = QLabel("Ждём событий от мыши")
        self.setCentralWidget(self.myLabel)
        self.resize(300, 300)


    def mouseMoveEvent(self, e):
        self.myLabel.setText("Мышь сдвинулась")

    def mousePressEvent(self, e):
        if e.button() == Qt.LeftButton:
            self.myLabel.setText("Нажата левая кнопка")

        elif e.button() == Qt.MiddleButton:
            self.myLabel.setText("Нажата средняя кнопка")

        elif e.button() == Qt.RightButton:
            self.myLabel.setText("Нажата правая кнопка")


    def mouseReleaseEvent(self, e):
        self.myLabel.setText("Отпущена кнопка мыши")

    def mouseDoubleClickEvent(self, e):
        self.myLabel.setText("Двойной клик мыши")


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

Qt.LeftButton — просто константа (=1)

События можно фильтровать, менять или игнорировать, передавая обычному обработчику путём вызова метода суперкласса.

PyQt5. Виджеты, тулбар, меню, строка статуса...

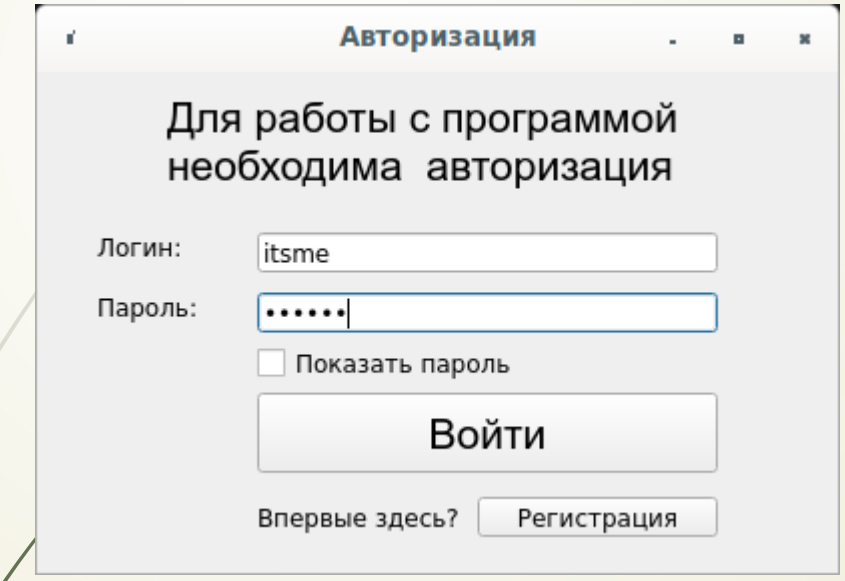

```python
import sys
from PyQt5.QtWidgets import ( QMainWindow,
    QApplication, QAction, QLabel, qApp)
from PyQt5.QtGui import QPixmap, QIcon

class HelloWorldWindow(QMainWindow):

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

    def designUI(self):
        self.setGeometry(100, 100, 350, 420)
        self.setWindowTitle('Окно в мир!')
        exitAction = QAction(QIcon('V.jpg'), 'Выход', self)
        exitAction.triggered.connect(qApp.quit)
        self.toolbar = self.addToolBar('Exit')
        self.toolbar.addAction(exitAction)
        text = QLabel(self)
        text.setText("И снова здравствуйте!")
        text.move(20, 55)
        text.resize(200, 20)

        worldLabel = QLabel(self)
        pixmap = QPixmap('earth.png')
        worldLabel.setPixmap(pixmap)
        worldLabel.move(20, 100)
        worldLabel.resize(300, 300)

        self.show()

app = QApplication([])
window = HelloWorldWindow()
sys.exit(app.exec ())
```

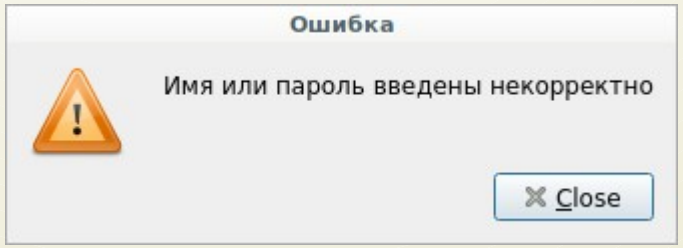
```python
# строка статуса - постоянно
ss = self.statusBar()
lb = QLabel('Привет!')
ss.insertPermanentWidget(0, lb)

# строка статуса - временно
self.statusBar().showMessage('Усё
готово, шеф!')

# меню
menubar = self.menuBar()
fileMenu =
menubar.addMenu('Файл')
fileMenu.addAction(exitAction)
```

PyQt5. Почти интересный проект. 1

23

Окно авторизации пользователя





Предлагается создать два класса:
LoginWindow — для основного окна авторизации
CreateNewUser — для окна регистрации новых пользователей

```
3   import sys
4   from PyQt5.QtWidgets import (QApplication, QWidget, QLabel,
5       QMessageBox, QLineEdit, QPushButton, QCheckBox)
6   from PyQt5.QtGui import QFont, QPixmap
7   from PyQt5.QtCore import Qt
8
9   # окно авторизации
10  class LoginWindow(QWidget):
11      def __init__(self):
12          super().__init__()
13          self.designUI()
14
15  >       def designUI(self):…
56
57  >       def clickLogin(self):…
81
82  >       def showPassword(self, state):…
88
89
90  >       def registerNewUser(self):…
94
95  # окно создания нового пользователя
96  class CreateNewUser(QWidget):
97      def __init__(self):
98          super().__init__()
99          self.designUI()
100
101 >       def designUI(self):…
140
141 >       def confirmSignUp(self):…
148
149 if __name__ == '__main__':
150     app = QApplication(sys.argv)
151     logWindow = LoginWindow()
152     sys.exit(app.exec_())
```

PyQt5. Почти интересный проект. 2

24 Используется дизайн с точным позиционированием отдельных виджетов

```
15      def designUI(self):
16          self.setGeometry(100, 100, 400, 250)
17          self.setWindowTitle('Авторизация')
18
19          # поле описания окна
20          loginLabel = QLabel(self)
21          loginLabel.setText(
22              "Для работы с программой\nнеобходима  авторизация")
23          loginLabel.move(65, 10)
24          loginLabel.setFont(QFont('Arial', 16))
25
26          # поле для ввода логина
27          nameLabel = QLabel("Логин:", self)
28          nameLabel.move(30, 80)
29          self.nameEntry = QLineEdit(self)
30          self.nameEntry.move(110, 80)
31          self.nameEntry.resize(230, 20)
32
33          # поле для ввода пароля
34          passwordLabel = QLabel("Пароль:", self)
35          passwordLabel.move(30, 110)
36          self.passwordEntry = QLineEdit(self)
37          self.passwordEntry.move(110, 110)
38          self.passwordEntry.resize(230, 20)
39          self.passwordEntry.setEchoMode(QLineEdit.Password)
40
```

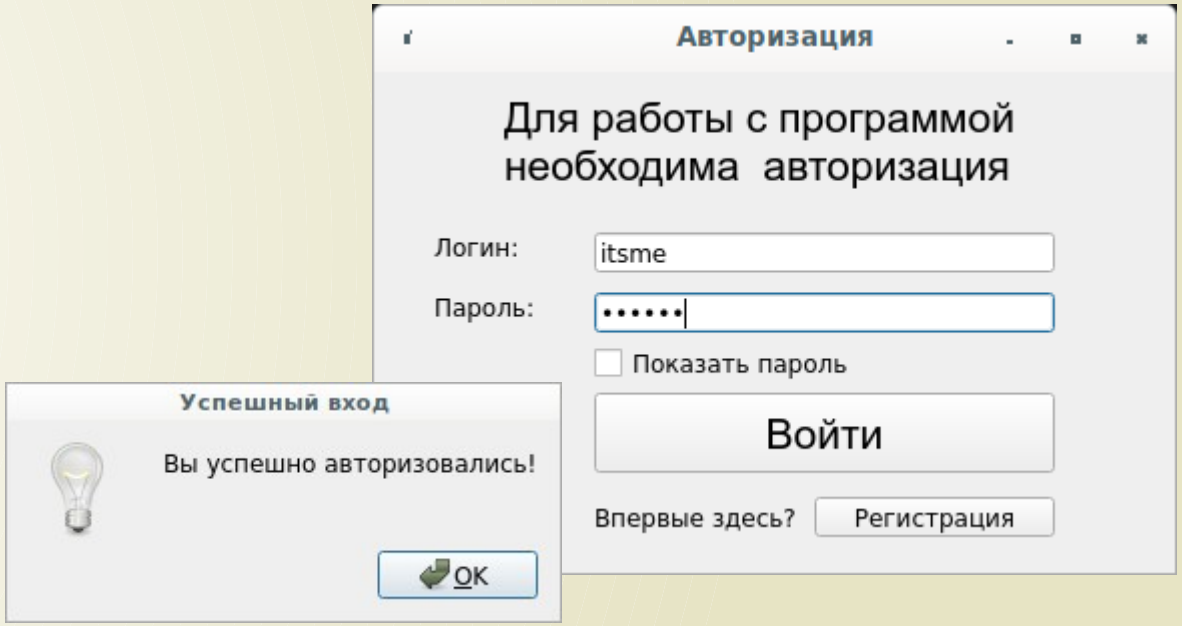
делаем ввод пароля защищённым

```
41          # кнопка входа
42          enterButton = QPushButton('Войти', self)
43          enterButton.move(110, 160)
44          enterButton.resize(230, 40)
45          enterButton.setFont(QFont('Arial', 16))
46          enterButton.clicked.connect(self.clickLogin)
47
48          # чекбокс "показать пароль"
49          pswdCB = QCheckBox("Показать пароль", self)
50          pswdCB.move(110, 135)
51          pswdCB.stateChanged.connect(self.showPassword)
52          pswdCB.setChecked(False)
53
54          # регистрация нового пользователя
55          notAmember = QLabel("Впервые здесь?", self)
56          notAmember.move(110, 215)
57          signUp = QPushButton("Регистрация", self)
58          signUp.move(220, 212)
59          signUp.resize(120, 20)
60          signUp.clicked.connect(self.registerNewUser)
61          self.show()
62
```

PyQt5. Почти интересный проект. 3

25 добавляем функциональность

```
62
63      def clickLogin(self):
64          # проверяем логин и пароль
65          # в соответствии со списком пользователей
66          users = {}
67          try:
68              with open("users.txt", 'r') as f:
69                  for line in f:
70                      fields = line.split(" ")
71                      username = fields[0]
72                      password = fields[1].strip('\n')
73                      users[username] = password
74          except FileNotFoundError:
75              print('А файла-то с паролями нет! Создаём новый.')
76              f = open ("users.txt", "w")
77          username = self.nameEntry.text()
78          password = self.passwordEntry.text()
79          if (username, password) in users.items():
80              QMessageBox.information(self,
81              "Успешный вход", "Вы успешно авторизовались!",
82              QMessageBox.Ok, QMessageBox.Ok)
83          else:
84              QMessageBox.warning(self,
85              "Ошибка", "Имя или пароль введены некорректно",
86              QMessageBox.Close, QMessageBox.Close)
```

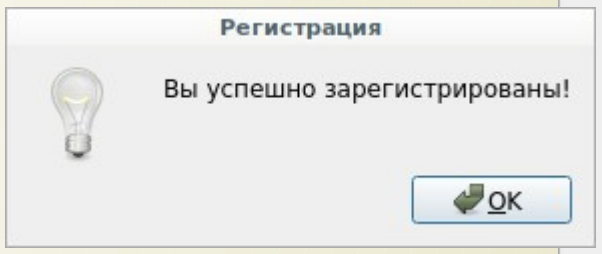


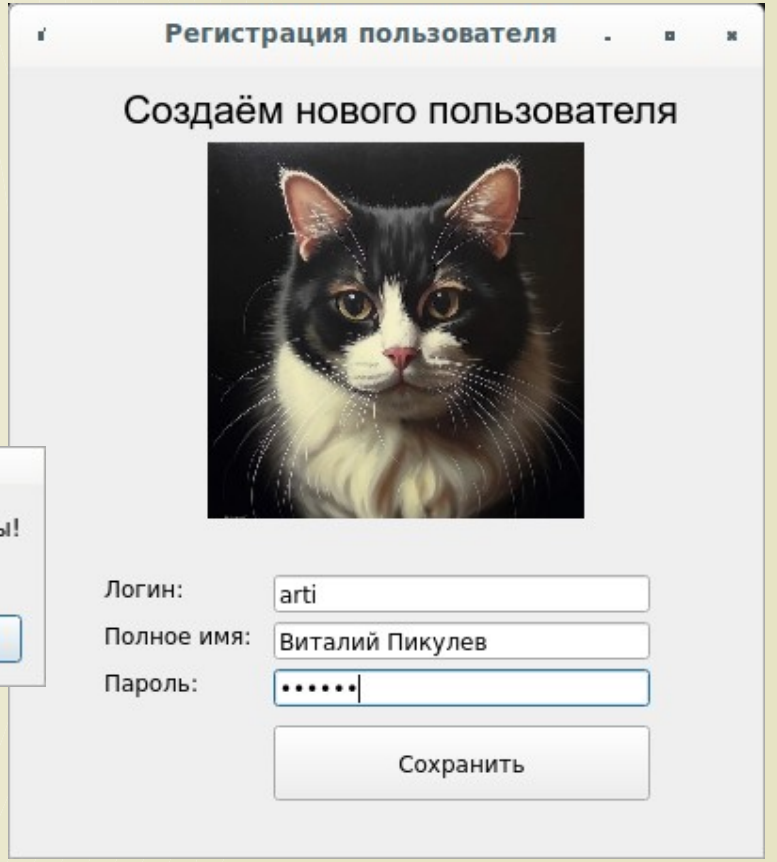

```
88      def showPassword(self, state):
89          # Маскировка и демаскировка символов пароля
90          if state == Qt.Checked:
91              self.passwordEntry.setEchoMode(QLineEdit.Normal)
92          else:
93              self.passwordEntry.setEchoMode(QLineEdit.Password)
94
95
96      def registerNewUser(self):
97          # создаём нового пользователя
98          self.newUserDialog = CreateNewUser()
99          self.newUserDialog.show()
```

PyQt5. Почти интересный проект. 4

```
107     def designUI(self):
108         self.setGeometry(100, 100, 400, 420)
109         self.setWindowTitle('Регистрация пользователя')
110         # добавляем нужные виджеты
111         userImage = "test_image.jpg"
112         newUser = QLabel(self)
113         pixmap = QPixmap(userImage)
114         newUser.setPixmap(pixmap.scaled(200, 200))
115         newUser.move(105, 40)
116         loginLabel = QLabel(self)
117         loginLabel.setText("Создаём нового пользователя")
118         loginLabel.move(60, 10)
119         loginLabel.setFont(QFont('Arial', 16))
120 
121         loginLabel = QLabel("Логин:", self)
122         loginLabel.move(50, 270)
123         self.nameEntry = QLineEdit(self)
124         self.nameEntry.move(140, 270)
125         self.nameEntry.resize(200, 20)
126 
127         nameLabel = QLabel("Полное имя:", self)
128         nameLabel.move(50, 295)
129         nameEntry = QLineEdit(self)
130         nameEntry.move(140, 295)
131         nameEntry.resize(200, 20)
132 
133         pswdLabel = QLabel("Пароль:", self)
134         pswdLabel.move(50, 320)
135         self.pswdEntry = QLineEdit(self)
136         self.pswdEntry.setEchoMode(QLineEdit.Password)
137         self.pswdEntry.move(140, 320)
138         self.pswdEntry.resize(200, 20)
```

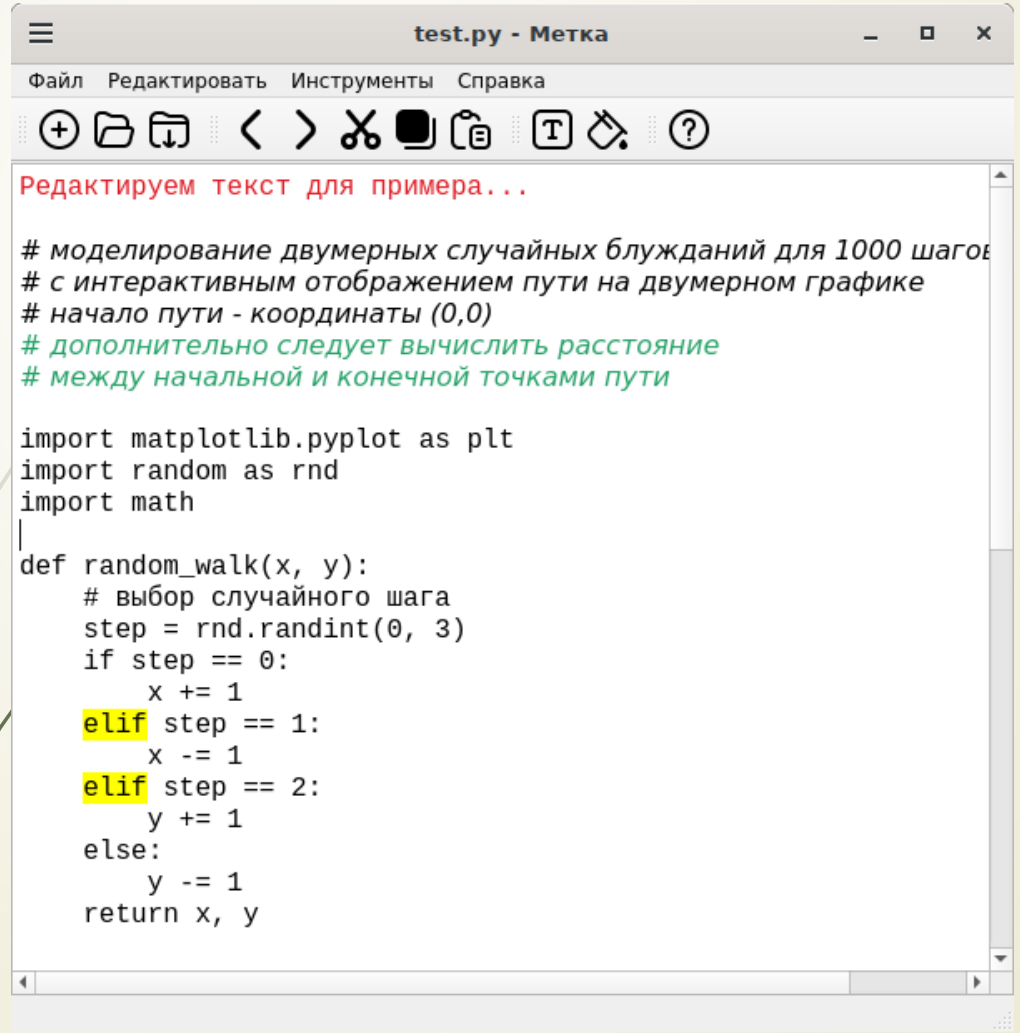
дизайн и функционал
окна регистрации
новых пользователей






```
140         enterButton = QPushButton("Сохранить", self)
141         enterButton.move(140, 350)
142         enterButton.resize(200, 40)
143         enterButton.clicked.connect(self.confirmSignUp)
144 
145         self.show()
146 
147     def confirmSignUp(self):
148         with open("users.txt", 'a+') as f:
149             f.write(self.nameEntry.text() + " " + self.pswdEntry.text() + "\n")
150         QMessageBox.information(self,
151             "Регистрация", "Вы успешно зарегистрированы!",
152             QMessageBox.Ok, QMessageBox.Ok)
153         self.close() # закрываем диалог
```

Полезный проект: редактор текста. (1)




```python
# простой текстовый редактор на Python
import sys, os
from PyQt5.QtWidgets import  (QApplication,  QMainWindow,
    QAction, QTextEdit, QMessageBox, QFontDialog, QColorDialog,
    QInputDialog, QFileDialog)
from PyQt5.QtGui import QIcon, QPixmap, QTextCursor, QColor, QFont
from PyQt5.QtCore import Qt

class EasyPad(QMainWindow):

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

    def designUI(self):···

    def clearText(self):···

    def openFile(self):···


    def saveFile(self):···

    def fontDialog(self):···

    def colorDialog(self):···

    def findTextDialog(self):···

    def aboutDialog(self):···

if __name__ == "__main__":
    wpath = os.path.dirname(os.path.realpath(__file__))
    app = QApplication(sys.argv)
    window = EasyPad()
    sys.exit(app.exec_())
```

Фактически мы просто добавляем интерфейс управления к функционально богатому методу **QTextEdit()**, который позволяет осуществлять просмотр и редактирование как простого текста, так и текста в формате HTML.

Полезный проект: редактор текста. (2)

В приложениях многие одинаковые команды можно вызывать с помощью меню (в том числе - контекстного), кнопок панели инструментов, сочетаний клавиш и др. Поэтому полезно представлять каждую команду как действие. Метод **QAction** является абстракцией для оформления таких действий. Действие должно быть добавлено в виджет (**QWidget.addAction()**) до того, как его можно будет использовать.

Основные методы класса **QTextEdit**, многие из которых являются слотами:

- ✔ **setText** (<Текст>) - помещает указанный текст в поле. Текст может быть простым или в формате HTML.
- ✔ **setPlainText** (<Текст>) - помещает в поле простой текст.
- ✔ **setHtml** (<Текст>) - помещает в поле текст в формате HTML.
- ✔ **toHtml** () - возвращает текст в формате HTML.
- ✔ **clear** () - удаляет весь текст из поля.
- ✔ **selectAll** () - выделяет весь текст в поле.
- ✔ **zoomIn** ([range=1]) - увеличивает размер шрифта.
- ✔ **zoomOut** ([range=1]) - уменьшает размер шрифта.
- ✔ **cut** () - копирует выделенный текст в буфер обмена и удаляет его из поля при условии, что есть выделенный фрагмент.
- ✔ **copy** () - копирует выделенный текст в буфер обмена при условии, что есть выделенный фрагмент.
- ✔ **paste** () - вставляет текст из буфера обмена в текущую позицию текстового курсора при условии, что поле доступно для редактирования.
- ✔ **undo** () - отменяет последнюю операцию ввода пользователем при условии, что отмена возможна.
- ✔ **redo** () - повторяет последнюю отмененную операцию ввода пользователем, если это возможно.
- ✔ **find** () - производит поиск фрагмента (по умолчанию в прямом направлений без учета регистра символов) в текстовом поле. Если фрагмент найден, то он выделяется, и метод возвращает значение True, в противном случае - значение False.

Полезный проект: редактор текста. (3)


```
def designUI(self):
    self.name = 'Метка'
    self.setGeometry(100, 100, 450, 550)
    self.setWindowTitle(self.name)
    #
    self.textField = QTextEdit(self)
    self.textField.setLineWrapMode(QTextEdit.NoWrap)
    self.textField.setFont(QFont('Liberation Mono', 12))
    self.setCentralWidget(self.textField)
    #
    newAct = QAction(QIcon(wpath + '/images/add.png'), 'Новый', self)
    newAct.setShortcut('Ctrl+N')
    newAct.triggered.connect(self.clearText)
    #
    openAct = QAction(QIcon(wpath + '/images/folder_open.png'), 'Открыть', self)
    openAct.setShortcut('Ctrl+O')
    openAct.triggered.connect(self.openFile)
    #
    saveAct = QAction(QIcon(wpath + '/images/folder_download.png'), 'Сохранить', self)
    saveAct.setShortcut('Ctrl+S')
    saveAct.triggered.connect(self.saveFile)
    #
    exitAct = QAction(QIcon(wpath + '/images/exit.png'), 'Выход', self)
    exitAct.setShortcut('Ctrl+Q')
    exitAct.triggered.connect(self.close)
```

Полезный проект: редактор текста. (4)

30

продолжение метода designUI

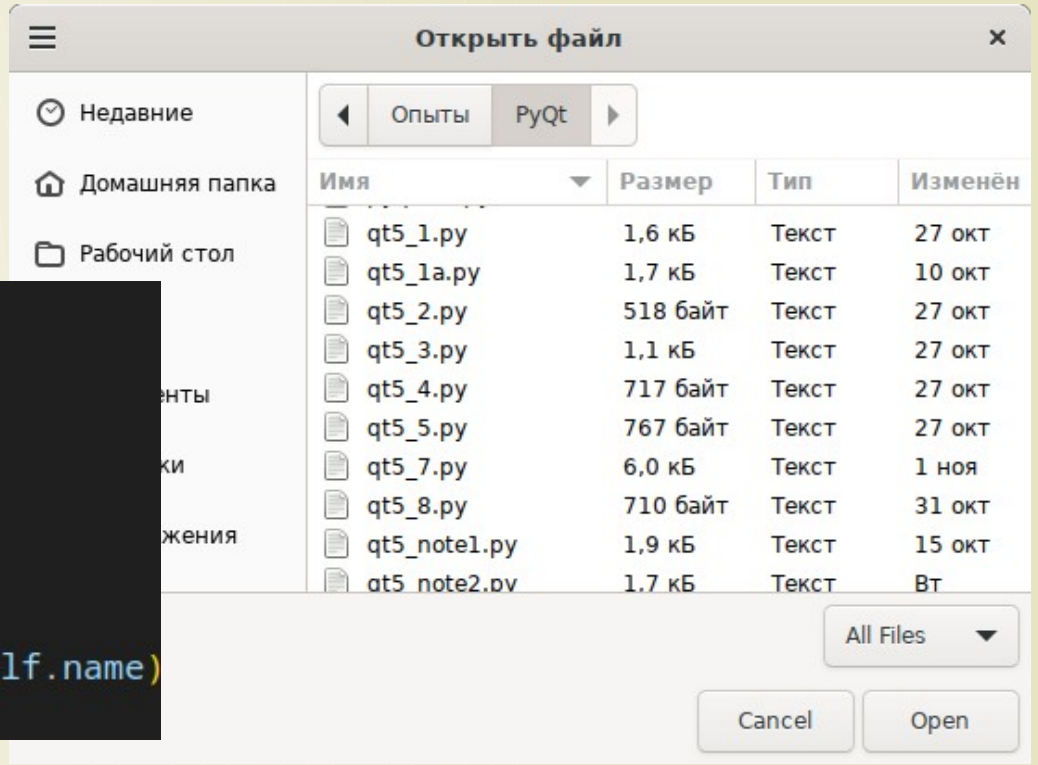
```
# работа с файлами: меню и панель ин
menuBar = self.menuBar()
fileMenu = menuBar.addMenu('Файл')
fileMenu.addAction(newAct)
fileMenu.addSeparator()
fileMenu.addAction(openAct)
fileMenu.addAction(saveAct)
fileMenu.addSeparator()
fileMenu.addAction(exitAct)
#
self.toolbar = self.addToolBar('Файл')
self.toolbar.addAction(newAct)
self.toolbar.addAction(openAct)
self.toolbar.addAction(saveAct)

# здесь пропущены все остальные пункты меню

# о программе: меню и панель инструментов
helpMenu = menuBar.addMenu('Справка')
helpMenu.addAction(aboutAct)
self.toolbar = self.addToolBar('Справка')
self.toolbar.addAction(aboutAct)

self.statusBar()
self.show()
```

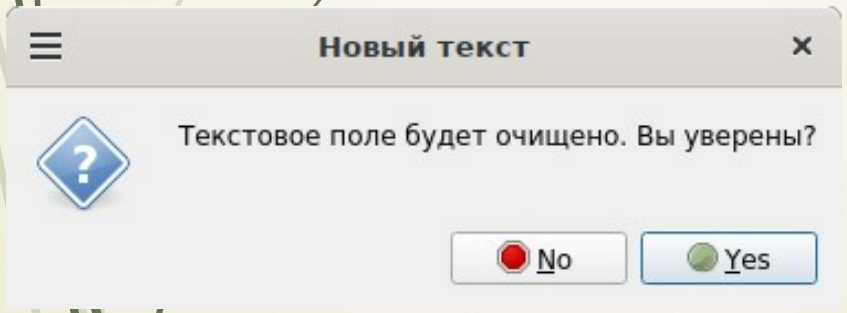
```
def openFile(self):
    fileName, _ = QFileDialog.getOpenFileName(self, "Открыть файл",
        "", "HTML Files (*.html);;Text Files (*.txt);;All Files (*)")
    if fileName:
        with open(fileName, 'r') as f:
            text = f.read()
            self.textField.setText(text)
            self.setWindowTitle(fileName.split("/")[-1] + " - " + self.name)
    else:
        QMessageBox.information(self, "Ошибка",
        "Файл не выбран!", QMessageBox.Ok)
```

QFileDialog - это предопределённое (полностью функциональное) модальное диалоговое окно, которое позволяет пользователям выбирать файлы или папки. Файлы могут быть указаны как для сохранения, так и для открытия. Параметры: *self, имя окна, начальный каталог, фильтр*. В фильтре (в случае выбора варианта) варианты отделяются двойной точкой с запятой.

Полезный проект: редактор текста. (5)


```
def saveFile(self):
    goodText = self.textField.toPlainText()
    options = QFileDialog.Options()
    fileName, _ = QFileDialog.getSaveFileName(self, 'Сохранить',
        "","All Files (*);;Text Files (*.txt)", options=options)
    if fileName:
        with open(fileName, 'w') as f:
            f.write(goodText)
            self.setWindowTitle(fileName.split("/")[-1] + " - " + self.name)
```



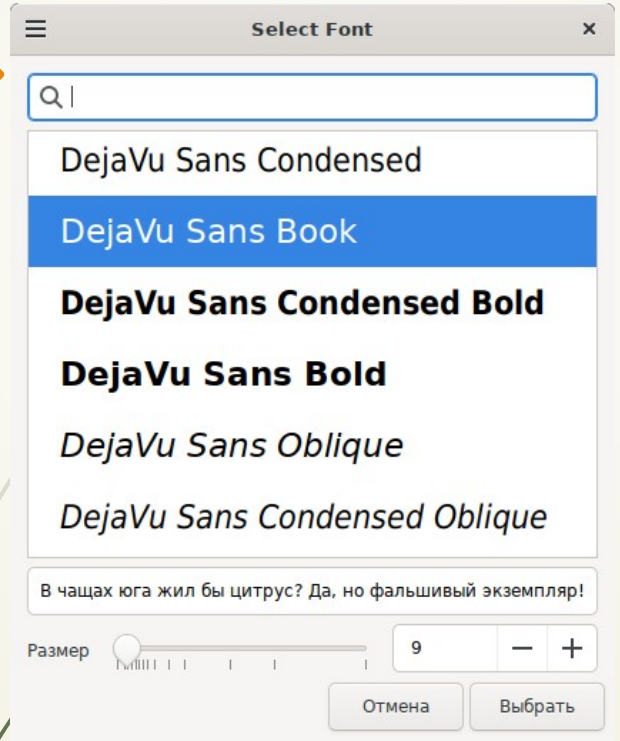



```
def clearText(self):
    answer = QMessageBox.question(self, "Новый текст",
    "Текстовое поле будет очищено. Вы уверены?",
    QMessageBox.No | QMessageBox.Yes, QMessageBox.Yes)
    if answer == QMessageBox.Yes:
        self.textField.clear()
```

нет метода - нет значка
Question - обслуживание вопроса
Information - обслуживание информационного сообщения
Warning - обслуживание предупреждающего сообщения
Critical - обслуживание критического сообщения

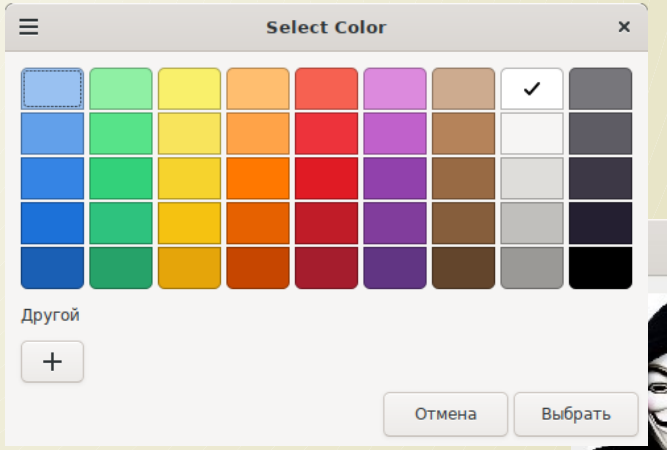
Полезный проект: редактор текста. (6)

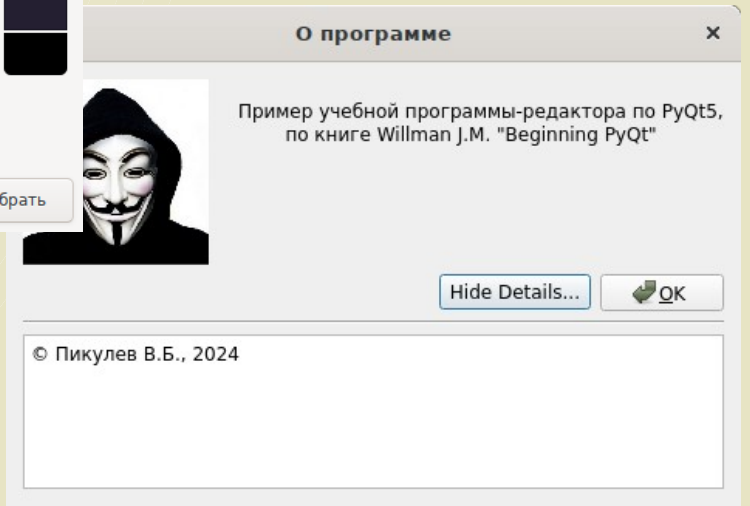
Ещё больше модальных диалоговых окон




```
def aboutDialog(self):
    msg = QMessageBox()
    pix = QPixmap(wpath + '/images/V.jpg').scaledToWidth(120)
    msg.setIconPixmap(pix)
    msg.setWindowTitle("О программе")
    msg.setText('''Пример учебной программы-редактора по PyQt5,
        по книге Willman J.M. "Beginning PyQt"''')
    msg.setDetailedText("© Пикулев В.Б., 2024")
    msg.setStandardButtons(QMessageBox.Ok)
    msg.exec_()
```

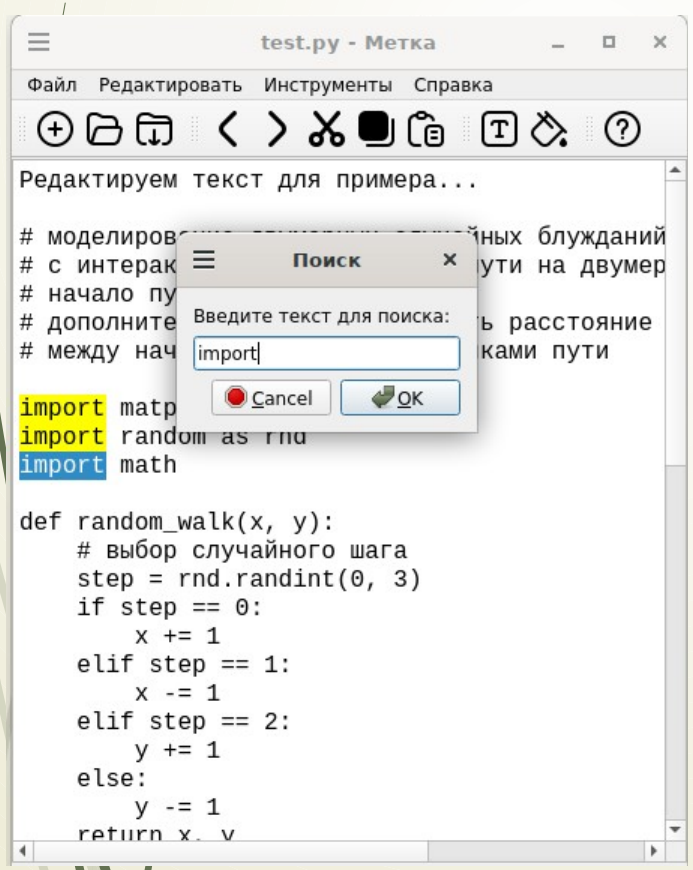
```
def fontDialog(self):
    font, ok = QFontDialog.getFont()
    if ok:
        self.textField.setCurrentFont(font)

def colorDialog(self):
    color = QColorDialog.getColor()
    if color.isValid():
        self.textField.setTextColor(color)
```





Полезный проект: редактор текста. (7)


```
def findTextDialog(self):
# поиск в тексте
    self.findText, ok = QInputDialog.getText(self, "Поиск",
    "Введите текст для поиска:", text = self.findText)
    if ok:
        self.textField.find(self.findText)
```



QTextEdit.find() — производит поиск фрагмента (по умолчанию в прямом направлении без учета регистра символов) в текстовом поле. Если фрагмент найден, он выделяется, и метод возвращает значение *True*, в противном случае — значение *False*. **QInputDialog** - простой удобный диалог для получения единственного значения от пользователя. Введённое значение может быть строкой, числом или пунктом списка.

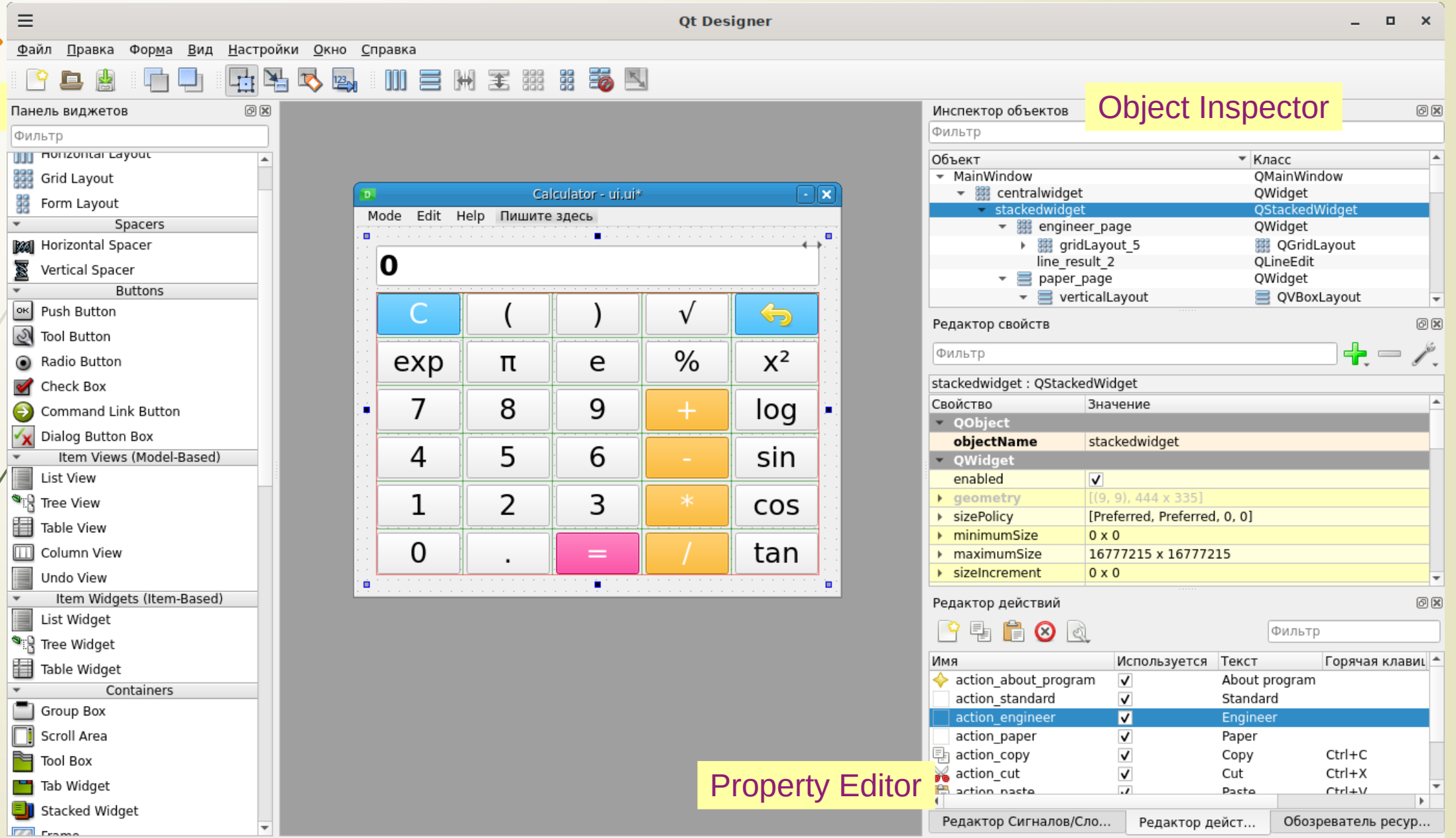
```
def findTextDialog(self):
    color = QColor(Qt.yellow)
    selected = []
    text, ok = QInputDialog.getText(self, "Поиск", "Введите текст для поиска:")
    if ok:
        self.textField.moveCursor(QTextCursor.Start)
        while self.textField.find(text):
            selection = QTextEdit.ExtraSelection()
            selection.format.setBackground(color)
            selection.cursor = self.textField.textCursor()
            selected.append(selection)
        for i in selected:
            self.textField.setExtraSelections(selected)
```

QTextEdit.ExtraSelection() - даёт возможность управлять форматом символов, к которым применяется выделение.

Использование Qt Designer. (1)

создаются файлы с расширением .ui

Widget Box

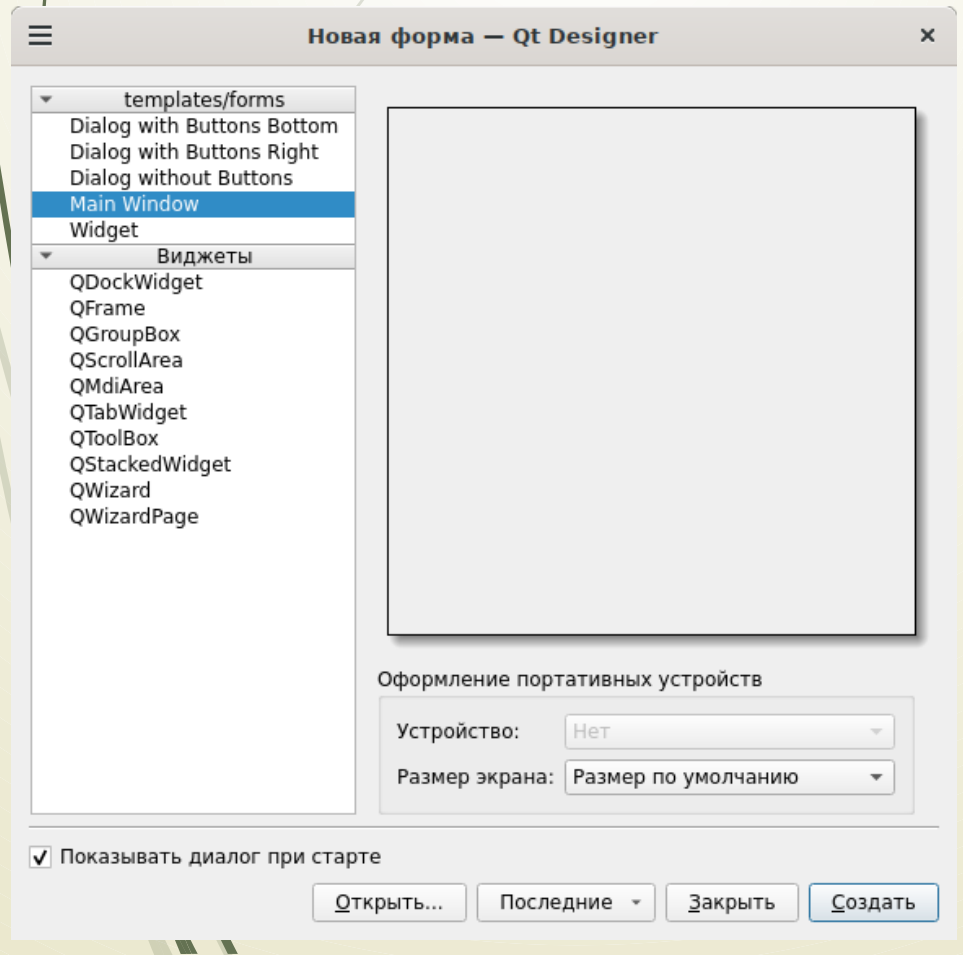


Object Inspector

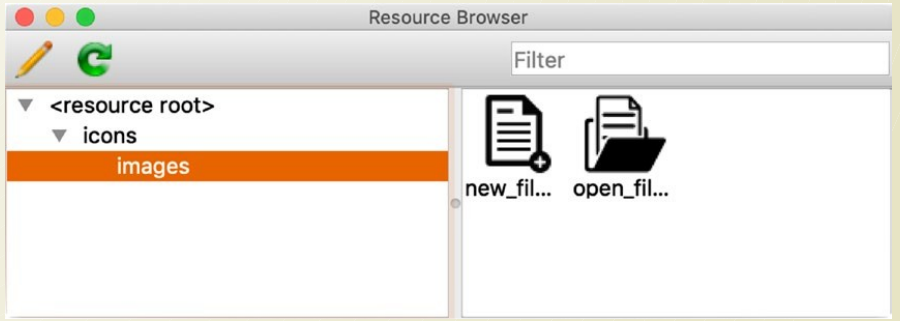
Property Editor

Использование Qt Designer. (2)

Виджеты и окна, которые можно создать с помощью Qt Designer, интегрируются с программным кодом, используя тот же самый механизм сигналов и слотов Qt. Однако при этом не тратится время на расчёт позиций виджетов, всё происходит интерактивно - методом перетаскивания виджетов из панели на создаваемое окно. Это означает, что экономится время и силы на стадии макетирования и дизайна, и можно быстрее приступить к backend-кодированию.



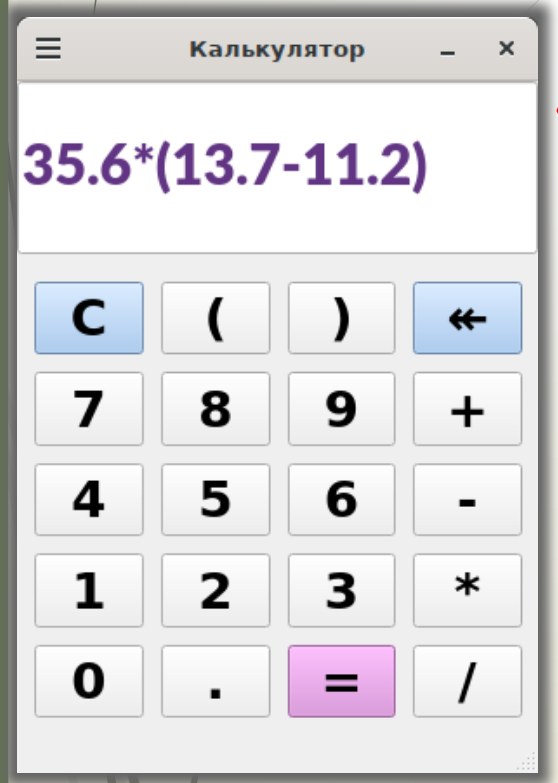
В Qt Designer также можно создавать, редактировать и удалять сигналы и слоты между объектами с помощью редактора сигналов/слотов. Однако несмотря на удобство интерфейса, далеко не всегда можно полностью настроить виджеты через Designer, поэтому окончательная настройка потребуется из Python-кода.

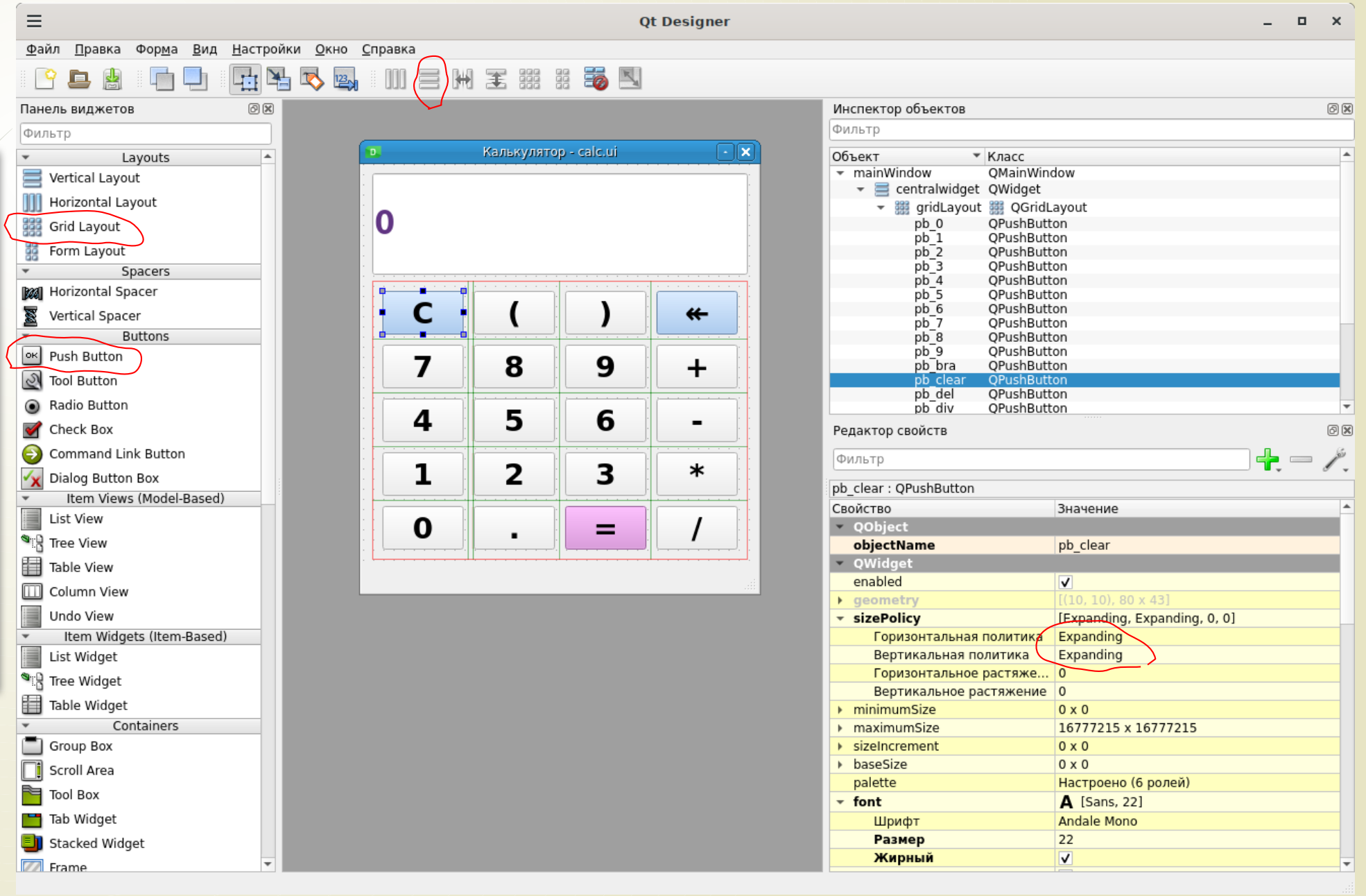


использование файла ресурсов

Существуют два пути для связи форм, приготовленных в Qt Designer, с программным кодом:
1) pyuic5 path/to/design.ui -o output/path/to/design.py
2) непосредственно подключить ui-файл к python-коду.

Проект с Qt Designer: калькулятор. (1)





Проект с Qt Designer: калькулятор. (2)

Внутри *ui*-файла содержится текст в XML-формате, а не программный код на языке Python. Следовательно, подключить файл с помощью инструкции import не получится. Чтобы использовать ui-файл внутри программы, следует воспользоваться модулем **uic**, который входит в состав библиотеки PyQt. Для загрузки *ui*-файла предназначен метод **loadUi()**.

Калькулятор предполагается сделать для чисел с фиксированной точкой, но при этом обработать (по-возможности) все ситуации, которые могут привести к счётной ошибке.

```python
# простой калькулятор на PyQt5
from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5.QtGui import QIcon
from PyQt5 import uic
import sys, os

class MyCalculator(QMainWindow):
    def __init__(self):
        super().__init__()
        uic.loadUi(wpath + '/calc.ui', self)
        self.designUI()
        self.show()

    def designUI(self):...

    def doIt(self, text):...

    def delIt(self):...

    def typeResult(self):...


if __name__ == "__main__":
    wpath = os.path.dirname(os.path.realpath(__file__))
    maxdigits = 15
    app = QApplication([])
    window = MyCalculator()
    sys.exit(app.exec_())
```

Проект с Qt Designer: калькулятор. (3)

Имена, которые вы дали виджетам (или забыли дать, и они установились по умолчанию), будут доступны методам класса через self.

Для связи сигналов и слотов с возможностью обрабатывать запрос от разных слотов одним методом (но с разными параметрами) можно в методе **connect()** представить имя метода-обработчика + параметр в виде лямбда-функции.

```
def designUI(self):
    self.setFixedSize(300, 400)
    self.setWindowIcon(
        QIcon(wpath + '/images/calc.png'))
    #
    self.pb_0.clicked.connect(lambda: self.doIt('0'))
    self.pb_1.clicked.connect(lambda: self.doIt('1'))
    self.pb_2.clicked.connect(lambda: self.doIt('2'))
    self.pb_3.clicked.connect(lambda: self.doIt('3'))
    self.pb_4.clicked.connect(lambda: self.doIt('4'))
    self.pb_5.clicked.connect(lambda: self.doIt('5'))
    self.pb_6.clicked.connect(lambda: self.doIt('6'))
    self.pb_7.clicked.connect(lambda: self.doIt('7'))
    self.pb_8.clicked.connect(lambda: self.doIt('8'))
    self.pb_9.clicked.connect(lambda: self.doIt('9'))
    self.pb_plus.clicked.connect(lambda: self.doIt('+'))
    self.pb_minus.clicked.connect(lambda: self.doIt('-'))
    self.pb_mul.clicked.connect(lambda: self.doIt('*'))
    self.pb_div.clicked.connect(lambda: self.doIt('/'))
    self.pb_point.clicked.connect(lambda: self.doIt('.'))
    self.pb_bra.clicked.connect(lambda: self.doIt('('))
    self.pb_ket.clicked.connect(lambda: self.doIt(')'))
    self.pb_equal.clicked.connect(self.typeResult)
    self.pb_del.clicked.connect(self.delIt)
    self.pb_clear.clicked.connect(
        lambda: self.lineEdit.setText('0'))
```

Проект с Qt Designer: калькулятор. (4)

maxdigits = 15 — число знаков, влезающих на дисплей калькулятора

```
39      def doIt(self, text):
40          # для кнопок 0 1 2 3 4 5 6 7 8 9 . ( ) + - * /
41          if self.lineEdit.text() in ['Error', '0']:
42              if text == '.':
43                  self.lineEdit.setText('0.')
44              else:
45                  if text not in ['*', '/', ')']:
46                      self.lineEdit.setText(text)
47          else:
48              self.lineEdit.setText(self.lineEdit.text() + text)
49
50      def delIt(self):
51          # для кнопки DEL
52          if len(self.lineEdit.text()) > 1:
53              self.lineEdit.setText(self.lineEdit.text()[:-1])
54          else:
55              self.lineEdit.setText('0')
```

Главной рабочей строкой является, безусловно: **self.lineEdit.setText(self.lineEdit.text() + text)** - где text передаёт символ, который добавляется в строку. Усложнение кода вызвано эстетическим моментом: на всех калькуляторах горит **0** перед началом выполнения действий, здесь он лишний, но поскольку он используется, то строки надо корректировать.

Проект с Qt Designer: калькулятор. (5)

Встроенная функция **eval()** используется для динамического исполнения выражений, записанных в текстовую строку. Функция анализирует строку, компилирует в байт-код и выполняет как выражение Python. Функция осуществляет только парсинг выражений, но не понимает составных конструкций.

Усложнения кода вызваны попыткой проанализировать ошибочные ситуации, например, когда общее количество знаков в числе (до и после запятой) превышает количество символов, которые можно вывести в информационное поле.

```
56
57      def typeResult(self):
58          # для кнопки =
59          try:
60              res = eval(self.lineEdit.text())
61              if res == int(res):
62                  if len(str(res)) > maxdigits:
63                      self.lineEdit.setText('Error')
64                  else:
65                      self.lineEdit.setText(str(res))
66              else:
67                  num = f'{res:40.20f}'.split('.')
68                  num[0] = num[0].lstrip(' ')
69                  num[1] = num[1].rstrip('0')
70                  print(num)
71                  if len(num[0]) > maxdigits:
72                      self.lineEdit.setText('Error')
73                  else:
74                      if len(num[1]) > maxdigits - len(num[0]):
75                          num[1] = num[1][:maxdigits-len(num[0])-1]
76                  self.lineEdit.setText(str(num[0] + '.' + num[1]))
77          except Exception as e:
78              self.lineEdit.setText('Error')
```

«Я придумал термин „объектно-ориентированный“, но я вовсе не имел в виду C++»

Alan Kay. создатель языка Smalltalk

Конец блока 2.

Питон в условиях карельской зимы. Шедеврум, 2024.