troduction to PyQt4 toolkit

t this tutorial

ls an introductory PyQt4 tutorial. The purpose of this tutorial is to get carted with the PyQt4 toolkit. The tutorial has been created and tested on .

t PyQt

ls a toolkit for creating GUI applications. It is a blending of python amming language and the successfull Qt library. Qt library is one of the powerful libraries on this planet. If not the most powerful. The official site for PyQt is on www.riverbankcomputing.co.uk It was developed by **Phil son.**

ls implemented as a set of python modules. It has over 300 classes and : 6000 functions and methods. It is a multiplatform toolkit. It runs on all operating systems. Including Unix, Windows and Mac. PyQt is dual licenced. pers can choose between GPL and commercial licence. Previously, GPL on was available only on Unix. Starting from PyQt version 4, GPL licence is able on all supported platforms.

se there are a lot of classes available, they have been divided into several modules.

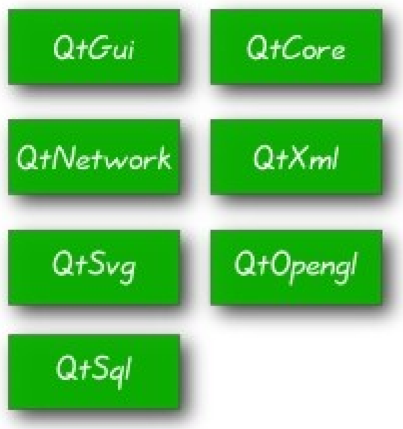


Figure: PyQt4 Modules

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** module contains the graphical components and related classes. These include for example buttons, windows, status bars, toolbars, sliders, bitmaps, colors, fonts etc. The **QtNetwork** module contains the classes for network programming. These classes allow to write TCP/IP and UDP clients and servers. They make the network programming easier and more portable. 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 **QtOpenGL** module is used for rendering 3D and 2D graphics using the OpenGL library. The module enables seamless integration of the Qt GUI libary and the OpenGL library. The **QtSql** module provides classes for working with databases.

Python

Python is a successful scripting language. It was initially developed by **Guido van Rossum**. It was first released in 1991. Python was inspired by ABC and Haskell programming languages. Python is a high level, general purpose, multiplatform, interpreted language. Some prefer to call it a dynamic language. It is easy to learn. Python is a minimalistic language. One of it's most visible features is that it does not use semicolons nor brackets. Python uses intendation instead. The most recent version of python is 2.5, which was released in September 2006. Today, Python is maintained by a large group of volunteers worldwide.

The TIOBE Programming Community Index gives us a theoretical usage of various programming languages. Java rules. The C++ language is detroned. But C++ will continue to be used in it's footholds for the coming decades and ther seems to be no real threat for it. We can clearly see specialization among programming languages. Java is used mainly in enterprise projects and portables, C is the king in system programming (OS, device drivers, small apps), PHP rules among small to medium size web sites, Javasript is used on the client site of a web application.

Position	Language	Ratings
1	Java	21.7%
2	C	14.9%
3	Visual Basic	10.7%
4	PHP	10.2%
5	C++	9.9%
6	Perl	5.4%
7	C#	3.4%
8	Python	3.0%
9	JavaScript	2.7%
10	Ruby	2.0%

Python is currently number 8. The Ruby language has made into the toplist. The closest competitors to Python are Ruby and Perl.

Python toolkits

For creating graphical user interfaces, python programmers can choose among three decent options. PyGTK, wxPython and PyQt. Which toolkit to choose, depends on the circumstances. There is also another "option", called TkInter. Avoid.

First programs in PyQt4 toolkit

In this part of the PyQt4 tutorial we will learn some basic functionality. The explanation will be slow, as if we would talk to a child. The first steps of a child are awkward, so are the very first attempts of a newbie programmer. Remember, there are no stupid people. There are only lazy people and people, that are not persistent enough.

Simple example

The code example is very simplistic. It only shows a small window. Yet we can do a lot with this window. We can resize it. Maximize it. Minimize 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 So it has been hidden from a programmer. PyQt is a high level toolkit. If we would code in a lower level toolkit, the following code example could easily have dozens of lines.

```
#!/usr/bin/python

# simple.py

import sys
from PyQt4 import QtGui

app = QtGui.QApplication(sys.argv)

widget = QtGui.QWidget()
widget.resize(250, 150)
widget.setWindowTitle('simple')
widget.show()

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

```
 import sys
 from PyQt4 import QtGui
```

Here we provide the necessary imports. The basic GUI widgets are located in *QtGui* module.

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

Every PyQt4 application must create an application object. The application object is located in the QtGui module. The *sys.argv* parameter is a list of arguments from the command line. Python scripts can be run from the shell. It is a way, how we can control the startup of our scripts.

```
widget = QtGui.QWidget()
```

The QWidget widget is the base class of all user interface objects in PyQt4. We provide the default constructor for QWidget. The default constructor has no parent. A widget with no parent is called a window.

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

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

```
widget.setWindowTitle('simple')
```

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

```
widget.show()
```

The *show()* method displays the widget 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.

You wonder why the exec_() method has the underscore? Everything has a meaning. This is obviously because the exec is a python keyword. And thus, exec_() was used instead.

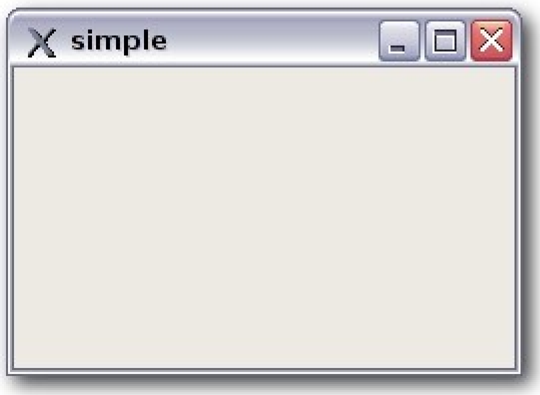


Figure: Simple

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 show, how we do it in PyQt4. We will also introduce some new methods.

```
#!/usr/bin/python

# icon.py

import sys
from PyQt4 import QtGui


class Icon(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Icon')
        self.setWindowIcon(QtGui.QIcon('icons/web.png'))


app = QtGui.QApplication(sys.argv)
icon = Icon()
icon.show()
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 PyQt4 means programming in OOP.

```
 class Icon(QtGui.QWidget):
     def __init__(self, parent=None):
         QtGui.QWidget.__init__(self, parent)
```

The three most important things in object oriented programming are classes, data and methods. Here we create a new class called Icon. The Icon class inherits from QtGui.QWidget class. This means, that we must call two constructors. The first one for the Icon class and the second one for the inherited class.

```
 self.setGeometry(300, 300, 250, 150)
 self.setWindowTitle('Icon')
 self.setWindowIcon(QtGui.QIcon('icons/web.png'))
```

All three classes have been inherited from the QtGui.QWidget class. The *setGeometry()* does two things. It locates the window on the screen and sets the size of the window. 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. 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.

Figure: Icon

Showing a tooltip

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

```
#!/usr/bin/python

# tooltip.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore


class Tooltip(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Tooltip')

        self.setToolTip('This is a <b>QWidget</b> widget')
        QtGui.QToolTip.setFont(QtGui.QFont('OldEnglish', 10))


app = QtGui.QApplication(sys.argv)
tooltip = Tooltip()
tooltip.show()
sys.exit(app.exec_())
```

In this example, we show a tooltip for a *QWidget* widget.

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

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

```
 QtGui.QToolTip.setFont(QtGui.QFont('OldEnglish', 10))
```

Because the default *QToolTip* font looks bad, we change it.

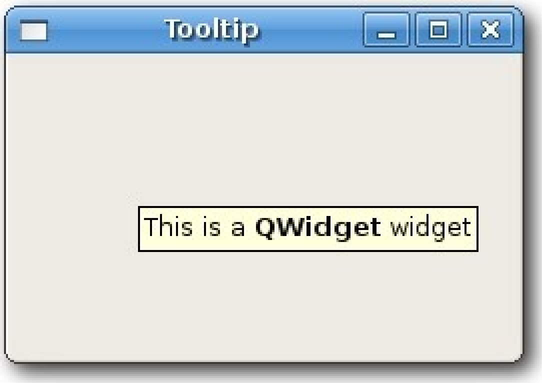


Figure: Tooltip

Closing a window

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

The following is the constructor of a QPushButton, that we will 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 the ancestor, onto which we place our button. In our case it is QWidget.

```
#!/usr/bin/python

# quitbutton.py

import sys
from PyQt4 import QtGui, QtCore


class QuitButton(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Icon')

        quit = QtGui.QPushButton('Close', self)
        quit.setGeometry(10, 10, 60, 35)

        self.connect(quit, QtCore.SIGNAL('clicked()'),
            QtGui.qApp, QtCore.SLOT('quit()'))


app = QtGui.QApplication(sys.argv)
qb = QuitButton()
qb.show()
sys.exit(app.exec_())
```

```
 quit = QtGui.QPushButton('Close', self)
 quit.setGeometry(10, 10, 60, 35)
```

We create a push button and position it on the QWidget just like we have positioned the QWidget on the screen.

```
 self.connect(quit, QtCore.SIGNAL('clicked()'),
     QtGui.qApp, QtCore.SLOT('quit()'))
```

The event processing system in PyQt4 is built with the signal & slot mechanism. If we click on the button, the signal *clicked()* is emitted. The slot can be a PyQt slot or any python callable. The *QtCore.QObject.connect()* method connects signals with slots. In our case the slot is a predefined PyQt *quit()* slot. The communication is done between two objects. The sender and the receiver. The sender is the push button, the receiver is the application object.

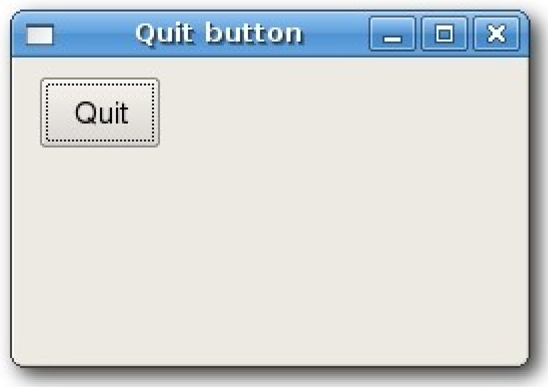


Figure: quit button

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/python

# messagebox.py

import sys
from PyQt4 import QtGui


class MessageBox(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('message box')


    def closeEvent(self, event):
        reply = QtGui.QMessageBox.question(self, 'Message',
            "Are you sure to quit?", QtGui.QMessageBox.Yes,
QtGui.QMessageBox.No)
```

```
        if reply == QtGui.QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()

app = QtGui.QApplication(sys.argv)
qb = MessageBox()
qb.show()
sys.exit(app.exec_())
```

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

```
 reply = QtGui.QMessageBox.question(self, 'Message',
     "Are you sure to quit?", QtGui.QMessageBox.Yes, QtGui.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 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 clicked 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.

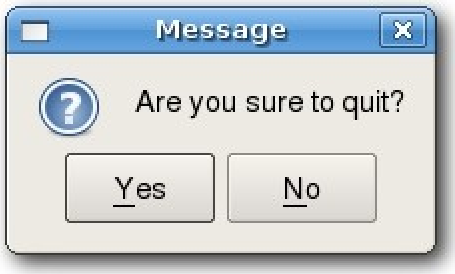


Figure: message box

Centering window on the screen

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

```
#!/usr/bin/python

# center.py

import sys
from PyQt4 import QtGui


class Center(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle('center')
        self.resize(250, 150)
        self.center()

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


app = QtGui.QApplication(sys.argv)
qb = Center()
qb.show()
sys.exit(app.exec_())
```

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

Here we resize the QWidget to be 250px wide and 150px heigh.

```
 screen = QtGui.QDesktopWidget().screenGeometry()
```

We figure out the screen resolution of our monitor.

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

Here we get the size of our QWidget.

```
 self.move((screen.width()-size.width())/2, (screen.height()-size.height())/2)
```

Here we move the window to the center of the screen.

Menus and Toolbars in PyQt4

Main Window

The QMainWindow class provides a main application window. This enables to create the classic application skeleton with a statusbar, toolbars and a menubar.

Statusbar

The statusbar is a widget that si used for displaying status information.

```
#!/usr/bin/python

# statusbar.py

import sys
from PyQt4 import QtGui

class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.resize(250, 150)
        self.setWindowTitle('statusbar')

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


app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

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

To get the statusbar, we call the *statusBar()* method of the QApplication class. The *showMessage()* displays message on the statusbar.

Menubar

A menubar is one of the most visible parts of the GUI application. It is a group of commands located in various menus. While in console applications you had to remember all those arcane commands, here we have most of the commands grouped into logical parts. There are accepted standards that further reduce the amount of time spending to learn a new application.

```
#!/usr/bin/python

# menubar.py

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)

        self.resize(250, 150)
        self.setWindowTitle('menubar')

        exit = QtGui.QAction(QtGui.QIcon('icons/exit.png'), 'Exit', self)
        exit.setShortcut('Ctrl+Q')
        exit.setStatusTip('Exit application')
        self.connect(exit, QtCore.SIGNAL('triggered()'), QtCore.SLOT('close()'))

        self.statusBar()

        menubar = self.menuBar()
        file = menubar.addMenu('&File')
        file.addAction(exit)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

```
 menubar = self.menuBar()
 file = menubar.addMenu('&File')
 file.addAction(exit)
```

First we create a menubar with the *menuBar()* method of the *QMainWindow* class. Then we add a menu with the *AddMenu()* method. In the end we plug the action object into the file menu.

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/python

# toolbar.py

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)

        self.resize(250, 150)
        self.setWindowTitle('toolbar')

        self.exit = QtGui.QAction(QtGui.QIcon('icons/exit.png'), 'Exit', self)
        self.exit.setShortcut('Ctrl+Q')
        self.connect(self.exit, QtCore.SIGNAL('triggered()'),
QtCore.SLOT('close()'))

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


app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

```
 self.exit = QtGui.QAction(QtGui.QIcon('icons/exit.png'), 'Exit', self)
 self.exit.setShortcut('Ctrl+Q')
```

GUI applications are controlled with commands. These commands can be launched from a menu, a context menu, a toolbar or with a shortcut. PyQt simplifies development with the introduction of **actions**. An action object can have menu text, an icon, a shortcut, status text, "What's This?" text and a tooltip. In our example, we define an action object with an icon, a tooltip and a shortcut.

```
 self.connect(self.exit, QtCore.SIGNAL('triggered()'), QtCore.SLOT('close()'))
```

Here we connect the action's *triggered()* signal to the predefined *close()* signal.

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

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

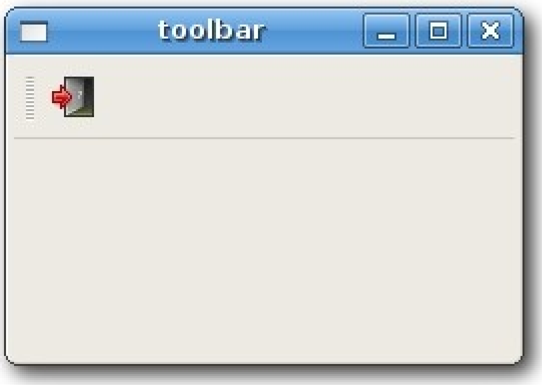


Figure: toolbar

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/python

# mainwindow.py

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)

        self.resize(350, 250)
        self.setWindowTitle('mainwindow')

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

        exit = QtGui.QAction(QtGui.QIcon('icons/exit.png'), 'Exit', self)
        exit.setShortcut('Ctrl+Q')
        exit.setStatusTip('Exit application')
        self.connect(exit, QtCore.SIGNAL('triggered()'), QtCore.SLOT('close()'))

        self.statusBar()

        menubar = self.menuBar()
        file = menubar.addMenu('&File')
        file.addAction(exit)

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


app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

```
 textEdit = QtGui.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.

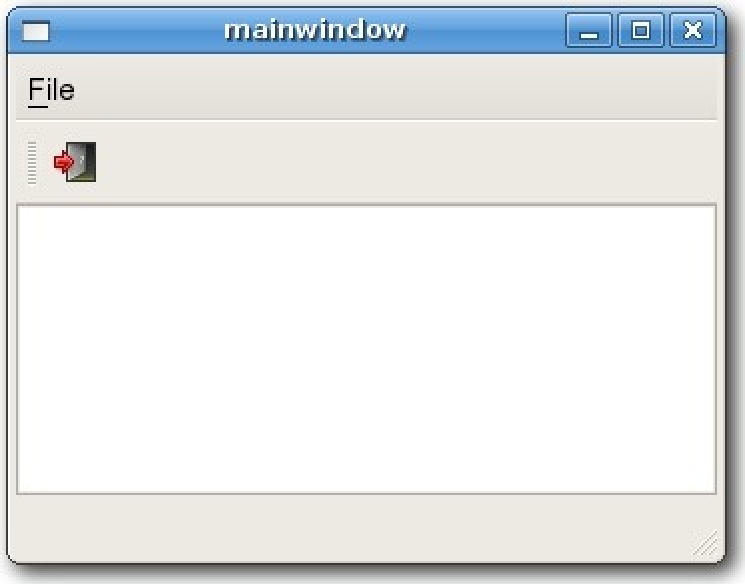


Figure: mainwindow

Layout management in PyQt4

Important thing in programming is the layout management. Layout management is the way how we place the widgets on the window. The management can be done in two 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, you have to understand several things.

- the size and the position of a widget do not change, if you resize a window
- applications might look different on various platforms
- changing fonts in your application might spoil the layout
- if you decide to change your layout, you must completely redo your layout, which is tedious and time consuming

```
#!/usr/bin/python

# absolute.py

import sys
```

```
from PyQt4 import QtGui


class Absolute(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('Communication')

        label = QtGui.QLabel('Couldn\'t', self)
        label.move(15, 10)

        label = QtGui.QLabel('care', self)
        label.move(35, 40)

        label = QtGui.QLabel('less', self)
        label.move(55, 65)

        label = QtGui.QLabel('And', self)
        label.move(115, 65)

        label = QtGui.QLabel('then', self)
        label.move(135, 45)

        label = QtGui.QLabel('you', self)
        label.move(115, 25)

        label = QtGui.QLabel('kissed', self)
        label.move(145, 10)

        label = QtGui.QLabel('me', self)
        label.move(215, 10)

        self.resize(250, 150)

app = QtGui.QApplication(sys.argv)
qb = Absolute()
qb.show()
sys.exit(app.exec_())
```

We simply call the *move()* method to position our widgets. In our case QLabel-s. We position them by providing the x and the 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.

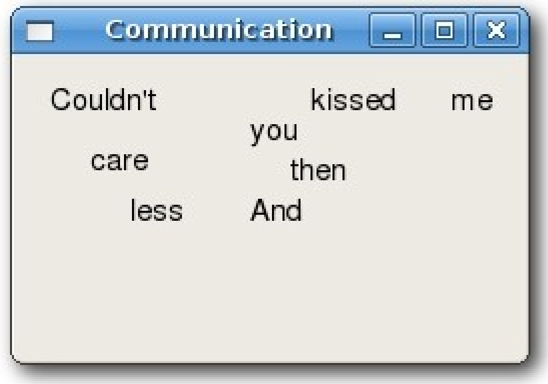


Figure: absolute positioning

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 basic layout classes are **QHBoxLayout** and **QVBoxLayout**. They 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 neccessary space, we will add a **stretch factor**.

```
#!/usr/bin/python

# boxlayout.py

import sys
from PyQt4 import QtGui


class BoxLayout(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('box layout')

        ok = QtGui.QPushButton("OK")
        cancel = QtGui.QPushButton("Cancel")

        hbox = QtGui.QHBoxLayout()
        hbox.addStretch(1)
        hbox.addWidget(ok)
        hbox.addWidget(cancel)

        vbox = QtGui.QVBoxLayout()
        vbox.addStretch(1)
        vbox.addLayout(hbox)

        self.setLayout(vbox)

        self.resize(300, 150)

app = QtGui.QApplication(sys.argv)
qb = BoxLayout()
qb.show()
sys.exit(app.exec_())
```

```
 ok = QtGui.QPushButton("OK")
 cancel = QtGui.QPushButton("Cancel")
```

Here we create two push buttons.

```
 hbox = QtGui.QHBoxLayout()
 hbox.addStretch(1)
 hbox.addWidget(ok)
 hbox.addWidget(cancel)
```

We create a horizontal box layout. Add a stretch factor and both buttons.

```
vbox = QtGui.QVBoxLayout()
vbox.addStretch(1)
vbox.addLayout(hbox)
```

To create the necessary layout, we put a horizontal lauout into a vertical one.

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

Finally, we set the main layout of the window.

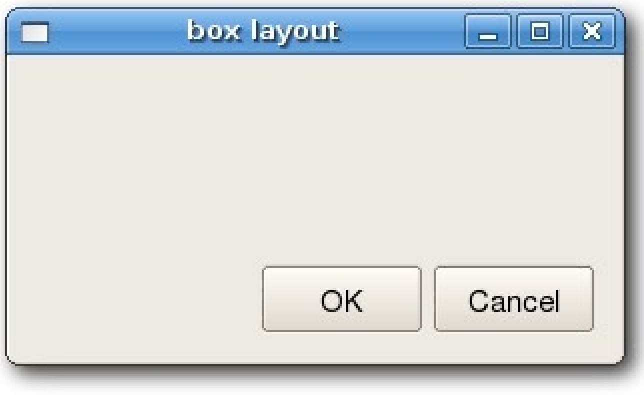


Figure: box layout

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/python

# gridlayout.py

import sys
from PyQt4 import QtGui


class GridLayout(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('grid layout')

        names = ['Cls', 'Bck', '', 'Close', '7', '8', '9', '/',
            '4', '5', '6', '*', '1', '2', '3', '-',
            '0', '.', '=', '+']

        grid = QtGui.QGridLayout()

        j = 0
        pos = [(0, 0), (0, 1), (0, 2), (0, 3),
                (1, 0), (1, 1), (1, 2), (1, 3),
                (2, 0), (2, 1), (2, 2), (2, 3),
                (3, 0), (3, 1), (3, 2), (3, 3 ),
                (4, 0), (4, 1), (4, 2), (4, 3)]

        for i in names:
            button = QtGui.QPushButton(i)
            if j == 2:
                grid.addWidget(QtGui.QLabel(''), 0, 2)
            else: grid.addWidget(button, pos[j][0], pos[j][1])
            j = j + 1

        self.setLayout(grid)



app = QtGui.QApplication(sys.argv)
qb = GridLayout()
qb.show()
sys.exit(app.exec_())
```

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

```
 grid = QtGui.QGridLayout()
```

Here we create a grid layout.

```
 if j == 2:
     grid.addWidget(QtGui.QLabel(''), 0, 2)
 else: grid.addWidget(button, pos[j][0], pos[j][1])
```

To add a widget to a grid, we call the *addWidget()* method. The arguments are the widget, the row and the column number.

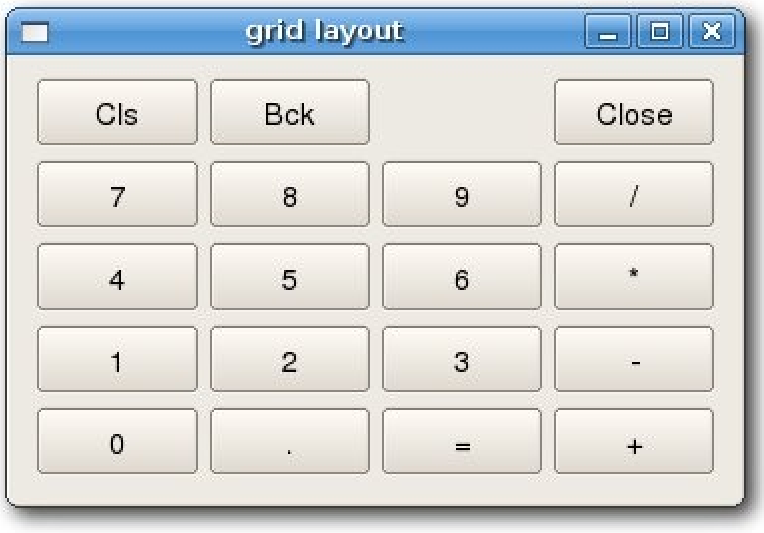


Figure: grid layout

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

```
#!/usr/bin/python
```

```
# gridlayout2.py

import sys
from PyQt4 import QtGui


class GridLayout2(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('grid layout')

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

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

        grid = QtGui.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.resize(350, 300)


app = QtGui.QApplication(sys.argv)
qb = GridLayout2()
qb.show()
sys.exit(app.exec_())
```

```
 grid = QtGui.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.

Events and Signals in PyQt4

In this part of the PyQt4 programming tutorial, we will explore events and singnals occuring in applications.

Events

Events are an important part in any GUI program. Events are generated by users or by the system. 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. Trolltech has introduced a unique signal and slot mechanism.

Signals & Slots

Signals are emitted, when users click on the button, drag a slider etc. Signals can be emitted also by the environment. For example, when a clock ticks. A slot is a method, that reacts to a signal. In python, a slot can be any python callable.

```
#!/usr/bin/python

# sigslot.py

import sys
from PyQt4 import QtGui, QtCore


class SigSlot(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('signal & slot')

        lcd = QtGui.QLCDNumber(self)
        slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)

        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(lcd)
        vbox.addWidget(slider)

        self.setLayout(vbox)
        self.connect(slider,  QtCore.SIGNAL('valueChanged(int)'), lcd,
                QtCore.SLOT('display(int)') )

        self.resize(250, 150)


app = QtGui.QApplication(sys.argv)
qb = SigSlot()
qb.show()
sys.exit(app.exec_())
```

In our example, we display an lcd number and a slider. We change the lcd number by dragging the slider.

```
 self.connect(slider,  QtCore.SIGNAL('valueChanged(int)'), lcd,
QtCore.SLOT('display(int)') )
```

Here we connect a *valueChanged()* signal of the slider to the *display()* slot of the lcd number.

The connect method has four parameters. The **sender** is an object that sends a signal. The **signal** is the signal, which is emitted. The **receiver** is the object, that receives the signal. Finally the **slot** is the method, that reacts to the signal.

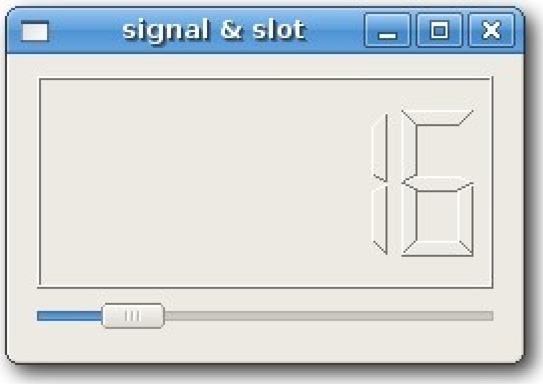


Figure: signal & slot

Reimplementing event handler

Events in PyQt are processed mainly by reimplementing event handlers .

```
#!/usr/bin/python

# escape.py

import sys
from PyQt4 import QtGui, QtCore

class Escape(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('escape')
        self.resize(250, 150)
        self.connect(self, QtCore.SIGNAL('closeEmitApp()'),
QtCore.SLOT('close()') )


    def keyPressEvent(self, event):
        if event.key() == QtCore.Qt.Key_Escape:
            self.close()

app = QtGui.QApplication(sys.argv)
qb = Escape()
qb.show()
sys.exit(app.exec_())
```

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

```
 def keyPressEvent(self, event):
     if event.key() == QtCore.Qt.Key_Escape:
         self.close()
```

If we click the escape button, we close the application.

Emitting signals

Objects created from *QtCore.QObject* can emit signals. If we click on the button, a clicked() signal is generated. In the following example we will see, how we can emit signals.

```
#!/usr/bin/python

# emit.py

import sys
from PyQt4 import QtGui, QtCore


class Emit(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('emit')
        self.resize(250, 150)
        self.connect(self, QtCore.SIGNAL('closeEmitApp()'),
QtCore.SLOT('close()') )

    def mousePressEvent(self, event):
        self.emit(QtCore.SIGNAL('closeEmitApp()'))

app = QtGui.QApplication(sys.argv)
qb = Emit()
qb.show()
sys.exit(app.exec_())
```

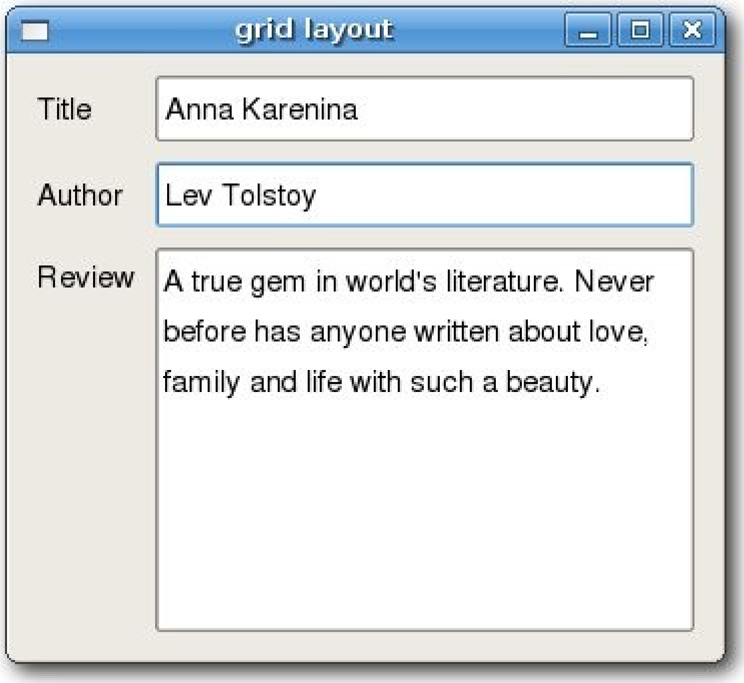
We create a new signal called *closeEmitApp()*. This signal is emitted, during a mouse press event.

```
 def mousePressEvent(self, event):
     self.emit(QtCore.SIGNAL('closeEmitApp()'))
```

Emitting a signal with the *emit()* method.

```
 self.connect(self, QtCore.SIGNAL('closeEmitApp()'), QtCore.SLOT('close()') )
```

Here we connect the manually created *closeEmitApp()* signal with the *close()* slot.



Anna Karenina

Lev Tolstoy

A true gem in world's literature. Never before has anyone written about love, family and life with such a beauty.

Figure: grid layout2

Events and Signals in PyQt4

In this part of the PyQt4 programming tutorial, we will explore events and singnals occuring in applications.

Events

Events are an important part in any GUI program. Events are generated by users or by the system. 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. Trolltech has introduced a unique signal and slot mechanism.

Signals & Slots

Signals are emitted, when users click on the button, drag a slider etc. Signals can be emitted also by the environment. For example, when a clock ticks. A slot is a method, that reacts to a signal. In python, a slot can be any python callable.

```
#!/usr/bin/python

# sigslot.py

import sys
from PyQt4 import QtGui, QtCore


class SigSlot(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('signal & slot')

        lcd = QtGui.QLCDNumber(self)
        slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)

        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(lcd)
        vbox.addWidget(slider)

        self.setLayout(vbox)
        self.connect(slider,  QtCore.SIGNAL('valueChanged(int)'), lcd,
                QtCore.SLOT('display(int)') )

        self.resize(250, 150)


app = QtGui.QApplication(sys.argv)
qb = SigSlot()
qb.show()
sys.exit(app.exec_())
```

In our example, we display an lcd number and a slider. We change the lcd number by dragging the slider.

```
 self.connect(slider,  QtCore.SIGNAL('valueChanged(int)'), lcd,
QtCore.SLOT('display(int)') )
```

Here we connect a *valueChanged()* signal of the slider to the *display()* slot of the lcd number.

The connect method has four parameters. The **sender** is an object that sends a signal. The **signal** is the signal, which is emitted. The **receiver** is the object, that receives the signal. Finally the **slot** is the method, that reacts to the signal.

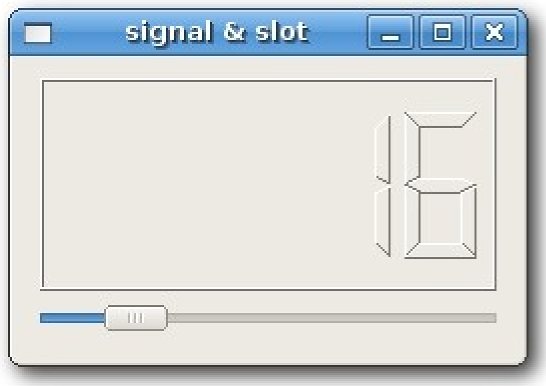


Figure: signal & slot

Reimplementing event handler

Events in PyQt are processed mainly by reimplementing event handlers .

```
#!/usr/bin/python
```

```
# escape.py

import sys
from PyQt4 import QtGui, QtCore

class Escape(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('escape')
        self.resize(250, 150)
        self.connect(self, QtCore.SIGNAL('closeEmitApp()'),
QtCore.SLOT('close()') )


    def keyPressEvent(self, event):
        if event.key() == QtCore.Qt.Key_Escape:
            self.close()

app = QtGui.QApplication(sys.argv)
qb = Escape()
qb.show()
sys.exit(app.exec_())
```

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

```
 def keyPressEvent(self, event):
     if event.key() == QtCore.Qt.Key_Escape:
         self.close()
```

If we click the escape button, we close the application.

Emitting signals

Objects created from *QtCore.QObject* can emit signals. If we click on the button, a clicked() signal is generated. In the following example we will see, how we can emit signals.

```
#!/usr/bin/python

# emit.py

import sys
from PyQt4 import QtGui, QtCore


class Emit(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setWindowTitle('emit')
        self.resize(250, 150)
        self.connect(self, QtCore.SIGNAL('closeEmitApp()'),
QtCore.SLOT('close()') )

    def mousePressEvent(self, event):
        self.emit(QtCore.SIGNAL('closeEmitApp()'))

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

```
qb = Emit()
qb.show()
sys.exit(app.exec_())
```

We create a new signal called *closeEmitApp()*. This signal is emitted, during a mouse press event.

```
 def mousePressEvent(self, event):
     self.emit(QtCore.SIGNAL('closeEmitApp()'))
```

Emitting a signal with the *emit()* method.

```
 self.connect(self, QtCore.SIGNAL('closeEmitApp()'), QtCore.SLOT('close()') )
```

Here we connect the manually created *closeEmitApp()* signal with the *close()* slot.

Dialogs in PyQt4

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. Dialogs are important means of communication between a user and a computer program.

There are essentially two types of dialogs. Predefined dialogs and custom dialogs.

Predefined Dialogs

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/python

# inputdialog.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore


class InputDialog(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 350, 80)
        self.setWindowTitle('InputDialog')
        self.button = QtGui.QPushButton('Dialog', self)
        self.button.setFocusPolicy(QtCore.Qt.NoFocus)

        self.button.move(20, 20)
        self.connect(self.button, QtCore.SIGNAL('clicked()'), self.showDialog)
        self.setFocus()

        self.label = QtGui.QLineEdit(self)
        self.label.move(130, 22)


    def showDialog(self):
        text, ok = QtGui.QInputDialog.getText(self, 'Input Dialog', 'Enter your 
name:')

        if ok:
            self.label.setText(unicode(text))


app = QtGui.QApplication(sys.argv)
icon = InputDialog()
icon.show()
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 = QtGui.QInputDialog.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 clicked ok button, the boolean value is true, otherwise false.

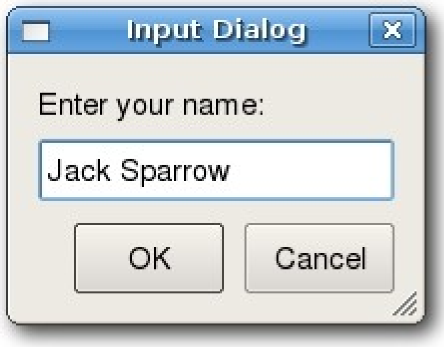


Figure: Input Dialog

QColorDialog

The *QColorDialog* provides a dialog widget for specifying colors.

```
#!/usr/bin/python
```

```
# colordialog.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore


class ColorDialog(QtGui.QWidget):
    def __init__(self, parent=None):

        QtGui.QWidget.__init__(self, parent)

        color = QtGui.QColor(0, 0, 0)

        self.setGeometry(300, 300, 250, 180)
        self.setWindowTitle('ColorDialog')

        self.button = QtGui.QPushButton('Dialog', self)
        self.button.setFocusPolicy(QtCore.Qt.NoFocus)
        self.button.move(20, 20)

        self.connect(self.button, QtCore.SIGNAL('clicked()'), self.showDialog)
        self.setFocus()

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


    def showDialog(self):
        col = QtGui.QColorDialog.getColor()

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

app = QtGui.QApplication(sys.argv)
cd = ColorDialog()
cd.show()
app.exec_()
```

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

```
 color = QtGui.QColorDialog.getColor()
```

This line will pop up the *QColorDialog*.

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

We check if the color is valid. If we click on the cancel button, no valid color is returned. If the color is valid, we change the background color using stylesheets.

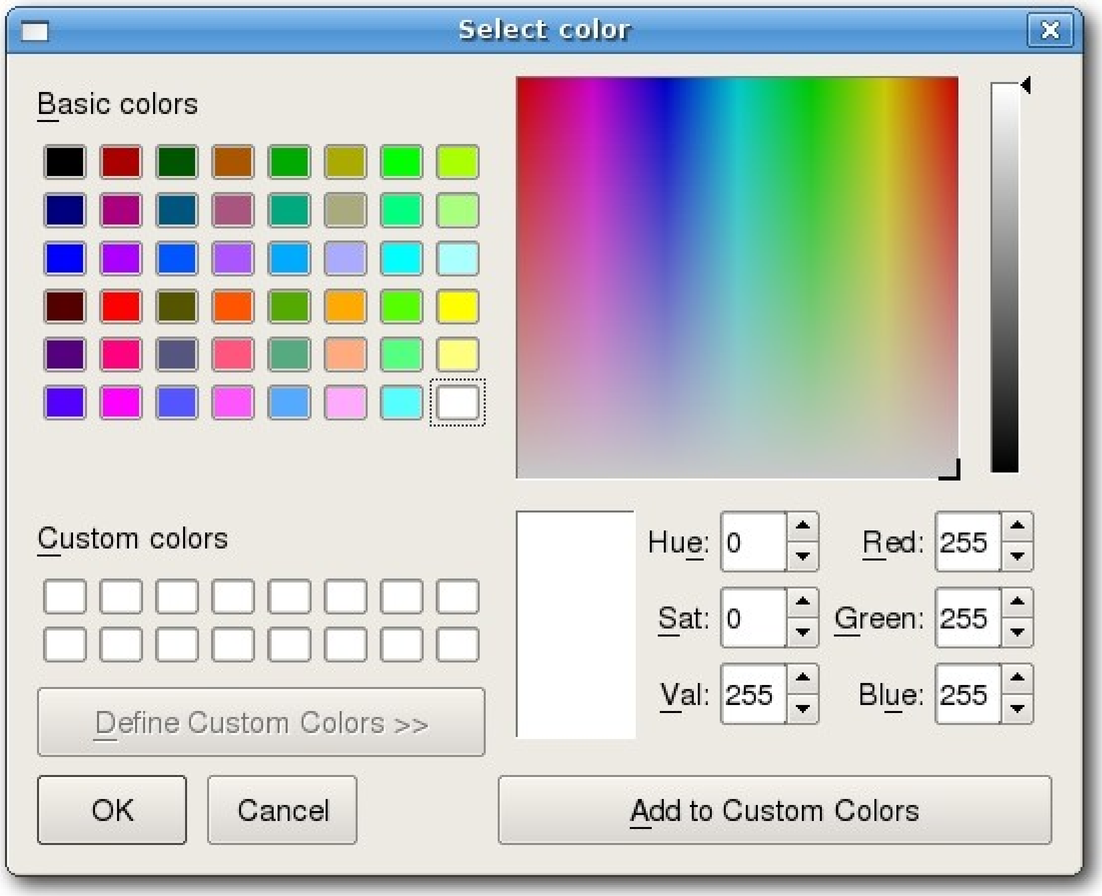


Figure: Color dialog

QFontDialog

The *QFontDialog* is a dialog widget for selecting font.

```
#!/usr/bin/python

# fontdialog.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore


class FontDialog(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        hbox = QtGui.QHBoxLayout()

        self.setGeometry(300, 300, 250, 110)
        self.setWindowTitle('FontDialog')

        button = QtGui.QPushButton('Dialog', self)
        button.setFocusPolicy(QtCore.Qt.NoFocus)
        button.move(20, 20)

        hbox.addWidget(button)
        self.connect(button, QtCore.SIGNAL('clicked()'), self.showDialog)

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

        hbox.addWidget(self.label, 1)
        self.setLayout(hbox)


    def showDialog(self):
        font, ok = QtGui.QFontDialog.getFont()
        if ok:
            self.label.setFont(font)


app = QtGui.QApplication(sys.argv)
cd = FontDialog()
cd.show()
app.exec_()
```

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

```
hbox.addWidget(self.label, 1)
```

We make the label resizable. It is necessary, because when we select a different font, the text may become larger. Otherwise the label might not be fully visible.

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

Here we pop up the font dialog.

```
if ok:
    self.label.setFont(font)
```

If we clicked ok, the font of the label was changed.

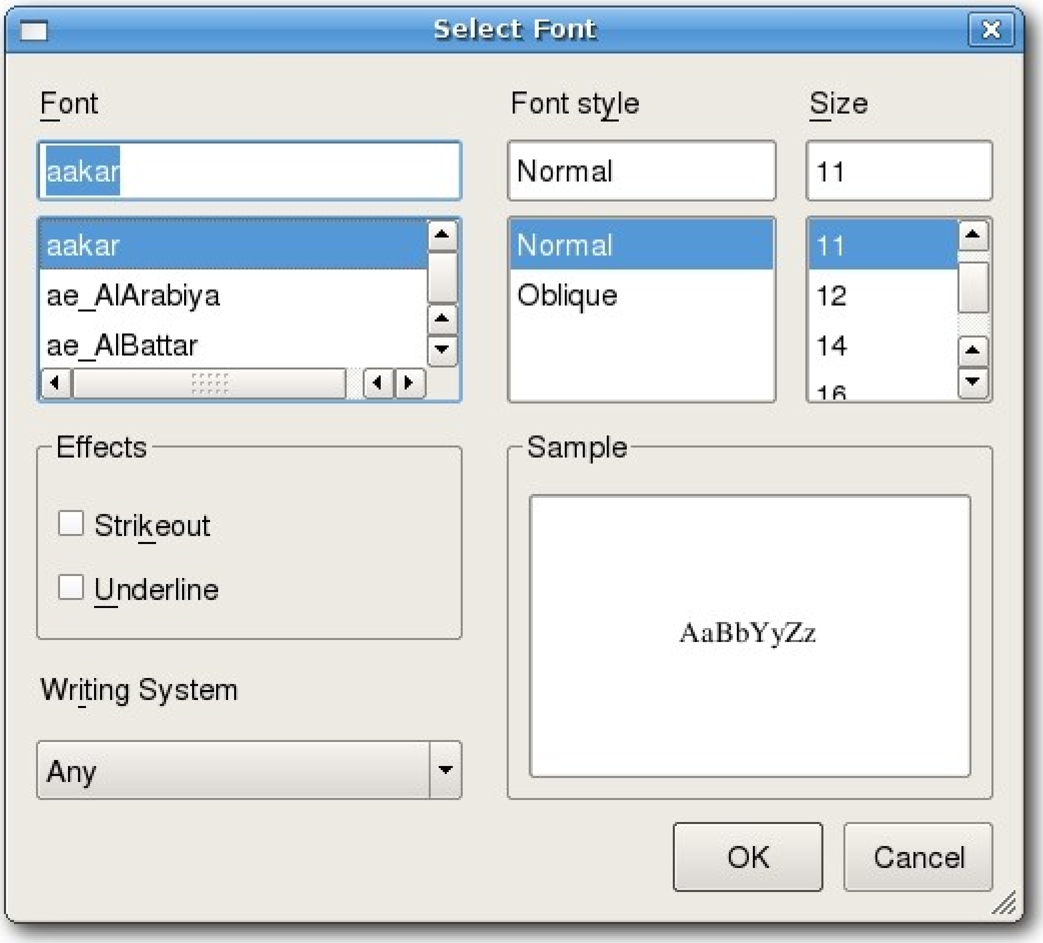


Figure: Font dialog

QFileDialog

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

```
#!/usr/bin/python

# openfiledialog.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore


class OpenFile(QtGui.QMainWindow):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)

        self.setGeometry(300, 300, 350, 300)
        self.setWindowTitle('OpenFile')

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

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

        menubar = self.menuBar()
        file = menubar.addMenu('&File')
        file.addAction(exit)

    def showDialog(self):
        filename = QtGui.QFileDialog.getOpenFileName(self, 'Open file',
                    '/home')
        file=open(filename)
        data = file.read()
        self.textEdit.setText(data)

app = QtGui.QApplication(sys.argv)
cd = OpenFile()
cd.show()
app.exec_()
```

The example shows a menubar, centrally set text edit widget and a statusbar. The statusbar is shown only for desing purposes. The the menu item shows the *QFileDialog* which is used to select a file. The contents of the file are loaded into the text edit widget.

```
class OpenFile(QtGui.QMainWindow):
...
        self.textEdit = QtGui.QTextEdit()
        self.setCentralWidget(self.textEdit)
```

The example is based on the *QMainWindow* widget, because we centrally set the text edit widget. This is easily done with the *QMainWindow* widget, without resorting to layouts.

```
 filename = QtGui.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 (*).

```
 file=open(filename)
 data = file.read()
 self.textEdit.setText(data)
```

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

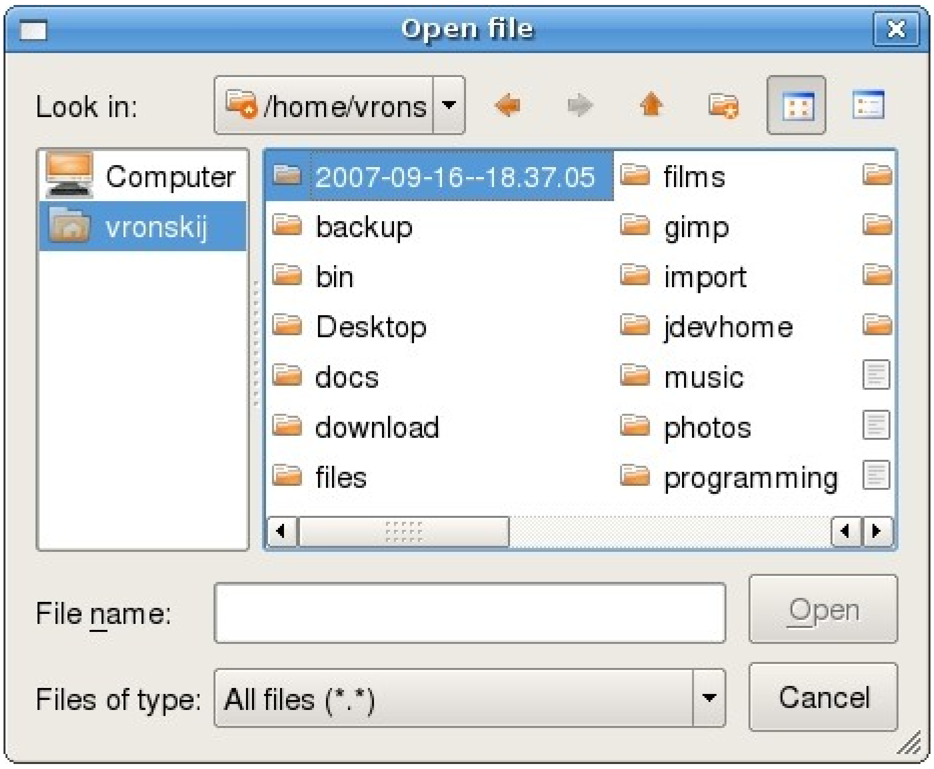


Figure: File dialog

PyQt4 Widgets

Widgets are basic building blocks of an application. The PyQt4 programming toolkit has a wide range of various widgets. Buttons, check boxes, sliders, list boxes etc. Everything a programmer needs for his job. In this section of the tutorial, we will describe several useful widgets.

QCheckBox

QCheckBox is a widget that has two states. On and Off. It is a box with a label. Whenever a checkbox is checked or cleared it emits the signal *stateChanged()*.

```
#!/usr/bin/python

# checkbox.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore


class CheckBox(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Checkbox')

        self.cb = QtGui.QCheckBox('Show title', self)
        self.cb.setFocusPolicy(QtCore.Qt.NoFocus)
        self.cb.move(10, 10)
        self.cb.toggle();
        self.connect(self.cb, QtCore.SIGNAL('stateChanged(int)'),
self.changeTitle)

    def changeTitle(self, value):
        if self.cb.isChecked():
            self.setWindowTitle('Checkbox')
        else:
            self.setWindowTitle('')

app = QtGui.QApplication(sys.argv)
icon = CheckBox()
icon.show()
app.exec_()
```

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

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

This is the *QCheckBox* constructor.

```
 self.cb.setFocusPolicy(QtCore.Qt.NoFocus)
```

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

```
 self.connect(self.cb, QtCore.SIGNAL('stateChanged(int)'), self.changeTitle)
```

By default, the *QCheckBox* accepts focus. It is represented by a thin rectangle over the checkbox label. The rectangle looks awful, so I disable it by setting the widget focus policy to *Qt.NoFocus*.

```
 self.cb.toggle();
```

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

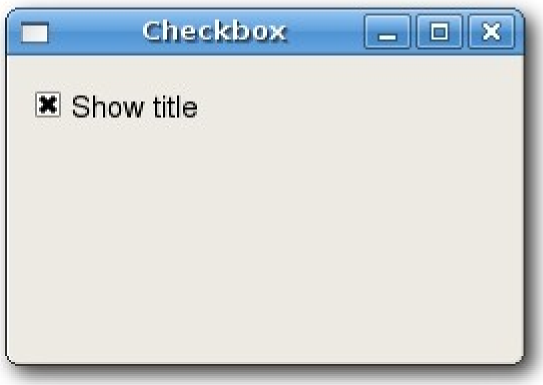


Figure: QCheckBox

ToggleButton

PyQt4 has no widget for a ToggleButton. To create a ToggleButton, we use a

QPushButton in a special mode. ToggleButton is a button that has two states. Pressed and not pressed. You toggle between these two states by clicking on it. There are situations where this functionality fits well.

```
#!/usr/bin/python

# togglebutton.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore


class ToggleButton(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.color = QtGui.QColor(0, 0, 0)

        self.setGeometry(300, 300, 280, 170)
        self.setWindowTitle('ToggleButton')

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

        self.connect(self.red, QtCore.SIGNAL('clicked()'), self.setRed)

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

        self.connect(self.green, QtCore.SIGNAL('clicked()'), self.setGreen)

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

        self.connect(self.blue, QtCore.SIGNAL('clicked()'), self.setBlue)

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

        QtGui.QApplication.setStyle(QtGui.QStyleFactory.create('cleanlooks'))

    def setRed(self):
        if self.red.isChecked():
            self.color.setRed(255)
        else: self.color.setRed(0)

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

    def setGreen(self):
        if self.green.isChecked():
            self.color.setGreen(255)
        else: self.color.setGreen(0)

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

    def setBlue(self):
        if self.blue.isChecked():
            self.color.setBlue(255)
        else: self.color.setBlue(0)

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



app = QtGui.QApplication(sys.argv)
tb = ToggleButton()
tb.show()
app.exec_()
```

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

```
 self.color = QtGui.QColor(0, 0, 0)
```

This is the initial color value. No red, green and blue equals to black. Theoretically speaking, black is not a color after all.

```
 self.red = QtGui.QPushButton('Red', self)
 self.red.setCheckable(True)
```

To create a ToggleButton, we create a *QPushButton* and make it checkable by calling *setCheckable()* method.

```
 self.connect(self.red, QtCore.SIGNAL('clicked()'), self.setRed)
```

We connect a *clicked()* signal to our user defined method.

```
 QtGui.QApplication.setStyle(QtGui.QStyleFactory.create('cleanlooks'))
```

I have set the style of the application to cleanlooks. I did it, because the default style for linux, plastique has a design bug. You cannot easily tell whether the ToggleButton is pressed or not. CleanLooks style is better.

```
 if self.red.isChecked():
     self.color.setRed(255)
 else: self.color.setRed(0)
```

We check, whether the button is pressed and change the color value accordingly.

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

To change the background color, we use stylesheets.

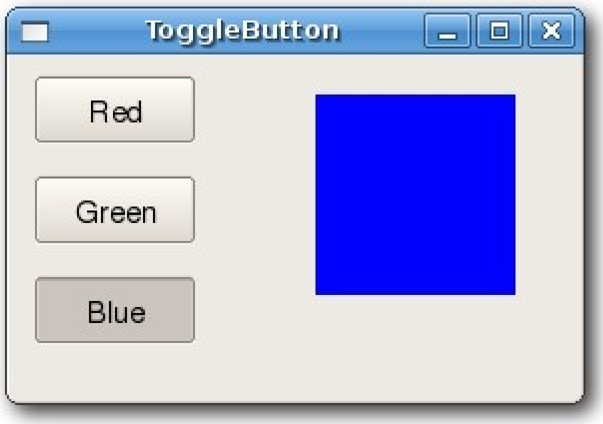


Figure: ToggleButton

QSlider, QLabel

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 simply providing a number or using a spin box. QLabel displays text or image.

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

```
#!/usr/bin/python

# slider-label.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore


class SliderLabel(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('SliderLabel')

        self.slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)
        self.slider.setFocusPolicy(QtCore.Qt.NoFocus)
        self.slider.setGeometry(30, 40, 100, 30)
        self.connect(self.slider, QtCore.SIGNAL('valueChanged(int)'),
self.changeValue)


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


    def changeValue(self, value):
        pos = self.slider.value()

        if pos == 0:
            self.label.setPixmap(QtGui.QPixmap('mute.png'))
        elif pos > 0 and pos <= 30:
            self.label.setPixmap(QtGui.QPixmap('min.png'))
        elif pos > 30 and pos < 80:
            self.label.setPixmap(QtGui.QPixmap('med.png'))
        else:
            self.label.setPixmap(QtGui.QPixmap('max.png'))

app = QtGui.QApplication(sys.argv)
icon = SliderLabel()
icon.show()
app.exec_()
```

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

```
self.slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)
```

Here we create a horizontal *QSlider*.

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

We create a *Qlabel*. And set an initial mute image to it.

```
self.connect(self.slider, QtCore.SIGNAL('valueChanged(int)'), self.changeValue)
```

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

```
pos = self.slider.value()
```

We get the position of the slider by calling the *value()* method. We change the image on the label accordingly.

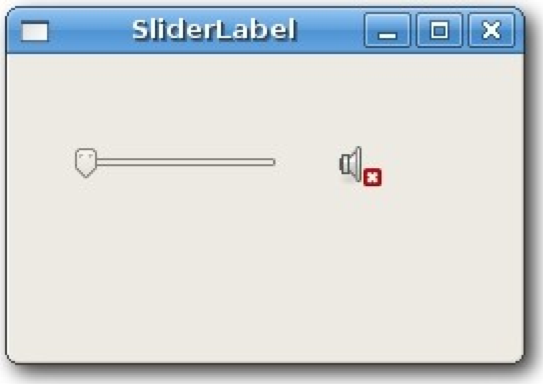


Figure: Slider and Label

QProgressBar

A progress bar is a widget that is used, when we process lengthy tasks. It is animated so that the user knows, that our task is progressing. The *QProgressBar* widget provides a horizontal or vertical progress bar in PyQt4 toolkit. The task is divided into steps. The programmer can set the minimum and maximum values for the progress bar. The default values are 0, 99.

```
#!/usr/bin/python

# progressbar.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore


class ProgressBar(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('ProgressBar')

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

        self.button = QtGui.QPushButton('Start', self)
        self.button.setFocusPolicy(QtCore.Qt.NoFocus)
        self.button.move(40, 80)

        self.connect(self.button, QtCore.SIGNAL('clicked()'), self.onStart)

        self.timer = QtCore.QBasicTimer()
        self.step = 0;


    def timerEvent(self, event):
        if self.step >= 100:
            self.timer.stop()
            return
        self.step = self.step + 1
        self.pbar.setValue(self.step)

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


app = QtGui.QApplication(sys.argv)
icon = ProgressBar()
icon.show()
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 = QtGui.QProgressBar(self)
```

QProgressBar constructor.

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

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

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

To launch the timer events, we call the *start()* method. This method has two parameters. The timeout and the object, which will receive the events.

```
 def timerEvent(self, event):
     if self.step >= 100:
         self.timer.stop()
         return
     self.step = self.step + 1
     self.pbar.setValue(self.step)
```

Each *QObject* and its descendants has a *QObject.timerEvent* event handler. In order to react to timer events, we reimplement the event handler.

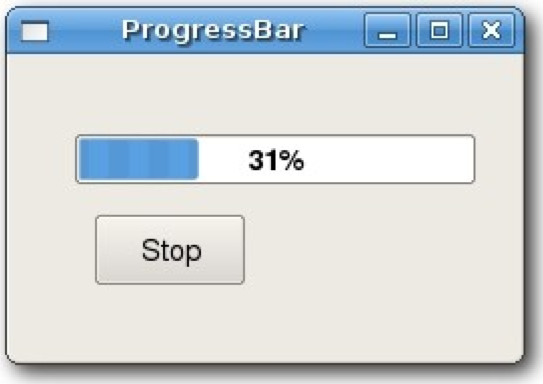


Figure: ProgressBar

QCalendarWidget

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

```
#!/usr/bin/python

# calendar.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore


class Calendar(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

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

        self.cal = QtGui.QCalendarWidget(self)
        self.cal.setGridVisible(True)
        self.cal.move(20, 20)
        self.connect(self.cal, QtCore.SIGNAL('selectionChanged()'),
self.showDate)


        self.label = QtGui.QLabel(self)
        date = self.cal.selectedDate()
        self.label.setText(str(date.toPyDate()))
        self.label.move(130, 260)


    def showDate(self):
        date = self.cal.selectedDate()
        self.label.setText(str(date.toPyDate()))


app = QtGui.QApplication(sys.argv)
icon = Calendar()
icon.show()
app.exec_()
```

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

```
 self.cal = QtGui.QCalendarWidget(self)
```

We construct a calendar widget.

```
 self.connect(self.cal, QtCore.SIGNAL('selectionChanged()'), self.showDate)
```

If we select a date from the widget, a *selectionChanged()* signal is emitted. We connect this method to the user defined *showDate()* method.

```
 def showDate(self):
     date = self.cal.selectedDate()
     self.label.setText(str(date.toPyDate()))
```

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

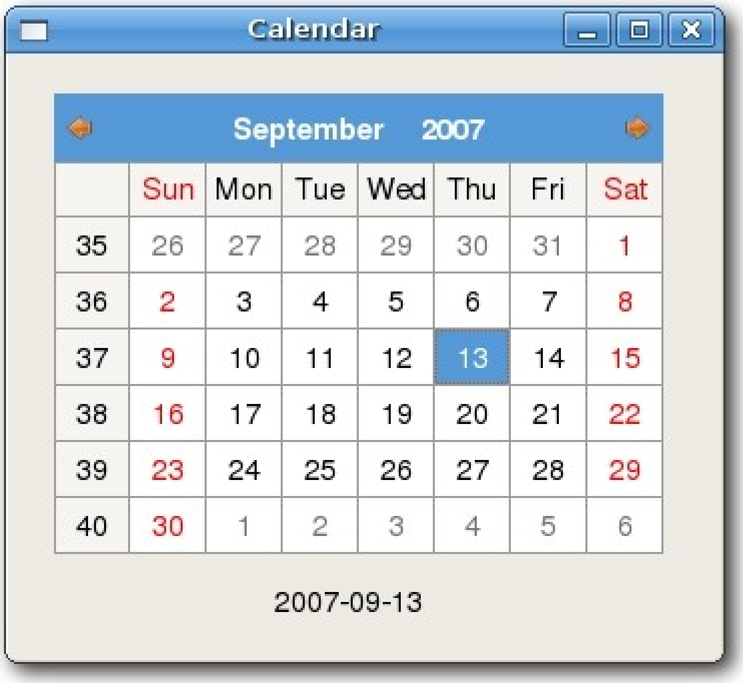


Figure: Calendar widget

Drag and Drop in PyQt4

In this part of the PyQt4 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. (Wikipedia)

Drag and drop functionality is one of the most visible aspects of the graphical user interface. Drag and drop operation enables 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 will have a **QLineEdit** and a **QPushButton**. We will drag plain text from the line edit widget and drop it onto the button widget.

```
#!/usr/bin/python

# dragdrop.py

import sys
```

```
from PyQt4 import QtGui

class Button(QtGui.QPushButton):
    def __init__(self, title, parent):
        QtGui.QPushButton.__init__(self, title, parent)
        self.setAcceptDrops(True)

    def dragEnterEvent(self, event):
        if event.mimeData().hasFormat('text/plain'):
            event.accept()
        else:
            event.ignore()

    def dropEvent(self, event):
        self.setText(event.mimeData().text())


class DragDrop(QtGui.QDialog):
    def __init__(self, parent=None):
        QtGui.QDialog.__init__(self, parent)

        self.resize(280, 150)
        self.setWindowTitle('Simple Drag & Drop')

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

        button = Button("Button", self)
        button.move(170, 65)

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

app = QtGui.QApplication(sys.argv)
icon = DragDrop()
icon.show()
app.exec_()
```

Simple drag & drop operation.

```
class Button(QtGui.QPushButton):
    def __init__(self, title, parent):
        QtGui.QPushButton.__init__(self, title, parent)
```

In order to drop text on the **QPushButton** widget, we must reimplement some methods. So 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, event):
    if event.mimeData().hasFormat('text/plain'):
        event.accept()
    else:
        event.ignore()
```

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

```
def dropEvent(self, event):
    self.setText(event.mimeData().text())
```

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

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

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

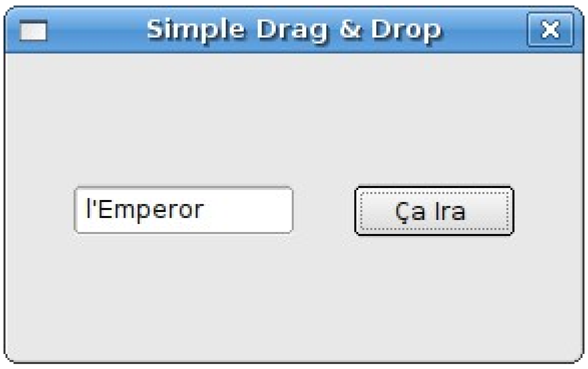


Figure: Simple Drag & Drop

Drag & drop a button widget

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

```
#!/usr/bin/python

# dragbutton.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore

class Button(QtGui.QPushButton):
    def __init__(self, title, parent):
        QtGui.QPushButton.__init__(self, title, parent)

    def mouseMoveEvent(self, event):

        if event.buttons() != QtCore.Qt.RightButton:
            return

        mimeData = QtCore.QMimeData()

        drag = QtGui.QDrag(self)
        drag.setMimeData(mimeData)
        drag.setHotSpot(event.pos() - self.rect().topLeft())

        dropAction = drag.start(QtCore.Qt.MoveAction)
        if dropAction == QtCore.Qt.MoveAction:
            self.close()


    def mousePressEvent(self, event):
        QtGui.QPushButton.mousePressEvent(self, event)
        if event.button() == QtCore.Qt.LeftButton:
            print 'press'



class DragButton(QtGui.QDialog):
    def __init__(self, parent=None):
        QtGui.QDialog.__init__(self, parent)

        self.resize(280, 150)
        self.setWindowTitle('Click or Move')
        self.setAcceptDrops(True)

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


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


    def dragEnterEvent(self, event):
        event.accept()

    def dropEvent(self, event):

        position = event.pos()
        button = Button('Button', self)

        button.move(position)
        button.show()

        event.setDropAction(QtCore.Qt.MoveAction)
        event.accept()


app = QtGui.QApplication(sys.argv)
db = DragButton()
db.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, we print 'press' to the console. By right clicking and moving the button, we perform a drag & drop operation on the button widget.

```
 class Button(QtGui.QPushButton):
     def __init__(self, title, parent):
         QtGui.QPushButton.__init__(self, title, parent)
```

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

```
if event.buttons() != QtCore.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.

```
mimeData = QtCore.QMimeData()

drag = QtGui.QDrag(self)
drag.setMimeData(mimeData)
drag.setHotSpot(event.pos() - self.rect().topLeft())
```

Here we create a **QDrag** object.

```
dropAction = drag.start(QtCore.Qt.MoveAction)

if dropAction == QtCore.Qt.MoveAction:
    self.close()
```

The **start()** method of the drag object starts the drag & drop operation. If we perform a move drop action, we destroy the button widget. Technically, we destroy a widget on the current position and recreate it on a new one.

```
def mousePressEvent(self, event):
    QtGui.QPushButton.mousePressEvent(self, event)
    if event.button() == QtCore.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 = event.pos()
button = Button('Close', self)
button.move(position)
button.show()
```

In the dropEvent() method we code, what happens after we release the mouse button and finish the drop operation. In our example, we create a new Button widget at the current position of the mouse pointer.

```
event.setDropAction(QtCore.Qt.MoveAction)
event.accept()
```

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

Drawing in PyQt4

Drawing is used, 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 drawing API provided by the PyQt4 toolkit.

The drawing is done within the *paintEvent()* method. The drawing code is placed between the *begin()* and *end()* methods of the *QPainter* object.

Drawing text

We begin with drawing some unicode text onto the window client area.

```
#!/usr/bin/python

# drawtext.py

import sys
from PyQt4 import QtGui, QtCore


class DrawText(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Draw Text')

        self.text = u'\u041b\u0435\u0432 \u041d\u0438\u043a\u043e\u043b\u0430\
\u0435\u0432\u0438\u0447 \u0422\u043e\u043b\u0441\u0442\u043e\u0439: \n\
\u0410\u043d\u043d\u0430 \u041a\u0430\u0440\u0435\u043d\u0438\u043d\u0430'



    def paintEvent(self, event):
        paint = QtGui.QPainter()
        paint.begin(self)
        paint.setPen(QtGui.QColor(168, 34, 3))
        paint.setFont(QtGui.QFont('Decorative', 10))
        paint.drawText(event.rect(), QtCore.Qt.AlignCenter, self.text)
        paint.end()


app = QtGui.QApplication(sys.argv)
dt = DrawText()
dt.show()
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 a paint event.

```
 paint = QtGui.QPainter()
 paint.begin(self)
 ...
 paint.end()
```

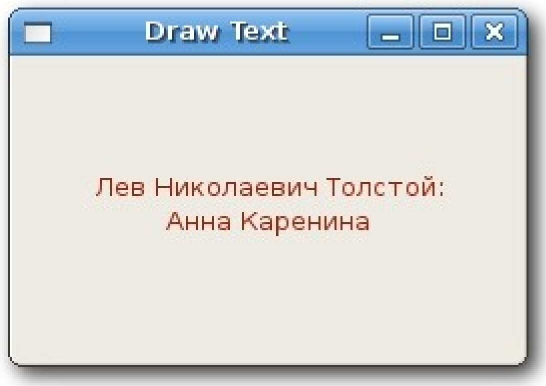
The *QPainter* class is responsible for all the low-level painting. All the painting methods go between *begin()* and *end()* methods.

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

Here we define pen and font, which we use to draw the text.

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

The *drawText()* method actually draws text on the window.



Лев Николаевич Толстой:
Анна Каренина

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/python

# points.py

import sys, random
from PyQt4 import QtGui, QtCore


class Points(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Points')

    def paintEvent(self, event):
        paint = QtGui.QPainter()
        paint.begin(self)
        paint.setPen(QtCore.Qt.red)
        size = self.size()
        for i in range(1000):
            x = random.randint(1, size.width()-1)
            y = random.randint(1, size.height()-1)
            paint.drawPoint(x, y)
        paint.end()
```

```
app = QtGui.QApplication(sys.argv)
dt = Points()
dt.show()
app.exec_()
```

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

```
 paint.setPen(QtCore.Qt.red)
```

We set the pen to red color. We use a predefined color 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.

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

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

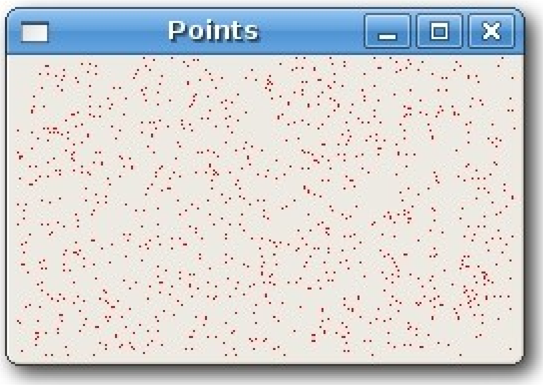


Figure: Points

Colors

A color is an object representing a combination of Red, Green, and Blue (RGB) intensity values. Valid RGB values are in the range 0 to 255. We can define a color 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, Alpha. Here we add some extra information, regarding transparency. Alpha value of 255 defines full opacity, 0 is for full transparency, eg the color is invisible.

```
#!/usr/bin/python

# colors.py

import sys, random
from PyQt4 import QtGui, QtCore


class Colors(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 350, 280)
        self.setWindowTitle('Colors')

    def paintEvent(self, event):
        paint = QtGui.QPainter()
        paint.begin(self)

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

        paint.setBrush(QtGui.QColor(255, 0, 0, 80))
        paint.drawRect(10, 15, 90, 60)

        paint.setBrush(QtGui.QColor(255, 0, 0, 160))
        paint.drawRect(130, 15, 90, 60)

        paint.setBrush(QtGui.QColor(255, 0, 0, 255))
        paint.drawRect(250, 15, 90, 60)

        paint.setBrush(QtGui.QColor(10, 163, 2, 55))
        paint.drawRect(10, 105, 90, 60)

        paint.setBrush(QtGui.QColor(160, 100, 0, 255))
        paint.drawRect(130, 105, 90, 60)

        paint.setBrush(QtGui.QColor(60, 100, 60, 255))
        paint.drawRect(250, 105, 90, 60)

        paint.setBrush(QtGui.QColor(50, 50, 50, 255))
        paint.drawRect(10, 195, 90, 60)

        paint.setBrush(QtGui.QColor(50, 150, 50, 255))
        paint.drawRect(130, 195, 90, 60)

        paint.setBrush(QtGui.QColor(223, 135, 19, 255))
        paint.drawRect(250, 195, 90, 60)

        paint.end()

app = QtGui.QApplication(sys.argv)
dt = Colors()
dt.show()
app.exec_()
```

In our example, we draw 9 colored rectangles. The first row shows a red color, with different alpha values.

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

Here we define a color using hexadecimal notation.

```
 paint.setBrush(QtGui.QColor(255, 0, 0, 80));
 paint.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 parameter. The first two are x, y values on the axis. The third and fourth parameters are width and height of the rectangle. The method draws a rectangle using current pen and current brush.

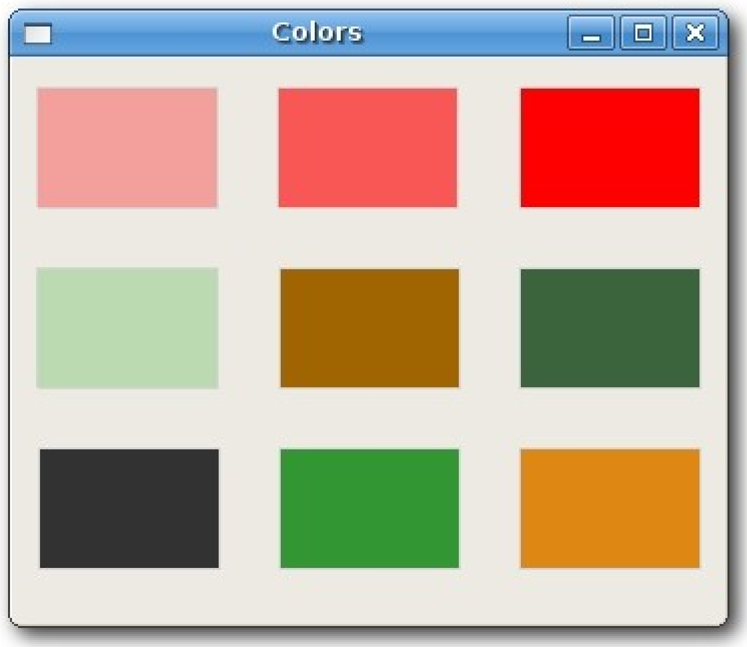


Figure: Colors

QPen

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

```
#!/usr/bin/python

# penstyles.py

import sys
from PyQt4 import QtGui, QtCore


class PenStyles(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 280, 270)
        self.setWindowTitle('penstyles')

    def paintEvent(self, event):
        paint = QtGui.QPainter()

        paint.begin(self)

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

        paint.setPen(pen)
        paint.drawLine(20, 40, 250, 40)
        pen.setStyle(QtCore.Qt.DashLine)
        paint.setPen(pen)
        paint.drawLine(20, 80, 250, 80)

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

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

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

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

        paint.end()

app = QtGui.QApplication(sys.argv)
dt = PenStyles()
dt.show()
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 custom pen style.

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

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

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

Here we define a custom pen style. We set a *QtCore.Qt.CustomDashLine* pen style and call a *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.

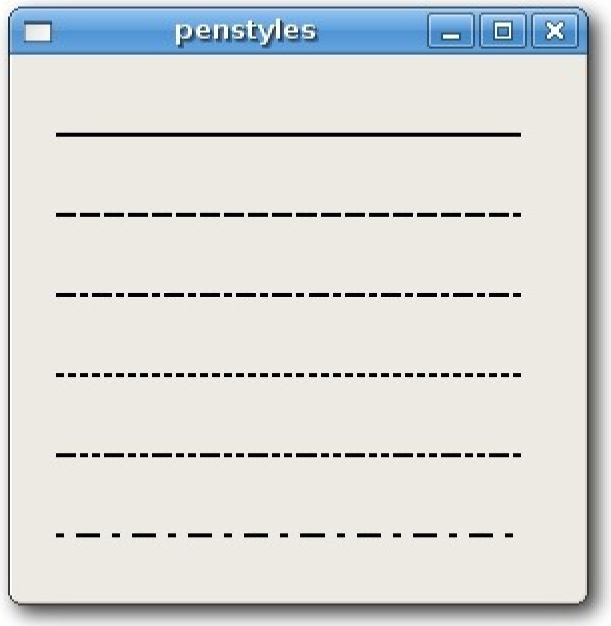


Figure: Pen Styles

QBrush

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 gradien or a texture pattern.

```
#!/usr/bin/python

# brushes.py

import sys
from PyQt4 import QtGui, QtCore


class Brushes(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

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

    def paintEvent(self, event):
        paint = QtGui.QPainter()

        paint.begin(self)

        brush = QtGui.QBrush(QtCore.Qt.SolidPattern)
        paint.setBrush(brush)
        paint.drawRect(10, 15, 90, 60)

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

        brush.setStyle(QtCore.Qt.Dense2Pattern)
        paint.setBrush(brush)
        paint.drawRect(250, 15, 90, 60)
        brush.setStyle(QtCore.Qt.Dense3Pattern)
        paint.setBrush(brush)
        paint.drawRect(10, 105, 90, 60)

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

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

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

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

        brush.setStyle(QtCore.Qt.VerPattern)
        paint.setBrush(brush)
        paint.drawRect(130, 195, 90, 60)

        brush.setStyle(QtCore.Qt.BDiagPattern)
        paint.setBrush(brush)
        paint.drawRect(250, 195, 90, 60)

        paint.end()

app = QtGui.QApplication(sys.argv)
dt = Brushes()
dt.show()
app.exec_()
```

In our example, we draw six different rectangles.

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

We define a brush object. Set it to the painter object. And draw the rectangle calling the *drawRect()* method.

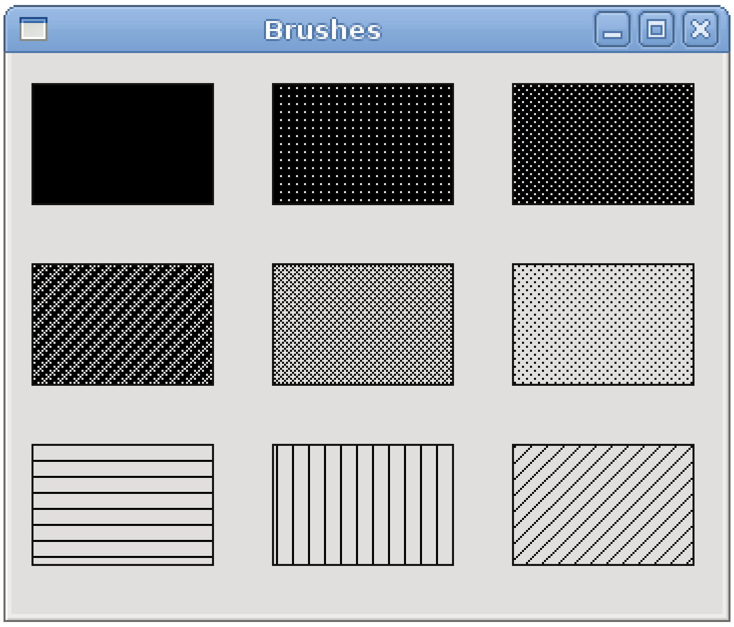


Figure: Brushes

Custom Widgets in PyQt4

Have you ever looked at an application and wondered, how a particular gui item was created? Probably every wannabe programmer has. Then you were looking at a list of widgets provided by your favourite gui library. But you couldn't find it. Toolkits usually provide only the most common widgets like buttons, text widgets, sliders etc. No toolkit can provide all possible widgets.

There are actually two kinds of toolkits. Spartan toolkits and heavy weight toolkits. The FLTK toolkit is a kind of a spartan toolkit. It provides only the very basic widgets and assumes, that the programemer will create the more complicated ones himself. PyQt4 is a heavy weight one. It has lots of widgets. Yet it does not provide the more specialized widgets. For example a speed meter widget, a widget that measures the capacity of a CD to be burned (found e.g. in nero). Toolkits also don't have usually charts.

Programmers must create such widgets by themselves. They do it by using the drawing tools provided by the toolkit. There are two 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/python

# burning.py

import sys
from PyQt4 import QtGui, QtCore
```

```
class Widget(QtGui.QLabel):
    def __init__(self, parent):
        QtGui.QLabel.__init__(self, parent)
        self.setMinimumSize(1, 30)
        self.parent = parent
        self.num = [75, 150, 225, 300, 375, 450, 525, 600, 675]

    def paintEvent(self, event):
        paint = QtGui.QPainter()
        paint.begin(self)

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

        size = self.size()
        w = size.width()
        h = size.height()
        cw = self.parent.cw
        step = int(round(w / 10.0))


        till = int(((w / 750.0) * cw))
        full = int(((w / 750.0) * 700))

        if cw >= 700:
            paint.setPen(QtGui.QColor(255, 255, 255))
            paint.setBrush(QtGui.QColor(255, 255, 184))
            paint.drawRect(0, 0, full, h)
            paint.setPen(QtGui.QColor(255, 175, 175))
            paint.setBrush(QtGui.QColor(255, 175, 175))
            paint.drawRect(full, 0, till-full, h)
        else:
            paint.setPen(QtGui.QColor(255, 255, 255))
            paint.setBrush(QtGui.QColor(255, 255, 184))
            paint.drawRect(0, 0, till, h)


        pen = QtGui.QPen(QtGui.QColor(20, 20, 20), 1, QtCore.Qt.SolidLine)
        paint.setPen(pen)
        paint.setBrush(QtCore.Qt.NoBrush)
        paint.drawRect(0, 0, w-1, h-1)

        j = 0

        for i in range(step, 10*step, step):
            paint.drawLine(i, 0, i, 5)
            metrics = paint.fontMetrics()
            fw = metrics.width(str(self.num[j]))
            paint.drawText(i-fw/2, h/2, str(self.num[j]))
            j = j + 1

        paint.end()

class Burning(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

        self.cw = 75

        self.slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)
        self.slider.setFocusPolicy(QtCore.Qt.NoFocus)
        self.slider.setRange(1, 750)
        self.slider.setValue(75)
        self.slider.setGeometry(30, 40, 150, 30)

        self.wid = Widget(self)

        self.connect(self.slider, QtCore.SIGNAL('valueChanged(int)'),
self.changeValue)
        hbox = QtGui.QHBoxLayout()
        hbox.addWidget(self.wid)
        vbox = QtGui.QVBoxLayout()
        vbox.addStretch(1)
        vbox.addLayout(hbox)
        self.setLayout(vbox)

        self.setGeometry(300, 300, 300, 220)
        self.setWindowTitle('Burning')

    def changeValue(self, event):
        self.cw = self.slider.value()
        self.wid.repaint()


app = QtGui.QApplication(sys.argv)
dt = Burning()
dt.show()
app.exec_()
```

In our example, we have a *QSlider* and a custom widget. The 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 Widget(QtGui.QLabel):
    def __init__(self, parent):
        QtGui.QLabel.__init__(self, parent)
```

The burning widget it based on the *QLabel* widget.

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

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

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

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

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

```
 cw = self.parent.cw
 step = int(round(w / 10.0))

 till = int(((w / 750.0) * cw))
 full = int(((w / 750.0) * 700))
```

We draw the widget dynamically. The greater the window, the greater 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 color. Notice the use of floating point arithmetics. This is to achieve greater precision.

The actual drawing consists of three steps. We draw the yellow or 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 = paint.fontMetrics()
 fw = metrics.width(str(self.num[j]))
 paint.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.

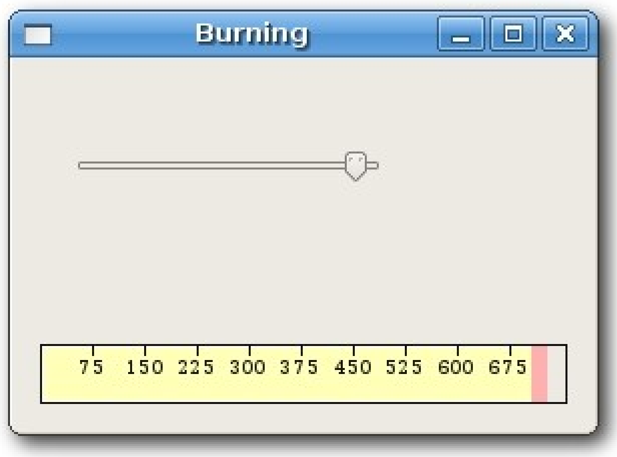


Figure: The burning widget

The Tetris game in PyQt4

Creating a computer game is very challenging. Sooner or later, a programmer will want to create a computer game one day. In fact, many people became interested in programming, because they played games and wanted to create their own. Creating a computer game will vastly help improving your programming skills.

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. Even my mobile phone has a modified version of the tetris game.

Tetris is called a falling block puzzle game. In this game, we have seven different shapes called **tetrominoes**. S-shape, Z-shape, T-shape, L-shape, Line-shape, 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.

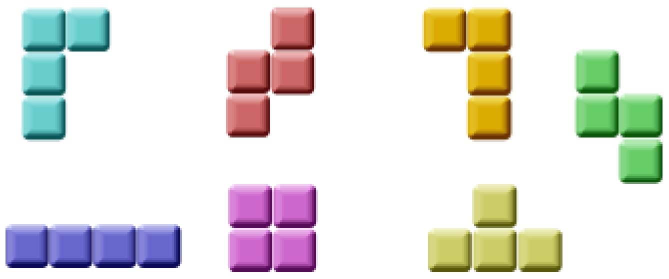


Figure: Tetrominoes

PyQt4 is a toolkit designed to create applications. There are other libraries which are targeted at creating computer games. Nevertheless, PyQt4 and other application toolkits can be used to create games.

The following example is a modified version of the tetris game, available with PyQt4 installation files.

The development

We do not have images for our tetris game, we draw the tetrominoes using the drawing API available in the PyQt4 programming toolkit. Behind every computer game, there is a mathematical model. So it is in tetris.

Some ideas behind the game.

- We use **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

```
#!/usr/bin/python

# tetris.py
```

```
import sys
import random
from PyQt4 import QtCore, QtGui


class Tetris(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)

        self.setGeometry(300, 300, 180, 380)
        self.setWindowTitle('Tetris')
        self.tetrisboard = Board(self)

        self.setCentralWidget(self.tetrisboard)

        self.statusbar = self.statusBar()
        self.connect(self.tetrisboard, 
QtCore.SIGNAL("messageToStatusbar(QString)"),
            self.statusbar, QtCore.SLOT("showMessage(QString)"))

        self.tetrisboard.start()
        self.center()

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

class Board(QtGui.QFrame):
    BoardWidth = 10
    BoardHeight = 22
    Speed = 300

    def __init__(self, parent):
        QtGui.QFrame.__init__(self, parent)

        self.timer = QtCore.QBasicTimer()
        self.isWaitingAfterLine = False
        self.curPiece = Shape()
        self.nextPiece = Shape()
        self.curX = 0
        self.curY = 0
        self.numLinesRemoved = 0
        self.board = []

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

        self.nextPiece.setRandomShape()

    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.emit(QtCore.SIGNAL("messageToStatusbar(QString)"),
            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.emit(QtCore.SIGNAL("messageToStatusbar(QString)"), "paused")
        else:
            self.timer.start(Board.Speed, self)
            self.emit(QtCore.SIGNAL("messageToStatusbar(QString)"),
                str(self.numLinesRemoved))

        self.update()

    def paintEvent(self, event):
        painter = QtGui.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 != Tetrominoes.NoShape:
                    self.drawSquare(painter,
                        rect.left() + j * self.squareWidth(),
                        boardTop + i * self.squareHeight(), shape)

        if self.curPiece.shape() != Tetrominoes.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() == Tetrominoes.NoShape:
            QtGui.QWidget.keyPressEvent(self, event)
            return

        key = event.key()
        if key == QtCore.Qt.Key_P:
            self.pause()
            return
        if self.isPaused:
            return
        elif key == QtCore.Qt.Key_Left:
            self.tryMove(self.curPiece, self.curX - 1, self.curY)
        elif key == QtCore.Qt.Key_Right:
            self.tryMove(self.curPiece, self.curX + 1, self.curY)
        elif key == QtCore.Qt.Key_Down:
            self.tryMove(self.curPiece.rotatedRight(), self.curX, self.curY)
        elif key == QtCore.Qt.Key_Up:
            self.tryMove(self.curPiece.rotatedLeft(), self.curX, self.curY)
        elif key == QtCore.Qt.Key_Space:
            self.dropDown()
        elif key == QtCore.Qt.Key_D:
            self.oneLineDown()
        else:
            QtGui.QWidget.keyPressEvent(self, event)

    def timerEvent(self, event):
        if event.timerId() == self.timer.timerId():
            if self.isWaitingAfterLine:
                self.isWaitingAfterLine = False
                self.newPiece()
            else:
                self.oneLineDown()
        else:
            QtGui.QFrame.timerEvent(self, event)

    def clearBoard(self):
        for i in range(Board.BoardHeight * Board.BoardWidth):
            self.board.append(Tetrominoes.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) == Tetrominoes.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.emit(QtCore.SIGNAL("messageToStatusbar(QString)"),
                str(self.numLinesRemoved))
            self.isWaitingAfterLine = True
            self.curPiece.setShape(Tetrominoes.NoShape)
            self.update()

    def newPiece(self):
        self.curPiece = self.nextPiece
        self.nextPiece.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(Tetrominoes.NoShape)
            self.timer.stop()
            self.isStarted = False
            self.emit(QtCore.SIGNAL("messageToStatusbar(QString)"), "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) != Tetrominoes.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 = QtGui.QColor(colorTable[shape])
        painter.fillRect(x + 1, y + 1, self.squareWidth() - 2,
            self.squareHeight() - 2, color)

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

        painter.setPen(color.dark())
        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 Tetrominoes(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 = Tetrominoes.NoShape

        self.setShape(Tetrominoes.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 rotatedLeft(self):
        if self.pieceShape == Tetrominoes.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 rotatedRight(self):
        if self.pieceShape == Tetrominoes.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


app = QtGui.QApplication(sys.argv)
tetris = Tetris()
```

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

I have simplified the game 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 immediately 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.statusbar = self.statusBar()
 self.connect(self.tetrisboard, QtCore.SIGNAL("messageToStatusbar(QString)"),
     self.statusbar, QtCore.SLOT("showMessage(QString)"))
```

We create a statusbar, where we will display messages. We will display three possible messages. The number of lines alredy removed. The paused message and the game over message.

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

Before we start the game cycle, we initialize some important variables. The **self.board** variable is a list of numbers from 0 ... 7. It represents the position of various shapes and remains of the shapes on the board.

```
 for j in range(Board.BoardWidth):
     shape = self.shapeAt(j, Board.BoardHeight - i - 1)
     if shape != Tetrominoes.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 rememberd in the **self.board** list variable. We access it using the shapeAt() method.

```
 if self.curPiece.shape() != Tetrominoes.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 drawing of the actual piece, that is falling down.

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

In the **keyPressEvent** 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.

```
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) != Tetrominoes.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:
        QtGui.QFrame.timerEvent(self, 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 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) == Tetrominoes.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))
...
```

If the piece hits the bottom, we call the **removeFullLines()** method. First we find out all full lines. And we 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 = self.nextPiece
    self.nextPiece.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(Tetrominoes.NoShape)
        self.timer.stop()
        self.isStarted = False
        self.emit(QtCore.SIGNAL("messageToStatusbar(QString)"), "Game over")
```

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

The **Shape** class saves information about the tetris piece.

```
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. For example, these tuples (0, -1), (0, 0), (1, 0), (1, 1) represent a rotated S-shape. The following diagram illustrates the shape.

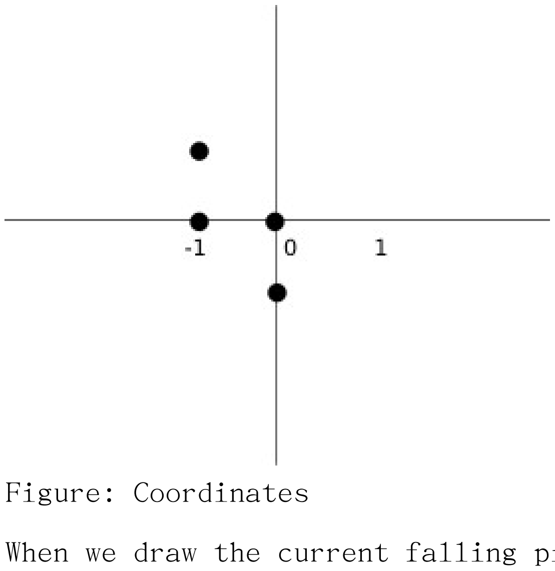


Figure: Coordinates

When we draw the current falling piece, we draw it at **self.curX, self.curY position**. Then we look at the coordinates table and draw all the four squares.

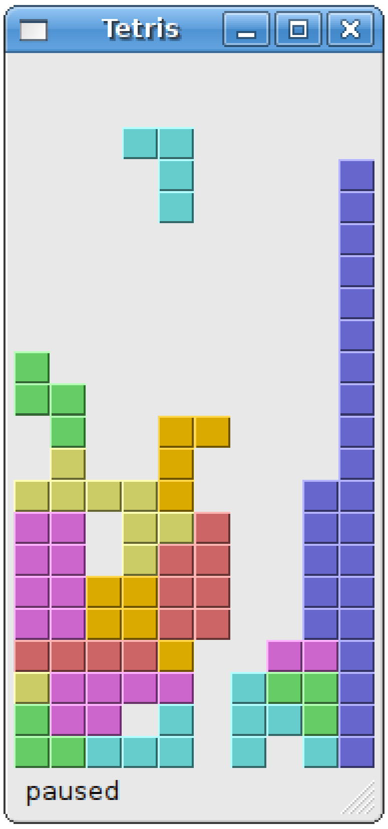


Figure: Tetris