# Разработка приложений с графическим интерфейсом с помощью PyQt5

Часто при решении практических задач Data Mining требуется в интерактивном режиме визуализировать данные или предоставить пользователю элементы контроля. В Python есть возможность создавать не только консольные скрипты, но и приложения с графическим интерфейсом пользователя. Есть разные библиотеки для решения этой задачи. Мы в данной практике познакомимся с одной из самых развитых, популярных и хорошо документированных библиотек – PyQt5.

Вот некоторые задачи, где может потребоваться создание приложений с графическим интерфейсом:

* Создание простой программы для задания настроек и генерации конфигурационных файлов для других более сложных программ,
* Создание программы для формирования обучающего множества, например, программы подготовки фрагментов для обучения нейросети в задачах машинного зрения,
* Создание интерактивных графиков и диаграмм, параметры которых могут задаваться с формы приложения с помощью окон ввода, кнопок, ползунков и т.д.,
* Создание программы диагностики журналов выполнения компьютерных программ с различными графиками и диаграммами.

## Библиотека PyQt5

Из Википедии: PyQt — набор «привязок» графического фреймворка Qt для языка программирования Python, выполненный в виде расширения Python.

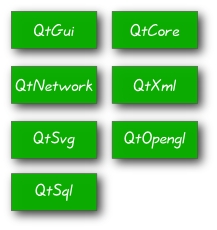
PyQt разработан британской компанией Riverbank Computing. PyQt работает на всех платформах, поддерживаемых Qt: Linux и другие UNIX-подобные ОС, Mac OS X и Windows. Существует 2 версии: PyQt5, поддерживающий Qt 5, и PyQt4, поддерживающий Qt 4. PyQt распространяется под лицензиями GPL (2 и 3 версии) и коммерческой.

PyQt практически полностью реализует возможности Qt. А это более 600 классов, более 6000 функций и методов, включая:

* Существующий набор виджетов графического интерфейса;
* стили виджетов;
* доступ к базам данных с помощью SQL (ODBC, MySQL, PostgreSQL, Oracle);
* QScintilla, основанный на Scintilla виджет текстового редактора;
* поддержку интернационализации (i18n);
* парсер XML;
* поддержку SVG;
* интеграцию с WebKit, движком рендеринга HTML;
* поддержку воспроизведения видео и аудио.

PyQt также включает в себя Qt Designer (Qt Creator) — дизайнер графического интерфейса пользователя. Программа pyuic генерирует Python код из файлов, созданных в Qt Designer. Это делает PyQt очень полезным инструментом для быстрого прототипирования. Кроме того, можно добавлять новые графические элементы управления, написанные на Python, в Qt Designer.

Библиотека содержит большое количество классов, которые распределены по разным модулям:



Модуль QtCore содержит ядро не-gui функциональности. Этот модуль используется для работы со временем, файлами и папками, различными типами данных, потоками, адресами URL, mime типами, потоками процессов. Модуль QtGui содержит графические компоненты и связанные классы. Сюда включены, например, кнопки, окна, строки состояния, панели инструментов, полосы прокрутки, изображения (bitmap), цвета, шрифты и др. Модуль QtNetwork содержит классы для сетевого программирования. Эти классы позволяют писать TCP/IP и UDP клиенты и серверы. Они делают сетевое программирование легче и более доступным. Модуль QtXml содержит классы для работы с xml файлами. Он предоставляет реализации API SAX и DOM. Модуль QtSvg предоставляет классы для отображения содержимого SVG файлов. Масштабируемая векторная графика (SVG) – это язык описания двумерной графики и графических приложений на языке XML. Модуль QtOpenGL используется для построения 3D и 2D графики с помощью библиотеки OpenGL. Модуль дает возможность бесшовной интеграции библиотек QtGui и OpenGL. Модуль QtSql содержит классы для работы с базами данных.

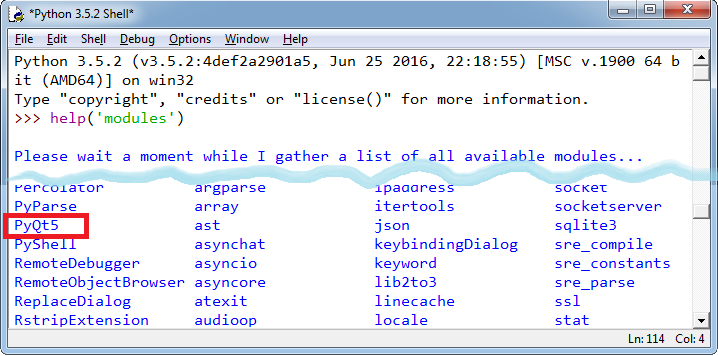
## Основы работы с библиотекой PyQt5

Для начала требуется установить библиотеку. При наличии интернет это делается как обычно через выполнение команды pip install:

pip install pyqt5

После завершения установки можно проверить наличие установленной библиотеки, выполнив в консоли Python команду:

help(‘modules’)



### Пример простой программы

В прилагаемом документе (pyqt5\_tutorial.pdf) приведены краткие основы по созданию главной формы приложения, задания положения окна, размеров и заголовка, добавления всплывающих подсказок, добавления кнопок и обработчиков нажатий на них и т.д.

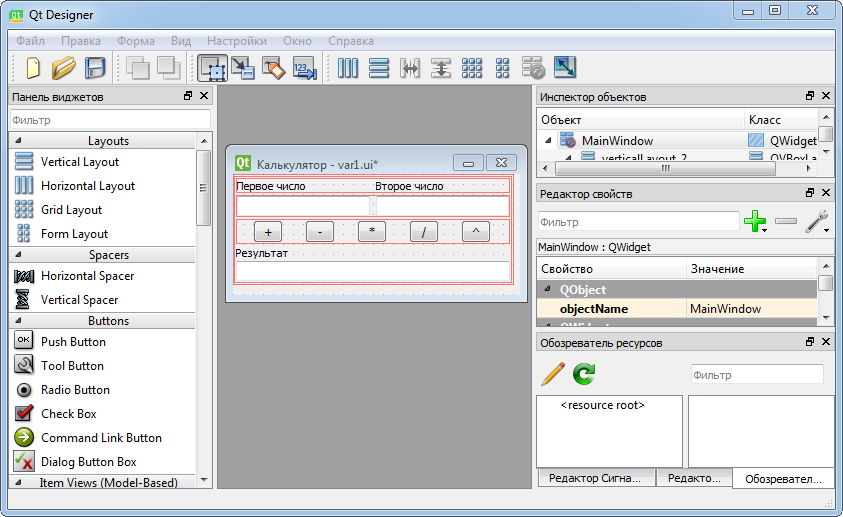
Более полный справочник находится по данному адресу в интернете: <http://zetcode.com/gui/pyqt5/>.

Далее в методичке будут рассмотрены только расширенные возможности, которых нет в приведённой справке, но которые потребуются при выполнении практики.

### Создание графического интерфейса с помощью QtDesigner

QtDesigner – это удобный редактор графического интерфейса пользователя в Qt. Вместо того, чтобы вручную создавать компоненты формы и задавать их свойства в коде, можно создать интерфейс в редакторе QtDesigner, задать все необходимые свойства, а затем подключить в Python-скрипт созданный файл интерфейса.

Вот как выглядит главное окно QtDesigner:



В левой части экрана расположены различные компоненты интерфейса: контейнеры, кнопки, окна ввода и т.п. В правой части перечислены компоненты и их свойства. В центральной части расположена сама форма приложения, на которой перетаскиванием располагаются компоненты интерфейса.

Освоиться с созданием интерфейса в QtDesigner не составит труда. При необходимости проверить созданную в дизайнере форму можно с помощью команды «Предпросмотр» (Ctrl+R).

Созданный графический интерфейс сохраняется в файле с расширением .ui, который представляет собой обычный xml-файл. Этот файл будет подключаться к скрипту на Python для создания приложения.

Для использования интерфейса, созданного в QtDesigner, нам потребуется модуль uic из библиотеки PyQt5 (uic = ui compiler). Чтобы его подключить, добавьте строчку в начале скрипта вашего приложения:

from PyQt5 import uic

Этот модуль позволяет преобразовать (скомпилировать) ui-файл с описанием интерфейса в формат, понятный интерпретатору Python.

Затем в классе окна вашего приложения (допустим, это MainWindow) в конструкторе добавьте код:

uic.loadUi('path\_to.ui', self)

Этот код позволяет использовать интерфейс, созданный в QtDesigner, вместе со всеми компонентами, объявленными там. path\_to.ui – путь до ui-файла с описанием интерфейса, который вы создали в дизайнере.

Вот как целиком может выглядеть код окна приложения, включая конструктор и функцию foo:

class MainWindow(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

uic.loadUi('path\_to.ui', self)

self.slider.valueChanged.connect(lambda x: self.foo(x))

def foo(self, value):

# обработчик изменения значения слайдера

В этом примере класс MainWindow наследует от базового класса QWidget. Функция \_\_init\_\_(self) является конструктором. С помощью строки super().\_\_init\_\_() мы выполняем инициализацию базового класса (QWidget).

В строке self.slider.valueChanged.connect(lambda x: self.foo(x)) мы подключаем обработчик сигнала изменения значения слайдера. При изменении слайдера будет вызываться функция self.foo, в которую будет передаваться параметр – текущее значение слайдера. Подробнее о механизме сигналов и слотов в PyQt5 будет сказано ниже.

Обратите внимание, что в конструкторе мы можем обращаться к слайдеру self.slider, который был создан в QtDesigner. Префикс self. означает, что объект или функция являются полем или методом класса MainWindow.

При создании графического интерфейса в QtDesigner старайтесь давать компонентам осмысленные имена, чтобы сделать код более ясным и читаемым. Также постарайтесь использовать контейнеры Layouts для компонентов (Vertical Layout, Horizontal Layout, Grid Layout) для того, чтобы размеры компонентов стали «резиновыми» и корректно подгонялись под размер формы при их изменении. При проверке практики этому требованию будет уделяться внимание.

### Использование графиков matplotlib в приложениях PyQt5

В PyQt5 существуют разные способы рисования графиков, например, matplotlib, QtCharts. Мы будем пользоваться первым вариантом, как уже знакомым нам способом построения мощных разнообразных графиков в Python.

В библиотеке matplotlib имеется механизм рисования графиков как виджетов PyQt5. Для этого нужно в начале вашего скрипта прописать строчки:

from matplotlib.backends.backend\_qt5agg import FigureCanvasQTAgg as FigureCanvas

import matplotlib.pyplot as plt

Затем в конструкторе пропишите код:

self.figure = plt.figure()

self.canvas = FigureCanvas(self.figure)

self.verticalLayout.insertWidget(0, self.canvas)

В первой строке создаётся объект Рисунок (figure), который мы будем использовать для рисования графиков. Во второй строке создаётся виджет канвы, на которой будет выполняться отрисовка графиков matplotlib. По сути, этот виджет и является связкой библиотек matplotlib и PyQt5. В третьей строке мы добавляем канву на форму внутрь контейнера self.verticalLayout.

Теперь необходимо построить какой-нибудь график. Для этого воспользуемся следующим кодом:

ax = self.figure.add\_subplot(111)

ax.hold(False) # discards the old graph

ax.plot(data, '-')

self.canvas.draw()

В первой строке создаётся объект, на котором будет рисоваться график. Во второй строке мы разрешаем перерисовку графика (это требуется, если график может меняться). В третьей строке выполняется построение графика (пока только в памяти компьютера), а в четвёртой – непосредственно рисование на форме. В первой строке в функции add\_subplot используется странный параметр 111. Расшифруем его смысл. Первая единица – это количество строк в создаваемой сетке графиков, вторая единица – количество столбцов в сетке. Третья единица – индекс графика в сетке (от 1 до числа ячеей в сетке). В данном случае создаётся сетка размера 1×1, и возвращается индекс первой и единственной ячейки.

### Обработка действий мыши и рисование

В PyQt5 имеется возможность обрабатывать события мыши, например, события нажатия на кнопку мыши, движения мыши над компонентом, отпускания кнопки мыши и т.д.

Для обработки событий мыши в классе, унаследованном от QWidget (или другого класса, являющегося производным от QWidget, например, QLabel), необходимо определить функции:

def mousePressEvent(self, event):

def mouseMoveEvent(self, event):

def mouseReleaseEvent(self, event):

Первая функция срабатывает при нажатии кнопки мыши на виджете, вторая – при движении мыши над виджетом, третья – при отпускании кнопки мыши.

**Важное замечание**: для того, чтобы срабатывал вызов функции mouseMoveEvent при движении мыши над виджетом, необходимо в конструкторе виджета вызвать функцию self.setMouseTracking(True).

Объект event, передаваемый в функции параметром, содержит множество методов, но чаще всего используются методы x() и y(), которые возвращают координаты указателя мыши относительно виджета.

### Работа с изображениями в PyQt5

Виджеты PyQt5 могут использоваться для отображения изображений и рисования. Чтобы вывести на экран изображение из файла, часто используют некоторые объекты-контейнеры, например, QLabel. Чтобы использовать класс QLabel в своём скрипте, его нужно подключить в начале файла:

from PyQt5.QtWidgets import QWidget, QApplication, **QLabel**

Для того, чтобы можно было рисовать на поверхности компонента, подключите следующие классы:

from PyQt5.QtGui import QPixmap, QPainter, QColor

Теперь попробуем отобразить картинку из файла на форме приложения:

self.image = QLabel(self)

self.pixmap = QPixmap('image.jpg')

self.image.setPixmap(self.pixmap)

self.verticalLayout.insertWidget(0, self.image)

В первой строке мы определяем, что для хранения картинки будет использоваться виджет класса QLabel. Во второй строке мы создаём объект типа QPixmap из картинки, расположенной в файле image.jpg. В третьей строке виджету картинки присваивается изображение, загруженное из файла, а в четвёртой строке созданный виджет с картинкой размещается в начале контейнера self.verticalLayout.

Помимо простого отображения картинок может потребоваться рисовать на них что-либо. Для этого используется объект класса QPainter.

Например, чтобы нарисовать окружность на картинке, которую вы ранее загрузили из файла, используйте следующий код:

painter = QPainter(self.image.pixmap())

color = QColor(128, 128, 128) # серый цвет в формате RGB

painter.setPen(color)

painter.drawEllipse(10, 10, 5, 5)

self.image.repaint()

Здесь вначале создаётся объект класса QPainter, на котором мы будем рисовать. Он инициализируется значением pixmap(), которое было загружено из файла. Затем создаётся объект класса QColor – цвет, который мы используем для рисования окружности. Этот цвет в 3 строке мы задаём перу, с помощью которого будем выполнять рисование. В 4 строке непосредственно рисуется окружность с радиусом 5 и центром в точке с координатами {10, 10}. В последней строке вызывается метод repaint(), чтобы отобразить на экране изображение с нарисованной поверх него окружностью.

Объект QPainter обладает большим количеством методов рисования. Можно рисовать линии, прямоугольники, окружности, сложные фигуры, задавать цвет и стиль линий и заливки, отображать текст. Предлагается самостоятельно ознакомиться с этими функциями по документации к QPainter.

### Создание собственных сигналов и их обработчиков

В библиотеке PyQt5 большинство компонентов, как-то кнопки, слайдеры, окна ввода и т.д., имеют определённые сигналы, на которые мы можем «повесить» свои обработчики. Например, кнопка QPushButton имеет сигнал clicked, на который можно повесить обработчик события нажатия на кнопку. Список сигналов, поддерживаемый классами PyQt5, можно найти в документации к библиотеке.

Вот как может выглядеть обработчик нажатия на кнопку «+», выполняющую сложение двух аргументов:

self.btn\_plus.clicked.connect(self.plus)

def plus(self):

try:

arg1 = float(self.arg1.text())

arg2 = float(self.arg2.text())

result = arg1 + arg2

self.result.setText(str(result))

except Exception:

self.result.setText('Invalid arguments')

Здесь сигналу clicked кнопки self.btn\_plus с помощью вызова функции connect присваивается обработчик self.plus. self.plus – это метод класса MainWindow, который считывает из окон ввода 2 значения, преобразует их к типу float, выполняет сложение и отображает в окне self.result результат операции. Если что-то пойдёт не так (например, вместо чисел в окна ввода будут введены строки или возникнет арифметическое переполнение), сработает исключение, и в окне результата отобразится текст 'Invalid arguments'.

Но бывают ситуации, когда необходимо определить свои собственные сигналы. Например, мы хотим, чтобы при клике мышью по картинке возникал сигнал с параметрами – координатами курсора. Для этого нужно создать собственный класс, который будет производным от одного из классов-потомков QWidget, например, QLabel, в этом классе определить сигнал, а при клике мышью на объекте этого класса генерировать сигнал, передавая ему координаты курсора. Вот как этот подход может быть реализован в коде (см. ниже). Ключевые места подсвечены жирным шрифтом.

from PyQt5.QtWidgets import QWidget, QApplication, **QLabel**

from PyQt5.QtCore import QObject, **pyqtSignal**

class MyLabel(QLabel):

**mouse\_pressed = pyqtSignal(int, int)**

def \_\_init\_\_(self, \*args):

super().\_\_init\_\_(\*args)

def mousePressEvent(self, event):

**self.mouse\_pressed.emit(event.x(), event.y())**

class MainWindow(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

uic.loadUi('path\_to.ui', self)

self.image = MyLabel(self)

**self.image.mouse\_pressed.connect(lambda x, y: self.handler(x, y))**

**def handler(self, x, y):**

# обработчик сигнала

pass

В первых двух строках мы подключаем необходимые классы. QLabel нам нужен для того, чтобы на его основе реализовать собственный класс MyLabel. Также во второй строке подключается pyqtSignal, необходимый для создания собственных сигналов.

Затем начинается объявление собственного класса MyLabel, который является производным по отношению к QLabel. И сразу после этого создаётся сигнал mouse\_pressed с помощью кода mouse\_pressed = pyqtSignal(int, int). Обратите внимание, что сигнал mouse\_pressed объявляется вне какого-либо метода класса.

Сигнал mouse\_pressed должен срабатывать, когда пользователь щёлкнул мышью по изображению. Поэтому в данном примере в классе MyLabel объявлена функция mousePressEvent, внутри которой с помощью функции emit генерируется сигнал. В качестве параметров в него передаются координаты курсора мыши.

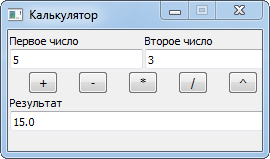
Созданный сигнал нужно как-то обработать. Для этого в Qt (и в PyQt5) используется механизм слотов. Слот – это обработчик, который связывается с сигналом и вызывается, когда получен соответствующий сигнал. С одним сигналом можно связывать несколько слотов. В приведённом выше примере мы с помощью вызова функции connect привязываем к сигналу mouse\_pressed обработчик self.handler, который является методом класса MainWindow. Обработчику передаются параметры x и y – координаты курсора.

## Варианты заданий

В качестве основы вашего приложения рекомендуется использовать каркас приложения, созданного по учебнику в pyqt5\_tutorial.pdf. Затем вы можете дорабатывать этот каркас под себя, редактируя интерфейс, добавляя обработчики кнопок, слайдеров и событий мыши.

Прочитайте подсказки для всех вариантов, так как они могут подсказать правильный путь решения при выполнении вашего варианта.

### Вариант 1:

Написать простой калькулятор, выполняющий сложение, вычитание, умножение, деление и возведение в степень для двух введённых чисел.

**Подсказки:**

Чтобы не писать множество разных функций для каждой из операций, сделайте одну функцию, в которую передавайте параметр – тип операции. Для типа операции создайте class Enum следующим образом:

class Operation(Enum):

plus = 1

minus = 2

mul = 3

div = 4

power = 5

Внутри функции-обработчика нажатий на кнопки операций включите блок try-except, чтобы гарантировать корректную работу программы при некорректных входных данных:

try:

arg1 = float(self.arg1.text())

arg2 = float(self.arg2.text())

result = 0

# код вычисления result

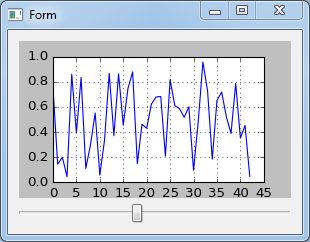
self.result.setText(str(result))

except Exception:

self.result.setText('Invalid arguments')

Если в блоке try произошла ошибка, будет выполнен код из блока except.

### Вариант 2:

Написать приложение, выполняющее построение синусоиды. Период синусоиды должен задаваться с помощью слайдера (QSlider).

**Подсказки:**

См. методические указания выше. Там дана вся необходима информация.

### Вариант 3:

Написать приложение, выполняющее просмотр изображений из указанного каталога. Должна быть возможность переходить от изображения к изображению с помощью кнопок «Влево» и «Вправо».

**Подсказки:**

Заранее подготовьте каталог с изображениями. В коде программы можете создать список файлов изображений, которые будет отображать программа.

Для загрузки изображений используйте функцию

def set\_pixmap(self):

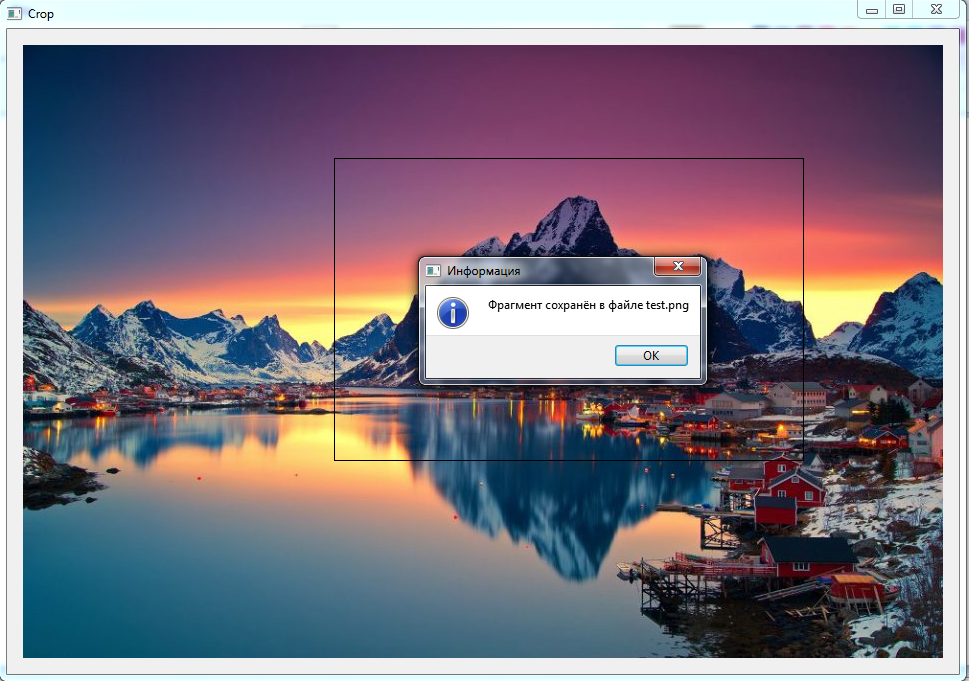
pixmap = QPixmap(self.images[self.index])

self.imageContainer.setPixmap(pixmap.scaled(800, 600))

Здесь self.images – это заранее подготовленный список путей к файлам изображений. self.index – индекс текущего изображения. self.imageContainer – объект класса QLabel, на который будет выводиться изображение.

pixmap.scaled(800, 600) – приведение изображения к размеру 800×600.

### Вариант 4:

Написать приложение, позволяющее открыть изображение, мышкой выделить фрагмент изображения и сохранить его в файл.

**Подсказки:**

Создайте собственный класс MyLabel, унаследованный от QLabel. В этом классе объявите сигнал image\_croped, имеющий 4 целочисленных параметра, означающих координаты точки начала выделенного фрагмента и конца выделенного фрагмента:

class MyLabel(QLabel):

**image\_cropped = pyqtSignal(int, int, int, int)**

def \_\_init\_\_(self, \*args):

super().\_\_init\_\_(\*args)

В обработчике события нажатия на кнопку мыши сохраняйте в переменных self.start\_x и self.start\_y координаты начала выделенного фрагмента.

В обработчике события отпускания кнопки мыши генерируйте сигнал image\_croped с помощью функции emit, передавая ему соответствующие координаты.

В конструкторе класса MainWindow привяжите сигнал image\_croped к обработчику, который будет сохранять изображение в заданный файл. Обработчик может выглядеть так:

def crop\_image(self, x1, y1, x2, y2):

cropped = self.pixmap.copy(x1, y1, x2 - x1, y2 - y1)

cropped.save('test.png')

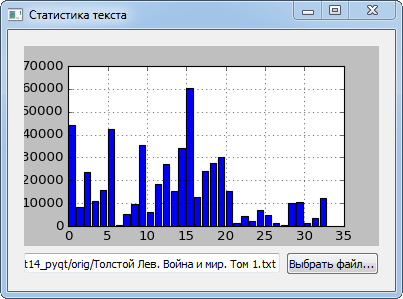
QMessageBox.information(self, 'Информация', 'Фрагмент сохранён в файле test.png')

Здесь self.pixmap – это изображение из файла, которое устанавливается в ваш класс MyLabel.

Чтобы использовать QMessageBox, подключите в начале скрипта соответствующий класс:

from PyQt5.QtWidgets import QWidget, QApplication, QLabel, **QMessageBox**

### Вариант 5:

Написать приложение, которое для указанного текстового файла считает, как часто встречается каждая буква русского алфавита, а затем отображает данные на экране в виде гистограммы.

**Подсказки:**

Чтобы отобразить диалог открытия файла, используйте код:

filename = QFileDialog.getOpenFileName(self, 'Выберите текстовый файл', 'D:\\', 'Текстовые файлы (\*.txt)')[0]

Не забудьте подключить в начале скрипта класс QFileDialog из PyQt5.QtWidgets.

Следующий код может использоваться для получения содержимого текстового файла:

with open(filename, 'r') as txt\_file:

data = txt\_file.read().replace('\n', '')

Ниже показано, как можно выполнить подсчёт количества букв в тексте:

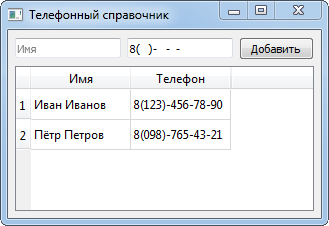
letters = list('абвгдеёжзийклмнопрстуфхцчшщъыьэюя')

hist = []

for letter in letters:

hist.append(data.count(letter))

### Вариант 6:

Написать приложение – телефонный справочник. Приложение должно содержать окна ввода имени и телефона, кнопку «Добавить» и таблицу, в которую будут вноситься данные.

**Подсказки:**

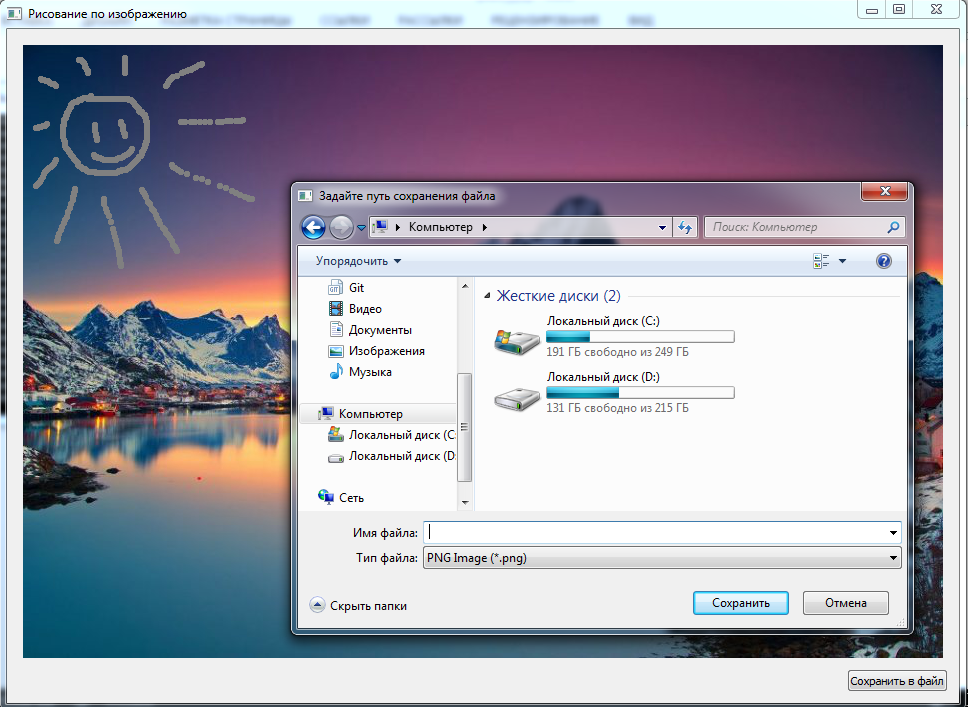
При создании формы в QtDesigner для окна ввода имени задайте свойство placeholderName и посмотрите, как оно работает. Для окна ввода имени задайте свойство inputMask равным 8(999)-999-99-99 и посмотрите, как оно работает. Для представления таблицы используйте компонент QTableWidget.

Для добавления данных в таблицу вам нужно будет реализовать обработчик кнопки «Добавить». При добавлении нужно будет добавить в таблицу новую строку (функция setRowCount). Для имени и номера телефона нужно будет создать экземпляры класса QTableWidgetItem. Не забудьте подключить класс QTableWidgetItem в начале скрипта кодом:

from PyQt5.QtWidgets import QWidget, QApplication, **QTableWidgetItem**

Затем устанавливайте значения в ячейках таблицы с помощью функции setItem, параметрами которой являются индекс строки таблицы, индекс столбца и ранее созданный экземпляр класса QTableWidgetItem.

### Вариант 7:

Написать приложение, которое позволяет открыть изображение, рисовать на нем линии с помощью мыши и сохранить получившийся рисунок в файл.

**Подсказки:**

Создайте класс MyLabel, наследующий от класса QLabel. В обработчике mousePressEvent устанавливайте флаг нажатия на кнопку мыши в True, в функции mouseReleaseEvent сбрасывайте флаг в 0. В функции mouseMoveEvent проверяйте, выставлен ли флаг, и если да, то используя объект QPainter рисуйте на изображении. Не забудьте после рисования вызвать функцию repaint.

Реализуйте сохранения итогового изображения в файл с возможностью выбора пути сохранения файла. Для этого используйте класс QFileDialog и приведённый ниже код:

filename = QFileDialog.getSaveFileName(self, 'Задайте путь сохранения файла', 'D:\\', 'PNG Image (\*.png)')[0]

self.image.pixmap().save(filename)

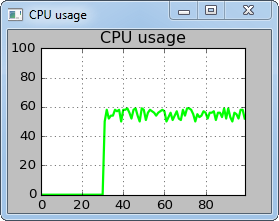
После сохранения изображения отобразите диалоговое окно с помощью приведённого кода:

QMessageBox.information(self, 'Информация', 'Изображение сохранено в файле ' + filename)

Не забудьте в начале файла подключить необходимые модули:

from PyQt5.QtWidgets import QWidget, QApplication, **QLabel**, **QFileDialog**, **QMessageBox**

### Вариант 8:

Напишите приложение, которое каждую секунду отображает на графике загрузку процессора. В качестве значений загрузки процессора используйте генератор случайных чисел.

**Подсказки**:

Для того, чтобы имитировать посекундное измерение загрузки процессора, используйте объект класса QTimer, к сигналу которого нужно подключить функцию-обработчик, и запустить таймер:

from PyQt5.QtCore import QTimer

…

self.timer = QTimer(self)

self.timer.timeout.connect(self.plot)

self.timer.start(1000) # таймер срабатывает каждую секунду

Для хранения значений загрузки процессора используйте очередь. Для этого в начале файла добавьте код:

import collections

Затем создайте очередь и обнулите её:

data\_len = 100

self.data = collections.deque(maxlen=data\_len)

for i in range(data\_len):

self.data.append(0)

Каждую секунду по таймеру добавляйте в хвост очереди очередное случайное значение в заданном диапазоне, например, от 50 до 60. Для этого можете воспользоваться функцией randint из библиотеки numpy.random.

Преобразовать очередь в массив можно с помощью кода:

list(self.data)

Каждому студенту необходимо выполнить по 2 задания из предложенных выше. Задания для каждого варианта приведены в таблице. Номер варианта соответствует номеру в таблице успеваемости.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| вариант | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  |  |
| 13 |  |  |  |  |  |  |  |  |
| 14 |  |  |  |  |  |  |  |  |
| 15 |  |  |  |  |  |  |  |  |