Учебное пособие по PyQt5 с примерами: разработка графического интерфейса с использованием PyQt на Python

PyQt — это привязка Python к набору инструментов виджетов с открытым исходным кодом Qt, который также функционирует как кроссплатформенная среда разработки приложений. Qt — популярная среда C++ для написания приложений с графическим интерфейсом для всех основных настольных, мобильных и встраиваемых платформ (поддерживает Linux, Windows, MacOS, Android, iOS, Raspberry Pi и другие).

PyQt — это бесплатное программное обеспечение, разработанное и поддерживаемое компанией Riverbank Computing, базирующейся в Англии, тогда как Qt разработан финской фирмой The Qt Company.

Вот важные особенности PyQt:

Изучите PyQt, который состоит из более чем шестисот классов, охватывающих ряд функций, таких как

- Графические интерфейсы пользователя
- Базы данных SQL
- Веб-инструментарии
- XML-обработка
- сеть

Эти функции можно комбинировать для создания расширенных пользовательских интерфейсов, а также автономных приложений. Многие крупные компании во всех отраслях используют Qt.

PyQt доступен в двух редакциях: PyQt4 и PyQt5. PyQt4 предоставляет связующий код для привязки версий платформы Qt 4.x и 5.x, тогда как PyQt5 предоставляет привязку только для версий 5.x. В результате PyQt5 не имеет обратной совместимости с устаревшими модулями старой версии. В этом руководстве по Qt GUI PyQt5 будет использоваться для демонстрации примеров. Помимо этих двух версий,

Riverbank Computing также предоставляет PyQt3D — привязки Python для платформы Qt3D. Qt3D — это прикладная среда, используемая для создания систем моделирования в реальном времени с 2D/3D-рендерингом.

Чтобы установить PyQt5,

Шаг 1) Откройте командную строку.

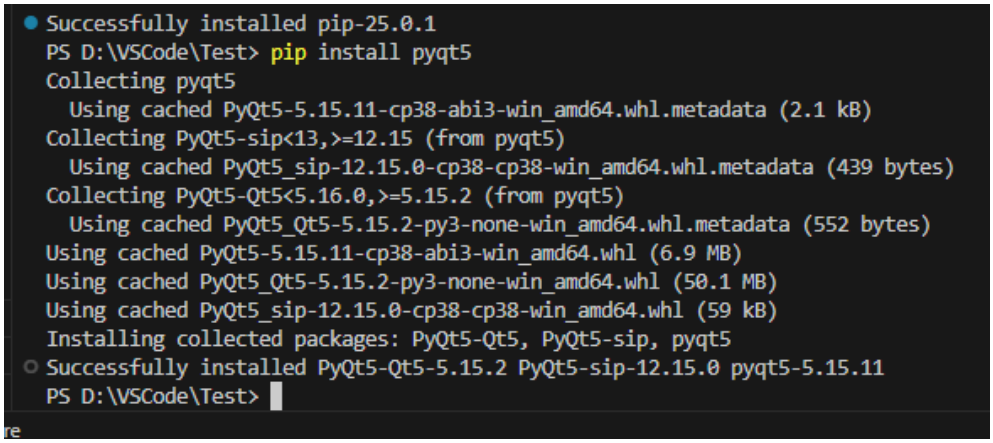
Откройте командную строку

Шаг 2) Введите следующее.

```
pip install pyqt5
```

Шаг 3) Установка прошла успешно.

На этом этапе этого руководства по PyQt5 вы загрузите пакет PyQt5и установите его в вашей системе.



После завершения перейдите к следующему разделу этого руководства по PyQt5, чтобы написать свое первое приложение с графическим интерфейсом.

Основные концепции и программы PyQt

Теперь, когда вы успешно установили PyQt5 на свой компьютер, вы готовы писать приложения для разработки графического пользовательского интерфейса Python.

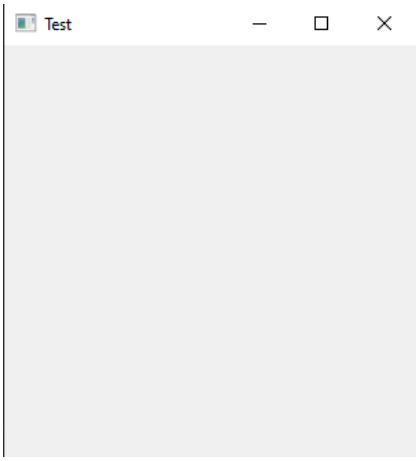
Давайте начнем с простого приложения в этом руководстве по PyQt5, которое будет отображать пустое окно на вашем экране.

Запустите средство программирования (VSCode, Pycharm) и введите следующее:

Программа 1

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

if __name__ == "__main__":
   app = QApplication(sys.argv)
   w = QWidget()
   w.resize(300, 300)
   w.setWindowTitle("Test")
   w.show()
   sys.exit(app.exec_())
```

Сохраните его как app.py (имя не имеет значения) и нажмите F5, чтобы запустить программу. Альтернативно, просто дважды щелкните сохраненный файл, чтобы запустить приложение. Если вы все сделали правильно, откроется новое окно с заголовком Test, как показано ниже.



Работает. Это немного, но достаточно, чтобы понять основы. Теперь в этом уроке по PyQt давайте подробно рассмотрим, что делает каждая строка в вашей программе.

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

Этот оператор импортирует все модули, необходимые для создания графического интерфейса, в текущее пространство имен. Модуль QtWidgets содержит все основные виджеты, которые вы будете использовать в этом руководстве по Python Qt.

```
app = QApplication(sys.argv)
```

Здесь вы создаете объект класса QApplication. Этот шаг необходим для PyQt5; каждое приложение пользовательского интерфейса должно создать экземпляр QApplication как своего рода точку входа в приложение. Если вы его не создадите, будут показаны ошибки.

sys.argv — это список параметров командной строки, которые вы можете передать приложению при его запуске через оболочку или при автоматизации интерфейса.

В этом примере PyQt5 вы не передали QApplications никаких аргументов. Следовательно, вы также можете заменить его приведенным ниже кодом и даже не импортировать модуль sys.

```
app = QApplication([])
w = QWidget()
```

Далее мы создаем объект класса QWidget. QWidget — это базовый класс всех объектов пользовательского интерфейса в Qt, и практически все, что вы видите в приложении, является виджетом. Сюда входят диалоги, тексты, кнопки, панели и т. д. Особенность, позволяющая проектировать сложные пользовательские интерфейсы, заключается в том, что виджеты могут быть вложенными, т. е. вы можете иметь виджет внутри виджета, который находится внутри другого виджета. Вы увидите это в действии в следующем разделе.

w.resize(300,300)

Метод resize класса QWidget позволяет вам установить размеры любого виджета. В данном случае вы изменили размер окна до 300 на 300 пикселей.

Здесь вы должны помнить, что виджеты могут быть вложены друг в друга, самый внешний виджет (т. е. виджет без родителя) называется окном.

w.setWindowTitle("Test")

Метод setWindowTitle() позволяет вам передавать строку в качестве аргумента, который установит заголовок окна в соответствии с переданной строкой. В примере PyQt5 в строке заголовка будет отображаться Test.

w.show()

show() просто отображает виджет на экране монитора.

sys.exit(app.exec_())

Метод app.exec_() запускает цикл событий Qt/C++. Как вы знаете, PyQt в основном написан на C++ и использует механизм цикла событий для реализации параллельного выполнения. app.exec_() передает управление Qt, который выйдет из приложения только тогда, когда пользователь закроет его из графического интерфейса. Вот почему сочетание клавиш Ctrl+C не приведет к выходу из приложения, как в других программах Python.

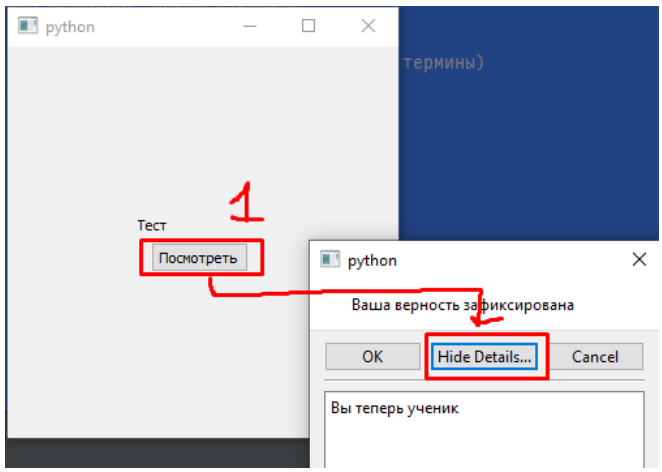
Поскольку Qt контролирует приложение, события Python не обрабатываются, если мы не настроим их внутри приложения. Также обратите внимание, что в названии метода exec есть подчеркивание; это связано с тем, что exec() уже было ключевым словом в Python, а подчеркивание разрешает конфликт имен.

В предыдущем разделе вы увидели, как создать базовый виджет в Qt. Пришло время создать более сложные интерфейсы, с которыми пользователи смогут по- настоящему взаимодействовать. Снова запустите IDLE и напишите следующее.

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


def dialog():
   mbox = QMessageBox()

   # Заголовок сообщения (перевод)
   mbox.setText("Ваша верность зафиксирована")
   # Подробное описание (перевод с учетом контекста)
```

```
    mbox.setDetailedText(
        "Вы теперь ученик")  # Пример перевода, можно изменить

    # Установка кнопок (Ok и Cancel остаются без перевода — стандартные термины)
    mbox.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)

    # Отображение окна сообщения
    mbox.exec_()


if __name__ == "__main__":
    app = QApplication(sys.argv)
    w = QWidget()
    w.resize(300, 300)
    label = QLabel(w)
    label.setText("Тест")  # Перевод "Test" на русский
    label.move(100, 130)
    label.show()

    btn = QPushButton(w)
    btn.setText('Посмотреть')  # Перевод "Beheld" (в зависимости от контекста)
    btn.move(110, 150)
    btn.show()
    btn.clicked.connect(dialog)

    w.show()
    sys.exit(app.exec_())
```

Сохраните файл как appone.py или, как угодно и запустите ПО. Если вы не допустили никаких ошибок, IDLE откроет новое окно с текстом и кнопкой, как показано ниже.



1. Как только вы нажмете кнопку в первом окне, откроется новое окно сообщения с написанным вами текстом.
2. Теперь вы можете нажать кнопку «Скрыть детали/Показать детали», чтобы переключить видимость дополнительного текста.

Как видите, поскольку мы не установили заголовок окна в окне сообщения,

заголовок по умолчанию был предоставлен самим Python.

Теперь, когда все работает, давайте посмотрим на дополнительный код, который вы добавили в предыдущий пример PyQt5.

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

При этом будут импортированы еще несколько виджетов, которые вы использовали в примерах PyQt5, а именно QLabel, QPushButton и QMessageBox.

```
def dialog():
    mbox = QMessageBox()

    # Заголовок сообщения (перевод)
    mbox.setText("Ваша верность зафиксирована")
    # Подробное описание (перевод с учетом контекста)
    mbox.setDetailedText(
        "Вы теперь ученик")  # Пример перевода, можно изменить

    # Установка кнопок (Ok и Cancel остаются без перевода — стандартные термины)
    mbox.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)

    # Отображение окна сообщения
    mbox.exec_()
```

Здесь вы определили метод с именем диалог, который создает виджет окна сообщения и задает текст для кнопок и других полей.

Метод диалога вызывается из основного блока программы при нажатии кнопки в конкретном виджете (в данном случае кнопка PushButton). Событие щелчка, вызванное этой кнопкой, вызывает выполнение этой функции. Такая функция в Qt называется слотом, и вы узнаете больше о **сигналах** и **слотах** в следующих параграфах.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    w = QWidget()
    w.resize(300, 300)
    label = QLabel(w)
    label.setText("Тест")  # Перевод "Test" на русский
    label.move(100, 130)
    label.show()
```

Это основной раздел приложения, и, как и в предыдущем примере, вы начинаете с создания экземпляра QApplication, за которым следует простой

виджет, то есть экземпляр QWidget.

```
label = QLabel(w)
btn = QPushButton(w)
```

Вы добавили в это приложение два новых виджета: QLabel и QPushButton. QLabel используется для печати нередактируемого текста или заполнителей внутри виджета, тогда как QPushButton используется для создания кнопки, на которую можно нажать.

Здесь важно отметить, что при создании объектов label и btn вы передаете объект окна (w) конструкторам QLabel и QPushButton. Вот как работает вложение в PyQt5. Чтобы создать виджет внутри другого виджета, вы передаете ссылку родительского виджета конструктору дочернего.

```
label.move(100,130)
btn.move(110,150)
```

move() используется для установки положения виджета относительно его родительского виджета. В первом случае метка будет перемещена на 100 пикселей слева и на 130 пикселей сверху окна.

Аналогично, кнопка будет размещена на расстоянии 110 пикселей слева и 150 пикселей сверху окна. Этот пример представляет собой грубый способ создания макетов и обычно не используется в производстве; оно включено сюда только в учебных целях. Qt поддерживает различные макеты, которые вы подробно увидите в следующих разделах этого руководства по PyQt.

```
btn.clicked.connect(dialog)
```

Наконец, это пример сигналов и слотов в Qt. В приложениях на основе графического пользовательского интерфейса функции выполняются на основе действий, выполняемых пользователем, таких как наведение курсора на элемент или нажатие кнопки. Эти действия называются **событиями**. Напомним, что метод app.exec_() передает управление **event**-циклу Qt. Именно для этого существует цикл событий: для прослушивания событий и выполнения ответных действий.

Всякий раз, когда происходит событие, например, когда пользователь нажимает кнопку, соответствующий виджет Qt выдает **сигнал**. Эти сигналы можно подключить к функциям Python (например, к функции диалога в этом примере), чтобы функция выполнялась при срабатывании сигнала. Эти функции на жаргоне Qt называются **слотами**.

Следовательно, основной синтаксис для запуска функции слота в ответ на сигнал события выглядит следующим образом:

widget.signal.connect(slot)

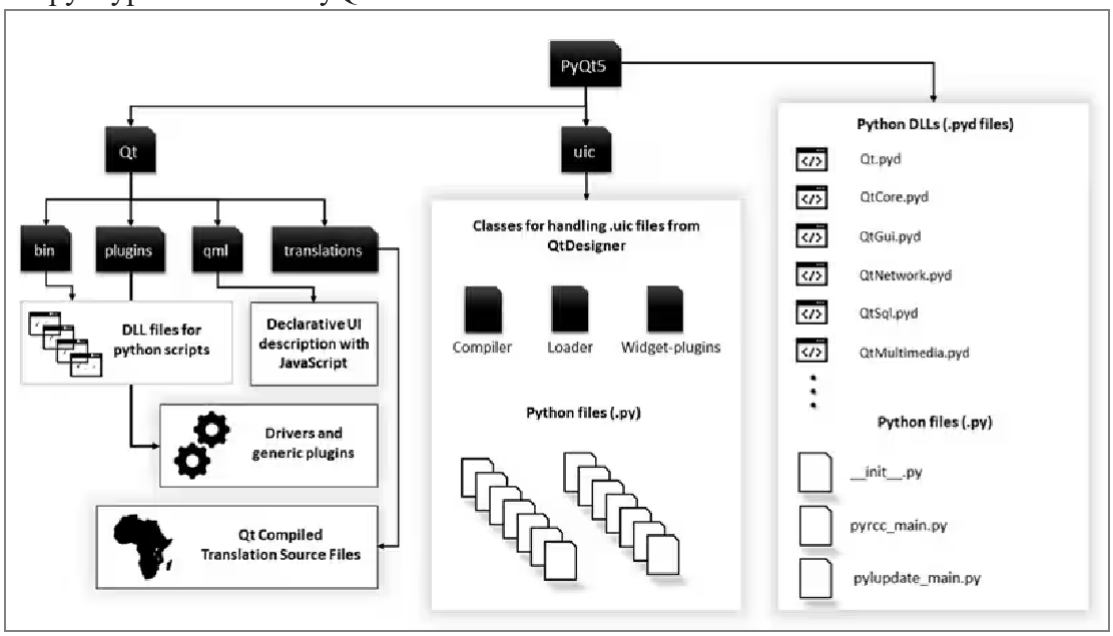
Это означает, что всякий раз, когда **сигнал** запускается **виджетом**, будет выполняться подключенная функция **слота**. Подводя итог, можно сказать, что сигналы и слоты используются Qt для связи между объектами и облегчения повторного использования и интерактивности компонентов.

Теперь, когда вы знаете, как вкладывать виджеты и реализовывать взаимодействия с помощью сигналов и слотов, вот список полезных виджетов и других классов, которые вы можете использовать в своих приложениях PyQt.

В PyQt доступно большое количество виджетов для создания приложений с графическим интерфейсом. Однако в PyQt5 произошла перестановка классов в разные модули и изменения в лицензиях.

Поэтому крайне важно иметь общее представление о структуре PyQt5. В этом разделе вы увидите, как PyQt5 организован внутри, и узнаете о различных модулях, библиотеках и классах API, предоставляемых PyQt5.

Структура каталогов PyQt5



Это основные модули, используемые привязкой Qt Python, в частности PyQt5.

- **Qt**: объединяет все упомянутые ниже классы/модули в один модуль. Это значительно увеличивает память, используемую приложением. Однако управлять платформой проще, импортировав только один модуль.
- **QtCore**: содержит основные неграфические классы, используемые другими модулями. Здесь реализован цикл событий Qt, сигналы, связь со слотами и т. д.

- **QtWidgets**: содержит большинство виджетов, доступных в Pyqt5.
- **QtGui**: содержит компоненты графического интерфейса и расширяет модуль QtCore.
- **QtNetwork**: содержит классы, используемые для реализации сетевого программирования посредством Qt. Он поддерживает TCP-серверы, TCP-сокеты, UDP-сокеты, обработку SSL, сетевые сеансы и поиск DNS.
- **QtMultimedia** обеспечивает низкоуровневую мультимедийную функциональность.
- **QtSql**: реализует интеграцию баз данных SQL. Поддерживает ODBC, MySQL, Oracle, SQLite и PostgreSQL.

Виджеты PyQt5

Вот список наиболее часто используемых виджетов в PyQt5.

- **QLineEdit**: это поле ввода, которое позволяет пользователю ввести одну строку текста.

```
line = QLineEdit()
```

- **QRadioButton**: это поле ввода с выбираемой кнопкой, похожее на переключатели в HTML.

```
rad = QRadioButton("button title")
rad.setChecked(True)       #to select the button by default.
```

- **QComboBox**: используется для отображения раскрывающегося меню со списком доступных для выбора элементов.

```
drop = QComboBox(w)
drop.addItems(["item one", "item two", "item three"])
```

- **QCheckBox**: отображает выбираемый квадрат перед меткой, которая отмечается галочкой, если выбрана, аналогично переключателям.

```
check = QCheckBox("button title")
```

- **QMenuBar**: отображает горизонтальную строку меню в верхней части окна. На эту панель можно добавлять только объекты класса QMenu. Эти объекты QMenu могут дополнительно содержать строки, объекты QAction или другие объекты QMenu.

- **QToolBar**: это горизонтальная полоса или панель, которую можно перемещать внутри окна. Он может содержать кнопки и другие виджеты.

- **QTab**: используется для разделения содержимого окна на несколько страниц, доступ к которым можно получить через разные вкладки в верхней части виджета. Он состоит из двух разделов: панели вкладок и страницы вкладок.
- **QScrollBar**: используется для создания полос прокрутки, которые позволяют пользователю прокручивать вверх и вниз внутри окна. Он состоит из подвижного ползунка, дорожки ползунка и двух кнопок для прокрутки ползунка вверх или вниз.

  scroll = QScrollBar()

- **QSplitter**: разделители используются для разделения содержимого окна, чтобы виджеты группировались правильно и не выглядели загроможденными. QSplitter — это один из основных обработчиков макета, доступных в PyQt5, который используется для разделения содержимого как по горизонтали, так и по вертикали.

- **QDock**: виджет закрепления представляет собой подокно с двумя свойствами:
  - Его можно перемещать в главном окне и
  - Его можно закрепить за пределами родительского окна в другом месте экрана.

Макеты и темы

В предыдущих примерах PyQt5 вы использовали только методы move() и resize() для установки положения виджетов в вашем графическом интерфейсе.

Однако PyQt имеет надежный механизм управления макетом, который можно использовать для создания расширенных пользовательских интерфейсов для приложений. В этом разделе вы узнаете о двух важных классах, которые используются в Qt для создания макетов и управления ими.

1. **QBoxLayout**
2. **QGridLayout**

QBoxLayout

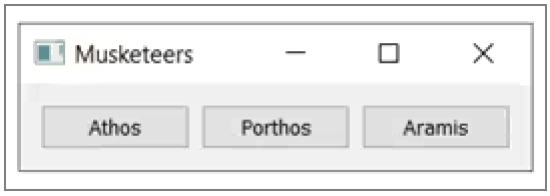
QBoxLayout используется для выравнивания дочерних виджетов макета в горизонтальный или вертикальный ряд. Два интересных класса, которые наследуются от QBoxLayout:

- QHBoxLayout: используется для горизонтального выравнивания дочерних

виджетов.

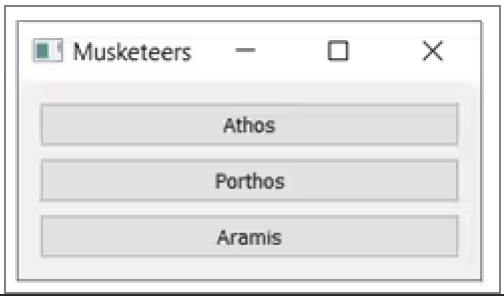
- QVBoxLayout: используется для вертикального выравнивания дочерних виджетов.

Например, так будут выглядеть три кнопки, выровненные по QHBoxLayout.




```
import sys
from PyQt5.QtWidgets import *

if __name__ == "__main__":
    app = QApplication([])
    w = QWidget()
    w.setWindowTitle("Musketeers")

    btn1 = QPushButton("Athos")
    btn2 = QPushButton("Porthos")
    btn3 = QPushButton("Aramis")
    hbox = QHBoxLayout(w)

    hbox.addWidget(btn1)
    hbox.addWidget(btn2)
    hbox.addWidget(btn3)

    w.show()

    sys.exit(app.exec_())
```

А вот как они будут выглядеть в QVBoxLayout.




```
import sys
from PyQt5.QtWidgets import *

if __name__ == "__main__":
    app = QApplication([])
    w = QWidget()
    w.setWindowTitle("Musketeers")

    btn1 = QPushButton("Athos")
```

```
    btn2 = QPushButton("Porthos")
    btn3 = QPushButton("Aramis")

    vb = QVBoxLayout(w)

    vb.addWidget(btn1)
    vb.addWidget(btn2)
    vb.addWidget(btn3)

    w.show()

    sys.exit(app.exec_())
```

Единственная функция, которая на данном этапе нуждается в пояснении, — это метод addWidget(). Он используется для вставки виджетов в макет HBox или VBox. Он также используется в других макетах, где принимает другое количество параметров, как вы увидите в следующем разделе. Виджеты появятся внутри макета в том порядке, в котором вы их вставите.

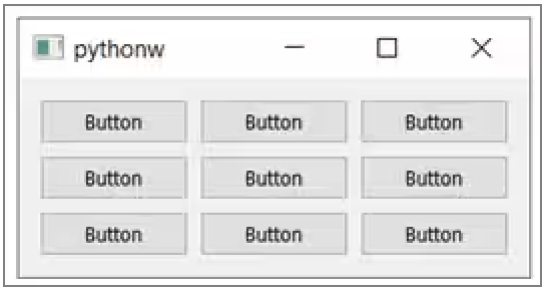
QGridLayout

QGridLayout используется для создания интерфейсов, в которых виджеты располагаются в виде сетки (например, матрицы или 2D-массива). Чтобы вставить элементы в макет сетки, вы можете использовать матричное представление, чтобы определить количество строк и столбцов в сетке, а также положение этих элементов.

Например, чтобы создать сетку 3*3 (т. е. сетку с тремя строками и тремя столбцами), вы напишете следующий код:

```
import sys
from PyQt5.QtWidgets import *

if __name__ == "__main__":
    app = QApplication([])

    w = QWidget()
    grid = QGridLayout(w)
    for i in range(3):
        for j in range(3):
            grid.addWidget(QPushButton("Button"), i, j)

    w.show()
    sys.exit(app.exec_())
```

Это будет результат:



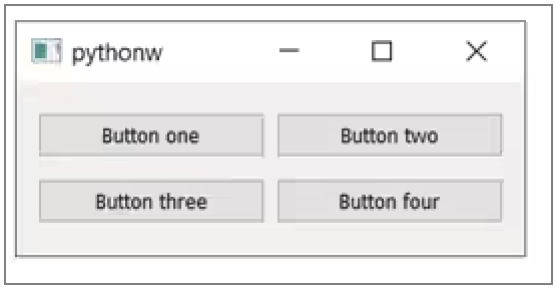
Метод addWidget() в макете сетки принимает следующие аргументы:

- Объект виджета, который вы хотите добавить в сетку.
- Координата X объекта
- Координата Y объекта
- Диапазон строк (по умолчанию=0)
- Диапазон столбцов (по умолчанию=0)

Чтобы лучше это понять, вы можете вручную вставить каждый виджет, как показано ниже.

```
import sys
from PyQt5.QtWidgets import *

if __name__ == "__main__":
   app = QApplication([])

   w = QWidget()

   grid = QGridLayout(w)
   grid.addWidget(QPushButton("Button one"), 0, 0)
   grid.addWidget(QPushButton("Button two"), 0, 1)
   grid.addWidget(QPushButton("Button three"), 1, 0)
   grid.addWidget(QPushButton("Button four"), 1, 1)

   w.show()
   sys.exit(app.exec_())
```

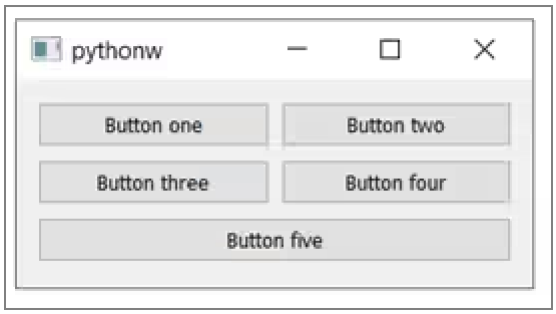
Вот как будет выглядеть сетка:



Вы также можете передать параметры rowspan и colspan в addWidget(), чтобы охватить более одной строки или столбца.

Например, grid.addWidget(QPushButton("Button five"),2,0,1,0)

Это создаст кнопку, которая растянется на оба столбца.

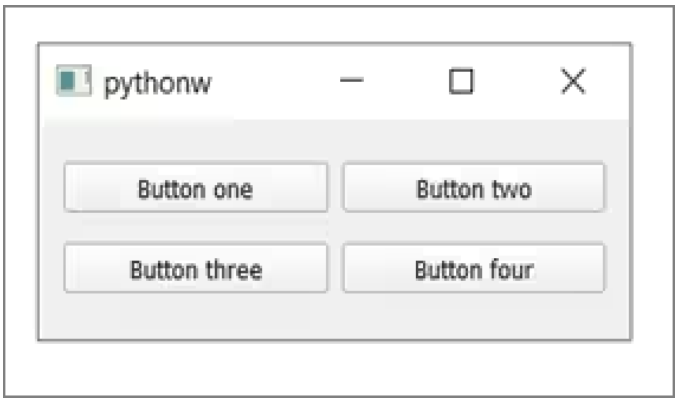


PyQt5 поставляется с некоторыми встроенными темами, которые вы можете использовать в своих приложениях. Метод **setStyle()**, вызываемый в экземпляре QApplication, используется для установки определенной темы для вашего приложения.

Например, добавление следующей строки кода изменит тему вашего приложения со стандартной на Fusion.

app.setStyle("Fusion")

Вот как будет выглядеть предыдущий пример в Fusion Theme

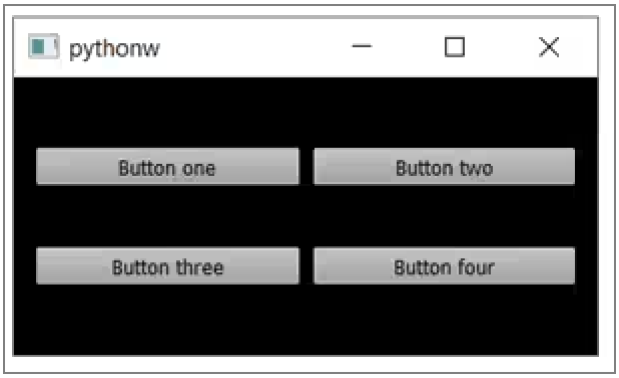


Еще одна полезная функция для оформления тем ваших приложений — метод setPalette(). Вот код для изменения цвета различных виджетов с помощью setPalette().

```
import sys
from PyQt5.QtCore import Qt

from PyQt5.QtWidgets import *
from PyQt5.QtGui import QPalette

if __name__ == "__main__":
    app = QApplication([])
    app.setStyle("Fusion")

    qp = QPalette()
```

```
qp.setColor(QPalette.ButtonText, Qt.black)
qp.setColor(QPalette.Window, Qt.black)
qp.setColor(QPalette.Button, Qt.gray)
app.setPalette(qp)

w = QWidget()

grid = QGridLayout(w)
grid.addWidget(QPushButton("Button one"), 0, 0)
grid.addWidget(QPushButton("Button two"), 0, 1)
grid.addWidget(QPushButton("Button three"), 1, 0)
grid.addWidget(QPushButton("Button four"), 1, 1)

w.show()
sys.exit(app.exec_())
```

Вот результат.



Чтобы использовать метод setPalette(), сначала необходимо определить палитру. Это делается путем создания объекта класса QPalette.

qp = QPalette()

Обратите внимание, что класс QPalette принадлежит модулю QtGui, и вам нужно будет его импортировать, чтобы это работало. После создания объекта QPalette используйте метод setColor(), чтобы передать имя виджета, цвет которого вы хотите изменить, и цвет, который вы хотите установить.

qp.setColor(QPalette.Window, Qt.black)

Это изменит цвет окна на черный. После того, как вы определили свою цветовую схему, используйте функцию setPalette(), чтобы применить палитру к вашему приложению.

app.setPalette(qp)

Это все, что вам нужно сделать, если вы хотите создать базовые темы для своего приложения. PyQt также позволяет вам использовать таблицы стилей для определения внешнего вида ваших виджетов. Если вы знакомы с CSS, вы можете

легко определить расширенные стили для своего приложения с помощью таблиц стилей Qt.

Дополнительные главы: GUI-приложение HR Manager на PyQt5

1. Форма авторизации (SQLite, таблица Admins)

Авторизация – это начальный экран, где пользователь вводит логин и пароль. В PyQt5 форма авторизации обычно реализуется как диалог (например, QDialog или отдельное окно QMainWindow) с полями QLineEdit для имени пользователя и пароля, кнопкой входа (QPushButton) и сообщениями об ошибках (QMessageBox). Поле пароля следует скрывать с помощью setEchoMode(QLineEdit.Password). Для хранения администраторов используется база SQLite с таблицей Admins (колонки: username, password_hash). Подключение к базе устанавливается через QSqlDatabase. Например:

```
db = QSqlDatabase.addDatabase("QSQLITE")
db.setDatabaseName("hrmanager.db")
if not db.open():
    QMessageBox.critical(self, "Ошибка", "Не удалось подключиться к БД")
```

После успешного подключения при нажатии кнопки «Войти» выполняется SQL-запрос SELECT password_hash FROM Admins WHERE username=?, где ? – введённое имя. Это делается с помощью QSqlQuery.

Если пользователь найден (query.next() возвращает True), получаем хэш пароля и сравниваем с введённым (рекомендуется хранить хэш, например через hashlib или QCryptographicHash). В случае совпадения открываем основное окно приложения, иначе показываем QMessageBox.warning с ошибкой. Пример метода авторизации:

```
def login(self):
    username = self.username_edit.text()
    password = self.password_edit.text()
    query = QSqlQuery()
    query.prepare("SELECT password_hash FROM Admins WHERE username = :user")
    query.bindValue(":user", username)
    if query.exec_() and query.next():
        stored_hash = query.value(0)
        if check_password(password, stored_hash):
            self.accept()  # вход успешен
        else:
            QMessageBox.warning(self, "Ошибка", "Неверный пароль")
    else:
        QMessageBox.warning(self, "Ошибка", "Пользователь не найден")
```

Здесь check_password – функция сравнения хэшей. Таким образом организуется форма входа с проверкой данных в SQLite. Ключевые классы: QLineEdit, QPushButton, QMessageBox, и из модуля QtSql – QSqlDatabase, QSqlQuery.

2. Управление справочниками «Отделы» и «Должности»

Справочники отделов и должностей – это отдельные таблицы в БД (например, Departments и Positions), связанные по идентификатору с таблицей сотрудников. Для управления ими создаются формы ввода и таблицы отображения. Для ввода нового элемента можно использовать диалог QDialog с текстовым полем QLineEdit для названия и кнопкой «Добавить». Для отображения списка применяется QTableView с моделью QSqlTableModel или QSqlQueryModel. Например, чтобы вывести таблицу «Departments», создаём модель:

```
model = QSqlTableModel(self, db)
model.setTable("Departments")
model.select()
view = QTableView()
view.setModel(model)
view.resizeColumnsToContents()
```

При добавлении новой записи через INSERT или через модель (model.insertRow() и установка полей), таблица автоматически обновляется. Можно настроить режим редактирования (OnFieldChange), чтобы изменения сразу сохранялись в БД. Благодаря архитектуре Model-View, стандартные классы QSqlTableModel и QTableView упрощают работу: они автоматически обрабатывают загрузку данных и обновление базы.

В качестве теоретической основы можно отметить, что QtSql разделён на слои: QSqlDatabase – для подключения, QSqlQuery – для выполнения SQL-запросов, а модели (QSqlTableModel, QSqlRelationalTableModel) – для отображения и редактирования данных через виджеты.

3. Форма добавления сотрудника (все поля, фото/документы, base64)

Форма добавления сотрудника содержит поля всех сведений (ФИО, дата рождения, отдел, должность и пр.), а также загрузку файла фотографии и дополнительных документов. Поля формы можно расположить с помощью QFormLayout или QGridLayout. Для выбора отдела и должности используются выпадающие списки QComboBox, заполненные запросами к таблицам Departments и Positions. Загрузка файла фото и документов производится через QFileDialog.getOpenFileName. Например:

```
photo_path, _ = QFileDialog.getOpenFileName(self, "Выбрать фото", "", "Images (*.png
*.jpg)")
if photo_path:
    with open(photo_path, "rb") as f:
        photo_data = base64.b64encode(f.read()).decode('utf-8')
```

Полученный в photo_data base64-код изображения сохраняется в текстовое поле таблицы (например, столбец photo в формате TEXT). При выборе документа аналогично читаем файл в байты и кодируем. Для хранения в БД удобно использовать TEXT или BLOB. При выводе данных фото из базы переводим из base64 обратно и загружаем в QPixmap:

```
img_data = base64.b64decode(employee_record.photo)
pixmap = QPixmap()
```

```
pixmap.loadFromData(img_data)
photo_label.setPixmap(pixmap)
```

Таким образом, фотографии и файлы сохраняются «в тексте» и легко извлекаются. Концептуально в этой главе объясняются классы QComboBox, QFileDialog, а также работа с модулями Python base64 и sqlite3. Кодирование изображения в строку и обратно выполняется модулем base64.

Привязка к отделу/должности задаётся через внешние ключи или ID, полученные из соответствующих таблиц. В итоге при нажатии «Сохранить» в БД выполняется INSERT со всеми полями, включая закодированные файлы.

4. Отображение списка сотрудников, фильтрация и поиск

Для просмотра сотрудников используется таблица employees в базе, отображаемая виджетом QTableView. Моделью данных служит QSqlTableModel (или QSqlQueryModel для произвольного запроса). Пример создания модели и вида (основано на примере из Real Python):

```
model = QSqlTableModel(self, db)
model.setTable("employees")
model.setEditStrategy(QSqlTableModel.OnFieldChange)
model.select()
view = QTableView()
view.setModel(model)
view.resizeColumnsToContents()
```

Здесь таблица автоматически загружает все записи; редактирование в ячейке сразу сохраняет изменения в БД благодаря OnFieldChange. Для фильтрации и поиска применяют QSortFilterProxyModel, который стоит над исходной моделью. Тогда при вводе текста поиска мы вызываем, например, proxy.setFilterFixedString(text) и proxy.setFilterKeyColumn(col), и виджеты обновляются без изменения данных в базе.

Таким образом, пользователь может искать сотрудников по имени, отделу и т.д., а данные подгружаются по модели. Ключевым преимуществом архитектуры «модель-представление» (Model-View) является автоматизация отображения и изменения данных: не нужно вручную перерисовывать таблицу или выполнять SQL-запросы после каждого действия – модель позаботится об этом.

5. Удаление сотрудника (подтверждение, проверка пароля)

Удаление сотрудника требует дополнительной осторожности. Логика такова: пользователь выбирает сотрудника по критериям (например, ищет по ФИО через тот же виджет или отдельное поле). Если найдено несколько совпадений, можно показать пользователю список (например, QTableWidget или диалог со списком QListWidget), чтобы он выбрал конкретную запись. После выбора запрашиваем подтверждение удаления. Диалог подтверждения выполняется с помощью QMessageBox.question, например:

```
reply = QMessageBox.question(self, "Удаление",
    "Удалить выбранного сотрудника?",
    QMessageBox.Yes | QMessageBox.No, QMessageBox.No)
```

```
if reply == QMessageBox.Yes:
    # проверяем пароль и удаляем
```

Далее появляется запрос на ввод пароля администратора. Для ввода пароля можно использовать QInputDialog.getText с флагом QLineEdit.Password (символы скрыты)[8]:

```
password, ok = QInputDialog.getText(self, "Пароль администратора",
    "Введите пароль администратора:", QLineEdit.Password)
```

Если пользователь ввёл правильный пароль (сравниваем с хранилищем пароля из таблицы Admins), выполняем SQL-запрос DELETE FROM employees WHERE id = ?. При этом полезно использовать подготовленные запросы (.prepare + .bindValue) во избежание SQL-инъекций. При удалении также можно вывести информацию о том, какую именно запись удалили. Таким образом, процесс удаления сотрудника включает выбор записи, подтверждение через QMessageBox, повторную аутентификацию через QInputDialog и выполнение DELETE в БД.

6. Экспорт данных в Excel, резервное копирование и создание Word-анкеты

Экспорт в Excel. Для выгрузки таблицы сотрудников в файл Excel удобно использовать библиотеку pandas: выполняем SQL-запрос через pd.read_sql(query, connection) и сохраняем DataFrame в файл .xlsx методом to_excel. Например:

```
import pandas as pd
import sqlite3
conn = sqlite3.connect("hrmanager.db")
df = pd.read_sql("SELECT * FROM employees", conn)
df.to_excel("employees.xlsx", index=False)
```

Это автоматически записывает все записи в Excel. Альтернативно можно использовать openpyxl или xlsxwriter напрямую, но pandas упрощает задачу.

Резервное копирование БД. Автоматическое резервное копирование базы SQLite можно реализовать, просто копируя файл базы в другую директорию или с новым именем. Например, при помощи модуля shutil и текущей даты-времени:

```
import shutil, datetime
ts = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
shutil.copyfile("hrmanager.db", f"backup_{ts}.db")
```

Также существует API sqlite3.Connection.backup(), который позволяет копировать базу, пока приложение работает[11]. В простейшем случае достаточно shutil.copyfile, что быстро создаст полный дубль базы[12].

Создание Word-документа (анкета). Для формата Word (.docx) используется библиотека python-docx. Создаём документ и наполняем его данными сотрудника. Пример:

```
from docx import Document
doc = Document()
doc.add_heading(f"Анкета сотрудника: {surname} {name}", level=1)
doc.add_paragraph(f"ФИО: {surname} {name} {patronymic}")
doc.add_paragraph(f"Дата рождения: {dob}")
doc.add_paragraph(f"Отдел: {department}, Должность: {position}")
```

```
# При желании можно добавить таблицу, фото и т.д.
doc.save(f"{surname}_{name}.docx")
```

С python-docx легко добавлять заголовки (add_heading), абзацы (add_paragraph), таблицы и изображения[13][14]. Библиотека автоматически упаковывает всё в правильный формат DOCX. Таким образом формируется персональная анкета сотрудника с нужными данными.

7. Генерация логинов и паролей для сотрудников

Для новых сотрудников приложение может автоматически генерировать учетные данные. Логин, например, можно строить как комбинацию имени и фамилии (или использовать номер карточки). Пароль стоит генерировать случайным образом. В Python 3.6+ рекомендуется использовать модуль secrets для криптографически стойкой генерации. Например, чтобы получить случайный пароль из букв и цифр:

```
import string, secrets
alphabet = string.ascii_letters + string.digits
password = ''.join(secrets.choice(alphabet) for i in range(8))
```

Это даст безопасный 8-символьный пароль. Можно добавить условия (минимум заглавных, цифр) как показано в документации. Затем сгенерированные логин и пароль сохраняются в базе (пароль – в хэшированном виде!). Модуль secrets обеспечивает равномерное распределение символов и защищённость от простых угадываний[15].

8. Сборка проекта в .exe с помощью PyInstaller

Чтобы распространить GUI-приложение, его компилируют в .exe. Сначала устанавливают PyInstaller: pip install pyinstaller. Затем выполняют команду из командной строки в каталоге проекта. Например:

pyinstaller --onefile --windowed --icon=app.ico main.py

Здесь флаг --onefile собирает всё в один исполняемый файл, а --windowed (или --noconsole) убирает консольное окно для GUI-приложения. Опция --icon добавляет иконку EXE. Если используются дополнительные файлы (UI-шаблоны, базы, изображения), их подключают флагом --add-data (с указанием пути).

PyInstaller создаст .spec файл, где можно тонко настроить сборку. После выполнения команды в папке dist появится готовый .exe с указанным именем. В целом упаковка PyQt-приложения сводится к установке PyInstaller и простому вызову команд сборки, при необходимости уточняя параметры (режим окна, добавить данные и т.д.).