Создание настольных Python приложений

с графическим интерфейсом пользователя

Тимур Машнин

12+

Тимур Машнин

Создание настольных Python приложений с графическим интерфейсом пользователя

«ЛитРес: Самиздат»

2021

Машнин Т.

Создание настольных Python приложений с графическим интерфейсом пользователя / Т. Машнин — «ЛитРес: Самиздат», 2021

ISBN 978-5-532-96908-7

Python является наиболее популярным языком программирования, используемым для объектно-ориентированного программирования. И конечно, Python — это интерактивный язык программирования, который предоставляет широкий спектр возможностей для создания GUI (Graphical User Interface) и разработки настольных приложений. В этой книге вы познакомитесь с различными фреймворками Python для создания настольных приложений с графическим интерфейсом пользователя, такими как PyQt, PySide, Tkinter, Kivy, WxPython и Dear PyGUI.

ISBN 978-5-532-96908-7

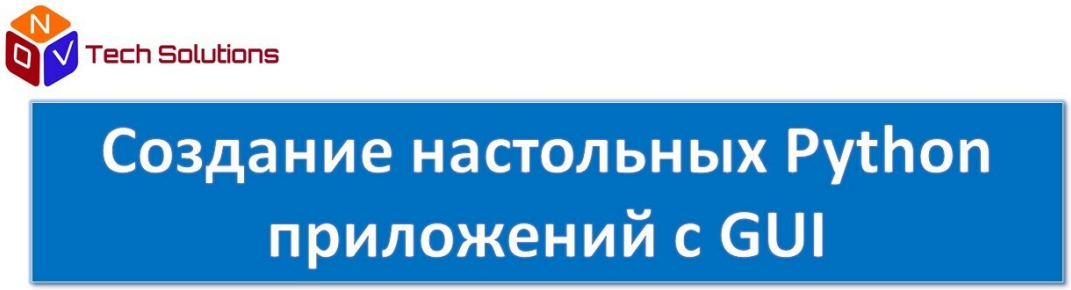
Содержание

Тимур Машнин
Создание настольных Python приложений с графическим интерфейсом пользователя

Исходный код

Исходный код к примерам можно скачать по адресу https://github.com/novts/python-gui.

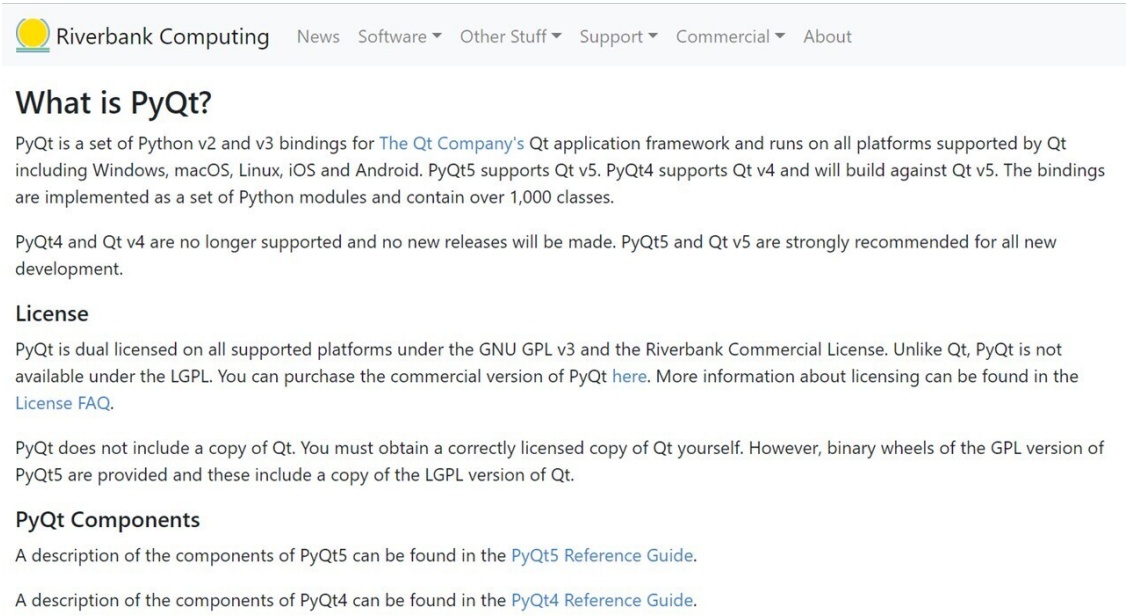
Введение



Введение

Де факто Python является наиболее популярным объектно-ориентированным языком программирования, который используется для веб-разработки и анализа больших данных.

И конечно, Python – это интерактивный язык программирования, который предоставляет широкий спектр возможностей для создания графического интерфейса пользователя.



PyQt – это библиотека графического фреймворка Qt для языка программирования Python.

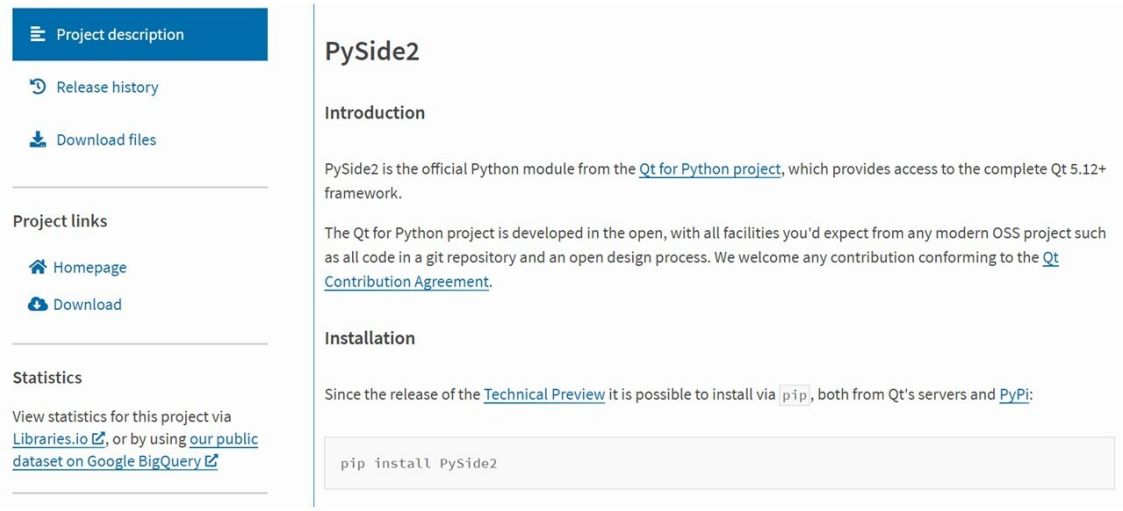
А Qt кью-ти – это кроссплатформенный инструментарий для разработки программного обеспечения на языке программирования C++, такого как графические интерфейсы, работа с сетью, базами данных и XML.

PyQt работает на всех платформах, поддерживаемых Qt – Linux и другие UNIX-подобные ОС, Mac OS и Windows.

И существуют 2 версии: PyQt5, поддерживающий Qt 5, и PyQt4, поддерживающий Qt 4.

PyQt практически полностью реализует возможности Qt, включая набор виджетов графического интерфейса, доступ к базам данных с помощью SQL, парсер XML и так далее.

PyQt также включает в себя Qt Designer— дизайнер графического интерфейса пользователя с генерацией Python кода из файлов, созданных в Qt Designer.



PySide – это также библиотека графического фреймворка Qt для языка программирования Python.

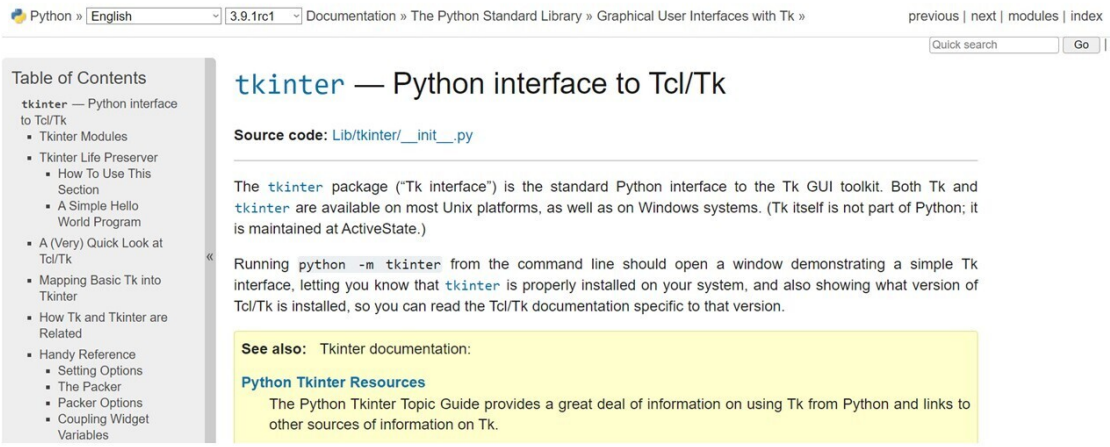
Основное отличие PySide от PyQt – это лицензии под которыми распространяются эти две обёртки Qt.

PyQt5 распространяется под GPL и коммерческой лицензией.

А PySide2 распространяется как Qt под GPL, LGPL и коммерческой лицензией.

То есть если вы пишите открытое ПО – можно использовать как PyQt5, так и PySide2.

Но если вы пишите закрытое/коммерческое ПО – бесплатно можно использовать только PySide2, а для использования PyQt5 потребуется покупать коммерческую лицензию.



Tkinter – это самая популярная библиотека для создания графического интерфейса пользователя или настольных приложений.

Tkinter – это комбинация стандартного графического интерфейса пользователя Tk и Python.

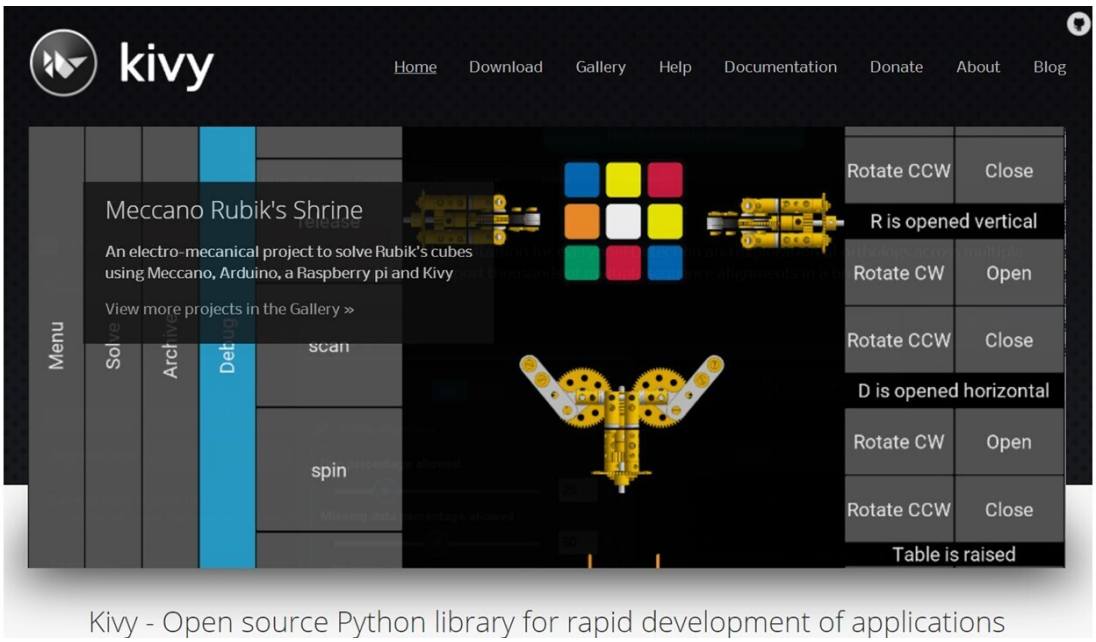
А Tk – это кроссплатформенная библиотека графического интерфейса с открытым исходным кодом.

Tkinter входит в стандартную библиотеку Python.

И Tkinter – это свободное программное обеспечение, распространяемое под Python-лицензией.

TKinter поставляется с хорошей документацией, что является основным ее достоинством.

И получить ответы на свои вопросы здесь легко, так как у Tkinter тысячи пользователей, потому что эта библиотека используется в течение очень долгого времени.



Kivy - Open source Python library for rapid development of applications

Kivy – это бесплатная среда Python с открытым исходным кодом для разработки кросс-платформенных приложений с поддержкой мультитач с пользовательским интерфейсом.

Kivy создана поверх OpenGL и для создания пользовательских интерфейсов дает возможность один раз написать код и запустить его на разных платформах Windows, MacOSX, Linux, Android, iOS и Raspberry.

WxPython – это обёртка библиотеки кроссплатформенного графического интерфейса пользователя wxWidgets, написанной на языке программирования C++.

Это еще одна из альтернатив Tkinter, которая поставляется вместе с Python.

И WxPython реализована в виде модуля расширения Python.

Dear

PyGUI – это простая и легкая библиотека графического интерфейса пользователя, так как она полностью связана с языком программирования Python.

Dear PyGui предоставляет оболочку библиотеки C++ Dear ImGui, которая имитирует традиционный графический интерфейс.

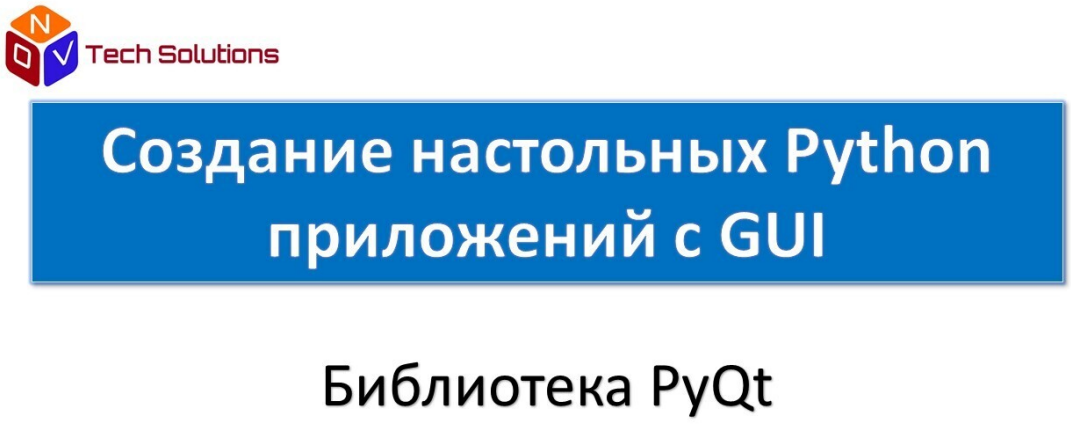
Это кроссплатформенная среда приложений с графическим интерфейсом пользователя, которая отображает естественный графический интерфейс платформы.

Здесь мы перечислили наиболее широко используемые и лучшие доступные фреймворки графического пользовательского интерфейса Python.

И вы можете выбрать наиболее подходящую вам среду для разработки графического интерфейса Python.

Далее мы более подробно разберем каждую библиотеку.

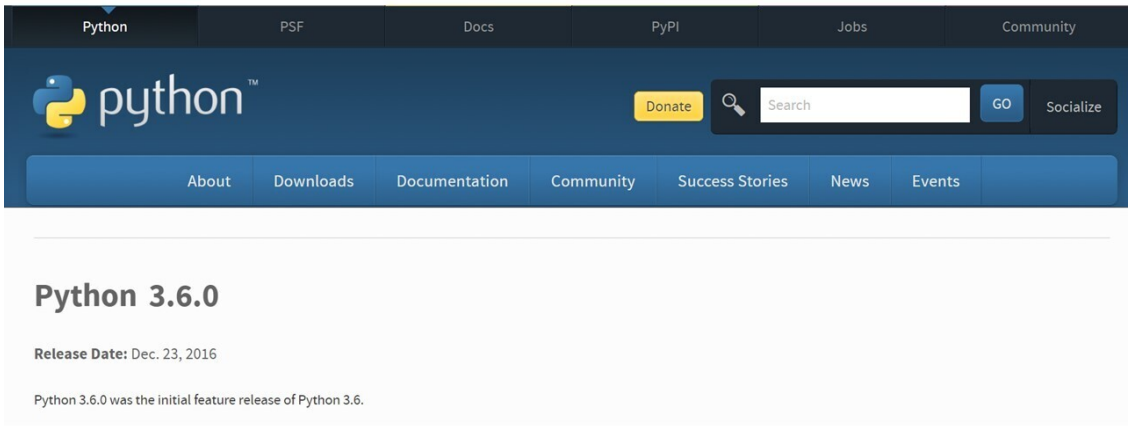
Библиотека PyQt



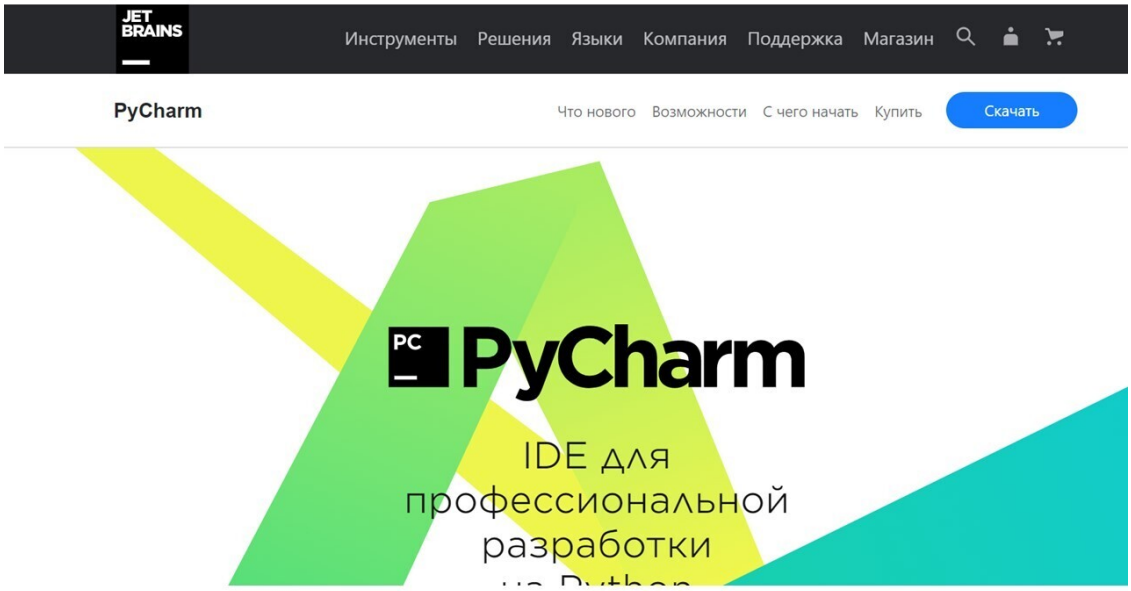
PyQt – это библиотека, которая позволяет использовать библиотеку графического интерфейса Qt в Python.

Сама библиотека Qt написана на C ++.

Самая последняя версия библиотеки – это PyQt5, и она поддерживает последнюю версию Qt5.



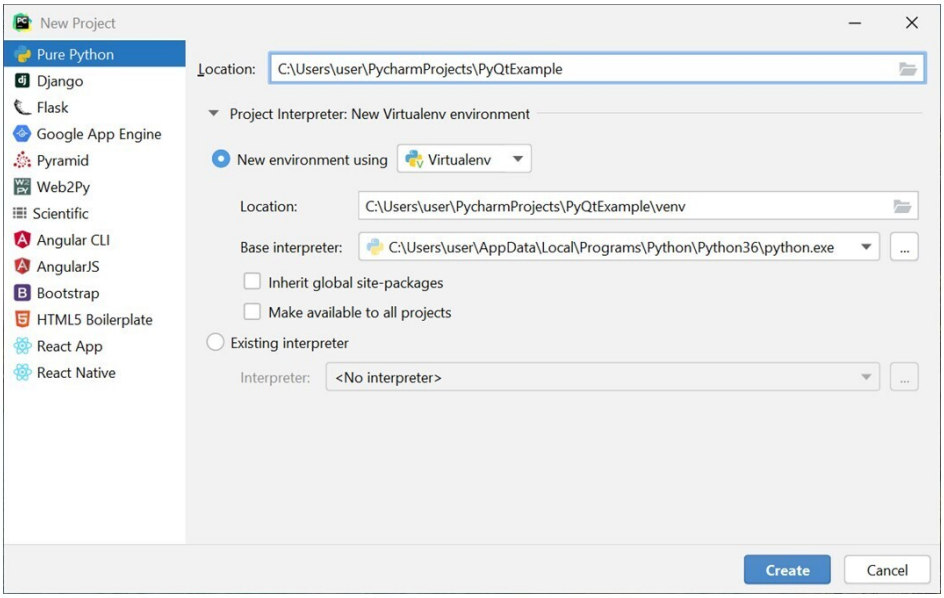
Для работы с библиотекой PyQt5, установим питон 3.6.



Для разработки приложений питон с графическим интерфейсом пользователя мы будем пользоваться средой разработки

PyCharm

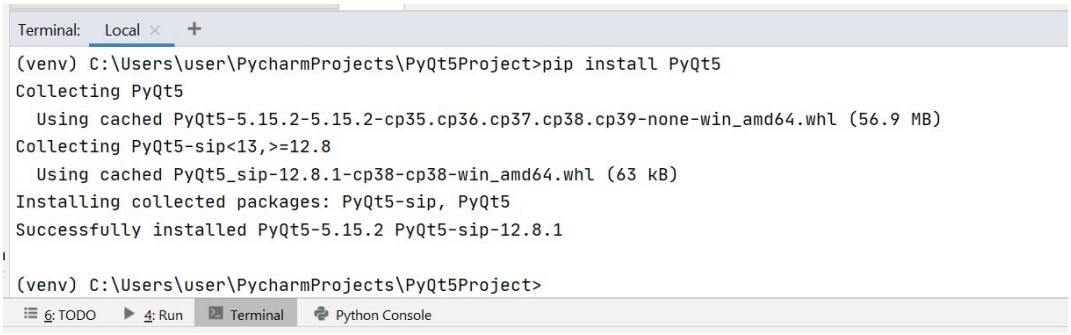
.



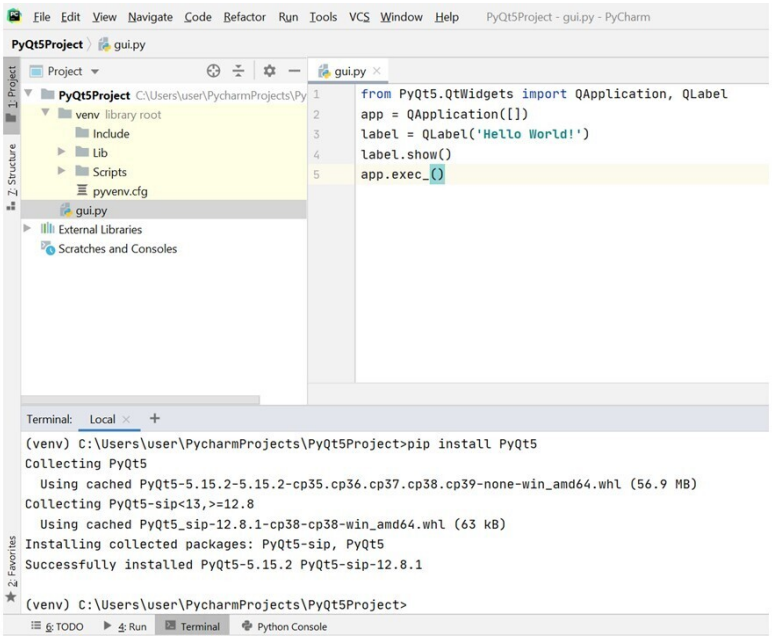
Создадим питон проект. При этом будет автоматически создана виртуальная среда.

Виртуальная среда – это просто локальный каталог, содержащий библиотеки для конкретного проекта.

pip install PyQt5



И для установки библиотеки PyQt просто наберите в окне терминала, в командной строке pip install PyQt5.



Далее в проекте создадим питон файл и наберем в нем код.

```
from PyQt5.QtWidgets import QApplication, Qlabel

app = QApplication([])

label = QLabel('Hello World!')

label.show()

app.exec_()
```

Сначала мы загружаем PyQt с помощью оператора импорта.

И из PyQt5 виджетов импортируем QApplication, QLabel.

Класс QApplication управляет потоком управления и основными настройками приложения с графическим интерфейсом.

Виджет QLabel обеспечивает отображение текста или изображения.

Затем мы создаем QApplication с помощью команды:

app = QApplication

Это требование Qt – каждое приложение с графическим интерфейсом должно иметь ровно один экземпляр QApplication.

Здесь квадратные скобки представляют аргументы командной строки, переданные приложению.

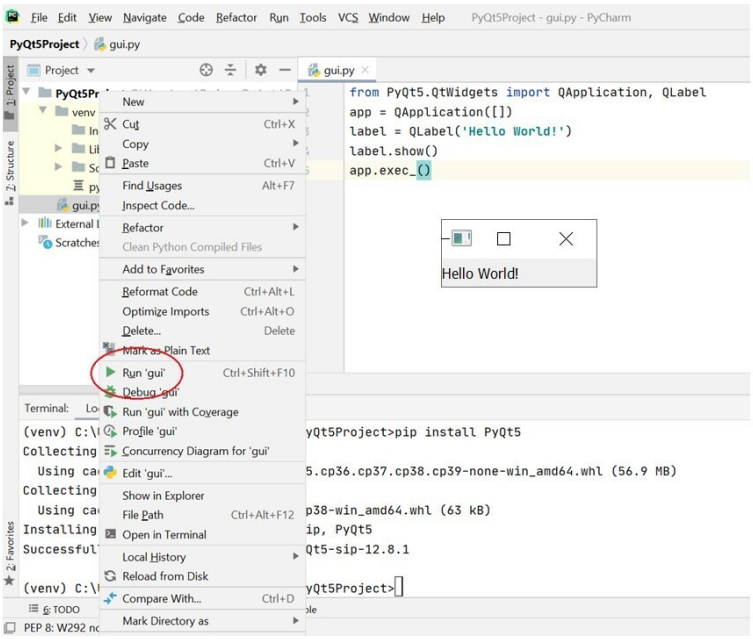
Так как наше приложение не использует никаких параметров, мы оставляем скобки пустыми.

Далее мы создаем простую метку 'Привет, мир!'.

И затем мы говорим Qt показать метку на экране с помощью команды show.

И последний шаг – это передать управление среде Qt и попросить ее «запустить приложение, пока пользователь не закроет его».

Это делается с помощью команды exec.

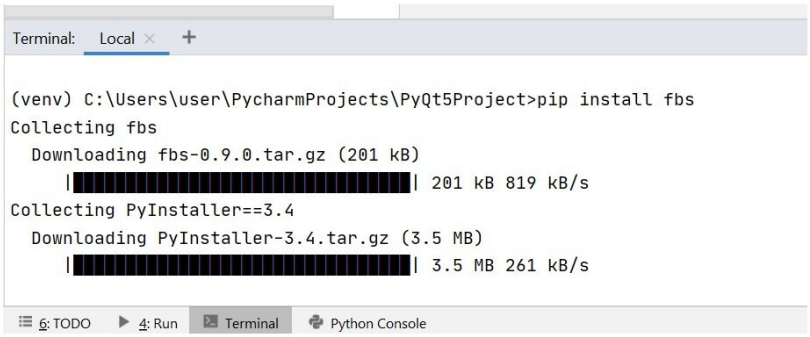


Далее нажмем правой кнопкой мыши на созданном питон файле и выберем команду run

.

В результате будет запущено приложение и откроется окно с меткой.

pip install fbs



Теперь у нас есть приложение с графическим интерфейсом пользователя.

И оно работает на вашем компьютере.

Вопрос – как его передать другим людям, чтобы они тоже могли его запустить?

Вы можете попросить пользователей вашего приложения установить Python и PyQt, а затем предоставить им свой исходный код.

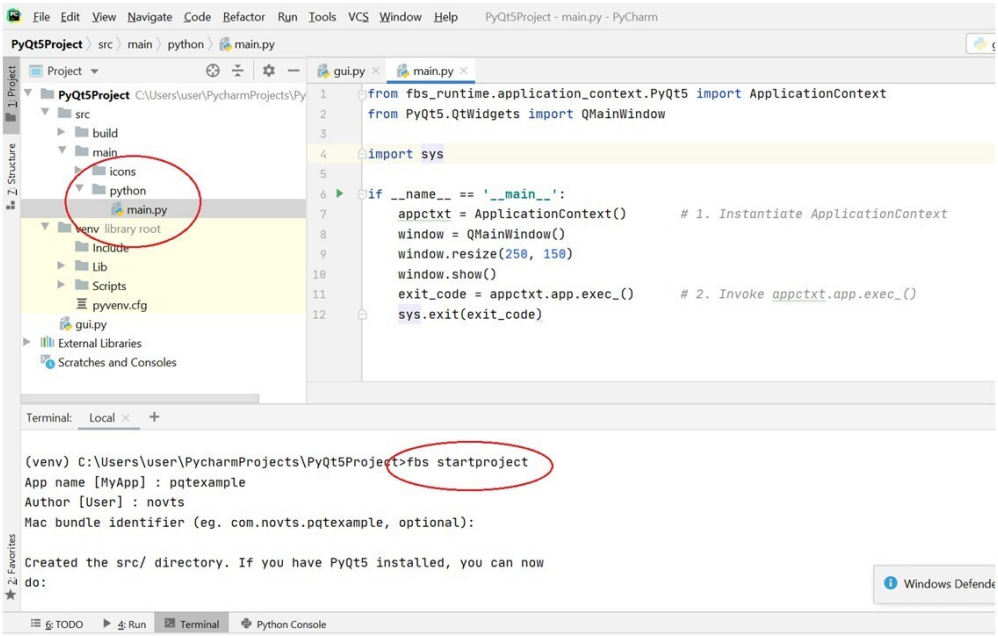
Но это очень неудобно.

Вместо этого нам нужен исполняемый файл, который другие люди могут запускать в своих системах, ничего не устанавливая.

В Python процесс превращения исходного кода в автономный исполняемый файл называется замораживанием.

Хотя существует множество библиотек, которые решают эту проблему, например PyInstaller, py2exe и так далее, здесь мы будем использовать библиотеку под названием fbs, которая позволяет создавать автономные исполняемые файлы для приложений PyQt.

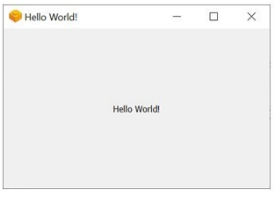
Поэтому для начала установим библиотеку fbs.



Далее мы в терминале запускаем команду fbs startproject.
В результате выполнения которой будет создана папка src/main/python/с файлом main.py.
Команда startproject создает необходимую структуру папок для приложения fbs.
Если мы наберем в терминале команду fbs run, откроется просто пустое окно.
Теперь, как нам вставить в это окно нашу метку.

```
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QMainWindow, QLabel
from PyQt5.QtCore import Qt

import sys

if __name__ == '__main__':
    appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
    window = QMainWindow()
    window.setWindowTitle("Hello World!")
    window.resize(500, 300)
    label = QLabel('Hello World!')
    label.setAlignment(Qt.AlignCenter)
    window.setCentralWidget(label)
    window.show()
    exit_code = appctxt.app.exec_()     # 2. Invoke appctxt.app.exec_()
    sys.exit(exit_code)
```

fbs run



Здесь вы можете заметить, что создание приложения с помощью fbs представляет новую концепцию – ApplicationContext.

При создании приложений PyQt5 обычно используется ряд компонентов или ресурсов, которые используются во всем приложении.

И ApplicationContext предоставляет центральное место для инициализации и хранения этих компонентов, а также предоставляет доступ к некоторым основным функциям fbs.

Объект ApplicationContext также создает и содержит ссылку на глобальный объект QApplication, доступный в ApplicationContext.app, так как каждое приложение Qt должно иметь один и только один объект QApplication для хранения цикла событий и основных настроек.

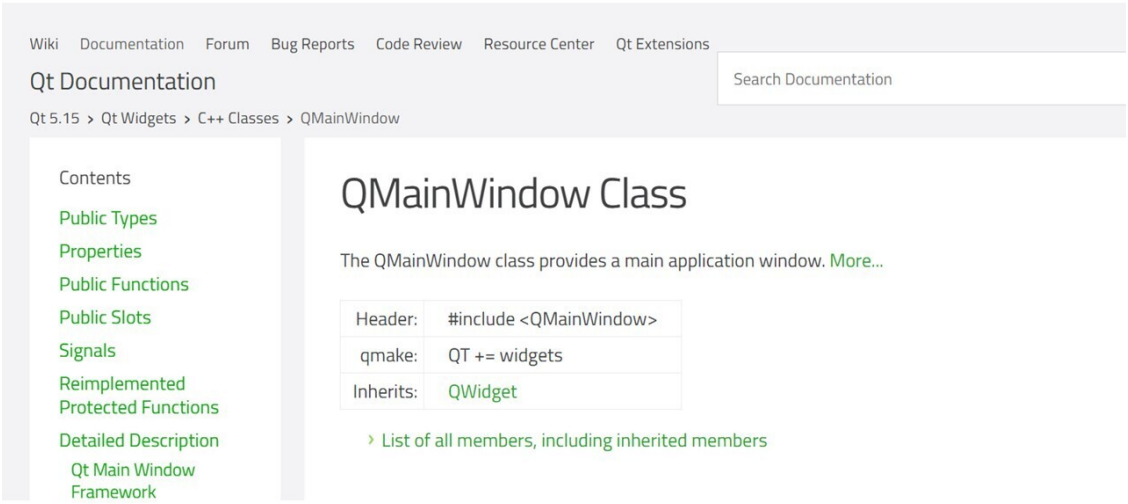
Теперь, чтобы вставить нашу метку, помимо QMainWindow импортируем метку.

Создадим метку и методом setAlignment установим ее посередине.

Методом setCentralWidget добавим метку в окно QMainWindow.

В результате после вызова команды fbs run мы увидим окно с меткой.

https://doc.qt.io/qt-5/qmainwindow.html



Более подробно про окно QMainWindow можно посмотреть в QT
документации.

```
appctxt = ApplicationContext()

im=QImage(appctxt.get_resource('images/im.jpg'))
```

```
src/main/resources/base/images/
```

Теперь, приложениям обычно требуются дополнительные файлы данных помимо исходного кода – например, изображения.

И вот здесь может быть полезен ApplicationContext.

Вы можете поместить ресурсы приложения в папку resources, и чтобы упростить загрузку ресурсов из папки resources, fbs предоставляет метод ApplicationContext.get_resource.

Этот метод принимает имя файла, который можно найти в папке resources, и возвращает абсолютный путь к этому файлу.

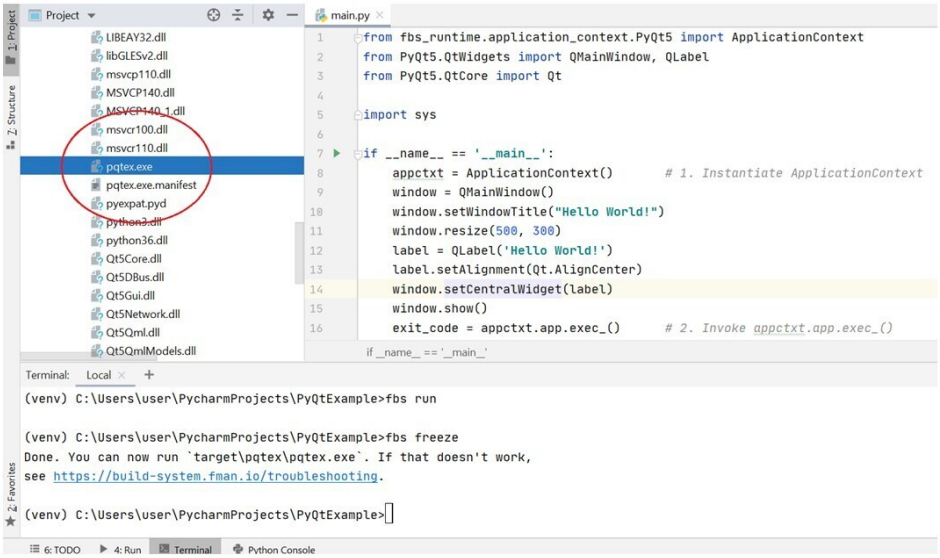
И вы можете использовать этот возвращенный абсолютный путь, чтобы открыть файл.

Папка resources должна содержать папку base плюс любую комбинацию других папок.

Базовая папка содержит файлы, общие для всех операционных систем, в то время как папки для конкретных платформ могут использоваться для файлов, специфичных для данной ОС.

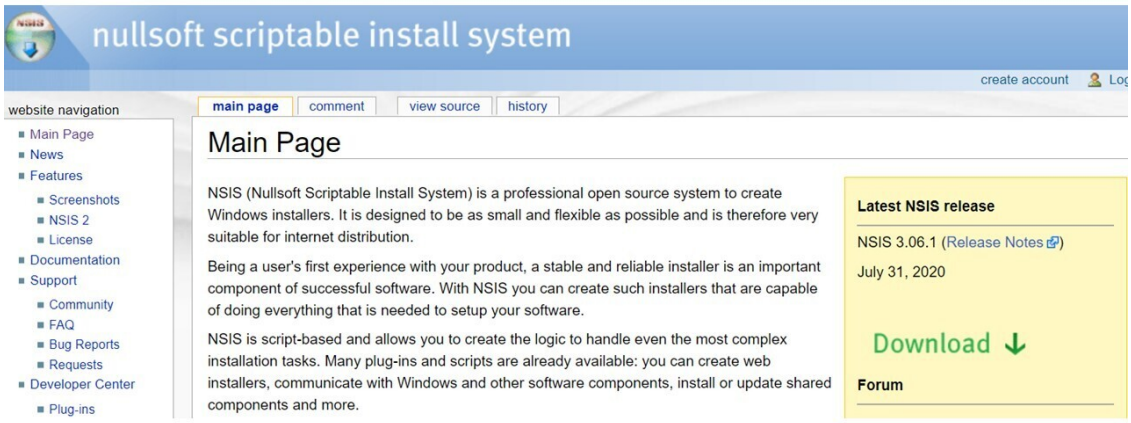
Теперь, далее мы можем использовать fbs, чтобы превратить файл питона в отдельный исполняемый файл.

fbs freeze

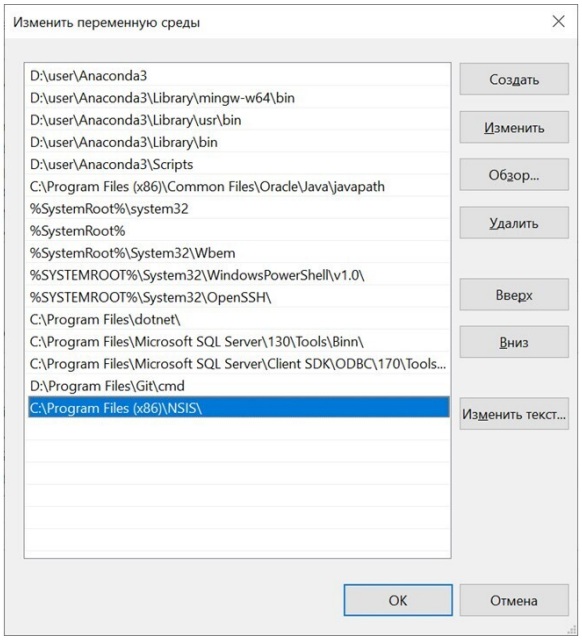


Для этого в терминале наберем команду fbs freeze.
Эта команда помещает исполняемый двоичный файл в целевую папку текущего каталога.
Далее мы можем создать установщик приложения.

https://nsis.sourceforge.io/Main_Page

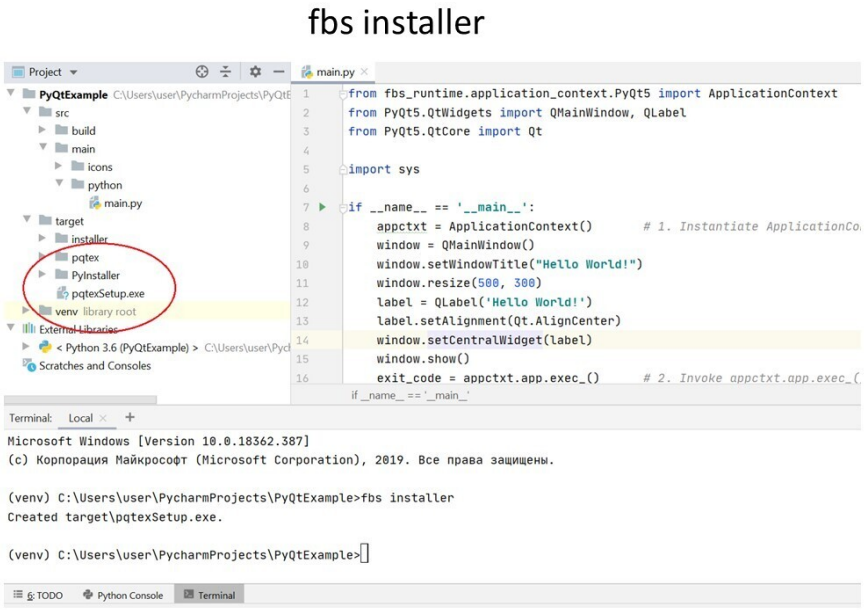


Но для начала мы должны установить NSIS – систему с открытым исходным кодом для создания установщиков Windows.



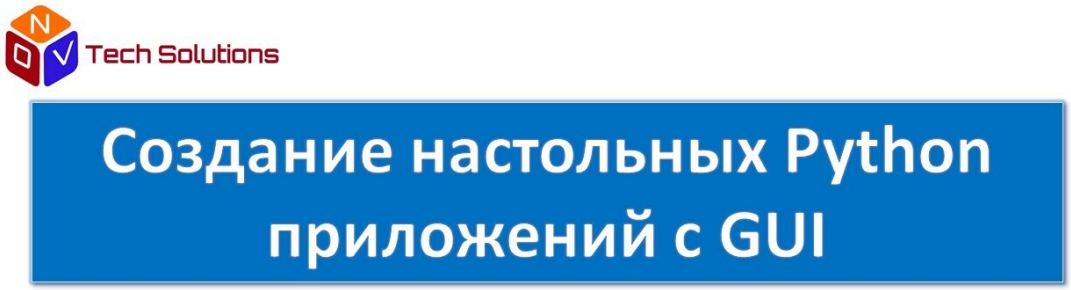
Также нужно добавить каталог NSIS в переменную среды
Windows
PATH.

После этого нужно перезапустить среду разработки PyCharm, чтобы она увидела эти изменения.



Далее создадим установщик с помощью команды fbs installer.
Эта команда помещает исполняемый двоичный файл в целевую папку текущего каталога.
Теперь вы можете отправлять его для установки приложения.

Виджеты и компоновки PyQt



Виджеты и компоновки PyQt

Система компоновки Qt предоставляет простой и мощный способ организации дочерних виджетов внутри родительского виджета.

QBoxLayout - выстраивает дочерние виджеты по горизонтали или вертикали
QButtonGroup - контейнер для организации групп виджетов кнопок
QFormLayout - управляет формами виджетов ввода и связанными с ними метками.
QGraphicsAnchorLayout - связывает виджеты между собой
QGridLayout - размещает виджеты в сетке
QGroupBox - групповой фрейм окна с заголовком
QHBoxLayout - выстраивает виджеты по горизонтали
QLayout - базовый класс компоновки
QLayoutItem - абстрактный элемент, которым управляет QLayout
QSizePolicy – описывает политику изменения размера по горизонтали и вертикали
QSpacerItem - пустое место в макете
QStackedLayout - стек виджетов, в котором одновременно виден только один виджет
QVBoxLayout - выстраивает виджеты вертикально
QWidgetItem - элемент, представляющий виджет

Qt предоставляет набор классов управления компоновкой.

Эти компоновки автоматически позиционируют и изменяют размер виджетов.

И все виджеты Qt могут использовать компоновки для управления своими дочерними элементами с помощью функции setLayout.

```
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QWidget, QLabel, QHBoxLayout
from PyQt5.QtCore import Qt
import sys

if __name__ == '__main__':
    appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
    window = QWidget()
    window.setWindowTitle("Hello World!")
    window.resize(500, 300)

    layout = QHBoxLayout()
    layout.addWidget(QLabel('One'))
    layout.addWidget(QLabel('Two'))
    layout.addWidget(QLabel('Three'))
    window.setLayout(layout)

    window.show()
    exit_code = appctxt.app.exec_()     # 2. Invoke appctxt.app.exec_()
    sys.exit(exit_code)
```

Здесь мы с помощью компоновки QHBoxLayout располагаем метки горизонтально в окне QWidget.

```
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QWidget, QLabel, QHBoxLayout, QVBoxLayout
from PyQt5.QtCore import Qt
import sys

if __name__ == '__main__':
    appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
    window = QWidget()
    window.setWindowTitle("Hello World!")
    window.resize(500, 300)

    layout = QVBoxLayout()
    layout.addWidget(QLabel('One'))
    layout.addWidget(QLabel('Two'))
    layout.addWidget(QLabel('Three'))
    window.setLayout(layout)

    window.show()
    exit_code = appctxt.app.exec_()     # 2. Invoke appctxt.app.exec_()
    sys.exit(exit_code)
```

Компоновка QVBoxLayout размещает метки вертикально.

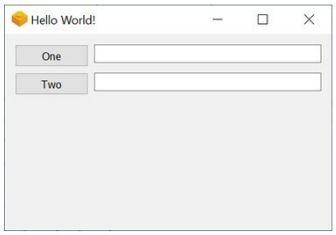
```
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QWidget, QLabel, QGridLayout
from PyQt5.QtCore import Qt
import sys

if __name__ == '__main__':
    appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
    window = QWidget()
    window.setWindowTitle("Hello World!")
    window.resize(500, 300)

    layout = QGridLayout()
    layout.addWidget(QLabel('One'),0,0,1,2, Qt.AlignHCenter)
    layout.addWidget(QLabel('Two'),1,0, Qt.AlignHCenter)
    layout.addWidget(QLabel('Three'),1,1, Qt.AlignHCenter)
    window.setLayout(layout)

    window.show()
    exit_code = appctxt.app.exec_()     # 2. Invoke appctxt.app.exec_()
    sys.exit(exit_code)
```

Компоновка QGridLayout располагает элементы в сетке.

При этом для каждого элемента можно указать строку, столбец, в которых должен находиться элемент.

Также можно указать затем сколько строк и столбцов должен заполнять элемент.

```
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QWidget, QFormLayout, QPushButton, QLineEdit
from PyQt5.QtCore import Qt
import sys

if __name__ == '__main__':
    appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
    window = QWidget()
    window.setWindowTitle("Hello World!")
    window.resize(500, 300)

    layout = QFormLayout()
    layout.addRow(QPushButton("One"), QLineEdit());
    layout.addRow(QPushButton("Two"), QLineEdit());
    window.setLayout(layout)

    window.show()
    exit_code = appctxt.app.exec_()     # 2. Invoke appctxt.app.exec_()
    sys.exit(exit_code)
```



Компоновка QFormLayout добавляет два виджета в строку, обычно QLabel и QLineEdit для создания форм.

В качестве резюме – виджеты могут иметь в качестве родительских только другие виджеты, но не компоновки.

Но вы можете вкладывать компоновки в родительскую компоновку с помощью метода addLayout, тогда внутренний макет становится дочерним по отношению к макету, в который он вставлен.

Теперь, модуль Qt Widgets предоставляет набор элементов пользовательского интерфейса для создания пользовательских интерфейсов приложения.

И все, что вы видите в приложении PyQt, представляет собой виджеты: кнопки, метки, окна, диалоговые окна, индикаторы выполнения и т. д.

```
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QWidget, QLabel
from PyQt5.QtCore import Qt

import sys

if __name__ == '__main__':
   appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
   window = QWidget()
   window.setWindowTitle("Hello World!")
   window.resize(500, 300)

   label = QLabel(window)
   label.setText("Hello PyQt5")
   label.adjustSize()
   label.move(200, 100)

   window.show()
   exit_code = appctxt.app.exec_()     # 2. Invoke appctxt.app.exec_()
   sys.exit(exit_code)
```

С меткой мы уже познакомились.

При создании метки, передавая в конструктор в качестве параметра объект окна, мы сообщаем, что метка является частью окна.

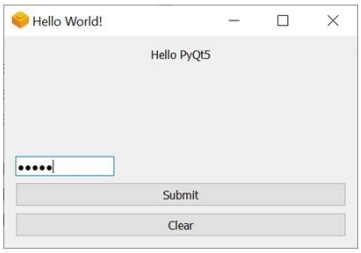
Метки имеют размер по умолчанию, и для длинных строк текста размер по умолчанию может быть слишком мал.

К счастью, у нас есть метод adjustSize, который автоматически настраивает размер метки.

В противном случае длинный текст будет отображаться на экране только частично.

И метод move определяет начальную позицию метки от левого верхнего угла экрана.

```
def show():
   print(line.text())


line = QLineEdit()
line.setEchoMode(QLineEdit.Password)
line.setFixedWidth(140)

buttonS = QPushButton()
buttonS.setText("Submit")
buttonS.clicked.connect(show)

buttonC = QPushButton()
buttonC.setText("Clear")
buttonC.setIcon(QIcon('im.jpg'))
buttonC.clicked.connect(line.clear)
```



Каждому графическому интерфейсу нужен какой-нибудь способ получения ввода от пользователя.

В PyQt такой способ ввода данных – это использование виджета QLineEdit.

Или если вы хотите получить ввод многострочного текста, вам нужно использовать виджет QTextEdit.

И конечно ввод как правило используется вместе с кнопкой QPushButton.

Чтобы получить введенный текст из виджета QLineEdit, вы должны использовать метод text.

Здесь мы создаем кнопку Submit, которая вызывает функцию show, использующую метод text виджета QLineEdit.

Мы также создаем кнопку Clear, которая вызывает метод clear виджета QLineEdit, который удаляет все его содержимое.

Метод SetEchoMode принимает несколько различных «режимов», одним из которых является режим пароля, который скрывает ввод.

Используя метод setFixedWidth, мы можем установить размер виджета QLineEdit в пикселях.

Значение по умолчанию для каждого виджета составляет около 100 пикселей.

Теперь о кнопке QPushButton.

Как следует из названия, это кнопка, которая запускает функцию при нажатии.

Кнопка, которая не связана с функцией, бесполезна.

Кнопки предназначены для подключения функции, которая будет выполняться после нажатия кнопки.

И такая функция подключается с помощью метода clicked.connect.

Также вы можете установить изображение на кнопку, с помощью виджета QIcon.

Просто передайте в него путь к файлу, и все готово.

```
def show():
  print(combo.currentText())


buttonS.clicked.connect(show)


combo = QComboBox()
combo.addItem("Python")
combo.addItem("Java")
combo.addItem("C++")
combo.setFixedWidth(140)


combo.addItems(["Python", "Java", "C++"])
```

Виджет QComboBox представляет собой раскрывающийся список элементов, из которых пользователь может выбрать свой вариант.

Преимущество этого виджета в том, что он занимает очень мало места на экране, при наличии большого списка элементов.

Здесь мы добавляем элементы в список по одному методом addItem.

Хотя мы можем добавить сразу все элементы кортежем с помощью метода addItems.

Зафиксировать ширину списка мы можем методом setFixedWidth.

Теперь, обработать выбор пользователем элемента в списке мы можем с помощью метода currentText(), который возвращает элемент списка в виде строки.

```
def show():
   print(check.text())
   print(check.checkState())
   print(check.isChecked())



check = QCheckBox()
check.setText("Option1")
check.stateChanged.connect(show)
```

Теперь перейдем к флажкам и радиокнопкам.

И здесь мы можем использовать сам флажок как кнопку, чтобы связать его с функцией обработки выбора флажка.

Мы делаем это с помощью метода stateChanged.connect.

Вы можете получить значение флажка с помощью метода text, который вернет текстовое значение флажка.

И вы можете использовать метод checkState, который возвращает целое число 0, если флажок не выбран и 2 – если он выбран.

Метод isChecked возвращает true, если флажок выбран.

```
def show():
   print(radio1.isChecked(), radio1.text())
   print(radio2.isChecked(), radio2.text())


radio1 = QRadioButton()
radio1.setText("Option1")
radio1.toggled.connect(show)
layout.addWidget(radio1)

radio2 = QRadioButton()
radio2.setText("Option2")
radio2.toggled.connect(show)
layout.addWidget(radio2)
```

В отличие от флажка, вы можете выбрать только одну радиокнопку из многих.

И здесь мы также можем использовать саму радиокнопку как кнопку, чтобы связать ее с функцией обработки выбора.

Мы делаем это с помощью метода toggled.connect.

Вы можете получить значение радиокнопки с помощью метода text, который вернет текстовое значение переключателя.

Есть еще один метод, который вы можете использовать – это метод isChecked, который возвращает true или false, показывая состояние выбора переключателя.

```
def show():
   print(text.toPlainText())

buttonS.clicked.connect(show)


text = QTextEdit()
text.setPlaceholderText("Enter some text here")
text.setUndoRedoEnabled(True)
layout.addWidget(text)
```

Как и виджет QLineEdit, виджет QTextEdit используется для ввода данных пользователем в виде текста.

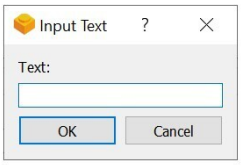
Однако, в отличие от QLineEdit, который вводит только одну строку, QTextEdit позволяет ввести несколько строк текста.

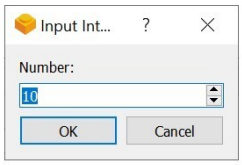
Чтобы получить от пользователя введенный текст, мы можем использовать метод toPlainText и метод кнопки clicked.connect.

Метод setPlaceholderText используется для установки в виджет выделенного серым цветом текста, который исчезает при взаимодействии с виджетом.

Метод setUndoRedoEnabled отключает / включает возможность для пользователя использовать функции Undo и Redo с помощью клавиш Ctrl + Z и Ctrl + Y.

```
buttonD.clicked.connect(display)

def display():
    text , pressed = QInputDialog.getText(window, "Input Text", "Text: ",QLineEdit.Normal, "")
    if pressed:
        print(text)
```






```
intV, pressed = QInputDialog.getInt(window, "Input Integer","Number:", 10, 0, 100, 1)
```

Помимо строки ввода и текстовой области, диалоговые окна обычно используются для ввода данных пользователя.

И в PyQt5 есть виджет QInputDialog, который позволяет создавать множество различных диалоговых окон для ввода данных различными способами.

И самое распространенное диалоговое окно – это ввод текста.

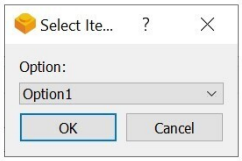
Оно представляет собой простое поле ввода и две кнопки, OK и Отмена.

Функция getText возвращает два значения, поэтому мы устанавливаем две переменные для них.

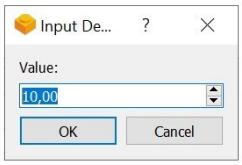
Первая переменная input получает ввод пользователя, вторая переменная получает значение True или False, указывающее, была ли нажата кнопка OK или нет.

Далее есть диалоговое окно ввода целого числа.

Здесь вы можете использовать стрелки для изменения целочисленного значения или непосредственно ввести значение самостоятельно.




```
options = ("Option1", "Option2", "Option3")
option, pressed = QInputDialog.getItem(window, "Select Item", "Option:", options, 0, False)
```




```
decimal, pressed = QInputDialog.getDouble(window, "Input Decimal", "Value:",10.00, 0, 100, 2)
```

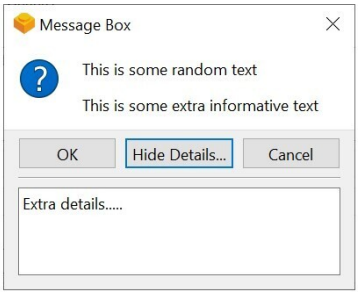
Следующее окно – это окно выбора элемента.

И здесь мы сначала определяем список / кортеж строк, отображаемых в качестве элементов.

И последнее диалоговое окно – это окно ввода чисел с плавающей запятой.

Здесь вы также можете использовать стрелки для изменения значения или непосредственно ввести значение самостоятельно.

```
def show_popup():
   msg = QMessageBox()
   msg.setWindowTitle("Message Box")
   msg.setText("This is some random text")
   msg.setIcon(QMessageBox.Question)

   msg.setStandardButtons(QMessageBox.Cancel | QMessageBox.Ok)
   msg.setDefaultButton(QMessageBox.Ok)

   msg.setDetailedText("Extra details.....")
   msg.setInformativeText("This is some extra informative text")

   msg.buttonClicked.connect(popup)

   x = msg.exec_()

def popup(i):
   print(i.text())
```



QMessageBox – это виджет, который используется для создания диалоговых окон для отображения различных сообщений и кнопок.

Как правило, в приложении определяется метод, который открывает окно сообщения.

И этот метод связывается с какой-нибудь кнопкой или событием приложения.

Здесь метод setWindowTitle устанавливает заголовок окна сообщения.

Метод setText устанавливает текст под заголовком.

Метод exec открывает окно сообщения.

Окно сообщения имеет 4 различных типа значков, которые можно изменить с помощью метода setIcon.

В окне сообщения по умолчанию используется кнопка «ОК».

Однако на самом деле существует более десятка различных кнопок, которые QMessageBox предлагает для использования.

Используя функцию setStandardButtons, мы можем установить другой тип кнопки.

Когда окно открывается, вокруг кнопки «ОК» есть синий контур.

Это обозначение кнопки по умолчанию.

Используя функцию setDefaultButton, мы можем изменить кнопку по умолчанию.

По умолчанию у нас есть только одна область в QMessageBox, где мы показываем текст.

Однако есть еще две дополнительные области, которые мы можем разблокировать, чтобы добавить больше текста.

Первая область – это дополнительный текстовый раздел в самом окне сообщения, который мы можем определить с помощью функции setInformativeText.

Вторая область текста отображается в расширении окна.

При настройке этого раздела автоматически создается кнопка, которая используется для отображения этой области.

И для создания этой области потребуется функция setDetailedText.

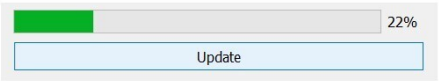
Теперь, если у нас есть 3 разные кнопки в окне сообщений, как мы узнаем, какая из них была нажата?

Используя метод buttonClicked.connect, мы можем вызвать запуск функции всякий раз, когда нажимается кнопка в окне сообщения.

Здесь когда нажимается кнопка окна сообщений, она автоматически запускает функцию popup, объявленную в buttonClicked.connect.

И вызов функции text кнопки вернет ее текстовое значение.

```
def update():
    value = prog_bar.value()
    prog_bar.setValue(value + 1)


prog_bar = QProgressBar()
prog_bar.setGeometry(50, 50, 250, 30)
prog_bar.setValue(0)
layout.addWidget(prog_bar)

buttonUp = QPushButton()
buttonUp.setText("Update")
buttonUp.clicked.connect(update)
layout.addWidget(buttonUp)
```



reset() - сбрасывает значение на индикаторе выполнения.

value() - возвращает текущее целочисленное значение.

setRange (n1, n2) - определяет начальную и конечную точки.
Значения по умолчанию - 0 и 100 соответственно.

Виджет ProgressBar, индикатор выполнения – это отличный способ визуализировать процесс выполнения длительных операций, таких как передача файлов, загрузка, выгрузка, копирование и т. д.

Здесь с помощью метода setGeometry, мы определяем расположение и размеры индикатора выполнения.

Первые два параметра представляют положение X и Y индикатора выполнения в окне.

Третий и четвертый параметры – это ширина и высота индикатора выполнения.

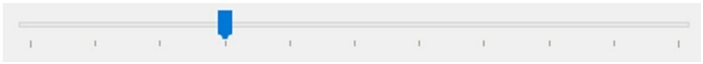
Метод setValue устанавливает значения индикатора.

Однако имейте в виду, что диапазон для индикатора выполнения составляет от 0 до 100.

Если вы хотите изменить этот диапазон, используйте метод setRange.

Метод reset сбрасывает значение на индикаторе выполнения.

Метод value возвращает текущее целочисленное значение индикатора.



```
slider = QSlider(Qt.Horizontal)
slider.setGeometry(50, 50, 200, 50)
slider.setMinimum(0)
slider.setMaximum(20)
slider.setTickPosition(QSlider.TicksBelow)
slider.setTickInterval(2)
slider.valueChanged.connect(show)
layout.addWidget(slider)

def show():
   print(str(slider.value()))
```

Виджет PyQt QSlider предоставляет графический интерфейс для выбора значения из диапазона различных значений.

У виджета Slider есть ползунок, который можно перемещать.

При перемещении ползунка выбранное значение соответственно изменяется.

По умолчанию ориентация слайдера вертикальная.

Для горизонтального ползунка вам нужно использовать значение Qt.Horizontal.

Затем нам нужно установить минимальный и максимальный диапазон с помощью методов setMinimum и setMaximum.

Используя метод setGeometry, мы определяем расположение и размеры слайдера.

Первые два параметра определяют положение ползунка по осям X и Y в окне.

Третий и четвертый параметры – это ширина и высота слайдера.

Затем нам нужно установить Ticks, маркеры по ползунку.

Вы можете выбрать место размещения тиков и установить интервал для расстояния между тиками.

Используя метод valueChanged, мы можем связать значение ползунка с функцией show.

Каждый раз, когда значение ползунка изменяется, вызывается функция show и показывается значение слайдера.


```
date = QDateEdit()
date.setMinimumDate(QDate(1900, 1, 1))
date.setMaximumDate(QDate(2100, 12, 31))
layout.addWidget(date)


def show():
   print(date.date().toPyDate())
```

Виджет PyQt5 QDateEdit – это интерактивный и визуальный способ ввести дату в приложении.

Здесь вы можете редактировать дату вручную или с помощью встроенных клавиш со стрелками.

И здесь вам не нужно назначать минимальное или максимальное значение, PyQt сам назначит значения по умолчанию.

Но вы можете сделать это с помощью методов setMinimumDate и setMaximumDate.

Чтобы получить введенную дату, вы можете использовать метод date и метод toPyDate, чтобы распечатать дату в более читаемом формате.

Конец ознакомительного фрагмента.