

PyQt5 tutorial

This is PyQt5 tutorial. The tutorial is suited for beginners and intermediate programmers. After reading this tutorial, you will be able to program non trivial PyQt5 applications.

Table of contents

- [Introduction](#)
- [First programs](#)
- [Menus and toolbars](#)
- [Layout management](#)
- [Events and signals](#)
- [Dialogs](#)
- [Widgets](#)
- [Widgets II](#)
- [Drag & drop](#)
- [Painting](#)
- [Custom widgets](#)
- [The Tetris game](#)

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 handling. 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.

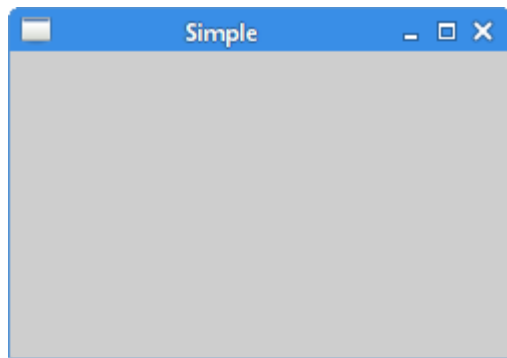


Figure: Simple

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 see 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 its 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.

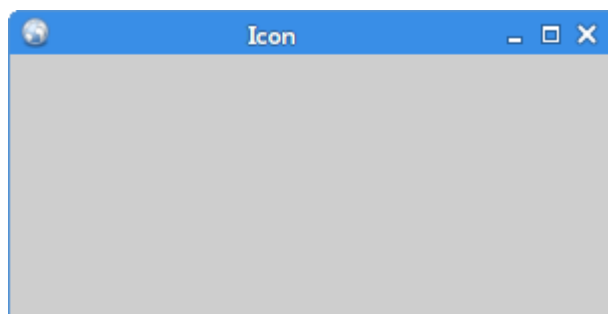


Figure: Icon

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.

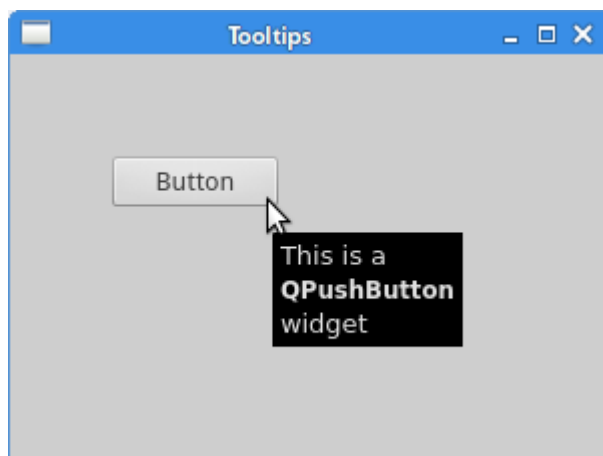


Figure: Tooltips

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 programmatically 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 QApplication
```

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.

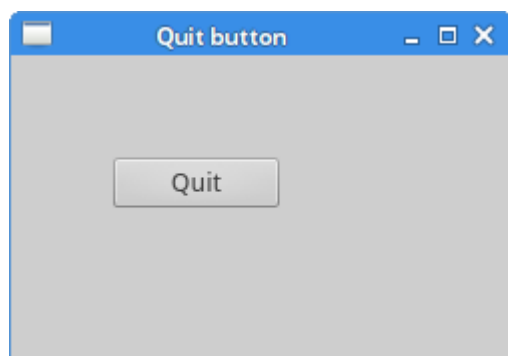


Figure: Quit button

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.

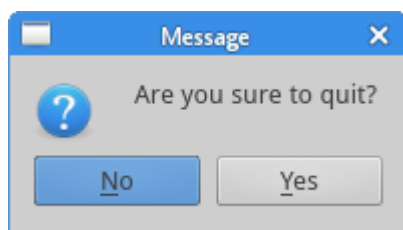


Figure: Message box

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, 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 shortcut. 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.

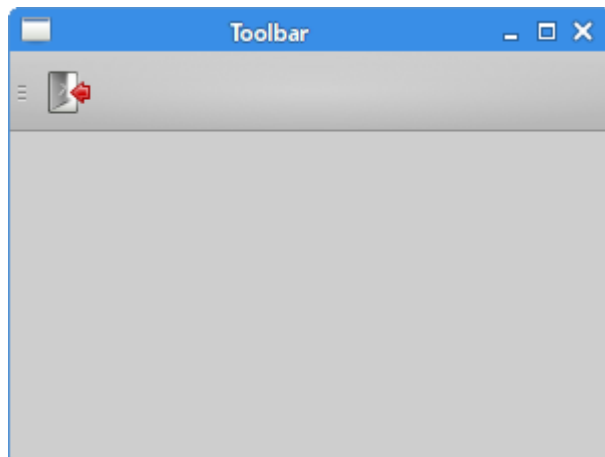


Figure: Toolbar

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.

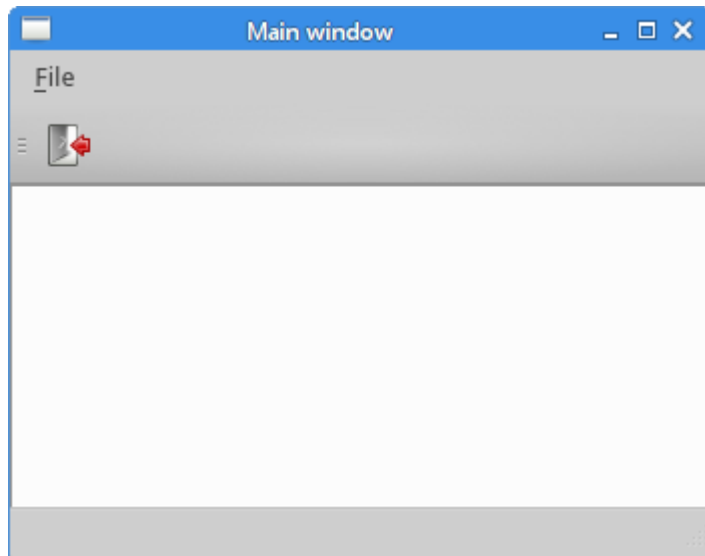


Figure: Main window

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.

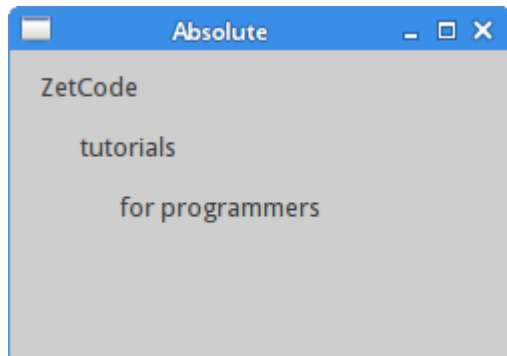


Figure: Absolute positioning

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.

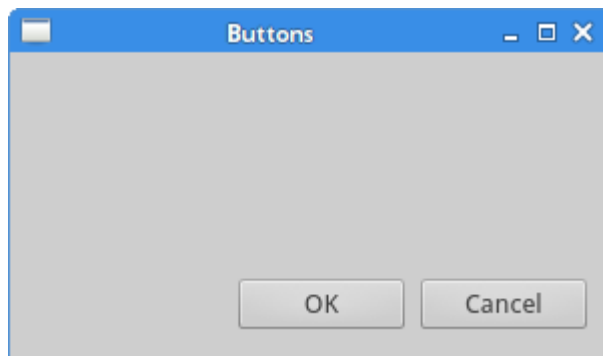


Figure: Buttons

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.

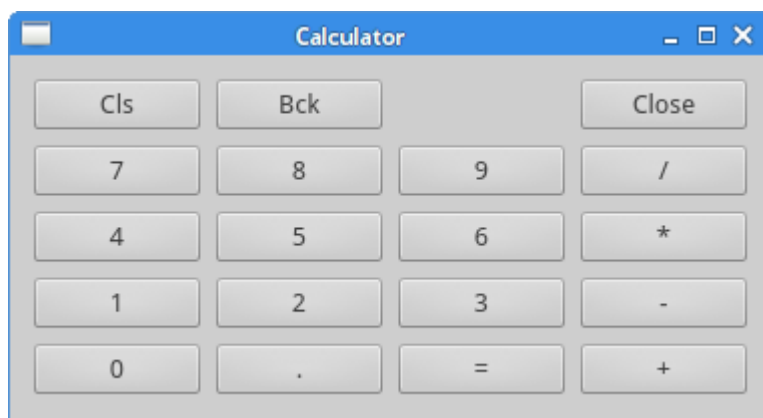


Figure: Calculator skeleton

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)

```

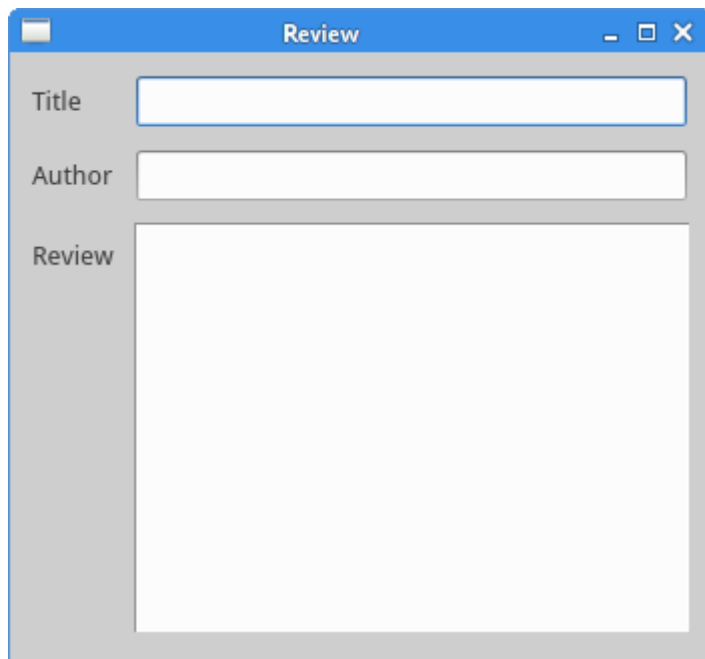
We create a grid layout and set spacing between widgets.

```

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.



The image shows a PyQt5 window titled "Review". The window has a blue title bar with standard minimize, maximize, and close buttons. The main content area is a light gray rectangle. On the left side of this area, there are three labels: "Title", "Author", and "Review". To the right of each label is a corresponding input field. The "Title" and "Author" fields are single-line text boxes, while the "Review" field is a larger, multi-line text area. All input fields are currently empty.

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. The *slot* is the method that reacts to the signal.

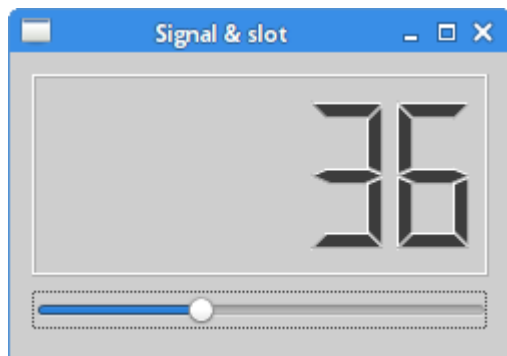


Figure: Signal & slot

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.

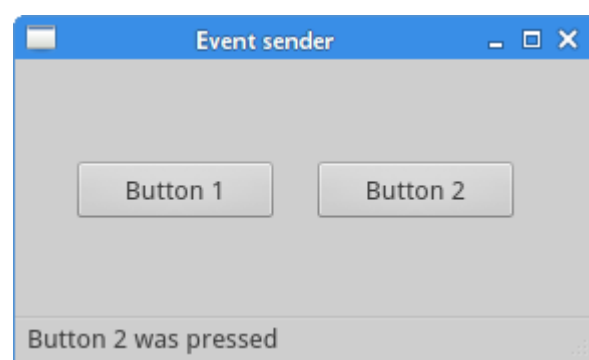


Figure: Event sender

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.

Dialogs in PyQt5

Dialog windows or dialogs are an indispensable part of most modern GUI applications. A dialog is defined as a conversation between two or more persons. In a computer application a dialog is a window which is used to "talk" to the application. A dialog is used to input data, modify data, change the application settings etc.

QInputDialog

The QInputDialog provides a simple convenience dialog to get a single value from the user. The input value can be a string, a number, or an item from a list.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

In this example, we receive data from
a QInputDialog dialog.

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

```
import sys
from PyQt5.QtWidgets import (QWidget, QPushButton, QLineEdit,
                             QInputDialog, QApplication)
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```

        self.btn = QPushButton('Dialog', self)
        self.btn.move(20, 20)
        self.btn.clicked.connect(self.showDialog)

        self.le = QLineEdit(self)
        self.le.move(130, 22)

        self.setGeometry(300, 300, 290, 150)
        self.setWindowTitle('Input dialog')
        self.show()

    def showDialog(self):

        text, ok = QDialog.getText(self, 'Input Dialog',
                                   'Enter your name:')

        if ok:
            self.le.setText(str(text))

if __name__ == '__main__':

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

```

The example has a button and a line edit widget. The button shows the input dialog for getting text values. The entered text will be displayed in the line edit widget.

```

text, ok = QDialog.getText(self, 'Input Dialog',
                           'Enter your name:')

```

This line displays the input dialog. The first string is a dialog title, the second one is a message within the dialog. The dialog returns the entered text and a boolean value. If we click the Ok button, the boolean value is true.

```

if ok:
    self.le.setText(str(text))

```

The text that we have received from the dialog is set to the line edit widget.

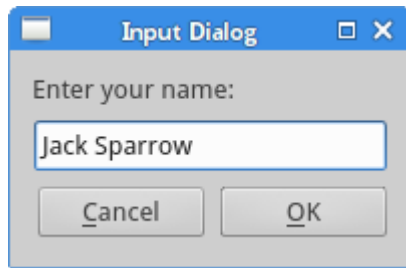


Figure: Input dialog

QColorDialog

The QColorDialog provides a dialog widget for selecting colour values.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

In this example, we select a color value from the QColorDialog and change the background color of a QFrame widget.

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

```
import sys
from PyQt5.QtWidgets import (QWidget, QPushButton, QFrame,
                             QColorDialog, QApplication)
from PyQt5.QtGui import QColor
```

```
class Example(QWidget):
```

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

        self.initUI()
```

```
    def initUI(self):
```

```

col = QColor(0, 0, 0)

self.btn = QPushButton('Dialog', self)
self.btn.move(20, 20)

self.btn.clicked.connect(self.showDialog)

self.frm = QFrame(self)
self.frm.setStyleSheet("QWidget { background-color: %s }"
    % col.name())
self.frm.setGeometry(130, 22, 100, 100)

self.setGeometry(300, 300, 250, 180)
self.setWindowTitle('Color dialog')
self.show()

def showDialog(self):

    col = QColorDialog.getColor()

    if col.isValid():
        self.frm.setStyleSheet("QWidget { background-color: %s }"
            % col.name())

if __name__ == '__main__':

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

```

The application example shows a push button and a QFrame. The widget background is set to black colour. Using the QColorDialog, we can change its background.

```
col = QColor(0, 0, 0)
```

This is an initial colour of the QtGui.QFrame background.

```
col = QColorDialog.getColor()
```

This line will pop up the QColorDialog.


```
if col.isValid():
    self.frm.setStyleSheet("QWidget { background-color: %s }"
        % col.name())
```

We check if the colour is valid. If we click on the Cancel button, no valid colour is returned. If the colour is valid, we change the background colour using style sheets.

QFontDialog

The QFontDialog is a dialog widget for selecting a font.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

In this example, we select a font name
and change the font of a label.

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

```
import sys
from PyQt5.QtWidgets import (QWidget, QVBoxLayout, QPushButton,
    QSizePolicy, QLabel, QFontDialog, QApplication)
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```
        vbox = QVBoxLayout()
```

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

```

        btn.setSizePolicy(QSizePolicy.Fixed,
                           QSizePolicy.Fixed)

        btn.move(20, 20)

        vbox.addWidget(btn)

        btn.clicked.connect(self.showDialog)

        self.lbl = QLabel('Knowledge only matters', self)
        self.lbl.move(130, 20)

        vbox.addWidget(self.lbl)
        self.setLayout(vbox)

        self.setGeometry(300, 300, 250, 180)
        self.setWindowTitle('Font dialog')
        self.show()

    def showDialog(self):

        font, ok = QFontDialog.getFont()
        if ok:
            self.lbl.setFont(font)

if __name__ == '__main__':

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

```

In our example, we have a button and a label. With the `QFontDialog`, we change the font of the label.

```
font, ok = QFontDialog.getFont()
```

Here we pop up the font dialog. The `getFont()` method returns the font name and the `ok` parameter. It is equal to `True` if the user clicked OK; otherwise it is `False`.

```

if ok:
    self.label.setFont(font)

```

If we clicked Ok, the font of the label would be changed.

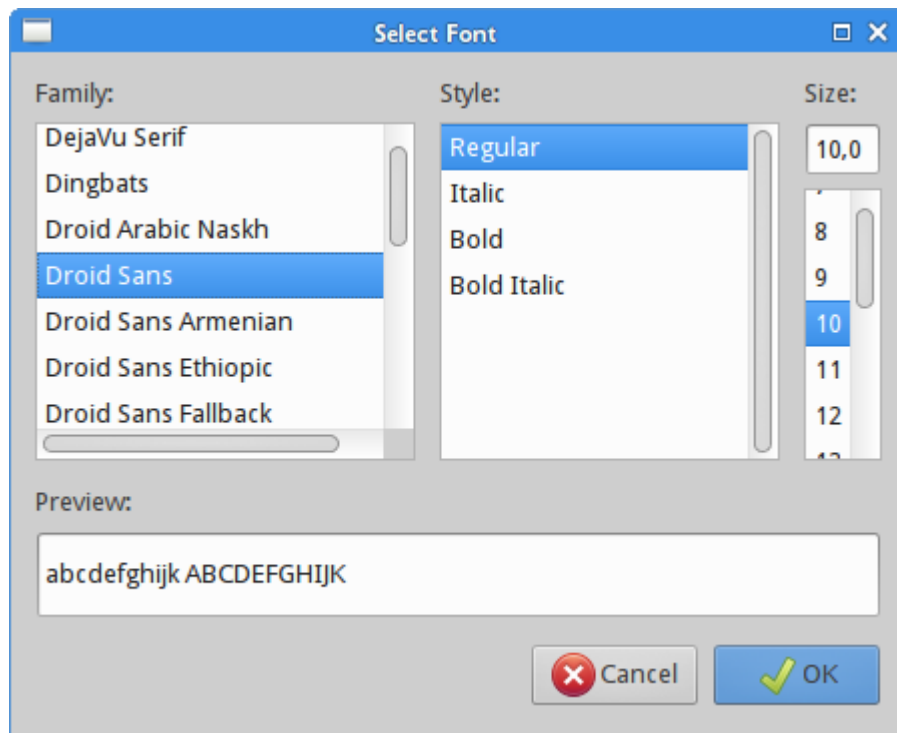


Figure: Font dialog

QFileDialog

The `QFileDialog` is a dialog that allows users to select files or directories. The files can be selected for both opening and saving.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

```
In this example, we select a file with a
QFileDialog and display its contents
in a QTextEdit.
```

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

```
import sys
from PyQt5.QtWidgets import (QMainWindow, QTextEdit,
```

```

    QAction, QFileDialog, QApplication)
from PyQt5.QtGui import QIcon

class Example(QMainWindow):

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

        self.initUI()

    def initUI(self):

        self.textEdit = QTextEdit()
        self.setCentralWidget(self.textEdit)
        self.statusBar()

        openFile = QAction(QIcon('open.png'), 'Open', self)
        openFile.setShortcut('Ctrl+O')
        openFile.setStatusTip('Open new File')
        openFile.triggered.connect(self.showDialog)

        menubar = self.menuBar()
        fileMenu = menubar.addMenu('&File')
        fileMenu.addAction(openFile)

        self.setGeometry(300, 300, 350, 300)
        self.setWindowTitle('File dialog')
        self.show()

    def showDialog(self):

        fname = QFileDialog.getOpenFileName(self, 'Open file', '/home')

        if fname[0]:
            f = open(fname[0], 'r')

            with f:
                data = f.read()
                self.textEdit.setText(data)

if __name__ == '__main__':

```

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

The example shows a menubar, centrally set text edit widget, and a statusbar. The menu item shows the `QtGui.QFileDialog` which is used to select a file. The contents of the file are loaded into the text edit widget.

```
class Example(QMainWindow):

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

        self.initUI()
```

The example is based on the `QMainWindow` widget because we centrally set a text edit widget.

```
fname = QFileDialog.getOpenFileName(self, 'Open file', '/home')
```

We pop up the `QFileDialog`. The first string in the `getOpenFileName()` method is the caption. The second string specifies the dialog working directory. By default, the file filter is set to All files (*).

```
if fname[0]:
    f = open(fname[0], 'r')

    with f:
        data = f.read()
        self.textEdit.setText(data)
```

The selected file name is read and the contents of the file are set to the text edit widget.

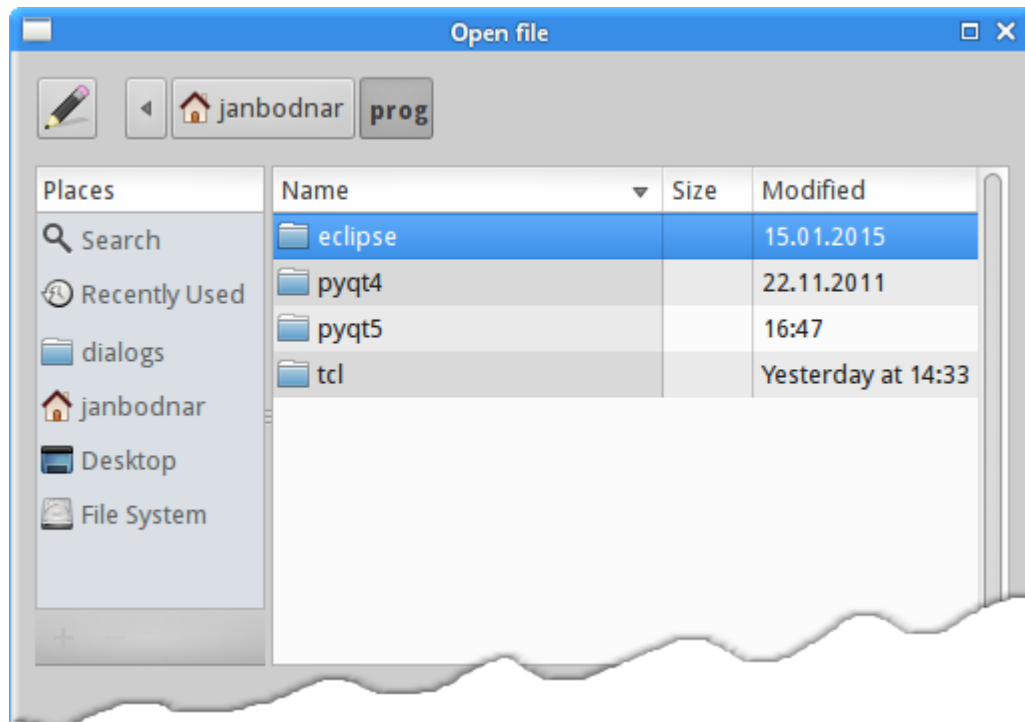


Figure: File dialog

In this part of the PyQt5 tutorial, we worked with dialogs.

PyQt5 widgets

Widgets are basic building blocks of an application. PyQt5 has a wide range of various widgets, including buttons, check boxes, sliders, or list boxes. In this section of the tutorial, we will describe several useful widgets: a `QCheckBox`, a `ToggleButton`, a `QSlider`, a `QProgressBar`, and a `QCalendarWidget`.

QCheckBox

A `QCheckBox` is a widget that has two states: on and off. It is a box with a label. Checkboxes are typically used to represent features in an application that can be enabled or disabled.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

In this example, a `QCheckBox` widget is used to toggle the title of a window.

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

```
import sys
from PyQt5.QtWidgets import QWidget, QCheckBox, QApplication
from PyQt5.QtCore import Qt
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```

        cb = QCheckBox('Show title', self)
        cb.move(20, 20)
        cb.toggle()
        cb.stateChanged.connect(self.changeTitle)

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

    def changeTitle(self, state):

        if state == Qt.Checked:
            self.setWindowTitle('QCheckBox')
        else:
            self.setWindowTitle('')

if __name__ == '__main__':

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

```

In our example, we will create a checkbox that will toggle the window title.

```
cb = QCheckBox('Show title', self)
```

This is a QCheckBox constructor.

```
cb.toggle()
```

We have set the window title, so we must also check the checkbox. By default, the window title is not set and the checkbox is unchecked.

```
cb.stateChanged.connect(self.changeTitle)
```

We connect the user defined changeTitle() method to the stateChanged signal. The changeTitle() method will toggle the window title.

```

def changeTitle(self, state):

    if state == Qt.Checked:
        self.setWindowTitle('QCheckBox')

```



```

else:
    self.setWindowTitle('')

```

The state of the widget is given to the `changeTitle()` method in the state variable. If the widget is checked, we set a title of the window. Otherwise, we set an empty string to the titlebar.

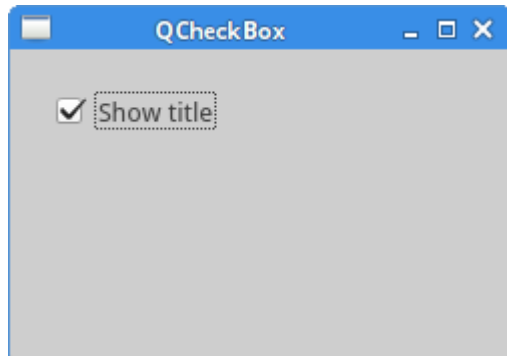


Figure: QCheckBox

Toggle button

A toggle button is a `QPushButton` in a special mode. It is a button that has two states: pressed and not pressed. We toggle between these two states by clicking on it. There are situations where this functionality fits well.

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

```

```

"""

```

```

ZetCode PyQt5 tutorial

```

In this example, we create three toggle buttons. They will control the background colour of a `QFrame`.

```

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

```

```

import sys
from PyQt5.QtWidgets import (QWidget, QPushButton,
                             QFrame, QApplication)
from PyQt5.QtGui import QColor

```

```

class Example(QWidget):

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

        self.initUI()

    def initUI(self):

        self.col = QColor(0, 0, 0)

        redb = QPushButton('Red', self)
        redb.setCheckable(True)
        redb.move(10, 10)

        redb.clicked[bool].connect(self.setColor)

        redb = QPushButton('Green', self)
        redb.setCheckable(True)
        redb.move(10, 60)

        redb.clicked[bool].connect(self.setColor)

        blueb = QPushButton('Blue', self)
        blueb.setCheckable(True)
        blueb.move(10, 110)

        blueb.clicked[bool].connect(self.setColor)

        self.square = QFrame(self)
        self.square.setGeometry(150, 20, 100, 100)
        self.square.setStyleSheet("QWidget { background-color: %s }" %
                                   self.col.name())

        self.setGeometry(300, 300, 280, 170)
        self.setWindowTitle('Toggle button')
        self.show()

    def setColor(self, pressed):

```

```

        source = self.sender()

        if pressed:
            val = 255
        else: val = 0

        if source.text() == "Red":
            self.col.setRed(val)
        elif source.text() == "Green":
            self.col.setGreen(val)
        else:
            self.col.setBlue(val)

        self.square.setStyleSheet("QFrame { background-color: %s }" %
                                   self.col.name())

if __name__ == '__main__':

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

```

In our example, we create three toggle buttons and a QWidget. We set the background colour of the QWidget to black. The toggle buttons will toggle the red, green, and blue parts of the colour value. The background colour will depend on which toggle buttons we have pressed.

```
self.col = QColor(0, 0, 0)
```

This is the initial, black colour value.

```

redb = QPushButton('Red', self)
redb.setCheckable(True)
redb.move(10, 10)

```

To create a toggle button, we create a QPushButton and make it checkable by calling the setCheckable() method.

```
redb.clicked[bool].connect(self.setColor)
```

We connect a clicked signal to our user defined method. We use the clicked signal that operates with a Boolean value.

```
source = self.sender()
```

We get the button which was toggled.

```
if source.text() == "Red":  
    self.col.setRed(val)
```

In case it is a red button, we update the red part of the colour accordingly.

```
self.square.setStyleSheet("QFrame { background-color: %s }" %  
    self.col.name())
```

We use style sheets to change the background colour.

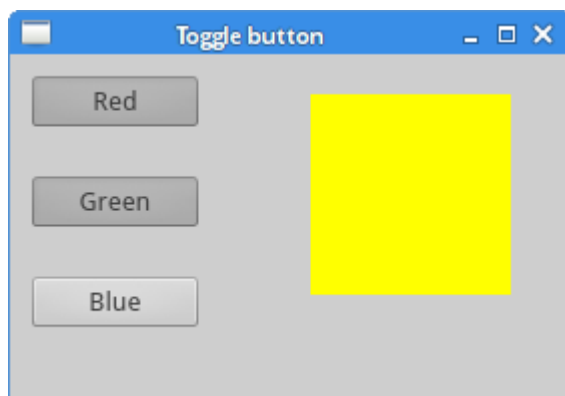


Figure: Toggle button

QSlider

A QSlider is a widget that has a simple handle. This handle can be pulled back and forth. This way we are choosing a value for a specific task. Sometimes using a slider is more natural than entering a number or using a spin box.

In our example we will show one slider and one label. The label will display an image. The slider will control the label.

```
#!/usr/bin/python3  
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

This example shows a QSlider widget.

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

```
import sys
from PyQt5.QtWidgets import (QWidget, QSlider,
                             QLabel, QApplication)
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QPixmap

class Example(QWidget):

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

        self.initUI()

    def initUI(self):

        sld = QSlider(Qt.Horizontal, self)
        sld.setFocusPolicy(Qt.NoFocus)
        sld.setGeometry(30, 40, 100, 30)
        sld.valueChanged[int].connect(self.changeValue)

        self.label = QLabel(self)
        self.label.setPixmap(QPixmap('mute.png'))
        self.label.setGeometry(160, 40, 80, 30)

        self.setGeometry(300, 300, 280, 170)
        self.setWindowTitle('QSlider')
        self.show()

    def changeValue(self, value):

        if value == 0:
            self.label.setPixmap(QPixmap('mute.png'))
        elif value > 0 and value <= 30:
            self.label.setPixmap(QPixmap('min.png'))
        elif value > 30 and value < 80:
```

```

        self.label.setPixmap(QPixmap('med.png'))
    else:
        self.label.setPixmap(QPixmap('max.png'))

if __name__ == '__main__':

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

```

In our example we simulate a volume control. By dragging the handle of a slider, we change an image on the label.

```
sld = QSlider(Qt.Horizontal, self)
```

Here we create a horizontal QSlider.

```

self.label = QLabel(self)
self.label.setPixmap(QPixmap('mute.png'))

```

We create a QLabel widget and set an initial mute image to it.

```
sld.valueChanged[int].connect(self.changeValue)
```

We connect the valueChanged signal to the user defined changeValue() method.

```

if value == 0:
    self.label.setPixmap(QPixmap('mute.png'))
...

```

Based on the value of the slider, we set an image to the label. In the above code, we set the mute.png image to the label if the slider is equal to zero.

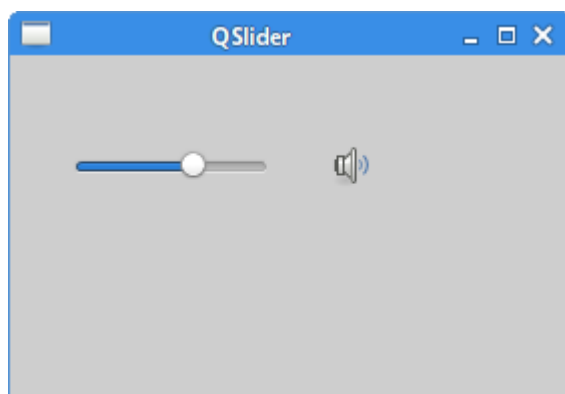


Figure: QSlider widget

QProgressBar

A progress bar is a widget that is used when we process lengthy tasks. It is animated so that the user knows that the task is progressing. The QProgressBar widget provides a horizontal or a vertical progress bar in PyQt5 toolkit. The programmer can set the minimum and maximum value for the progress bar. The default values are 0 and 99.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

This example shows a QProgressBar widget.

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

```
import sys
from PyQt5.QtWidgets import (QWidget, QProgressBar,
                             QPushButton, QApplication)
from PyQt5.QtCore import QBasicTimer
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```
        self.pbar = QProgressBar(self)
        self.pbar.setGeometry(30, 40, 200, 25)
```

```
        self.btn = QPushButton('Start', self)
```

```

        self.btn.move(40, 80)
        self.btn.clicked.connect(self.doAction)

        self.timer = QTimer()
        self.step = 0

        self.setGeometry(300, 300, 280, 170)
        self.setWindowTitle('QProgressBar')
        self.show()

    def timerEvent(self, e):

        if self.step >= 100:
            self.timer.stop()
            self.btn.setText('Finished')
            return

        self.step = self.step + 1
        self.pbar.setValue(self.step)

    def doAction(self):

        if self.timer.isActive():
            self.timer.stop()
            self.btn.setText('Start')
        else:
            self.timer.start(100, self)
            self.btn.setText('Stop')

if __name__ == '__main__':

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

```

In our example we have a horizontal progress bar and a push button. The push button starts and stops the progress bar.

```
self.pbar = QProgressBar(self)
```

This is a `QProgressBar` constructor.


```
self.timer = QtCore.QBasicTimer()
```

To activate the progress bar, we use a timer object.

```
self.timer.start(100, self)
```

To launch a timer event, we call its `start()` method. This method has two parameters: the timeout and the object which will receive the events.

```
def timerEvent(self, e):  
  
    if self.step >= 100:  
  
        self.timer.stop()  
        self.btn.setText('Finished')  
        return  
  
    self.step = self.step + 1  
    self.pbar.setValue(self.step)
```

Each `QObject` and its descendants have a `timerEvent()` event handler. In order to react to timer events, we reimplement the event handler.

```
def doAction(self):  
  
    if self.timer.isActive():  
        self.timer.stop()  
        self.btn.setText('Start')  
  
    else:  
        self.timer.start(100, self)  
        self.btn.setText('Stop')
```

Inside the `doAction()` method, we start and stop the timer.

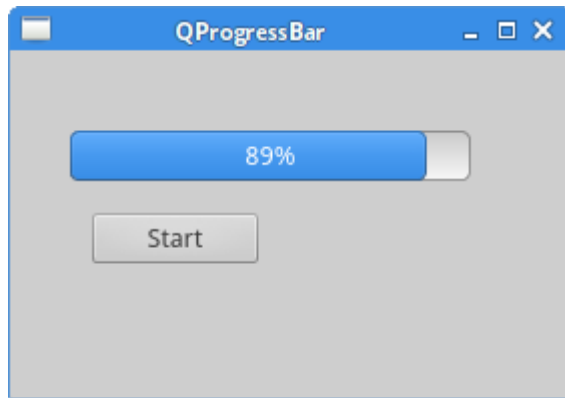


Figure: QProgressBar

QCalendarWidget

A `QCalendarWidget` provides a monthly based calendar widget. It allows a user to select a date in a simple and intuitive way.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

This example shows a `QCalendarWidget` widget.

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

```
import sys
from PyQt5.QtWidgets import (QWidget, QCalendarWidget,
                             QLabel, QApplication)
from PyQt5.QtCore import QDate
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```

def initUI(self):

    cal = QCalendarWidget(self)
    cal.setGridVisible(True)
    cal.move(20, 20)
    cal.clicked[QDate].connect(self.showDate)

    self.lbl = QLabel(self)
    date = cal.selectedDate()
    self.lbl.setText(date.toString())
    self.lbl.move(130, 260)

    self.setGeometry(300, 300, 350, 300)
    self.setWindowTitle('Calendar')
    self.show()

def showDate(self, date):

    self.lbl.setText(date.toString())

if __name__ == '__main__':

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

```

The example has a calendar widget and a label widget. The currently selected date is displayed in the label widget.

```
cal = QCalendarWidget(self)
```

The QCalendarWidget is created.

```
cal.clicked[QDate].connect(self.showDate)
```

If we select a date from the widget, a clicked[QDate] signal is emitted. We connect this signal to the user defined showDate() method.

```

def showDate(self, date):

    self.lbl.setText(date.toString())

```

We retrieve the selected date by calling the `selectedDate()` method. Then we transform the date object into string and set it to the label widget.

In this part of the PyQt5 tutorial, we covered several widgets.

PyQt5 widgets II

Here we will continue introducing PyQt5 widgets. We will cover a QPixmap, a QLineEdit, a QSplitter, and a QComboBox.

QPixmap

A QPixmap is one of the widgets used to work with images. It is optimized for showing images on screen. In our code example, we will use the QPixmap to display an image on the window.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

```
In this example, we display an image
on the window.
```

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

```
import sys
from PyQt5.QtWidgets import (QWidget, QHBoxLayout,
                             QLabel, QApplication)
from PyQt5.QtGui import QPixmap
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```
        hbox = QHBoxLayout(self)
        pixmap = QPixmap("redrock.png")
```

```

        lbl = QLabel(self)
        lbl.setPixmap(pixmap)

        hbox.addWidget(lbl)
        self.setLayout(hbox)

        self.move(300, 200)
        self.setWindowTitle('Red Rock')
        self.show()

if __name__ == '__main__':

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

```

In our example, we display an image on the window.

```
pixmap = QPixmap("redrock.png")
```

We create a QPixmap object. It takes the name of the file as a parameter.

```

lbl = QLabel(self)
lbl.setPixmap(pixmap)

```

We put the pixmap into the QLabel widget.

QLineEdit

A QLineEdit is a widget that allows to enter and edit a single line of plain text. There are undo and redo, cut and paste, and drag & drop functions available for the widget.

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

```

```
"""
```

```
ZetCode PyQt5 tutorial
```

```

This example shows text which
is entered in a QLineEdit

```

in a QLabel widget.

author: Jan Bodnar

website: zetcode.com

last edited: January 2015

"""

```
import sys
```

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

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```
        self.lbl = QLabel(self)  
        qle = QLineEdit(self)
```

```
        qle.move(60, 100)  
        self.lbl.move(60, 40)
```

```
        qle.textChanged[str].connect(self.onChanged)
```

```
        self.setGeometry(300, 300, 280, 170)  
        self.setWindowTitle('QLineEdit')  
        self.show()
```

```
    def onChanged(self, text):
```

```
        self.lbl.setText(text)  
        self.lbl.adjustSize()
```

```
if __name__ == '__main__':
```

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

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

This example shows a line edit widget and a label. The text that we key in the line edit is displayed immediately in the label widget.

```
qle = QLineEdit(self)
```

The QLineEdit widget is created.

```
qle.textChanged[str].connect(self.onChanged)
```

If the text in the line edit widget changes, we call the onChanged() method.

```
def onChanged(self, text):
```

```
    self.lbl.setText(text)
    self.lbl.adjustSize()
```

Inside the onChanged() method, we set the typed text to the label widget. We call the adjustSize() method to adjust the size of the label to the length of the text.

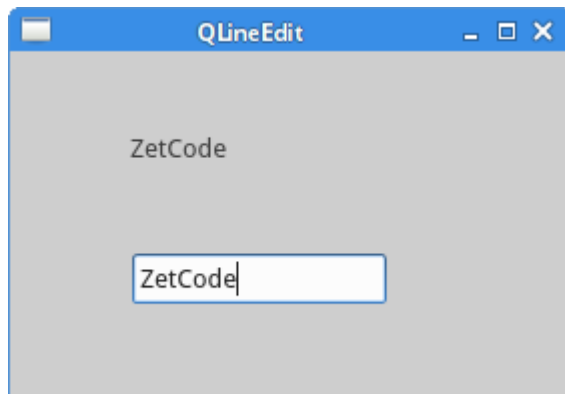


Figure: QLineEdit

QSplitter

A QSplitter lets the user control the size of child widgets by dragging the boundary between its children. In our example, we show three QFrame widgets organized with two splitters.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```



```
"""
```

```
ZetCode PyQt5 tutorial
```

```
This example shows  
how to use QSplitter widget.
```

```
author: Jan Bodnar
```

```
website: zetcode.com
```

```
last edited: January 2015
```

```
"""
```

```
import sys
```

```
from PyQt5.QtWidgets import (QWidget, QHBoxLayout, QFrame,  
                             QSplitter, QStyleFactory, QApplication)
```

```
from PyQt5.QtCore import Qt
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```
        hbox = QHBoxLayout(self)
```

```
        topleft = QFrame(self)  
        topleft.setFrameShape(QFrame.StyledPanel)
```

```
        topright = QFrame(self)  
        topright.setFrameShape(QFrame.StyledPanel)
```

```
        bottom = QFrame(self)  
        bottom.setFrameShape(QFrame.StyledPanel)
```

```
        splitter1 = QSplitter(Qt.Horizontal)  
        splitter1.addWidget(topleft)  
        splitter1.addWidget(topright)
```

```
        splitter2 = QSplitter(Qt.Vertical)
```

```

        splitter2.addWidget(splitter1)
        splitter2.addWidget(bottom)

        hbox.addWidget(splitter2)
        self.setLayout(hbox)

        self.setGeometry(300, 300, 300, 200)
        self.setWindowTitle('QSplitter')
        self.show()

    def onChanged(self, text):

        self.lbl.setText(text)
        self.lbl.adjustSize()

if __name__ == '__main__':

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

```

In our example, we have three frame widgets and two splitters. Note that under some themes, the splitters may not be visible very well.

```

topleft = QFrame(self)
topleft.setFrameShape(QFrame.StyledPanel)

```

We use a styled frame in order to see the boundaries between the QFrame widgets.

```

splitter1 = QSplitter(Qt.Horizontal)
splitter1.addWidget(topleft)
splitter1.addWidget(topright)

```

We create a QSplitter widget and add two frames into it.

```

splitter2 = QSplitter(Qt.Vertical)
splitter2.addWidget(splitter1)

```

We can also add a splitter to another splitter widget.

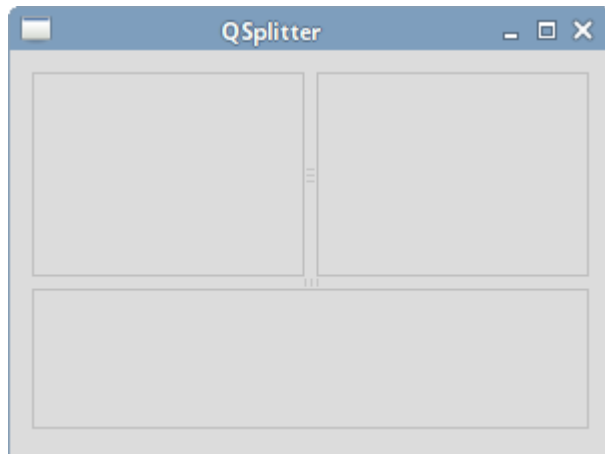


Figure: QSplitter widget

QComboBox

The QComboBox is a widget that allows a user to choose from a list of options.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

```
This example shows how to use
a QComboBox widget.
```

```
author: Jan Bodnar
```

```
website: zetcode.com
```

```
last edited: January 2015
```

```
"""
```

```
import sys
from PyQt5.QtWidgets import (QWidget, QLabel,
                             QComboBox, QApplication)
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```

def initUI(self):

    self.lbl = QLabel("Ubuntu", self)

    combo = QComboBox(self)
    combo.addItem("Ubuntu")
    combo.addItem("Mandriva")
    combo.addItem("Fedora")
    combo.addItem("Arch")
    combo.addItem("Gentoo")

    combo.move(50, 50)
    self.lbl.move(50, 150)

    combo.activated[str].connect(self.onActivated)

    self.setGeometry(300, 300, 300, 200)
    self.setWindowTitle('QComboBox')
    self.show()

def onActivated(self, text):

    self.lbl.setText(text)
    self.lbl.adjustSize()

if __name__ == '__main__':

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

```

The example shows a QComboBox and a QLabel. The combo box has a list of five options. These are the names of Linux distros. The label widget displays the selected option from the combo box.

```

combo = QComboBox(self)
combo.addItem("Ubuntu")
combo.addItem("Mandriva")
combo.addItem("Fedora")
combo.addItem("Arch")
combo.addItem("Gentoo")

```

We create a `QComboBox` widget with five options.

```
combo.activated[str].connect(self.onActivated)
```

Upon an item selection, we call the `onActivated()` method.

```
def onActivated(self, text):  
  
    self.lbl.setText(text)  
    self.lbl.adjustSize()
```

Inside the method, we set the text of the chosen item to the label widget. We adjust the size of the label.

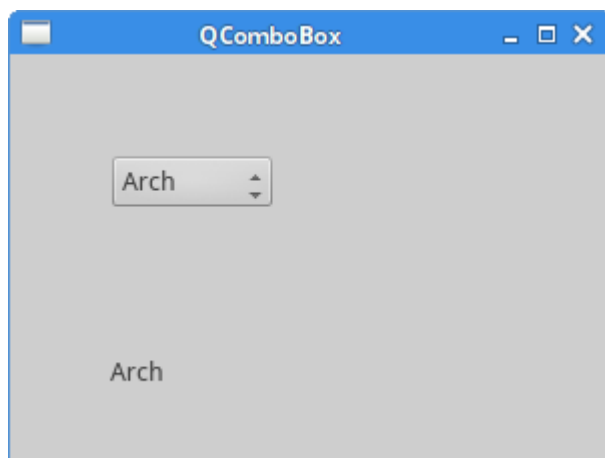


Figure: `QComboBox`

In this part of the PyQt5 tutorial, we covered other four PyQt5 widgets.

Drag and drop in PyQt5

In this part of the PyQt5 tutorial, we will talk about drag & drop operations.

In computer graphical user interfaces, drag-and-drop is the action of (or support for the action of) clicking on a virtual object and dragging it to a different location or onto another virtual object. In general, it can be used to invoke many kinds of actions, or create various types of associations between two abstract objects.

Drag and drop is part of the graphical user interface. Drag and drop operations enable users to do complex things intuitively.

Usually, we can drag and drop two things: data or some graphical objects. If we drag an image from one application to another, we drag and drop binary data. If we drag a tab in Firefox and move it to another place, we drag and drop a graphical component.

Simple drag and drop

In the first example, we have a QLineEdit and a QPushButton. We drag plain text from the line edit widget and drop it onto the button widget. The button's label will change.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

"""
ZetCode PyQt5 tutorial

This is a simple drag and
drop example.

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

import sys
from PyQt5.QtWidgets import (QPushButton, QWidget,
                             QLineEdit, QApplication)
```

```

class Button(QPushButton):

    def __init__(self, title, parent):
        super().__init__(title, parent)

        self.setAcceptDrops(True)

    def dragEnterEvent(self, e):

        if e.mimeData().hasFormat('text/plain'):
            e.accept()
        else:
            e.ignore()

    def dropEvent(self, e):

        self.setText(e.mimeData().text())

class Example(QWidget):

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

        self.initUI()

    def initUI(self):

        edit = QLineEdit('', self)
        edit.setDragEnabled(True)
        edit.move(30, 65)

        button = Button("Button", self)
        button.move(190, 65)

        self.setWindowTitle('Simple drag & drop')
        self.setGeometry(300, 300, 300, 150)

if __name__ == '__main__':

    app = QApplication(sys.argv)

```

```
ex = Example()
ex.show()
app.exec_()
```

The example presents a simple drag & drop operation.

```
class Button(QPushButton):

    def __init__(self, title, parent):
        super().__init__(title, parent)

        self.setAcceptDrops(True)
```

In order to drop text on the QPushButton widget, we must reimplement some methods. Therefore, we create our own Button class which will inherit from the QPushButton class.

```
self.setAcceptDrops(True)
```

We enable drop events for the widget.

```
def dragEnterEvent(self, e):

    if e.mimeData().hasFormat('text/plain'):
        e.accept()
    else:
        e.ignore()
```

First, we reimplement the dragEnterEvent() method. We inform about the data type that we accept. In our case it is plain text.

```
def dropEvent(self, e):

    self.setText(e.mimeData().text())
```

By reimplementing the dropEvent() method we define what we will do upon the drop event. Here we change the text of the button widget.

```
edit = QLineEdit('', self)
edit.setDragEnabled(True)
```

The QLineEdit widget has a built-in support for drag operations. All we need to do is to call the setDragEnabled() method to activate it.

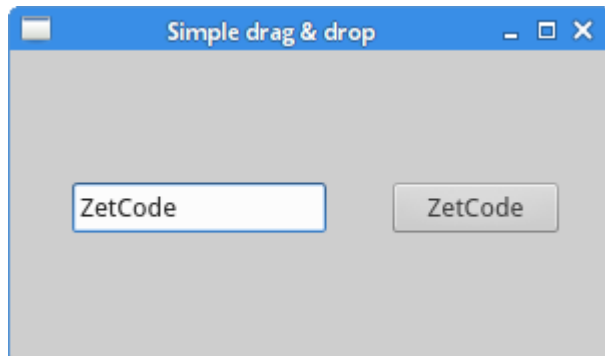


Figure: Simple drag & drop

Drag & drop a button widget

In the following example, we will demonstrate how to drag & drop a button widget.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

In this program, we can press on a button with a left mouse click or drag and drop the button with the right mouse click.

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

```
import sys
from PyQt5.QtWidgets import QPushButton, QWidget, QApplication
from PyQt5.QtCore import Qt, QMimeData
from PyQt5.QtGui import QDrag
```

```
class Button(QPushButton):
```

```
    def __init__(self, title, parent):
        super().__init__(title, parent)
```

```
    def mouseMoveEvent(self, e):
```

```

        if e.buttons() != Qt.RightButton:
            return

        mimeType = QMimeData()

        drag = QDrag(self)
        drag.setMimeData(mimeType)
        drag.setHotSpot(e.pos() - self.rect().topLeft())

        dropAction = drag.exec_(Qt.MoveAction)

def mousePressEvent(self, e):

    QPushButton.mousePressEvent(self, e)

    if e.button() == Qt.LeftButton:
        print('press')

class Example(QWidget):

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

        self.initUI()

    def initUI(self):

        self.setAcceptDrops(True)

        self.button = Button('Button', self)
        self.button.move(100, 65)

        self.setWindowTitle('Click or Move')
        self.setGeometry(300, 300, 280, 150)

    def dragEnterEvent(self, e):

        e.accept()

```

```

def dropEvent(self, e):

    position = e.pos()
    self.button.move(position)

    e.setDropAction(Qt.MoveAction)
    e.accept()

if __name__ == '__main__':

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

```

In our code example, we have a QPushButton on the window. If we click on the button with a left mouse button, the 'press' message is printed to the console. By right clicking and moving the button, we perform a drag & drop operation on the button widget.

```

class Button(QPushButton):

    def __init__(self, title, parent):
        super().__init__(title, parent)

```

We create a Button class which derives from the QPushButton. We also reimplement two methods of the QPushButton: the mouseMoveEvent() and the mousePressEvent(). The mouseMoveEvent() method is the place where the drag & drop operation begins.

```

if e.buttons() != Qt.RightButton:
    return

```

Here we decide that we can perform drag & drop only with a right mouse button. The left mouse button is reserved for clicking on the button.

```

mimeType = QMimeData()

drag = QDrag(self)
drag.setMimeData(mimeType)
drag.setHotSpot(e.pos() - self.rect().topLeft())

```

The QDrag object is created. The class provides support for MIME-based drag and drop data transfer.

```
dropAction = drag.exec_(Qt.MoveAction)
```

The start() method of the drag object starts the drag & drop operation.

```
def mousePressEvent(self, e):  
  
    QPushButton.mousePressEvent(self, e)  
  
    if e.button() == Qt.LeftButton:  
        print('press')
```

We print 'press' to the console if we left click on the button with the mouse. Notice that we call mousePressEvent() method on the parent as well. Otherwise, we would not see the button being pushed.

```
position = e.pos()  
self.button.move(position)
```

In the dropEvent() method we code what happens after we release the mouse button and finish the drop operation. We find out the current mouse pointer position and move the button accordingly.

```
e.setDropAction(Qt.MoveAction)  
e.accept()
```

We specify the type of the drop action. In our case it is a move action.

This part of the PyQt5 tutorial was dedicated to drag and drop operations.

Painting in PyQt5

PyQt5 painting system is able to render vector graphics, images, and outline font-based text. Painting is needed in applications when we want to change or enhance an existing widget, or if we are creating a custom widget from scratch. To do the drawing, we use the painting API provided by the PyQt5 toolkit.

The painting is done within the `paintEvent()` method. The painting code is placed between the `begin()` and `end()` methods of the `QPainter` object. It performs low-level painting on widgets and other paint devices.

Drawing text

We begin with drawing some Unicode text on the client area of a window.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

In this example, we draw text in Russian azbuka.

```
author: Jan Bodnar
```

```
website: zetcode.com
```

```
last edited: September 2015
```

```
"""
```

```
import sys
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QColor, QFont
from PyQt5.QtCore import Qt
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```

def initUI(self):

    self.text = u'\u041b\u0435\u0432\u0414\u0438\u0430\u0435\u043b\u0430\u0432\u0435\u0432\u0447\u0422\u0435\u043b\u0441\u0442\u0435\u0439: \n\u0410\u0434\u0430\u0430\u0435\u0434\u0438\u0434\u0438\u0430\u0430\u0430\u0434\u0435\u0434\u0438\u0434\u0430\u0430'

    self.setGeometry(300, 300, 280, 170)
    self.setWindowTitle('Draw text')
    self.show()

def paintEvent(self, event):

    qp = QPainter()
    qp.begin(self)
    self.drawText(event, qp)
    qp.end()

def drawText(self, event, qp):

    qp.setPen(QColor(168, 34, 3))
    qp.setFont(QFont('Decorative', 10))
    qp.drawText(event.rect(), Qt.AlignCenter, self.text)

```

```

if __name__ == '__main__':

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

```

In our example, we draw some text in Azbuka. The text is vertically and horizontally aligned.

```

def paintEvent(self, event):
...

```

Drawing is done within the paint event.

```

qp = QPainter()

```

```
qp.begin(self)
self.drawText(event, qp)
qp.end()
```

The QPainter class is responsible for all the low-level painting. All the painting methods go between begin() and end() methods. The actual painting is delegated to the drawText() method.

```
qp.setPen(QColor(168, 34, 3))
qp.setFont(QFont('Decorative', 10))
```

Here we define a pen and a font which are used to draw the text.

```
qp.drawText(event.rect(), Qt.AlignCenter, self.text)
```

The drawText() method draws text on the window. The rect() method of the paint event returns the rectangle that needs to be updated.

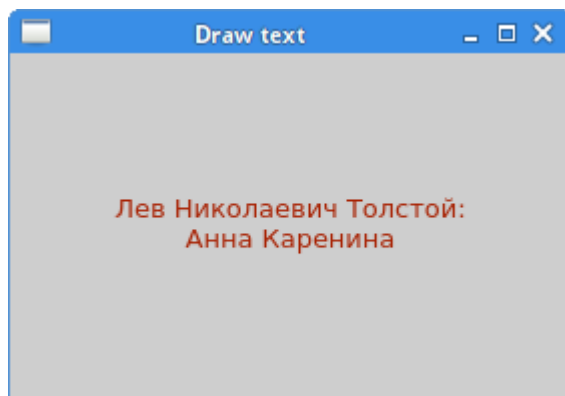


Figure: Drawing text

Drawing points

A point is the most simple graphics object that can be drawn. It is a small spot on the window.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

"""
ZetCode PyQt5 tutorial
```

In the example, we draw randomly 1000 red points on the window.

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

```
import sys, random
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QColor, QPen
from PyQt5.QtCore import Qt
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```
        self.setGeometry(300, 300, 280, 170)
        self.setWindowTitle('Points')
        self.show()
```

```
    def paintEvent(self, e):
```

```
        qp = QPainter()
        qp.begin(self)
        self.drawPoints(qp)
        qp.end()
```

```
    def drawPoints(self, qp):
```

```
        qp.setPen(Qt.red)
        size = self.size()
```

```
        for i in range(1000):
            x = random.randint(1, size.width()-1)
            y = random.randint(1, size.height()-1)
            qp.drawPoint(x, y)
```



```
if __name__ == '__main__':  
  
    app = QApplication(sys.argv)  
    ex = Example()  
    sys.exit(app.exec_())
```

In our example, we draw randomly 1000 red points on the client area of the window.

```
qp.setPen(Qt.red)
```

We set the pen to red colour. We use a predefined `Qt.red` colour constant.

```
size = self.size()
```

Each time we resize the window, a paint event is generated. We get the current size of the window with the `size()` method. We use the size of the window to distribute the points all over the client area of the window.

```
qp.drawPoint(x, y)
```

We draw the point with the `drawPoint()` method.

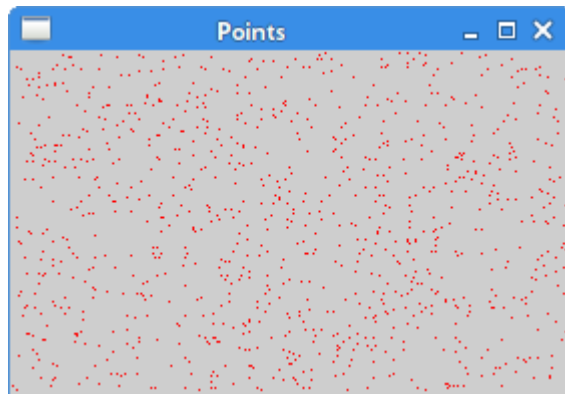


Figure: Points

Colours

A colour is an object representing a combination of Red, Green, and Blue (RGB) intensity values. Valid RGB values are in the range from 0 to 255. We can define a colour in various ways. The most common are RGB decimal values or hexadecimal values. We can also use an RGBA

value which stands for Red, Green, Blue, and Alpha. Here we add some extra information regarding transparency. Alpha value of 255 defines full opacity, 0 is for full transparency, e.g. the colour is invisible.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

"""
ZetCode PyQt5 tutorial

This example draws three rectangles in three
different colours.

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

import sys
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QColor, QBrush

class Example(QWidget):

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

        self.initUI()

    def initUI(self):

        self.setGeometry(300, 300, 350, 100)
        self.setWindowTitle('Colours')
        self.show()

    def paintEvent(self, e):

        qp = QPainter()
        qp.begin(self)
        self.drawRectangles(qp)
```

```

qp.end()

def drawRectangles(self, qp):

    col = QColor(0, 0, 0)
    col.setNamedColor('#d4d4d4')
    qp.setPen(col)

    qp.setBrush(QColor(200, 0, 0))
    qp.drawRect(10, 15, 90, 60)

    qp.setBrush(QColor(255, 80, 0, 160))
    qp.drawRect(130, 15, 90, 60)

    qp.setBrush(QColor(25, 0, 90, 200))
    qp.drawRect(250, 15, 90, 60)

if __name__ == '__main__':

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

```

In our example, we draw 3 coloured rectangles.

```

color = QColor(0, 0, 0)
color.setNamedColor('#d4d4d4')

```

Here we define a colour using a hexadecimal notation.

```

qp.setBrush(QColor(200, 0, 0))
qp.drawRect(10, 15, 90, 60)

```

Here we define a brush and draw a rectangle. A *brush* is an elementary graphics object which is used to draw the background of a shape. The `drawRect()` method accepts four parameters. The first two are x and y values on the axis. The third and fourth parameters are the width and height of the rectangle. The method draws the rectangle using the current pen and brush.

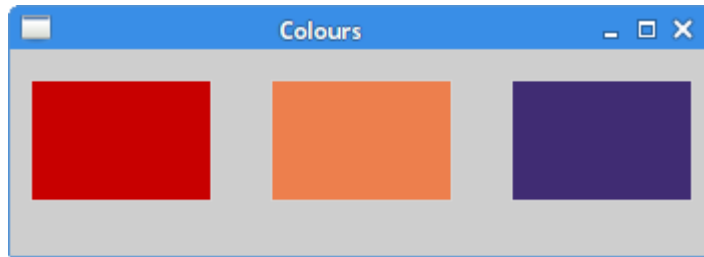


Figure: Colours

QPen

The QPen is an elementary graphics object. It is used to draw lines, curves and outlines of rectangles, ellipses, polygons, or other shapes.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

```
In this example we draw 6 lines using
different pen styles.
```

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

```
import sys
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QColor, QPen
from PyQt5.QtCore import Qt
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```

        self.setGeometry(300, 300, 280, 270)
        self.setWindowTitle('Pen styles')
        self.show()

def paintEvent(self, e):

    qp = QPainter()
    qp.begin(self)
    self.drawLines(qp)
    qp.end()

def drawLines(self, qp):

    pen = QPen(Qt.black, 2, Qt.SolidLine)

    qp.setPen(pen)
    qp.drawLine(20, 40, 250, 40)

    pen.setStyle(Qt.DashLine)
    qp.setPen(pen)
    qp.drawLine(20, 80, 250, 80)

    pen.setStyle(Qt.DashDotLine)
    qp.setPen(pen)
    qp.drawLine(20, 120, 250, 120)

    pen.setStyle(Qt.DotLine)
    qp.setPen(pen)
    qp.drawLine(20, 160, 250, 160)

    pen.setStyle(Qt.DashDotDotLine)
    qp.setPen(pen)
    qp.drawLine(20, 200, 250, 200)

    pen.setStyle(Qt.CustomDashLine)
    pen.setDashPattern([1, 4, 5, 4])
    qp.setPen(pen)
    qp.drawLine(20, 240, 250, 240)

if __name__ == '__main__':

```

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

In our example, we draw six lines. The lines are drawn in six different pen styles. There are five predefined pen styles. We can create also custom pen styles. The last line is drawn using a custom pen style.

```
pen = QPen(Qt.black, 2, Qt.SolidLine)
```

We create a QPen object. The colour is black. The width is set to 2 pixels so that we can see the differences between the pen styles. The Qt.SolidLine is one of the predefined pen styles.

```
pen.setStyle(Qt.CustomDashLine)
pen.setDashPattern([1, 4, 5, 4])
qp.setPen(pen)
```

Here we define a custom pen style. We set a Qt.CustomDashLine pen style and call the setDashPattern() method. The list of numbers defines a style. There must be an even number of numbers. Odd numbers define a dash, even numbers space. The greater the number, the greater the space or the dash. Our pattern is 1px dash, 4px space, 5px dash, 4px space etc.

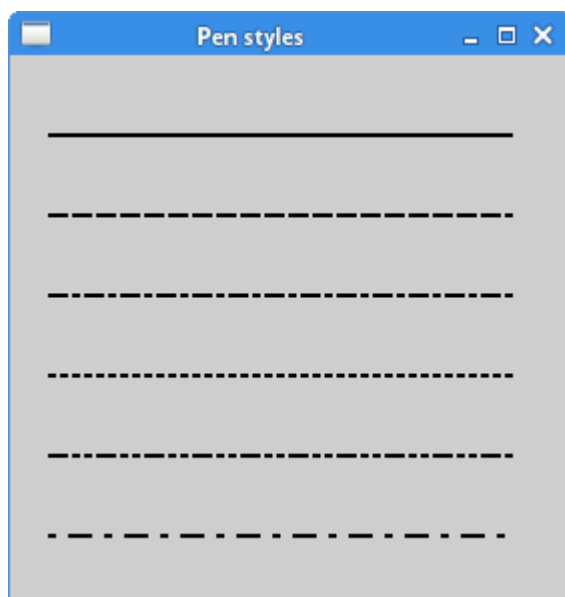


Figure: Pen styles

QBrush

The QBrush is an elementary graphics object. It is used to paint the background of graphics shapes, such as rectangles, ellipses, or polygons. A brush can be of three different types: a predefined brush, a gradient, or a texture pattern.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

This example draws 9 rectangles in different brush styles.

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

```
import sys
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QBrush
from PyQt5.QtCore import Qt
```

```
class Example(QWidget):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```
        self.setGeometry(300, 300, 355, 280)
        self.setWindowTitle('Brushes')
        self.show()
```

```
    def paintEvent(self, e):
```

```
qp = QPainter()
qp.begin(self)
self.drawBrushes(qp)
qp.end()
```

```
def drawBrushes(self, qp):

    brush = QBrush(Qt.SolidPattern)
    qp.setBrush(brush)
    qp.drawRect(10, 15, 90, 60)

    brush.setStyle(Qt.Dense1Pattern)
    qp.setBrush(brush)
    qp.drawRect(130, 15, 90, 60)

    brush.setStyle(Qt.Dense2Pattern)
    qp.setBrush(brush)
    qp.drawRect(250, 15, 90, 60)

    brush.setStyle(Qt.Dense3Pattern)
    qp.setBrush(brush)
    qp.drawRect(10, 105, 90, 60)

    brush.setStyle(Qt.DiagCrossPattern)
    qp.setBrush(brush)
    qp.drawRect(10, 105, 90, 60)

    brush.setStyle(Qt.Dense5Pattern)
    qp.setBrush(brush)
    qp.drawRect(130, 105, 90, 60)

    brush.setStyle(Qt.Dense6Pattern)
    qp.setBrush(brush)
    qp.drawRect(250, 105, 90, 60)

    brush.setStyle(Qt.HorPattern)
    qp.setBrush(brush)
    qp.drawRect(10, 195, 90, 60)

    brush.setStyle(Qt.VerPattern)
    qp.setBrush(brush)
    qp.drawRect(130, 195, 90, 60)
```



```
brush.setStyle(Qt.BDiagPattern)
qp.setBrush(brush)
qp.drawRect(250, 195, 90, 60)
```

```
if __name__ == '__main__':

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

In our example, we draw nine different rectangles.

```
brush = QBrush(Qt.SolidPattern)
qp.setBrush(brush)
qp.drawRect(10, 15, 90, 60)
```

We define a brush object. We set it to the painter object and draw the rectangle by calling the `drawRect()` method.

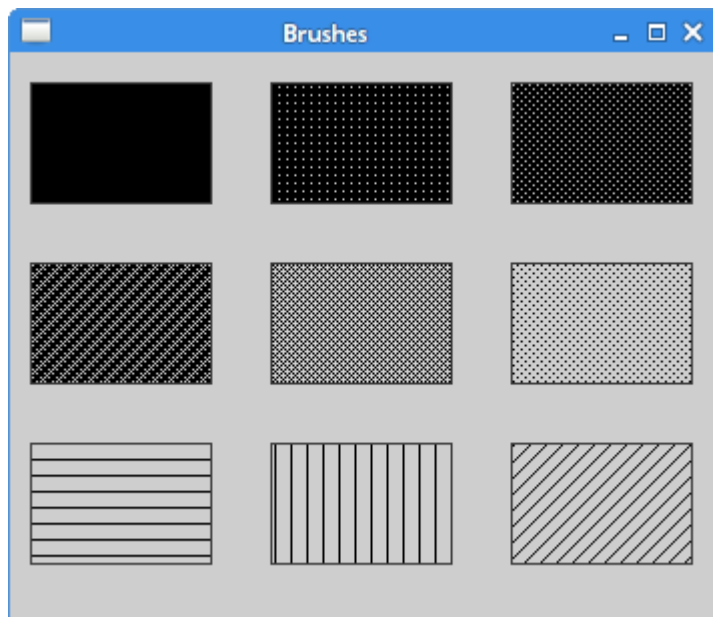


Figure: Brushes

In this part of the PyQt5 tutorial, we did some basic painting.

Custom widgets in PyQt5

PyQt5 has a rich set of widgets. However, no toolkit can provide all widgets that programmers might need in their applications. Toolkits usually provide only the most common widgets like buttons, text widgets, or sliders. If there is a need for a more specialised widget, we must create it ourselves.

Custom widgets are created by using the drawing tools provided by the toolkit. There are two basic possibilities: a programmer can modify or enhance an existing widget or he can create a custom widget from scratch.

Burning widget

This is a widget that we can see in Nero, K3B, or other CD/DVD burning software.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

In this example, we create a custom widget.

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

```
import sys
from PyQt5.QtWidgets import (QWidget, QSlider, QApplication,
                             QHBoxLayout, QVBoxLayout)
from PyQt5.QtCore import QObject, Qt, pyqtSignal
from PyQt5.QtGui import QPainter, QFont, QColor, QPen
```

```
class Communicate(QObject):
```

```
    updateBW = pyqtSignal(int)
```

```

class BurningWidget(QWidget):

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

        self.initUI()

    def initUI(self):

        self.setMinimumSize(1, 30)
        self.value = 75
        self.num = [75, 150, 225, 300, 375, 450, 525, 600, 675]

    def setValue(self, value):

        self.value = value

    def paintEvent(self, e):

        qp = QPainter()
        qp.begin(self)
        self.drawWidget(qp)
        qp.end()

    def drawWidget(self, qp):

        font = QFont('Serif', 7, QFont.Light)
        qp.setFont(font)

        size = self.size()
        w = size.width()
        h = size.height()

        step = int(round(w / 10.0))

        till = int(((w / 750.0) * self.value))
        full = int(((w / 750.0) * 700))

        if self.value >= 700:

```

```

qp.setPen(QColor(255, 255, 255))
qp.setBrush(QColor(255, 255, 184))
qp.drawRect(0, 0, full, h)
qp.setPen(QColor(255, 175, 175))
qp.setBrush(QColor(255, 175, 175))
qp.drawRect(full, 0, till-full, h)

```

```

else:

```

```

qp.setPen(QColor(255, 255, 255))
qp.setBrush(QColor(255, 255, 184))
qp.drawRect(0, 0, till, h)

```

```

pen = QPen(QColor(20, 20, 20), 1,
           Qt.SolidLine)

```

```

qp.setPen(pen)
qp.setBrush(Qt.NoBrush)
qp.drawRect(0, 0, w-1, h-1)

```

```

j = 0

```

```

for i in range(step, 10*step, step):

```

```

    qp.drawLine(i, 0, i, 5)
    metrics = qp.fontMetrics()
    fw = metrics.width(str(self.num[j]))
    qp.drawText(i-fw/2, h/2, str(self.num[j]))
    j = j + 1

```

```

class Example(QWidget):

```

```

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

```

```

        self.initUI()

```

```

    def initUI(self):

```

```

        sld = QSlider(Qt.Horizontal, self)

```

```

sld.setFocusPolicy(Qt.NoFocus)
sld.setRange(1, 750)
sld.setValue(75)
sld.setGeometry(30, 40, 150, 30)

self.c = Communicate()
self.wid = BurningWidget()
self.c.updateBW[int].connect(self.wid.setValue)

sld.valueChanged[int].connect(self.changeValue)
hbox = QHBoxLayout()
hbox.addWidget(self.wid)
vbox = QVBoxLayout()
vbox.addStretch(1)
vbox.addLayout(hbox)
self.setLayout(vbox)

self.setGeometry(300, 300, 390, 210)
self.setWindowTitle('Burning widget')
self.show()

def changeValue(self, value):

    self.c.updateBW.emit(value)
    self.wid.repaint()

if __name__ == '__main__':

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

```

In our example, we have a QSlider and a custom widget. A slider controls the custom widget. This widget shows graphically the total capacity of a medium and the free space available to us. The minimum value of our custom widget is 1, the maximum is 750. If we reach value 700, we begin drawing in red colour. This normally indicates overburning.

The burning widget is placed at the bottom of the window. This is achieved using one QHBoxLayout and one QVBoxLayout.

```

class BurningWidget(QWidget):

```

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

The burning widget is based on the QWidget widget.

```
self.setMinimumSize(1, 30)
```

We change the minimum size (height) of the widget. The default value is a bit small for us.

```
font = QFont('Serif', 7, QFont.Light)  
qp.setFont(font)
```

We use a smaller font than the default one. This better suits our needs.

```
size = self.size()  
w = size.width()  
h = size.height()
```

```
step = int(round(w / 10.0))
```

```
till = int(((w / 750.0) * self.value))  
full = int(((w / 750.0) * 700))
```

We draw the widget dynamically. The greater is the window, the greater is the burning widget and vice versa. That is why we must calculate the size of the widget onto which we draw the custom widget. The till parameter determines the total size to be drawn. This value comes from the slider widget. It is a proportion of the whole area. The full parameter determines the point where we begin to draw in red colour. Notice the use of floating point arithmetics to achieve greater precision in drawing.

The actual drawing consists of three steps. We draw the yellow or the red and yellow rectangle. Then we draw the vertical lines which divide the widget into several parts. Finally, we draw the numbers which indicate the capacity of the medium.

```
metrics = qp.fontMetrics()  
fw = metrics.width(str(self.num[j]))  
qp.drawText(i-fw/2, h/2, str(self.num[j]))
```

We use font metrics to draw the text. We must know the width of the text in order to center it around the vertical line.

```
def changeValue(self, value):  
  
    self.c.updateBW.emit(value)  
    self.wid.repaint()
```

When we move the slider, the `changeValue()` method is called. Inside the method, we send a custom `updateBW` signal with a parameter. The parameter is the current value of the slider. The value is later used to calculate the capacity of the Burning widget to be drawn. The custom widget is then repainted.

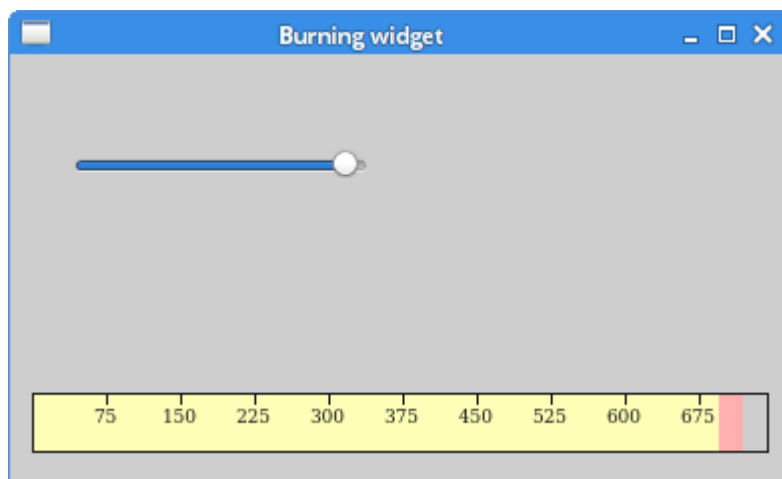


Figure: The burning widget

In this part of the PyQt5 tutorial, we created a custom widget.

The Tetris game in PyQt5

In this chapter, we will create a Tetris game clone.

Tetris

The Tetris game is one of the most popular computer games ever created. The original game was designed and programmed by a Russian programmer *Alexey Pajitnov* in 1985. Since then, Tetris is available on almost every computer platform in lots of variations.

Tetris is called a falling block puzzle game. In this game, we have seven different shapes called *tetrominoes*: an S-shape, a Z-shape, a T-shape, an L-shape, a Line-shape, a MirroredL-shape, and a Square-shape. Each of these shapes is formed with four squares. The shapes are falling down the board. The object of the Tetris game is to move and rotate the shapes so that they fit as much as possible. If we manage to form a row, the row is destroyed and we score. We play the Tetris game until we top out.



Figure: Tetrominoes

PyQt5 is a toolkit designed to create applications. There are other libraries which are targeted at creating computer games. Nevertheless, PyQt5 and other application toolkits can be used to create simple games.

Creating a computer game is a good way for enhancing programming skills.

The development

We do not have images for our Tetris game, we draw the tetrominoes using the drawing API available in the PyQt5 programming toolkit.

Behind every computer game, there is a mathematical model. So it is in Tetris.

Some ideas behind the game:

- We use a `QtCore.QBasicTimer()` to create a game cycle.
- The tetrominoes are drawn.
- The shapes move on a square by square basis (not pixel by pixel).
- Mathematically a board is a simple list of numbers.

The code consists of four classes: Tetris, Board, Tetrominoe and Shape. The Tetris class sets up the game. The Board is where the game logic is written. The Tetrominoe class contains names for all tetris pieces and the Shape class contains the code for a tetris piece.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```
"""
```

```
ZetCode PyQt5 tutorial
```

```
This is a Tetris game clone.
```

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

```
import sys, random
from PyQt5.QtWidgets import QMainWindow, QFrame, QDesktopWidget,
QApplication
from PyQt5.QtCore import Qt, QBasicTimer, pyqtSignal
from PyQt5.QtGui import QPainter, QColor
```

```
class Tetris(QMainWindow):
```

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

```
        self.initUI()
```

```
    def initUI(self):
```

```

        self.tboard = Board(self)
        self.setCentralWidget(self.tboard)

        self.statusbar = self.statusBar()

self.tboard.msg2Statusbar[str].connect(self.statusbar.showMessage)

        self.tboard.start()

        self.resize(180, 380)
        self.center()
        self.setWindowTitle('Tetris')
        self.show()

    def center(self):

        screen = QDesktopWidget().screenGeometry()
        size = self.geometry()
        self.move((screen.width()-size.width())/2,
                  (screen.height()-size.height())/2)

class Board(QFrame):

    msg2Statusbar = pyqtSignal(str)

    BoardWidth = 10
    BoardHeight = 22
    Speed = 300

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

        self.initBoard()

    def initBoard(self):

        self.timer = QBasicTimer()
        self.isWaitingAfterLine = False

        self.curX = 0

```

```

        self.curY = 0
        self.numLinesRemoved = 0
        self.board = []

        self.setFocusPolicy(Qt.StrongFocus)
        self.isStarted = False
        self.isPaused = False
        self.clearBoard()

    def shapeAt(self, x, y):
        return self.board[(y * Board.BoardWidth) + x]

    def setShapeAt(self, x, y, shape):
        self.board[(y * Board.BoardWidth) + x] = shape

    def squareWidth(self):
        return self.contentsRect().width() // Board.BoardWidth

    def squareHeight(self):
        return self.contentsRect().height() // Board.BoardHeight

    def start(self):

        if self.isPaused:
            return

        self.isStarted = True
        self.isWaitingAfterLine = False
        self.numLinesRemoved = 0
        self.clearBoard()

        self.msg2Statusbar.emit(str(self.numLinesRemoved))

        self.newPiece()
        self.timer.start(Board.Speed, self)

    def pause(self):

```

```

if not self.isStarted:
    return

self.isPaused = not self.isPaused

if self.isPaused:
    self.timer.stop()
    self.msg2Statusbar.emit("paused")

else:
    self.timer.start(Board.Speed, self)
    self.msg2Statusbar.emit(str(self.numLinesRemoved))

self.update()

def paintEvent(self, event):

    painter = QPainter(self)
    rect = self.contentsRect()

    boardTop = rect.bottom() - Board.BoardHeight * self.squareHeight()

    for i in range(Board.BoardHeight):
        for j in range(Board.BoardWidth):
            shape = self.shapeAt(j, Board.BoardHeight - i - 1)

            if shape != Tetrominoe.NoShape:
                self.drawSquare(painter,
                                rect.left() + j * self.squareWidth(),
                                boardTop + i * self.squareHeight(), shape)

    if self.curPiece.shape() != Tetrominoe.NoShape:

        for i in range(4):

            x = self.curX + self.curPiece.x(i)
            y = self.curY - self.curPiece.y(i)
            self.drawSquare(painter, rect.left() + x *
self.squareWidth(),
                            boardTop + (Board.BoardHeight - y - 1) *
self.squareHeight(),
                            self.curPiece.shape())

```

```

def keyPressEvent(self, event):

    if not self.isStarted or self.curPiece.shape() ==
Tetrominoe.NoShape:
        super(Board, self).keyPressEvent(event)
        return

    key = event.key()

    if key == Qt.Key_P:
        self.pause()
        return

    if self.isPaused:
        return

    elif key == Qt.Key_Left:
        self.tryMove(self.curPiece, self.curX - 1, self.curY)

    elif key == Qt.Key_Right:
        self.tryMove(self.curPiece, self.curX + 1, self.curY)

    elif key == Qt.Key_Down:
        self.tryMove(self.curPiece.rotateRight(), self.curX,
self.curY)

    elif key == Qt.Key_Up:
        self.tryMove(self.curPiece.rotateLeft(), self.curX, self.curY)

    elif key == Qt.Key_Space:
        self.dropDown()

    elif key == Qt.Key_D:
        self.oneLineDown()

    else:
        super(Board, self).keyPressEvent(event)

def timerEvent(self, event):

    if event.timerId() == self.timer.timerId():

```

```

        if self.isWaitingAfterLine:
            self.isWaitingAfterLine = False
            self.newPiece()
        else:
            self.oneLineDown()

    else:
        super(Board, self).timerEvent(event)

def clearBoard(self):

    for i in range(Board.BoardHeight * Board.BoardWidth):
        self.board.append(Tetrominoe.NoShape)

def dropDown(self):

    newY = self.curY

    while newY > 0:

        if not self.tryMove(self.curPiece, self.curX, newY - 1):
            break

        newY -= 1

    self.pieceDropped()

def oneLineDown(self):

    if not self.tryMove(self.curPiece, self.curX, self.curY - 1):
        self.pieceDropped()

def pieceDropped(self):

    for i in range(4):

        x = self.curX + self.curPiece.x(i)
        y = self.curY - self.curPiece.y(i)
        self.setShapeAt(x, y, self.curPiece.shape())

```

```

self.removeFullLines()

if not self.isWaitingAfterLine:
    self.newPiece()

def removeFullLines(self):

    numFullLines = 0
    rowsToRemove = []

    for i in range(Board.BoardHeight):

        n = 0
        for j in range(Board.BoardWidth):
            if not self.shapeAt(j, i) == Tetrominoe.NoShape:
                n = n + 1

        if n == 10:
            rowsToRemove.append(i)

    rowsToRemove.reverse()

    for m in rowsToRemove:

        for k in range(m, Board.BoardHeight):
            for l in range(Board.BoardWidth):
                self.setShapeAt(l, k, self.shapeAt(l, k + 1))

    numFullLines = numFullLines + len(rowsToRemove)

    if numFullLines > 0:

        self.numLinesRemoved = self.numLinesRemoved + numFullLines
        self.msg2StatusBar.emit(str(self.numLinesRemoved))

        self.isWaitingAfterLine = True
        self.curPiece.setShape(Tetrominoe.NoShape)
        self.update()

def newPiece(self):

```

```

self.curPiece = Shape()
self.curPiece.setRandomShape()
self.curX = Board.BoardWidth // 2 + 1
self.curY = Board.BoardHeight - 1 + self.curPiece.minY()

if not self.tryMove(self.curPiece, self.curX, self.curY):

    self.curPiece.setShape(Tetrominoe.NoShape)
    self.timer.stop()
    self.isStarted = False
    self.msg2StatusBar.emit("Game over")

def tryMove(self, newPiece, newX, newY):

    for i in range(4):

        x = newX + newPiece.x(i)
        y = newY - newPiece.y(i)

        if x < 0 or x >= Board.BoardWidth or y < 0 or y >=
Board.BoardHeight:
            return False

        if self.shapeAt(x, y) != Tetrominoe.NoShape:
            return False

    self.curPiece = newPiece
    self.curX = newX
    self.curY = newY
    self.update()

    return True

def drawSquare(self, painter, x, y, shape):

    colorTable = [0x000000, 0xCC6666, 0x66CC66, 0x6666CC,
                  0xCCCC66, 0xCC66CC, 0x66CCCC, 0xDAAA00]

    color = QColor(colorTable[shape])
    painter.fillRect(x + 1, y + 1, self.squareWidth() - 2,
                    self.squareHeight() - 2, color)

```



```

        painter.setPen(color.lighter())
        painter.drawLine(x, y + self.squareHeight() - 1, x, y)
        painter.drawLine(x, y, x + self.squareWidth() - 1, y)

        painter.setPen(color.darker())
        painter.drawLine(x + 1, y + self.squareHeight() - 1,
            x + self.squareWidth() - 1, y + self.squareHeight() - 1)
        painter.drawLine(x + self.squareWidth() - 1,
            y + self.squareHeight() - 1, x + self.squareWidth() - 1, y +
1)

```

```

class Tetrominoe(object):

```

```

    NoShape = 0
    ZShape = 1
    SShape = 2
    LineShape = 3
    TShape = 4
    SquareShape = 5
    LShape = 6
    MirroredLShape = 7

```

```

class Shape(object):

```

```

    coordsTable = (
        ((0, 0), (0, 0), (0, 0), (0, 0)),
        ((0, -1), (0, 0), (-1, 0), (-1, 1)),
        ((0, -1), (0, 0), (1, 0), (1, 1)),
        ((0, -1), (0, 0), (0, 1), (0, 2)),
        ((-1, 0), (0, 0), (1, 0), (0, 1)),
        ((0, 0), (1, 0), (0, 1), (1, 1)),
        ((-1, -1), (0, -1), (0, 0), (0, 1)),
        ((1, -1), (0, -1), (0, 0), (0, 1))
    )

```

```

    def __init__(self):

```

```

        self.coords = [[0,0] for i in range(4)]
        self.pieceShape = Tetrominoe.NoShape

        self.setShape(Tetrominoe.NoShape)

```

```

def shape(self):
    return self.pieceShape

def setShape(self, shape):

    table = Shape.coordsTable[shape]

    for i in range(4):
        for j in range(2):
            self.coords[i][j] = table[i][j]

    self.pieceShape = shape

def setRandomShape(self):
    self.setShape(random.randint(1, 7))

def x(self, index):
    return self.coords[index][0]

def y(self, index):
    return self.coords[index][1]

def setX(self, index, x):
    self.coords[index][0] = x

def setY(self, index, y):
    self.coords[index][1] = y

def minX(self):

    m = self.coords[0][0]
    for i in range(4):
        m = min(m, self.coords[i][0])

    return m

```

```
def maxX(self):
```

```
    m = self.coords[0][0]
    for i in range(4):
        m = max(m, self.coords[i][0])

    return m
```

```
def minY(self):
```

```
    m = self.coords[0][1]
    for i in range(4):
        m = min(m, self.coords[i][1])

    return m
```

```
def maxY(self):
```

```
    m = self.coords[0][1]
    for i in range(4):
        m = max(m, self.coords[i][1])

    return m
```

```
def rotateLeft(self):
```

```
    if self.pieceShape == Tetrominoe.SquareShape:
        return self

    result = Shape()
    result.pieceShape = self.pieceShape

    for i in range(4):

        result.setX(i, self.y(i))
        result.setY(i, -self.x(i))

    return result
```

```

def rotateRight(self):

    if self.pieceShape == Tetrominoe.SquareShape:
        return self

    result = Shape()
    result.pieceShape = self.pieceShape

    for i in range(4):

        result.setX(i, -self.y(i))
        result.setY(i, self.x(i))

    return result

if __name__ == '__main__':

    app = QApplication([])
    tetris = Tetris()
    sys.exit(app.exec_())

```

The game is simplified a bit so that it is easier to understand. The game starts immediately after it is launched. We can pause the game by pressing the p key. The Space key will drop the tetris piece instantly to the bottom. The game goes at constant speed, no acceleration is implemented. The score is the number of lines that we have removed.

```

self.tboard = Board(self)
self.setCentralWidget(self.tboard)

```

An instance of the Board class is created and set to be the central widget of the application.

```

self.statusbar = self.statusBar()
self.tboard.msg2Statusbar[str].connect(self.statusbar.showMessage)

```

We create a statusbar where we will display messages. We will display three possible messages: the number of lines already removed, the paused message, or the game over message. The msg2Statusbar is a custom signal that is implemented in the Board class. The showMessage() is a built-in method that displays a message on a statusbar.

```
self.tboard.start()
```

This line initiates the game.

```
class Board(QFrame):
```

```
    msg2Statusbar = pyqtSignal(str)
```

```
    ...
```

A custom signal is created. The msg2Statusbar is a signal that is emitted when we want to write a message or the score to the statusbar.

```
BoardWidth = 10
```

```
BoardHeight = 22
```

```
Speed = 300
```

These are Board's class variables. The BoardWidth and the BoardHeight define the size of the board in blocks. The Speed defines the speed of the game. Each 300 ms a new game cycle will start.

```
    ...
```

```
    self.curX = 0
```

```
    self.curY = 0
```

```
    self.numLinesRemoved = 0
```

```
    self.board = []
```

```
    ...
```

In the initBoard() method we initialize some important variables. The self.board variable is a list of numbers from 0 to 7. It represents the position of various shapes and remains of the shapes on the board.

```
def shapeAt(self, x, y):
```

```
    return self.board[(y * Board.BoardWidth) + x]
```

The shapeAt() method determines the type of a shape at a given block.

```
def squareWidth(self):
```

```
    return self.contentsRect().width() // Board.BoardWidth
```

The board can be dynamically resized. As a consequence, the size of a block may change. The squareWidth() calculates the width of the single square in pixels and returns it. The Board.BoardWidth is the size of the board in blocks.

```

for i in range(Board.BoardHeight):
    for j in range(Board.BoardWidth):
        shape = self.shapeAt(j, Board.BoardHeight - i - 1)

        if shape != Tetrominoe.NoShape:
            self.drawSquare(painter,
                            rect.left() + j * self.squareWidth(),
                            boardTop + i * self.squareHeight(), shape)

```

The painting of the game is divided into two steps. In the first step, we draw all the shapes, or remains of the shapes that have been dropped to the bottom of the board. All the squares are remembered in the `self.board` list variable. The variable is accessed using the `shapeAt()` method.

```

if self.curPiece.shape() != Tetrominoe.NoShape:

    for i in range(4):

        x = self.curX + self.curPiece.x(i)
        y = self.curY - self.curPiece.y(i)
        self.drawSquare(painter, rect.left() + x * self.squareWidth(),
                        boardTop + (Board.BoardHeight - y - 1) * self.squareHeight(),
                        self.curPiece.shape())

```

The next step is the drawing of the actual piece that is falling down.

```

elif key == Qt.Key_Right:
    self.tryMove(self.curPiece, self.curX + 1, self.curY)

```

In the `keyPressEvent()` method we check for pressed keys. If we press the right arrow key, we try to move the piece to the right. We say try because the piece might not be able to move.

```

elif key == Qt.Key_Up:
    self.tryMove(self.curPiece.rotateLeft(), self.curX, self.curY)

```

The Up arrow key will rotate the falling piece to the left.

```

elif key == Qt.Key_Space:
    self.dropDown()

```

The Space key will drop the falling piece instantly to the bottom.

```
elif key == Qt.Key_D:  
    self.oneLineDown()
```

Pressing the d key, the piece will go one block down. It can be used to accelerate the falling of a piece a bit.

```
def tryMove(self, newPiece, newX, newY):  
  
    for i in range(4):  
  
        x = newX + newPiece.x(i)  
        y = newY - newPiece.y(i)  
  
        if x < 0 or x >= Board.BoardWidth or y < 0 or y >= Board.BoardHeight:  
            return False  
  
        if self.shapeAt(x, y) != Tetrominoe.NoShape:  
            return False  
  
    self.curPiece = newPiece  
    self.curX = newX  
    self.curY = newY  
    self.update()  
    return True
```

In the tryMove() method we try to move our shapes. If the shape is at the edge of the board or is adjacent to some other piece, we return False. Otherwise we place the current falling piece to a new position.

```
def timerEvent(self, event):  
  
    if event.timerId() == self.timer.timerId():  
  
        if self.isWaitingAfterLine:  
            self.isWaitingAfterLine = False  
            self.newPiece()  
        else:  
            self.oneLineDown()  
  
    else:  
        super(Board, self).timerEvent(event)
```

In the timer event, we either create a new piece after the previous one was dropped to the bottom or we move a falling piece one line down.

```
def clearBoard(self):
```

```
    for i in range(Board.BoardHeight * Board.BoardWidth):
        self.board.append(Tetrominoe.NoShape)
```

The `clearBoard()` method clears the board by setting `Tetrominoe.NoShape` at each block of the board.

```
def removeFullLines(self):
```

```
    numFullLines = 0
    rowsToRemove = []
```

```
    for i in range(Board.BoardHeight):
```

```
        n = 0
        for j in range(Board.BoardWidth):
            if not self.shapeAt(j, i) == Tetrominoe.NoShape:
                n = n + 1
```

```
        if n == 10:
            rowsToRemove.append(i)
```

```
    rowsToRemove.reverse()
```

```
    for m in rowsToRemove:
```

```
        for k in range(m, Board.BoardHeight):
            for l in range(Board.BoardWidth):
                self.setShapeAt(l, k, self.shapeAt(l, k + 1))
```

```
    numFullLines = numFullLines + len(rowsToRemove)
```

```
    ...
```

If the piece hits the bottom, we call the `removeFullLines()` method. We find out all full lines and remove them. We do it by moving all lines above the current full line to be removed one line down. Notice that we reverse the order of the lines to be removed. Otherwise, it would not work correctly. In our case we use a *naive gravity*. This means that the pieces may be floating above empty gaps.


```

def newPiece(self):

    self.curPiece = Shape()
    self.curPiece.setRandomShape()
    self.curX = Board.BoardWidth // 2 + 1
    self.curY = Board.BoardHeight - 1 + self.curPiece.minY()

    if not self.tryMove(self.curPiece, self.curX, self.curY):

        self.curPiece.setShape(Tetrominoe.NoShape)
        self.timer.stop()
        self.isStarted = False
        self.msg2Statusbar.emit("Game over")

```

The newPiece() method creates randomly a new tetris piece. If the piece cannot go into its initial position, the game is over.

```

class Tetrominoe(object):

```

```

    NoShape = 0
    ZShape = 1
    SShape = 2
    LineShape = 3
    TShape = 4
    SquareShape = 5
    LShape = 6
    MirroredLShape = 7

```

The Tetrominoe class holds names of all possible shapes. We have also a NoShape for an empty space.

The Shape class saves information about a tetris piece.

```

class Shape(object):

```

```

    coordsTable = (
        ((0, 0), (0, 0), (0, 0), (0, 0)),
        ((0, -1), (0, 0), (-1, 0), (-1, 1)),
        ...
    )
    ...

```

The coordsTable tuple holds all possible coordinate values of our tetris pieces. This is a template from which all pieces take their coordinate values.

```
self.coords = [[0,0] for i in range(4)]
```

Upon creation we create an empty coordinates list. The list will save the coordinates of the tetris piece.

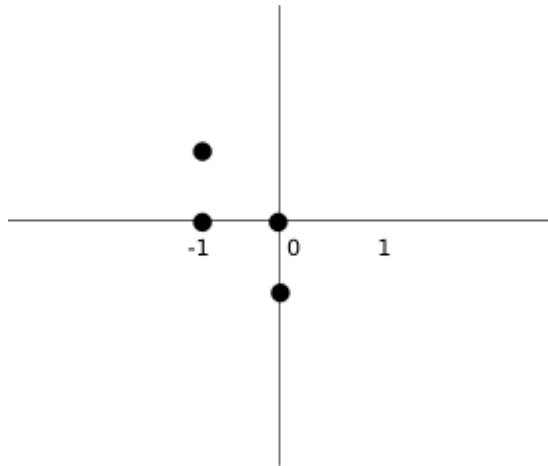


Figure: Coordinates

The above image will help understand the coordinate values a bit more. For example, the tuples (0, -1), (0, 0), (-1, 0), (-1, -1) represent a Z-shape. The diagram illustrates the shape.

```
def rotateLeft(self):  
  
    if self.pieceShape == Tetrominoe.SquareShape:  
        return self  
  
    result = Shape()  
    result.pieceShape = self.pieceShape  
  
    for i in range(4):  
  
        result.setX(i, self.y(i))  
        result.setY(i, -self.x(i))  
  
    return result
```

The rotateLeft() method rotates a piece to the left. The square does not have to be rotated. That is why we simply return the reference to the current object. A new piece is created and its coordinates are set to the ones of the rotated piece.

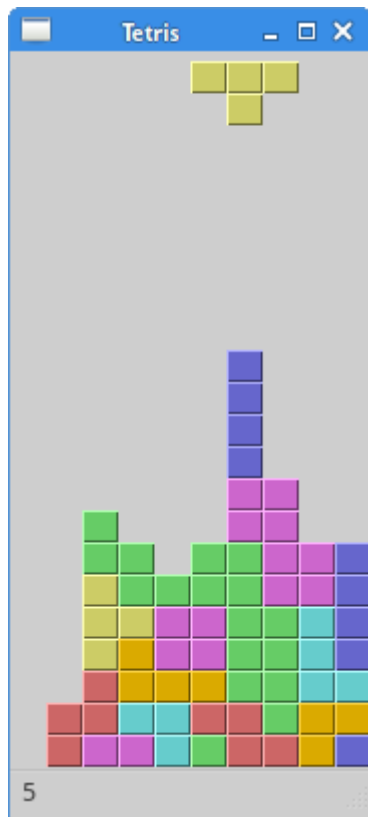


Figure: Tetris

This was a Tetris game in PyQt5.