Introduction to PyQt5

This is an introductory PyQt5 tutorial. The purpose of this tutorial is to get you started with the PyQt5 toolkit. The tutorial has been created and tested on Linux. PyQt4 tutorial covers PyQt4, which is a blending of the Python language (2.x and 3.x) to the Qt4 library.

About PyQt5

PyQt5 is a set of Python bindings for Qt5 application framework from Digia. It is available for the Python 2.x and 3.x. This tutorial uses Python 3. Qt library is one of the most powerful GUI libraries. The official home site for PyQt5 is www.riverbankcomputing.co.uk/news. PyQt5 is developed by Riverbank Computing.

PyQt5 is implemented as a set of Python modules. It has over 620 classes and 6000 functions and methods. It is a multiplatform toolkit which runs on all major operating systems, including Unix, Windows, and Mac OS. PyQt5 is dual licensed. Developers can choose between a GPL and a commercial license.

PyQt5's classes are divided into several modules, including the following:

- QtCore
- QtGui
- QtWidgets
- QtMultimedia
- QtBluetooth
- QtNetwork
- QtPositioning
- Enginio
- QtWebSockets
- QtWebKit
- QtWebKitWidgets
- QtXml
- QtSvg
- QtSql
- QtTest

The *QtCore* module contains the core non GUI functionality. This module is used for working with time, files and directories, various data types, streams, URLs, mime types, threads or processes. The *QtGui* contains classes for windowing system integration, event handling, 2D graphics, basic imaging, fonts and text. The *QtWidgets* module contains classes that provide a set of UI elements to create classic desktop-style user interfaces.
The *QtMultimedia* contains classes to handle multimedia content and APIs to access camera and radio functionality. The *QtBluetooth* module contains classes to scan for devices and connect and interact with them.
The *QtNetwork* module contains the classes for network programming. These classes facilitate the coding of TCP/IP and UDP clients and servers by making the network programming easier and more portable.
The *QtPositioning* contains classes to determine a position by using a variety of possible sources, including satellite, Wi-Fi, or a text file.
The *Enginio* module implements the client-side library for accessing the Qt Cloud Services Managed Application Runtime. The *QtWebSockets* module contains classes that implement the WebSocket protocol.
The *QtWebKit* contains classes for a web browser implementation based on the WebKit2 library. The *QtWebKitWidgets* contains classes for a WebKit1 based implementation of a web browser for use in *QtWidgets* based applications. The *QtXml* contains classes for working with XML files. This module provides implementation for both SAX and DOM APIs.
The *QtSvg* module provides classes for displaying the contents of SVG files. Scalable Vector Graphics (SVG) is a language for describing two-dimensional graphics and graphical applications in XML. The *QtSql* module provides classes for working with databases. The *QtTest* contains functions that enable unit testing of PyQt5 applications.

PyQt4 and PyQt5 differences

The PyQt5 is not backward compatible with PyQt4; there are several significant changes in PyQt5. However, it is not very difficult to adjust older code to the new library. The differences are, among others, the following:

- Python modules have been reorganized. Some modules have been dropped (*QtScript*), others have been split into submodules (*QtGui*, *QtWebKit*).
- New modules have been introduced, including *QtBluetooth*, *QtPositioning*, or *Enginio*.
- PyQt5 supports only the new-style signal and slots handlig. The calls to *SIGNAL()* or *SLOT()* are no longer supported.
- PyQt5 does not support any parts of the Qt API that are marked as deprecated or obsolete in Qt v5.0.

Python

Python is a general-purpose, dynamic, object-oriented programming language. The design purpose of the Python language emphasizes programmer productivity and code readability. Python was initially developed by Guido van Rossum. It was first released in 1991. Python was inspired by ABC, Haskell, Java, Lisp, Icon, and Perl programming languages. Python is a high-level, general purpose, multiplatform, interpreted language. Python is a minimalistic language. One of its most visible features is that it does not use semicolons nor brackets. It uses indentation instead. There are two main branches of Python currently: Python 2.x and Python 3.x. Python 3.x breaks backward compatibility with previous releases of Python. It was created to correct some design flaws of the language and make the language more clean. The most recent version of Python 2.x is 2.7.9, and of

Python 3.x is 3.4.2. Python is maintained by a large group of volunteers worldwide. Python is open source software. Python is an ideal start for those who want to learn programming.

This tutorial uses Python 3.x version.

Python programming language supports several programming styles. It does not force a programmer to a specific paradigm. Python supports object-oriented and procedural programming. There is also a limited support for functional programming.

The official web site for the Python programming language is python.org

Perl, Python, and Ruby are widely used scripting languages. They share many similarities and they are close competitors.

Python toolkits

For creating graphical user interfaces, Python programmers can choose among three decent options: PyQt4, PyGTK, and wxPython.

This chapter was an introduction to PyQt4 toolkit.

First programs in PyQt5

In this part of the PyQt5 tutorial we learn some basic functionality.

Simple example

This is a simple example showing a small window. Yet we can do a lot with this window. We can resize it, maximise it or minimise it. This requires a lot of coding. Someone already coded this functionality. Because it repeats in most applications, there is no need to code it over again. PyQt5 is a high level toolkit. If we would code in a lower level toolkit, the following code example could easily have hundreds of lines.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-
```

```
"""
ZetCode PyQt5 tutorial

In this example, we create a simple
window in PyQt5.

author: Jan Bodnar
website: zetcode.com
last edited: January 2015
"""

import sys
from PyQt5.QtWidgets import QApplication, QWidget


if __name__ == '__main__':

    app = QApplication(sys.argv)

    w = QWidget()
    w.resize(250, 150)
    w.move(300, 300)

    w.setWindowTitle('Simple')

    w.show()



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

The above code example shows a small window on the screen.

```
import sys

from PyQt5.QtWidgets import QApplication, QWidget
```

Here we provide the necessary imports. The basic widgets are located in `PyQt5.QtWidgets` module.

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

Every PyQt5 application must create an application object.

The `sys.argv` parameter is a list of arguments from a command line. Python scripts can be run from the shell. It is a way how we can control the startup of our scripts.

```
w = QWidget()
```

The `QWidget` widget is the base class of all user interface objects in PyQt5.

We provide the default constructor for `QWidget`. The default constructor has no parent. A widget with no parent is called a window.

```
w.resize(250, 150)
```

The `resize()` method resizes the widget. It is 250px wide and 150px high.

```
w.move(300, 300)
```

The `move()` method moves the widget to a position on the screen at x=300, y=300 coordinates.

```
w.setWindowTitle('Simple')
```

Here we set the title for our window. The title is shown in the titlebar.

```
w.show()
```

The *show*() method displays the widget on the screen. A widget is first created in memory and later shown on the screen.

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

Finally, we enter the mainloop of the application. The event handling starts from this point. The mainloop receives events from the window system and dispatches them to the application widgets. The mainloop ends if we call the *exit*() method or the main widget is destroyed. The *sys.exit*()method ensures a clean exit. The environment will be informed how the application ended.

The *exec_*() method has an underscore. It is because the *exec* is a Python keyword. And thus, *exec_*()was used instead.

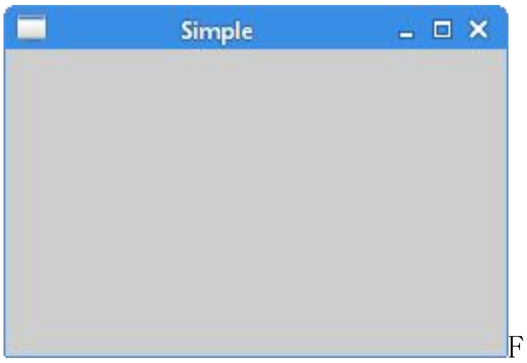


Figure: Simple

An application icon

The application icon is a small image which is usually displayed in the top left corner of the titlebar. In the following example we will s how how we do it in PyQt5. We will also introduce some new methods.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-
```

```
"""
ZetCode PyQt5 tutorial

This example shows an icon
in the titlebar of the window.

author: Jan Bodnar
website: zetcode.com
last edited: January 2015
"""

import sys
from PyQt5.QtWidgets import QApplication, QWidget
from PyQt5.QtGui import QIcon


class Example(QWidget):

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



    def initUI(self):


        self.setGeometry(300, 300, 300, 220)

        self.setWindowTitle('Icon')

        self.setWindowIcon(QIcon('web.png'))


        self.show()



if __name__ == '__main__':


    app = QApplication(sys.argv)

    ex = Example()

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

The previous example was coded in a procedural style. Python programming language supports both procedural and object oriented programming styles. Programming in PyQt5 means programming in OOP.

```
class Example(QWidget):



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

        ...
```

Three important things in object oriented programming are classes, data, and methods. Here we create a new class called *Example*. The *Example* class inherits from the *QWidget* class. This means that we call two constructors: the first one for the *Example* class and the second one for the inherited class. The *super()* method returns the parent object of the *Example* class and we call its constructor. The*__init__()* method is a constructor method in Python language.

```
self.initUI()
```

The creation of the GUI is delegated to the *initUI()* method.

```
self.setGeometry(300, 300, 300, 220)

self.setWindowTitle('Icon')

self.setWindowIcon(QIcon('web.png'))
```

All three methods have been inherited from the *QWidget* class.

The *setGeometry()* does two things: it locates the window on the screen and sets it size. The first two parameters are the x and y positions of the window. The third is the width and the fourth is the height of the window. In fact, it combines the *resize()* and *move()* methods in one method. The last method sets the application icon. To do this, we have created a *QIcon* object.

The *QIcon* receives the path to our icon to be displayed.

```
if __name__ == '__main__':
    app = QApplication(sys.argv)

    ex = Example()

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

The application and example objects are created. The main loop is started.

Figure: Icon

Showing a tooltip

We can provide a balloon help for any of our widgets.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



This example shows a tooltip on

a window and a button.



author: Jan Bodnar

website: zetcode.com
```

```
last edited: January 2015
"""

import sys
from PyQt5.QtWidgets import (QWidget, QToolTip,
    QPushButton, QApplication)
from PyQt5.QtGui import QFont


class Example(QWidget):

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

        self.initUI()


    def initUI(self):

        QToolTip.setFont(QFont('SansSerif', 10))

        self.setToolTip('This is a <b>QWidget</b> widget')
        btn = QPushButton('Button', self)

        btn.setToolTip('This is a <b>QPushButton</b> widget')

        btn.resize(btn.sizeHint())

        btn.move(50, 50)



        self.setGeometry(300, 300, 300, 200)

        self.setWindowTitle('Tooltips')

        self.show()




if __name__ == '__main__':



    app = QApplication(sys.argv)

    ex = Example()

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

In this example, we show a tooltip for two PyQt5 widgets.

```
QToolTip.setFont(QFont('SansSerif', 10))
```

This static method sets a font used to render tooltips. We use a *10px* SansSerif font.

```
self.setToolTip('This is a <b>QWidget</b> widget')
```

To create a tooltip, we call the *setTooltip()* method. We can use rich text formatting.

```
btn = QPushButton('Button', self)

btn.setToolTip('This is a <b>QPushButton</b> widget')
```

We create a push button widget and set a tooltip for it.

```
btn.resize(btn.sizeHint())

btn.move(50, 50)
```

The button is being resized and moved on the window. The *sizeHint()* method gives a recommended size for the button.

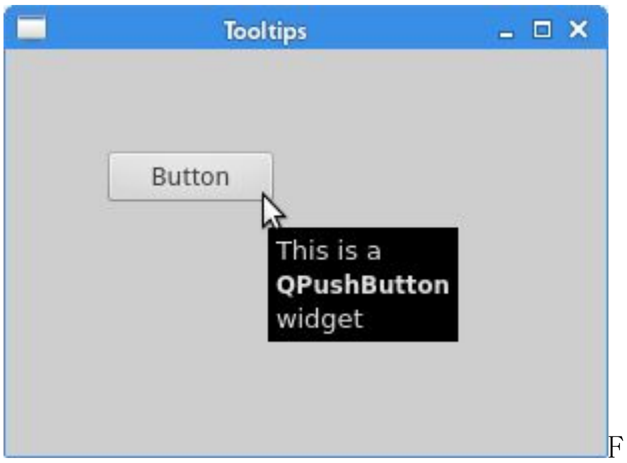


Figure: Tooltips

Closing a window

The obvious way to close a window is to click on the x mark on the titlebar. In the next example, we will show how we can programatically close our window. We will briefly touch signals and slots.

The following is the constructor of a *QPushButton* widget that we use in our example.

```
QPushButton(string text, QWidget parent = None)
```

The *text* parameter is a text that will be displayed on the button.

The *parent* is a widget on which we place our button. In our case it will be a *QWidget*. Widgets of an application form a hierarchy. In this hierarchy, most widgets have their parents. Widgets without parents are toplevel windows.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



This program creates a quit

button. When we press the button,

the application terminates.



author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""



import sys

from PyQt5.QtWidgets import QWidget, QPushButton, QApplication

from PyQt5.QtCore import QCoreApplication





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

        self.initUI()


    def initUI(self):

        qbtn = QPushButton('Quit', self)
        qbtn.clicked.connect(QCoreApplication.instance().quit)
        qbtn.resize(qbtn.sizeHint())
        qbtn.move(50, 50)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Quit button')
        self.show()


if __name__ == '__main__':

    app = QApplication(sys.argv)
    ex = Example()

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

In this example, we create a quit button. Upon clicking on the button, the application terminates.

```
from PyQt5.QtCore import QCoreApplication
```

We need an object from the *QtCore* module.

```
qbtn = QPushButton('Quit', self)
```

We create a push button. The button is an instance of the *QPushButton* class. The first parameter of the constructor is the label of the button. The second parameter is the parent widget. The parent widget is the *Example* widget, which is a *QWidget* by inheritance.

```
qbtn.clicked.connect(QCoreApplication.instance().quit)
```

The event processing system in PyQt5 is built with the signal & slot mechanism. If we click on the button, the signal *clicked* is emitted. The slot can be a Qt slot or any Python callable. The*QCoreApplication* contains the main event loop; it processes and dispatches all events.
The *instance()*method gives us its current instance. Note that *QCoreApplication* is created with the *QApplication*. The clicked signal is connected to the *quit()* method which terminates the application. The communication is done between two objects: the sender and the receiver. The sender is the push button, the receiver is the application object.

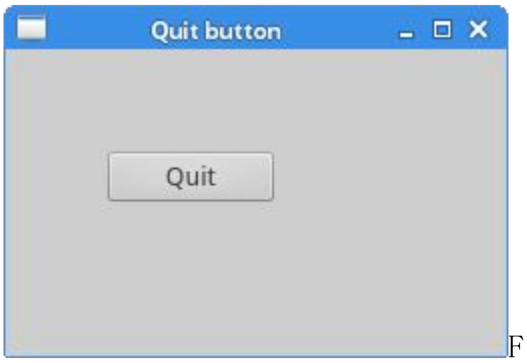


Figure: Quit button

Message Box

By default, if we click on the x button on the titlebar, the *QWidget* is closed.

Sometimes we want to modify this default behaviour. For example, if we have a file opened in an editor to which we did some changes. We show a message box to confirm the action.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



This program shows a confirmation

message box when we click on the close

button of the application window.



author: Jan Bodnar

website: zetcode.com

last edited: January 2015
```

```
"""


import sys

from PyQt5.QtWidgets import QWidget, QMessageBox, QApplication




class Example(QWidget):



    def __init__(self):

        super().__init__()


        self.initUI()




    def initUI(self):



        self.setGeometry(300, 300, 250, 150)

        self.setWindowTitle('Message box')

        self.show()




    def closeEvent(self, event):
        reply = QMessageBox.question(self, 'Message',

            "Are you sure to quit?", QMessageBox.Yes |

            QMessageBox.No, QMessageBox.No)



        if reply == QMessageBox.Yes:

            event.accept()

        else:

            event.ignore()




if __name__ == '__main__':



    app = QApplication(sys.argv)

    ex = Example()

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

If we close a *QWidget*, the *QCloseEvent* is generated. To modify the widget behaviour we need to reimplement the *closeEvent()* event handler.

```
reply = QMessageBox.question(self, 'Message',

    "Are you sure to quit?", QMessageBox.Yes |

    QMessageBox.No, QMessageBox.No)
```

We show a message box with two buttons: Yes and No. The first string appears on the titlebar. The second string is the message text displayed by the dialog.

The third argument specifies the combination of buttons appearing in the dialog. The last parameter is the default button. It is the button which has initially the keyboard focus. The return value is stored in the *reply* variable.

```
if reply == QtGui.QMessageBox.Yes:

    event.accept()

else:

    event.ignore()
```

Here we test the return value. If we click the Yes button, we accept the event which leads to the closure of the widget and to the termination of the application. Otherwise we ignore the close event.

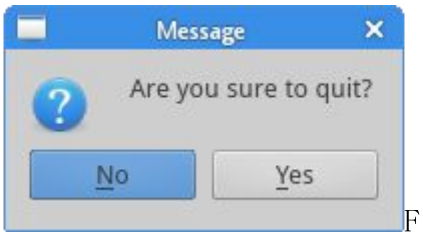


Figure: Message box

Centering window on the screen

The following script shows how we can center a window on the desktop screen.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



This program centers a window

on the screen.
```

```
author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""


import sys

from PyQt5.QtWidgets import QWidget, QDesktopWidget, QApplication




class Example(QWidget):



    def _init_(self):

        super()._init_()



        self.initUI()





    def initUI(self):



        self.resize(250, 150)

        self.center()
        self.setWindowTitle('Center')

        self.show()



    def center(self):


        qr = self.frameGeometry()

        cp = QDesktopWidget().availableGeometry().center()

        qr.moveCenter(cp)

        self.move(qr.topLeft())



if __name__ == '__main__':


    app = QApplication(sys.argv)

    ex = Example()

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

The *QtGui.QDesktopWidget* class provides information about the user's desktop, including the screen size.

```
self.center()
```

The code that will center the window is placed in the custom *center()* method.

```
qr = self.frameGeometry()
```

We get a rectangle specifying the geometry of the main window. This includes any window frame.

```
cp = QDesktopWidget().availableGeometry().center()
```

We figure out the screen resolution of our monitor. And from this resolution, we get the center point.

```
qr.moveCenter(cp)
```

Our rectangle has already its width and height. Now we set the center of the rectangle to the center of the screen. The rectangle's size is unchanged.

```
self.move(qr.topLeft())
```

We move the top-left point of the application window to the top-left point of the qr rectangle, thus centering the window on our screen.

In this part of the PyQt5 tutorial, we covered some basics.

Menus and toolbars in PyQt5

In this part of the PyQt5 tutorial, we will create menus and toolbars. A menu is a group of commands located in a menubar. A toolbar has buttons with some common commands in the application.

Main Window

The *QMainWindow* class provides a main application window. This enables

to create a classic application skeleton with a statusbar, toolbars, and a menubar.

Statusbar

A statusbar is a widget that is used for displaying status information.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-


"""

ZetCode PyQt5 tutorial



This program creates a statusbar.



author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""



import sys

from PyQt5.QtWidgets import QMainWindow, QApplication





class Example(QMainWindow):



    def __init__(self):

        super().__init__()
        self.initUI()



    def initUI(self):


        self.statusBar().showMessage('Ready')


        self.setGeometry(300, 300, 250, 150)

        self.setWindowTitle('Statusbar')

        self.show()



if __name__ == '__main__':


    app = QApplication(sys.argv)

    ex = Example()

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

The statusbar is created with the help of the *QMainWindow* widget.

```
self.statusBar().showMessage('Ready')
```

To get the statusbar, we call the *statusBar()* method of

the *QtGui.QMainWindow* class. The first call of the method creates a status bar. Subsequent calls return the statusbar object. The *showMessage*()displays a message on the statusbar.

Menubar

A menubar is a common part of a GUI application. It is a group of commands located in various menus. (Mac OS treats menubars differently. To get a similar outcome, we can add the following line: *menubar.setNativeMenuBar(False)*.)

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



This program creates a menubar. The

menubar has one menu with an exit action.



author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""



import sys

from PyQt5.QtWidgets import QMainWindow, QAction, qApp, QApplication
from PyQt5.QtGui import QIcon



class Example(QMainWindow):


    def __init__(self):

        super().__init__()


        self.initUI()




    def initUI(self):



        exitAction = QAction(QIcon('exit.png'), '&Exit', self)

        exitAction.setShortcut('Ctrl+Q')

        exitAction.setStatusTip('Exit application')

        exitAction.triggered.connect(qApp.quit)


        self.statusBar()


        menubar = self.menuBar()

        fileMenu = menubar.addMenu('&File')
        fileMenu.addAction(exitAction)



        self.setGeometry(300, 300, 300, 200)

        self.setWindowTitle('Menubar')

        self.show()




if __name__ == '__main__':



    app = QApplication(sys.argv)

    ex = Example()

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

In the above example, we create a menubar with one menu. This menu will contain one action which will terminate the application if selected. A statusbar is created as well. The action is accessible with the Ctrl+Q shortcut.

```
exitAction = QAction(QIcon('exit.png'), '&Exit', self)

exitAction.setShortcut('Ctrl+Q')

exitAction.setStatusTip('Exit application')
```

A `QAction` is an abstraction for actions performed with a menubar, toolbar,

or with a custom keyboard shortcut. In the above three lines, we create an action with a specific icon and an 'Exit' label. Furthermore, a shortcut is defined for this action. The third line creates a status tip which is shown in the statusbar when we hover a mouse pointer over the menu item.

```
exitAction.triggered.connect(qApp.quit)
```

When we select this particular action, a triggered signal is emitted. The signal is connected to the*quit()* method of the *QApplication* widget. This terminates the application.

```
menubar = self.menuBar()

fileMenu = menubar.addMenu('&File')

fileMenu.addAction(exitAction)
```

The *menuBar()* method creates a menubar. We create a file menu and append the exit action to it.

Toolbar

Menus group all commands that we can use in an application. Toolbars provide a quick access to the most frequently used commands.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



This program creates a toolbar.

The toolbar has one action, which

terminates the application, if triggered.



author: Jan Bodnar

website: zetcode.com
```

```
last edited: January 2015
"""


import sys
from PyQt5.QtWidgets import QMainWindow, QAction, qApp, QApplication
from PyQt5.QtGui import QIcon



class Example(QMainWindow):

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

        self.initUI()


    def initUI(self):

        exitAction = QAction(QIcon('exit24.png'), 'Exit', self)
        exitAction.setShortcut('Ctrl+Q')
        exitAction.triggered.connect(qApp.quit)

        self.toolbar = self.addToolBar('Exit')

        self.toolbar.addAction(exitAction)



        self.setGeometry(300, 300, 300, 200)

        self.setWindowTitle('Toolbar')

        self.show()




if __name__ == '__main__':



    app = QApplication(sys.argv)

    ex = Example()

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

In the above example, we create a simple toolbar. The toolbar has one tool action, an exit action which terminates the application when triggered.

```
exitAction = QAction(QIcon('exit24.png'), 'Exit', self)

exitAction.setShortcut('Ctrl+Q')

exitAction.triggered.connect(qApp.quit)
```

Similar to the menubar example above, we create an action object. The object has a label, icon, and a shorcut. A *quit()* method of

the *QtGui.QMainWindow* is connected to the triggered signal.

```
self.toolbar = self.addToolBar('Exit')

self.toolbar.addAction(exitAction)
```

Here we create a toolbar and plug and action object into it.

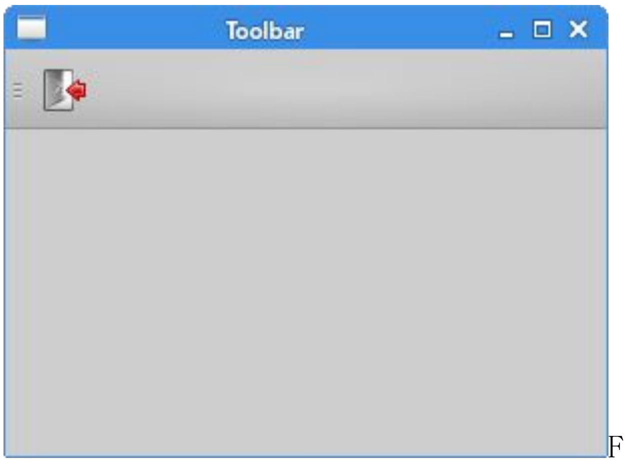


Figure: Toolbar

Putting it together

In the last example of this section, we will create a menubar, toolbar, and a statusbar. We will also create a central widget.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



This program creates a skeleton of

a classic GUI application with a menubar,

toolbar, statusbar, and a central widget.



author: Jan Bodnar

website: zetcode.com
```

```
last edited: January 2015
"""

import sys
from PyQt5.QtWidgets import QMainWindow, QTextEdit, QAction, QApplication
from PyQt5.QtGui import QIcon


class Example(QMainWindow):

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

        self.initUI()


    def initUI(self):

        textEdit = QTextEdit()
        self.setCentralWidget(textEdit)

        exitAction = QAction(QIcon('exit24.png'), 'Exit', self)
        exitAction.setShortcut('Ctrl+Q')

        exitAction.setStatusTip('Exit application')

        exitAction.triggered.connect(self.close)



        self.statusBar()



        menubar = self.menuBar()

        fileMenu = menubar.addMenu('&File')

        fileMenu.addAction(exitAction)



        toolbar = self.addToolBar('Exit')

        toolbar.addAction(exitAction)



        self.setGeometry(300, 300, 350, 250)

        self.setWindowTitle('Main window')

        self.show()




if __name__ == '__main__':



    app = QApplication(sys.argv)
    ex = Example()

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

This code example creates a skeleton of a classic GUI application with a menubar, toolbar, and a statusbar.

```
textEdit = QTextEdit()

self.setCentralWidget(textEdit)
```

Here we create a text edit widget. We set it to be the central widget of the `QMainWindow`. The central widget will occupy all space that is left.

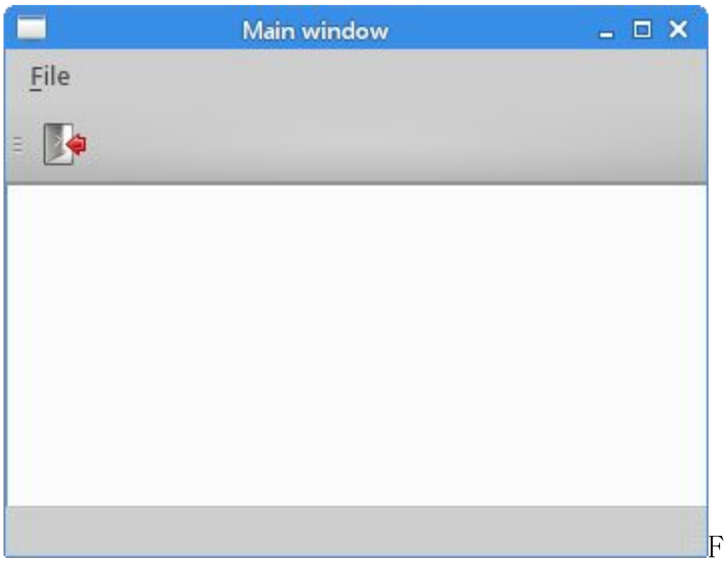


Figure: Main window

In this part of the PyQt5 tutorial, we worked with menus, toolbars, a statusbar, and a main application window.

Layout management in PyQt5

An important aspect in GUI programming is the layout management. Layout management is the way how we place the widgets on the application window. The management can be done in two basic ways. We can use absolute positioning or layout classes.

Absolute positioning

The programmer specifies the position and the size of each widget in pixels. When you use absolute positioning, we have to understand the following limitations:

- The size and the position of a widget do not change if we resize a window
- Applications might look different on various platforms
- Changing fonts in our application might spoil the layout
- If we decide to change our layout, we must completely redo our layout, which is tedious and time consuming

The following example positions widgets in absolute coordinates.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



This example shows three labels on a window

using absolute positioning.



author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""
```

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



class Example(QWidget):

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

        self.initUI()


    def initUI(self):

        lbl1 = QLabel('Zetcode', self)
        lbl1.move(15, 10)

        lbl2 = QLabel('tutorials', self)
        lbl2.move(35, 40)

        lbl3 = QLabel('for programmers', self)
        lbl3.move(55, 70)
        self.setGeometry(300, 300, 250, 150)

        self.setWindowTitle('Absolute')

        self.show()




if __name__ == '__main__':



    app = QApplication(sys.argv)

    ex = Example()

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

We use the `move()` method to position our widgets. In our case these are labels. We position them by providing the x and y coordinates. The beginning of the coordinate system is at the left top corner. The x values grow from left to right. The y values grow from top to bottom.

```
lbl1 = QLabel('Zetcode', self)

lbl1.move(15, 10)
```

The label widget is positioned at `x=15` and `y=10`.

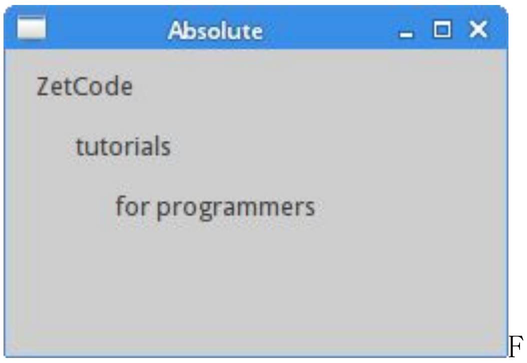


Figure: Absolute positioning

Box layout

Layout management with layout classes is much more flexible and practical. It is the preferred way to place widgets on a window.

The *QHBoxLayout* and *QVBoxLayout* are basic layout classes that line up widgets horizontally and vertically.

Imagine that we wanted to place two buttons in the right bottom corner. To create such a layout, we will use one horizontal and one vertical box. To create the necessary space, we will add a stretch factor.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



In this example, we position two push

buttons in the bottom-right corner

of the window.



author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""



import sys
```

```
from PyQt5.QtWidgets import (QWidget, QPushButton,
    QHBoxLayout, QVBoxLayout, QApplication)



class Example(QWidget):

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

        self.initUI()


    def initUI(self):

        okButton = QPushButton("OK")
        cancelButton = QPushButton("Cancel")

        hbox = QHBoxLayout()
        hbox.addStretch(1)
        hbox.addWidget(okButton)
        hbox.addWidget(cancelButton)
        vbox = QVBoxLayout()

        vbox.addStretch(1)

        vbox.addLayout(hbox)



        self.setLayout(vbox)



        self.setGeometry(300, 300, 300, 150)

        self.setWindowTitle('Buttons')

        self.show()




if __name__ == '__main__':



    app = QApplication(sys.argv)

    ex = Example()

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

The example places two buttons in the bottom-right corner of the window. They stay there when we resize the application window. We use both

a `HBoxLayout` and a `QVBoxLayout`.

```
okButton = QPushButton("OK")

cancelButton = QPushButton("Cancel")
```

Here we create two push buttons.

```
hbox = QHBoxLayout()

hbox.addStretch(1)

hbox.addWidget(okButton)

hbox.addWidget(cancelButton)
```

We create a horizontal box layout and add a stretch factor and both buttons. The stretch adds a stretchable space before the two buttons. This will push them to the right of the window.

```
vbox = QVBoxLayout()

vbox.addStretch(1)

vbox.addLayout(hbox)
```

To create the necessary layout, we put a horizontal layout into a vertical one. The stretch factor in the vertical box will push the horizontal box with the buttons to the bottom of the window.

```
self.setLayout(vbox)
```

Finally, we set the main layout of the window.

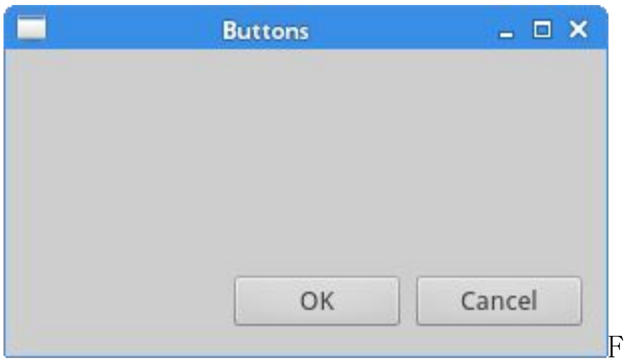


Figure: Buttons

QGridLayout

The most universal layout class is the grid layout. This layout divides the space into rows and columns. To create a grid layout, we use

the *QGridLayout* class.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-


"""

ZetCode PyQt5 tutorial


In this example, we create a skeleton

of a calculator using a QGridLayout.


author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""


import sys

from PyQt5.QtWidgets import (QWidget, QGridLayout,

    QPushButton, QApplication)



class Example(QWidget):


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


        self.initUI()



    def initUI(self):


        grid = QGridLayout()

        self.setLayout(grid)


        names = ['Cls', 'Bck', '', 'Close',

                 '7', '8', '9', '/',

                '4', '5', '6', '*',

                 '1', '2', '3', '-',

                '0', '.', '=', '+']


        positions = [(i,j) for i in range(5) for j in range(4)]


        for position, name in zip(positions, names):


            if name == '':

                continue
            button = QPushButton(name)

            grid.addWidget(button, *position)



        self.move(300, 150)

        self.setWindowTitle('Calculator')

        self.show()




if __name__ == '__main__':



    app = QApplication(sys.argv)

    ex = Example()

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

In our example, we create a grid of buttons. To fill one gap, we add one *QLabel* widget.

```
grid = QGridLayout()

self.setLayout(grid)
```

The instance of a *QGridLayout* is created and set to be the layout for the application window.

```
names = ['Cls', 'Bck', '', 'Close',

            '7', '8', '9', '/',

        '4', '5', '6', '*',
            '1', '2', '3', '-',

        '0', '.', '=', '+']
```

These are the labels used later for buttons.

```
positions = [(i,j) for i in range(5) for j in range(4)]
```

We create a list of positions in the grid.

```
for position, name in zip(positions, names):



    if name == '':

        continue

    button = QPushButton(name)

    grid.addWidget(button, *position)
```

Buttons are created and added to the layout with the *addWidget*() method.

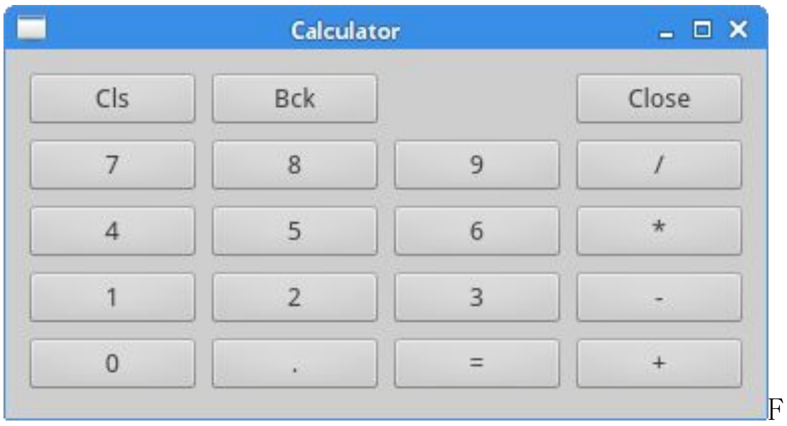


Figure: Calculator skeleton

Review example

Widgets can span multiple columns or rows in a grid. In the next example we illustrate this.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-


"""

ZetCode PyQt5 tutorial



In this example, we create a bit

more complicated window layout using

the QGridLayout manager.



author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""


import sys

from PyQt5.QtWidgets import (QWidget, QLabel, QLineEdit,

    QTextEdit, QGridLayout, QApplication)




class Example(QWidget):

```

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

    self.initUI()


def initUI(self):

    title = QLabel('Title')
    author = QLabel('Author')
    review = QLabel('Review')

    titleEdit = QLineEdit()
    authorEdit = QLineEdit()
    reviewEdit = QTextEdit()

    grid = QGridLayout()
    grid.setSpacing(10)

    grid.addWidget(title, 1, 0)
    grid.addWidget(titleEdit, 1, 1)
```

```
        grid.addWidget(author, 2, 0)

        grid.addWidget(authorEdit, 2, 1)


        grid.addWidget(review, 3, 0)

        grid.addWidget(reviewEdit, 3, 1, 5, 1)


        self.setLayout(grid)


        self.setGeometry(300, 300, 350, 300)

        self.setWindowTitle('Review')

        self.show()



if __name__ == '__main__':


    app = QApplication(sys.argv)

    ex = Example()

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

We create a window in which we have three labels, two line edits and one text edit widget. The layout is done with the *QGridLayout*.

```
grid = QGridLayout()

grid.setSpacing(10)
```

We create a grid layout and set spacing between widgets.

```
grid.addWidget(reviewEdit, 3, 1, 5, 1)
```

If we add a widget to a grid, we can provide row span and column span of the widget. In our case, we make the *reviewEdit* widget span 5 rows.

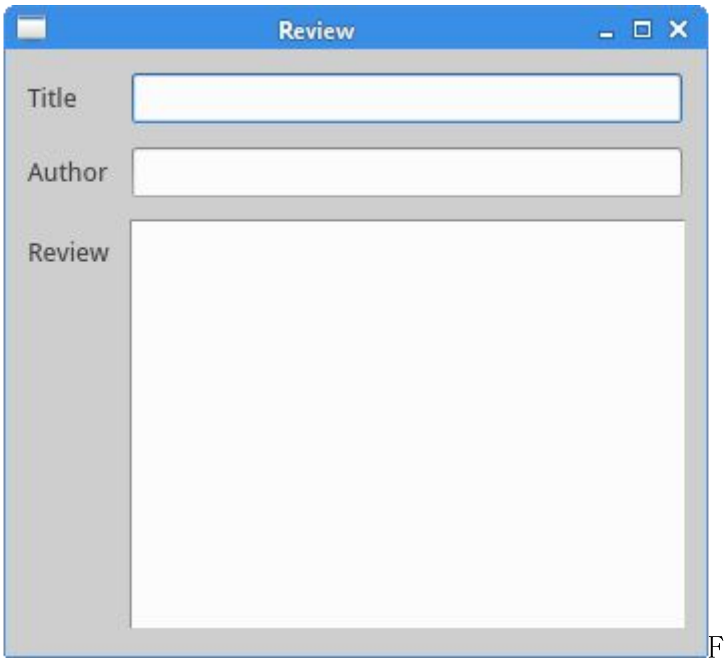


Figure: Review example

This part of the PyQt5 tutorial was dedicated to layout management.

Events and signals in PyQt5

In this part of the PyQt5 programming tutorial, we will explore events and signals occurring in applications.

Events

All GUI applications are event-driven. Events are generated mainly by the user of an application. But they can be generated by other means as well: e.g. an Internet connection, a window manager, or a timer. When we call the application's *exec_()* method, the application enters the main loop. The main loop fetches events and sends them to the objects.

In the event model, there are three participants:

- event source
- event object
- event target

The event source is the object whose state changes. It generates events.
The event object (event) encapsulates the state changes in the event source.
The event target is the object that wants to be notified. Event source object delegates the task of handling an event to the event target.

PyQt5 has a unique signal and slot mechanism to deal with events. Signals and slots are used for communication between objects. A signal is emitted when a particular event occurs. A slot can be any Python callable. A slot is called when its connected signal is emitted.

Signals & slots

This is a simple example demonstrating signals and slots in PyQt5.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



In this example, we connect a signal

of a QSlider to a slot of a QLCDNumber.



author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""
```

```
import sys
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import (QWidget, QLCDNumber, QSlider,
    QVBoxLayout, QApplication)


class Example(QWidget):

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

        self.initUI()


    def initUI(self):

        lcd = QLCDNumber(self)
        sld = QSlider(Qt.Horizontal, self)

        vbox = QVBoxLayout()
        vbox.addWidget(lcd)
        vbox.addWidget(sld)


        self.setLayout(vbox)

        sld.valueChanged.connect(lcd.display)


        self.setGeometry(300, 300, 250, 150)

        self.setWindowTitle('Signal & slot')

        self.show()



if __name__ == '__main__':


    app = QApplication(sys.argv)

    ex = Example()

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

In our example, we display a *QtGui.QLCDNumber* and a *QtGui.QSlider*.

We change the *lcd* number by dragging the slider knob.

```
sld.valueChanged.connect(lcd.display)
```

Here we connect a *valueChanged* signal of the slider to the *display* slot of

the *lcd* number.

The sender is an object that sends a signal. The receiver is the object that receives the signal. Theslot is the method that reacts to the signal.

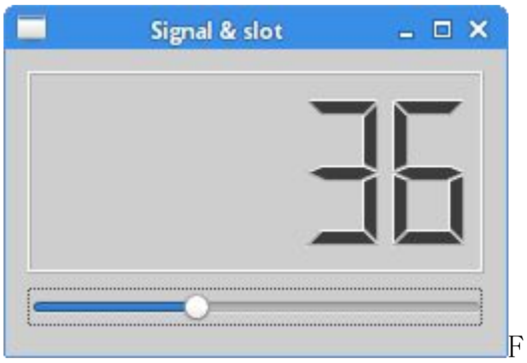


Figure: Signal & slot

Reimplementing event handler

Events in PyQt5 are processed often by reimplementing event handlers.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



In this example, we reimplement an

event handler.



author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""



import sys
```

```
from PyQt5.QtCore import Qt

from PyQt5.QtWidgets import QWidget, QApplication




class Example(QWidget):



    def __init__(self):

        super().__init__()



        self.initUI()




    def initUI(self):



        self.setGeometry(300, 300, 250, 150)

        self.setWindowTitle('Event handler')

        self.show()




    def keyPressEvent(self, e):



        if e.key() == Qt.Key_Escape:
            self.close()




if __name__ == '__main__':



    app = QApplication(sys.argv)

    ex = Example()

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

In our example, we reimplement the *keyPressEvent()* event handler.

```
def keyPressEvent(self, e):



    if e.key() == Qt.Key_Escape:

        self.close()
```

If we click the Escape button, the application terminates.

Event sender

Sometimes it is convenient to know which widget is the sender of a signal. For this, PyQt5 has the*sender()* method.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""
```

```
ZetCode PyQt5 tutorial


In this example, we determine the event sender

object.


author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""


import sys

from PyQt5.QtWidgets import QMainWindow, QPushButton, QApplication




class Example(QMainWindow):


    def __init__(self):

        super().__init__()


        self.initUI()
```

```
def initUI(self):


    btn1 = QPushButton("Button 1", self)

    btn1.move(30, 50)


    btn2 = QPushButton("Button 2", self)

    btn2.move(150, 50)


    btn1.clicked.connect(self.buttonClicked)

    btn2.clicked.connect(self.buttonClicked)


    self.statusBar()


    self.setGeometry(300, 300, 290, 150)

    self.setWindowTitle('Event sender')

    self.show()



def buttonClicked(self):


    sender = self.sender()

    self.statusBar().showMessage(sender.text() + ' was pressed')
```

```
if __name__ == '__main__':



    app = QApplication(sys.argv)

    ex = Example()

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

We have two buttons in our example. In the *buttonClicked()* method we determine which button we have clicked by calling the *sender()* method.

```
btn1.clicked.connect(self.buttonClicked)

btn2.clicked.connect(self.buttonClicked)
```

Both buttons are connected to the same slot.

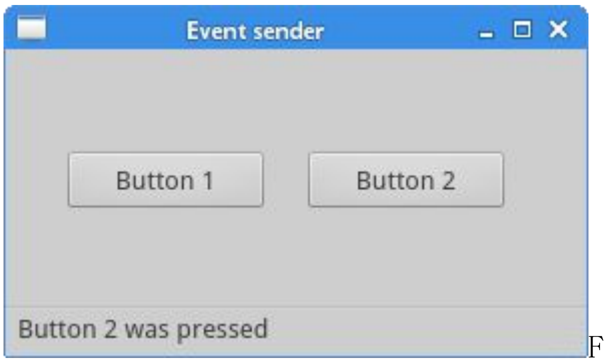
```
def buttonClicked(self):



    sender = self.sender()

    self.statusBar().showMessage(sender.text() + ' was pressed')
```

We determine the signal source by calling the *sender()* method. In the statusbar of the application, we show the label of the button being pressed.



Button 2 was pressed

Figure: Event sender

Emitting signals

Objects created from a *QObject* can emit signals. In the following example we will see how we can emit custom signals.

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-



"""

ZetCode PyQt5 tutorial



In this example, we show how to emit a

signal.



author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""



import sys

from PyQt5.QtCore import pyqtSignal, QObject

from PyQt5.QtWidgets import QMainWindow, QApplication
```

```
class Communicate(QObject):


    closeApp = pyqtSignal()



class Example(QMainWindow):


    def __init__(self):

        super().__init__()


        self.initUI()



    def initUI(self):


        self.c = Communicate()

        self.c.closeApp.connect(self.close)


        self.setGeometry(300, 300, 290, 150)

        self.setWindowTitle('Emit signal')

        self.show()
    def mousePressEvent(self, event):

        self.c.closeApp.emit()


if __name__ == '__main__':

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

We create a new signal called *closeApp*. This signal is emitted during a mouse press event. The signal is connected to the *close*() slot of the *QMainWindow*.

```
class Communicate(QObject):

    closeApp = pyqtSignal()
```

A signal is created with the *pyqtSignal*() as a class attribute of the external *Communicate* class.

```
self.c = Communicate()
self.c.closeApp.connect(self.close)
```

The custom *closeApp* signal is connected to the *close()* slot of

the *QMainWindow*.

```
def mousePressEvent(self, event):



    self.c.closeApp.emit()
```

When we click on the window with a mouse pointer, the *closeApp* signal is emitted. The application terminates.
In this part of the PyQt5 tutorial, we have covered signals and slots.

本资料来源：

http://zetcode.com/gui/pyqt5/