tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

PyQt is a GUI widgets toolkit. It is a Python interface for **Qt**, one of the most powerful, and popular cross-platform GUI library. PyQt is a blend of Python programming language and the Qt library. This introductory tutorial will assist you in creating graphical applications with the help of PyQt.

Audience

This tutorial is designed for software programmers who are keen on learning how to develop graphical applications using PyQt.

Prerequisites

You should have a basic understanding of computer programming terminologies. A basic understanding of Python and any of the programming languages is a plus.

Disclaimer & Copyright

Table of Contents

1. PyQt – Introduction

PyQt is a GUI widgets toolkit. It is a Python interface for **Qt**, one of the most powerful, and popular cross-platform GUI library. PyQt was developed by RiverBank Computing Ltd. The latest version of PyQt can be downloaded from its official website:

www.riverbankcomputing.com/software/pyqt/download

PyQt API is a set of modules containing a large number of classes and functions. While **QtCore** module contains non-GUI functionality for working with file and directory etc., **QtGui** module contains all the graphical controls. In addition, there are modules for working with XML (**QtXml**), SVG (**QtSvg**), and SQL (**QtSql**), etc.

Supporting Environments

PyQt is compatible with all the popular operating systems including Windows, Linux, and Mac OS. It is dual licensed, available under GPL as well as commercial license.

Windows

You can download and install an appropriate installer from the above download link corresponding to Python version (2.7 or 3.4) and hardware architecture (32 bit or 64 bit). Note that there are two versions of PyQt that are available namely, **PyQt 4.8** and **PyQt 5.5**.

While PyQt4 is available for Python 2 as well as Python 3, PyQt5 can be used along with Python 3.* only.

PyQt4 Windows Binaries

PyQt4-4.11.4-gpl-Py3.4-Qt4.8.7-x64.exe	Windows 64 bit installer
PyQt4-4.11.4-gpl-Py3.4-Qt4.8.7-x32.exe	Windows 32 bit installer
PyQt4-4.11.4-gpl-Py3.4-Qt5.5.0-x64.exe	Windows 64 bit installer
PyQt4-4.11.4-gpl-Py3.4-Qt5.5.0-x32.exe	Windows 32 bit installer
PyQt4-4.11.4-gpl-Py2.7-Qt4.8.7-x64.exe	Windows 64 bit installer
PyQt4-4.11.4-gpl-Py2.7-Qt4.8.7-x32.exe	Windows 32 bit installer

PyQt5 Windows Binaries

PyQt5-5.5-gpl-Py3.4-Qt5.5.0-x64.exe	Windows 64 bit installer
PyQt5-5.5-gpl-Py3.4-Qt5.5.0-x32.exe	Windows 32 bit installer

Linux

For Ubuntu or any other debian Linux distribution, use the following command to install PyQt:

```
sudo apt-get install python-qt4


or


sudo apt-get install python-qt5
```

You can also build from the source code available on the 'download' page.

PyQt-x11-gpl-4.11.4.tar.gz	Linux, UNIX source for PyQt4
PyQt-gpl-5.5.tar.gz	Linux, UNIX, MacOS/X source for PyQt5

Mac OS

PyQtX project (http://sourceforge.net/projects/pyqtx/) hosts binaries of PyQt for Mac. Use Homebrew installer as per the following command:

```
brew install pyqt
```

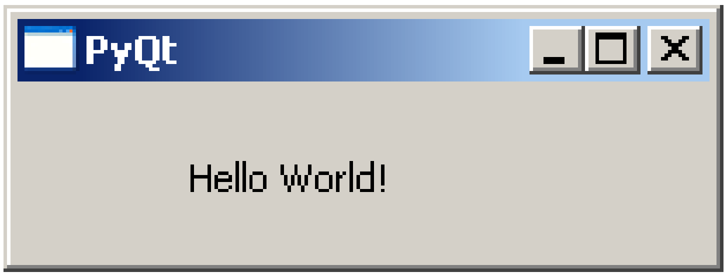
2. Hello World

Creating a simple GUI application using PyQt involves the following steps:

- Import QtGui module.
- Create an application object.
- A QWidget object creates top level window. Add QLabel object in it.
- Set the caption of label as “hello world”.
- Define the size and position of window by setGeometry() method.
- Enter the mainloop of application by **app.exec_()** method.

```
import sys
from PyQt4 import QtGui
def window():
    app = QtGui.QApplication(sys.argv)
    w = QtGui.QWidget()
    b= QtGui.QLabel(w)
    b.setText("Hello World!")
    w.setGeometry(100,100,200,50)
    b.move(50,20)
    w.setWindowTitle(“PyQt”)
    w.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    window()
```

The above code produces the following output:



3. Major Classes

PyQt API is a large collection of classes and methods. These classes are defined in more than 20 modules. Following are some of the frequently used modules:

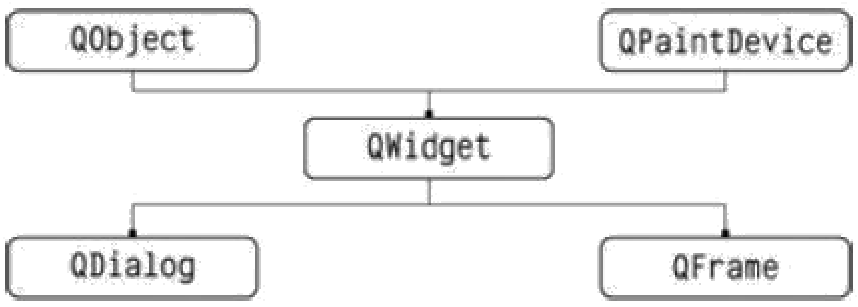
QtCore	Core non-GUI classes used by other modules
QtGui	Graphical user interface components
QtMultimedia	Classes for low-level multimedia programming
QtNetwork	Classes for network programming
QtOpenGL	OpenGL support classes
QtScript	Classes for evaluating Qt Scripts
QtSql	Classes for database integration using SQL
QtSvg	Classes for displaying the contents of SVG files
QtWebKit	Classes for rendering and editing HTML
QtXml	Classes for handling XML
QtAssistant	Support for online help
QtDesigner	Classes for extending Qt Designer

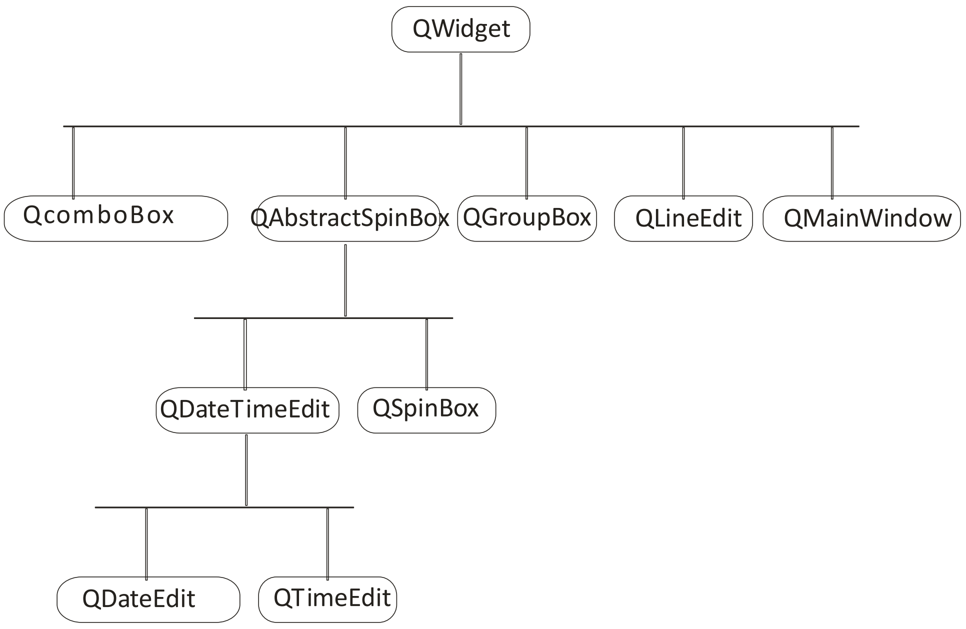
PyQt API contains more than 400 classes. The **QObject** class is at the top of class hierarchy. It is the base class of all Qt objects. Additionally, **QPaintDevice** class is the base class for all objects that can be painted.

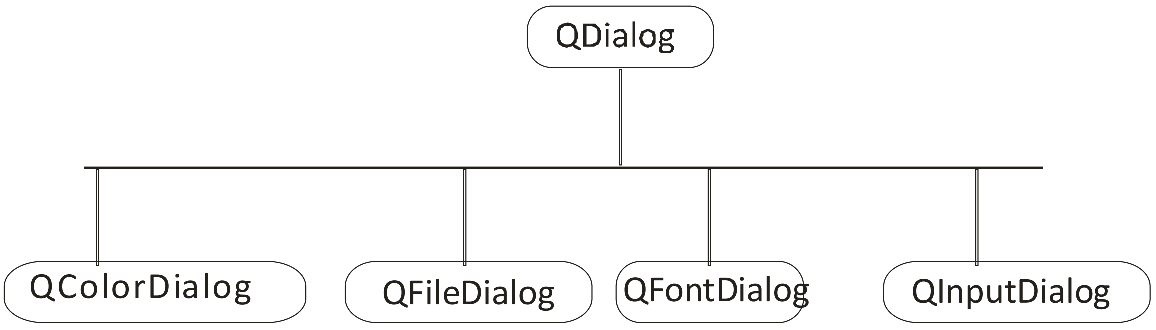
QApplication class manages the main settings and control flow of a GUI application. It contains main event loop inside which events generated by window elements and other sources are processed and dispatched. It also handles system-wide and application-wide settings.

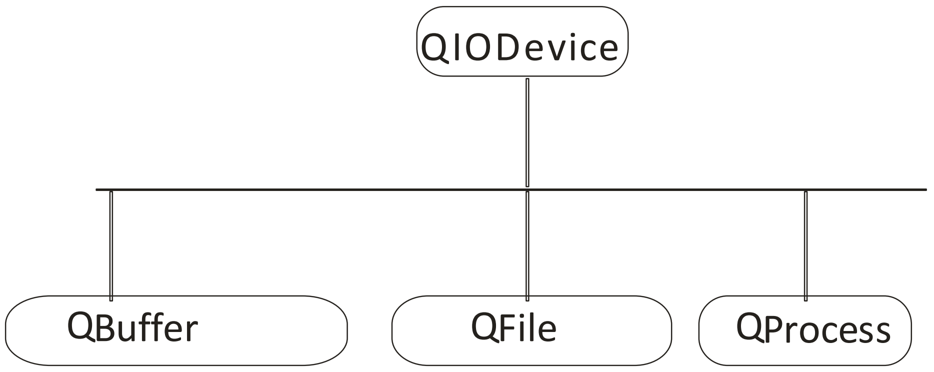
QWidget class, derived from QObject and QPaintDevice classes is the base class for all user interface objects. **QDialog** and **QFrame** classes are also derived from QWidget class. They have their own sub-class system.

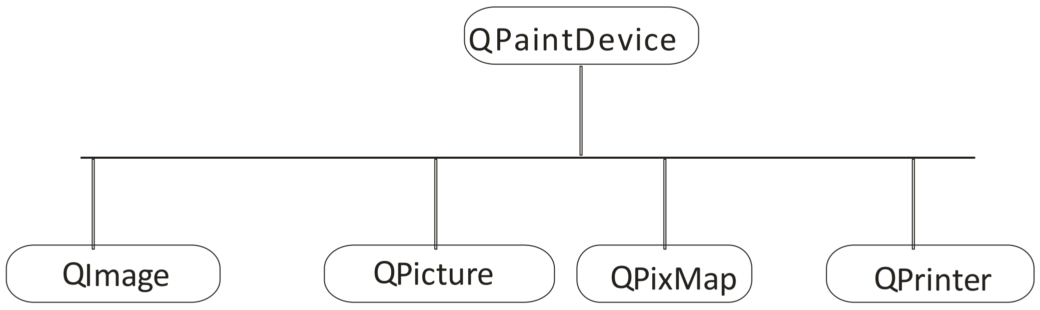
Following diagrams depict some important classes in their hierarchy.




QWidget
QcomboBox
QAbstractSpinBox
QGroupBox
QLineEdit
QMainWindow
QDateTimeEdit
QSpinBox
QDateEdit
QTimeEdit


QDialog
QColorDialog
QFileDialog
QFontDialog
QInputDialog


QIODevice
QBuffer
QFile
QProcess

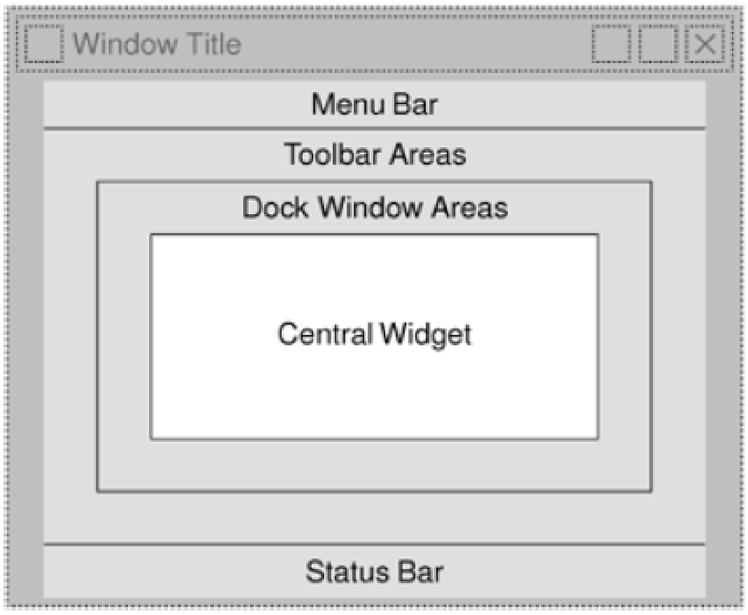


Here is a select list of frequently used widgets:

QLabel	Used to display text or image
QLineEdit	Allows the user to enter one line of text
QTextEdit	Allows the user to enter multi-line text
QPushButton	A command button to invoke action
QRadioButton	Enables to choose one from multiple options
QCheckBox	Enables choice of more than one options
QSpinBox	Enables to increase/decrease an integer value
QScrollBar	Enables to access contents of a widget beyond display aperture
QSlider	Enables to change the bound value linearly.
QComboBox	Provides a dropdown list of items to select from
QMenuBar	Horizontal bar holding QMenu objects
QStatusBar	Usually at bottom of QMainWindow, provides status information.
QToolBar	Usually at top of QMainWindow or floating. Contains action buttons
QListView	Provides a selectable list of items in ListMode or IconMode
QPixmap	Off-screen image representation for display on QLabel or QPushButton object
QDialog	Modal or modeless window which can return information to parent window

A typical GUI based application's top level window is created by **QMainWindow** widget object. Some widgets as listed above take their appointed place in this main window, while others are placed in the central widget area using various layout managers.

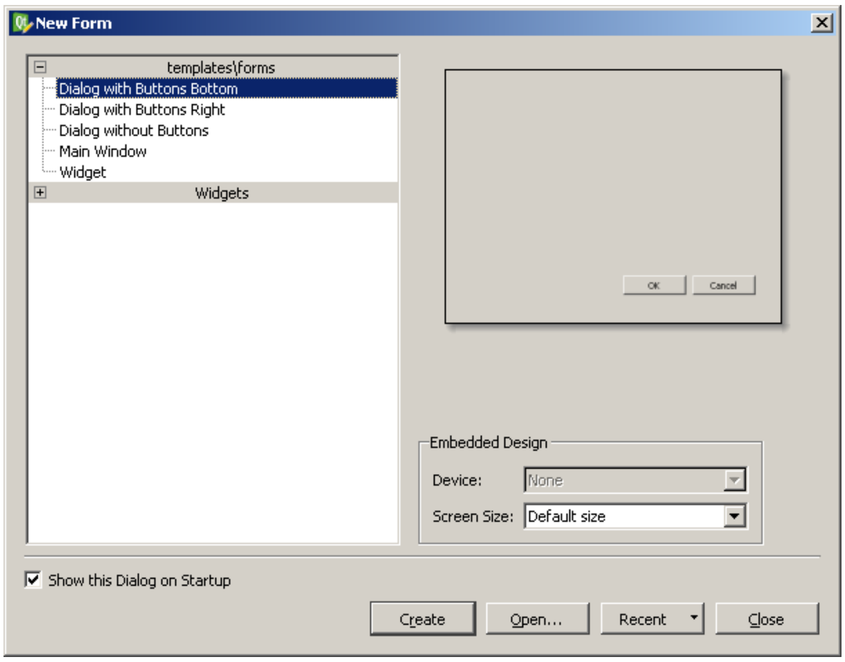
The following diagram shows the QMainWindow framework:


Window Title
Menu Bar
Toolbar Areas
Dock Window Areas
Central Widget
Status Bar

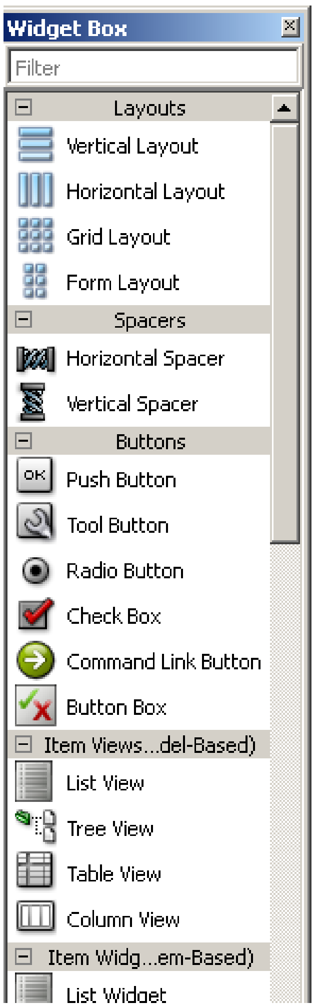
4. Using Qt Designer

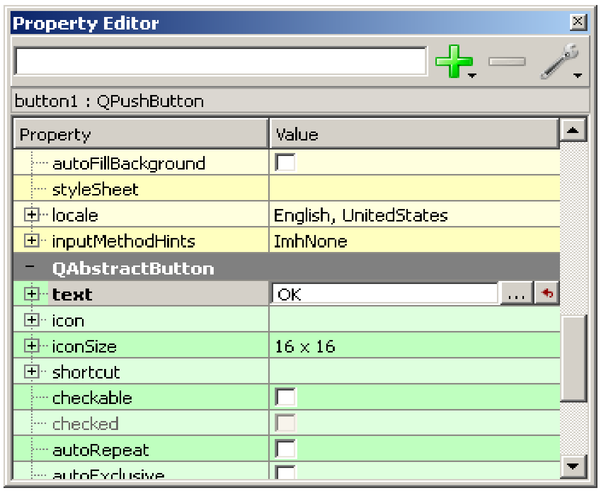
The PyQt installer comes with a GUI builder tool called **Qt Designer**. Using its simple drag and drop interface, a GUI interface can be quickly built without having to write the code. It is however, not an IDE such as Visual Studio. Hence, Qt Designer does not have the facility to debug and build the application.

Creation of a GUI interface using Qt Designer starts with choosing a top level window for the application.



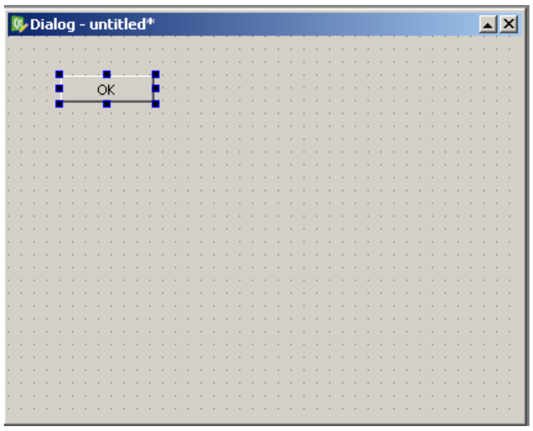
You can then drag and drop required widgets from the widget box on the left pane. You can also assign value to properties of widget laid on the form.





button1 : QPushButton

Property	Value
autoFillBackground	☐
styleSheet	
locale	English, UnitedStates
inputMethodHints	ImhNone
QAbstractButton	
text	OK
icon	
iconSize	16 x 16
shortcut	
checkable	☐
checked	☐
autoRepeat	☐
autoExclusive	☐



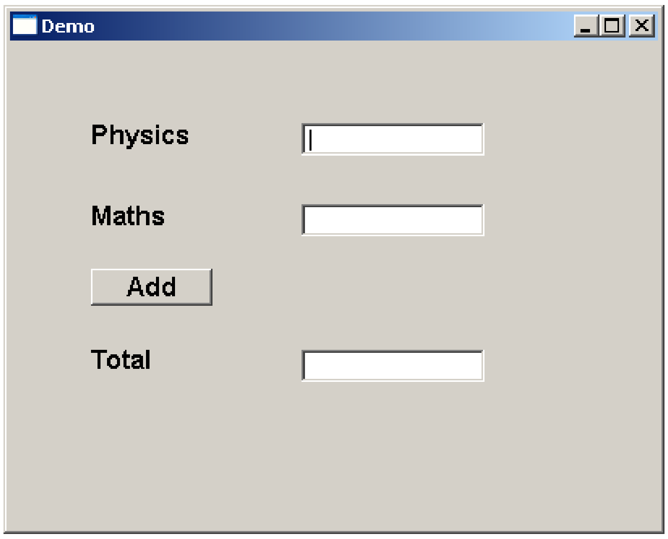
The designed form is saved as demo.ui. This ui file contains XML representation of widgets and their properties in the design. This design is translated into Python equivalent by using pyuic4 command line utility. This utility is a wrapper for uic module. The usage of pyuic4 is as follows:

```
pyuic4 –x demo.ui –o demo.py
```

In the above command, -x switch adds a small amount of additional code to the generated XML so that it becomes a self-executable standalone application.

```
if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    Dialog = QtGui.QDialog()
    ui = Ui_Dialog()
    ui.setupUi(Dialog)
    Dialog.show()
    sys.exit(app.exec_())
```

The resultant python script is executed to show the following dialog box:



The user can input data in input fields but clicking on Add button will not generate any action as it is not associated with any function. Reacting to user-generated response is called as **event handling**.

5. Signals and Slots

Unlike a console mode application, which is executed in a sequential manner, a GUI based application is event driven. Functions or methods are executed in response to user's actions like clicking on a button, selecting an item from a collection or a mouse click etc., called **events**.

Widgets used to build the GUI interface act as the source of such events. Each PyQt widget, which is derived from QObject class, is designed to emit '**signal'** in response to one or more events. The signal on its own does not perform any action. Instead, it is 'connected' to a '**slot'**. The slot can be any **callable Python function**.

In PyQt, connection between a signal and a slot can be achieved in different ways. Following are most commonly used techniques:

```
QtCore.QObject.connect(widget, QtCore.SIGNAL(‘signalname’), slot_function)
```

A more convenient way to call a slot_function, when a signal is emitted by a widget is as follows:

```
widget.signal.connect(slot_function)
```

Suppose if a function is to be called when a button is clicked. Here, the clicked signal is to be connected to a callable function. It can be achieved in any of the following two techniques:

```
QtCore.QObject.connect(button, QtCore.SIGNAL(“clicked()”), slot_function)
```

or

```
button.clicked.connect(slot_function)
```

Example

In the following example, two QPushButton objects (b1 and b2) are added in QDialog window. We want to call functions b1_clicked() and b2_clicked() on clicking b1 and b2 respectively.

When b1 is clicked, the clicked() signal is connected to b1_clicked() function

```
b1.clicked.connect(b1_clicked())
```

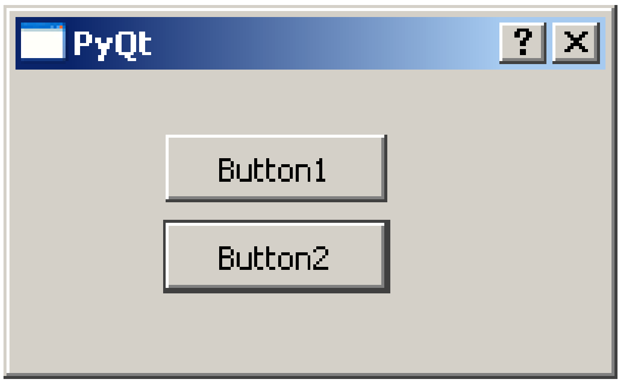
When b2 is clicked, the clicked() signal is connected to b2_clicked() function

```
 QObject.connect(b2, SIGNAL("clicked()"), b2_clicked)
```

Example

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
def window():
    app = QApplication(sys.argv)
    win = QDialog()
    b1= QPushButton(win)
    b1.setText("Button1")
    b1.move(50,20)
    b1.clicked.connect(b1_clicked)

    b2=QPushButton(win)
    b2.setText("Button2")
    b2.move(50,50)
    QObject.connect(b2,SIGNAL("clicked()"),b2_clicked)

    win.setGeometry(100,100,200,100)
    win.setWindowTitle("PyQt")
    win.show()
    sys.exit(app.exec_())

def b1_clicked():
    print "Button 1 clicked"

def b2_clicked():
    print "Button 2 clicked"

if __name__ == '__main__':
    window()
```

The above code produces the following output:



Output:

```
Button 1 clicked

Button 2 clicked
```

6. Layout Managers

A GUI widget can be placed inside the container window by specifying its absolute coordinates measured in pixels. The coordinates are relative to the dimensions of the window defined by setGeometry() method.

setGeometry() syntax:

```
QWidget.setGeometry(xpos, ypos, width, height)
```

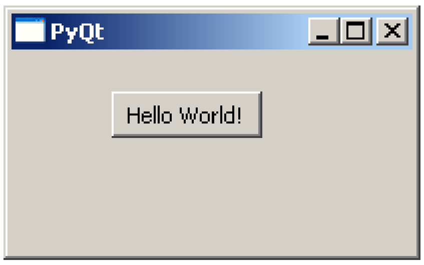
In the following code snippet, the top level window of 300 by 200 pixels dimensions is displayed at position (10, 10) on the monitor.

```
import sys
from PyQt4 import QtGui
def window():
    app = QtGui.QApplication(sys.argv)
    w = QtGui.QWidget()
    b = QtGui.QPushButton(w)
    b.setText("Hello World!")
    b.move(50,20)
    w.setGeometry(10,10,300,200)
    w.setWindowTitle(“PyQt”)
    w.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    window()
```

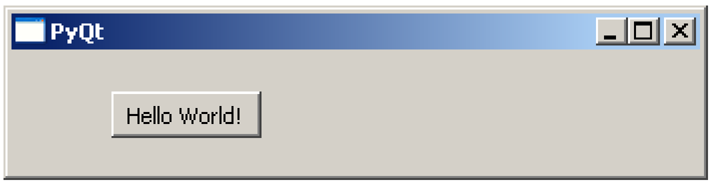
A **PushButton** widget is added in the window and placed at a position 50 pixels towards right and 20 pixels below the top left position of the window.

This **Absolute Positioning**, however, is not suitable because of following reasons:

- The position of the widget does not change even if the window is resized.
- The appearance may not be uniform on different display devices with different resolutions.
- Modification in the layout is difficult as it may need redesigning the entire form.



Original window



Resized window. Position of button is unchanged.

PyQt API provides layout classes for more elegant management of positioning of widgets inside the container. The advantages of Layout managers over absolute positioning are:

- Widgets inside the window are automatically resized.
- Ensures uniform appearance on display devices with different resolutions
- Adding or removing widget dynamically is possible without having to redesign.

QLayout class is the base class from which QBoxLayout, QGridLayout and QFormLayout classes are derived.

7. QBoxLayout Class

QBoxLayout class lines up the widgets vertically or horizontally. Its derived classes are **QVBoxLayout** (for arranging widgets vertically) and **QHBoxLayout** (for arranging widgets horizontally). Following table shows the important methods of QBoxLayout class:

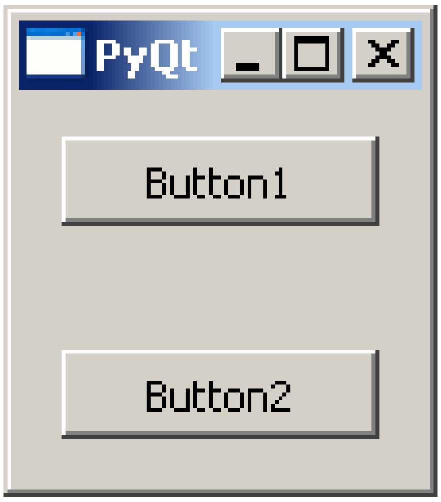
addWidget()	Add a widget to the BoxLayout
addStretch()	Creates empty stretchable box
addLayout()	Add another nested layout

Example 1

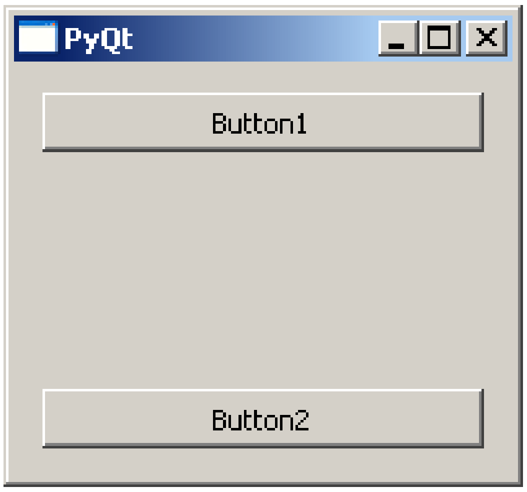
Here two buttons are added in the vertical box layout. A stretchable empty space is added between them by addStretch() method. Therefore, if the top level window is resized, the position of buttons automatically gets relocated.

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
def window():
    app = QApplication(sys.argv)
    win = QWidget()

    b1=QPushButton("Button1")
    b2=QPushButton("Button2")

    vbox=QVBoxLayout()
    vbox.addWidget(b1)
    vbox.addStretch()
    vbox.addWidget(b2)
    win.setLayout(vbox)

    win.setWindowTitle("PyQt")
    win.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    window()
```

The above code produces the following output:



Original window



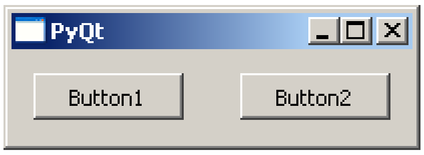
Resized window. Position and size changes dynamically

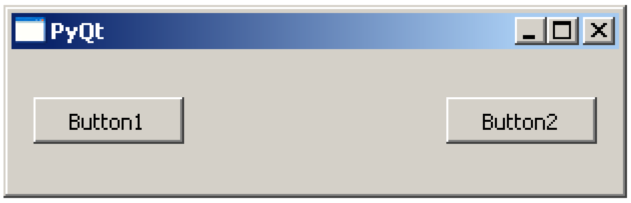
Example 2

This example uses horizontal box layout. addStretch() method inserts a stretchable empty space between the two button objects. Hence, as the window is resized, the size and position of the button changes dynamically.

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
def window():
    app = QApplication(sys.argv)
    win = QWidget()

    b1= QPushButton("Button1")
    b2=QPushButton("Button2")

    hbox=QHBoxLayout()

    hbox.addWidget(b1)
    hbox.addStretch()
    hbox.addWidget(b2)
    win.setLayout(hbox)

    win.setWindowTitle("PyQt")
    win.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    window()
```

The above code produces the following output:





Original window

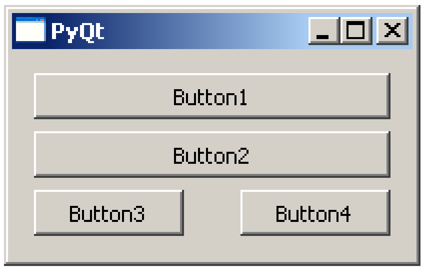
Resized window. Position and size of buttons changes dynamically

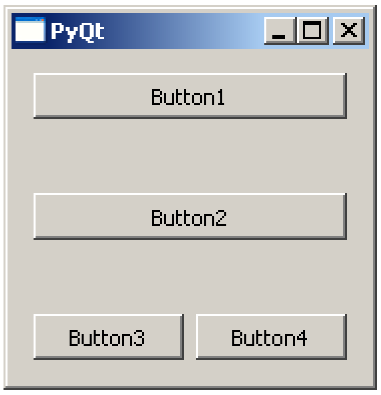
Example 3

This example shows how the layouts can be nested. Here, two buttons are added to vertical box layout. Then, a horizontal box layout object with two buttons and a stretchable empty space is added to it. Finally, the vertical box layout object is applied to the top level window by the setLayout() method.

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
def window():
    app = QApplication(sys.argv)
    win = QWidget()

    b1=QPushButton("Button1")
    b2=QPushButton("Button2")
    vbox=QVBoxLayout()
    vbox.addWidget(b1)
    vbox.addStretch()
    vbox.addWidget(b2)
    hbox=QHBoxLayout()
    b3=QPushButton("Button3")
    b4=QPushButton("Button4")
    hbox.addWidget(b3)
    hbox.addStretch()
    hbox.addWidget(b4)

    vbox.addStretch()
    vbox.addLayout(hbox)
    win.setLayout(vbox)

    win.setWindowTitle("PyQt")
    win.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    window()
```

The above code produces the following output:





Original window

Resized window. Position and size of buttons changes dynamically

8. QGridLayout Class

A **GridLayout** class object presents with a grid of cells arranged in rows and columns. The class contains addWidget() method. Any widget can be added by specifying the number of rows and columns of the cell. Optionally, a spanning factor for row as well as column, if specified makes the widget wider or taller than one cell. Two overloads of addWidget() method are as follows:

addWidget(QWidget, int r, int c)	Adds a widget at specified row and column
addWidget(QWidget, int r, int c, int rowspan, int columnspan)	Adds a widget at specified row and column and having specified width and/or height

A child layout object can also be added at any cell in the grid.

addLayout(QLayout, int r, int c)	Adds a layout object at specified row and column

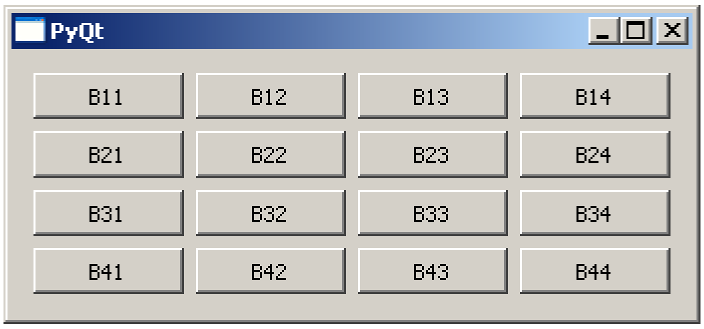
Example

The following code creates a grid layout of 16 push buttons arranged in a grid layout of 4 rows and 4 columns.

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
def window():
    app = QApplication(sys.argv)
    win = QWidget()
    grid=QGridLayout()
    for i in range(1,5):
        for j in range(1,5):
            grid.addWidget(QPushButton("B"+str(i)+str(j)),i,j)

    win.setLayout(grid)
    win.setGeometry(100,100,200,100)
    win.setWindowTitle("PyQt")
    win.show()
    sys.exit(app.exec_())


if __name__ == '__main__':
    window()
```

The code uses two nested for loops for row and column numbers, denoted by variables *i* and *j*. They are converted to string to concatenate the caption of each push button to be added at *i*th row and *j*th column.

The above code produces the following output:



9. QFormLayout Class

QFormLayout is a convenient way to create two column form, where each row consists of an input field associated with a label. As a convention, the left column contains the label and the right column contains an input field. Mainly three overloads of addRow() method addLayout() are commonly used.

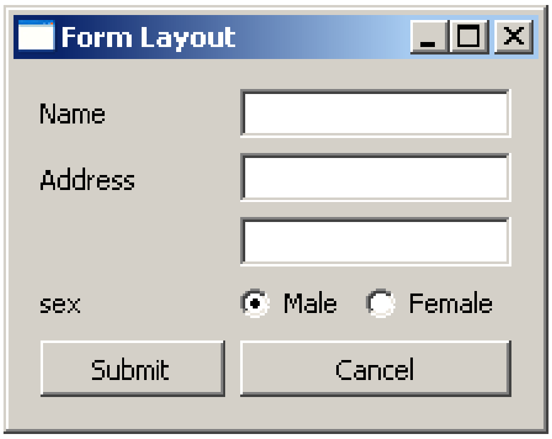
addRow(QLabel, QWidget)	Adds a row containing label and input field
addRow(QLabel, QLayout)	Adds a child layout in the second column
addRow(QWidget)	Adds a widget spanning both columns

Example

This code adds a LineEdit field to input name in the first row. Then it adds a vertical box layout for two address fields in the second column of the next row. Next, a horizontal box layout object containing two Radio button fields is added in the second column of the third row. The fourth row shows two buttons 'Submit' and 'Cancel'.

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
def window():
    app = QApplication(sys.argv)
    win = QWidget()

    l1=QLabel("Name")
    nm=QLineEdit()

    l2=QLabel("Address")
    add1=QLineEdit()
    add2=QLineEdit()
    fbox=QFormLayout()
    fbox.addRow(l1,nm)
    vbox=QVBoxLayout()

    vbox.addWidget(add1)
    vbox.addWidget(add2)
    fbox.addRow(l2,vbox)
    hbox=QHBoxLayout()
    r1=QRadioButton("Male")
    r2=QRadioButton("Female")
    hbox.addWidget(r1)
    hbox.addWidget(r2)
    hbox.addStretch()
    fbox.addRow(QLabel("sex"),hbox)
    fbox.addRow(QPushButton("Submit"),QPushButton("Cancel"))

    win.setLayout(fbox)

    win.setWindowTitle("PyQt")
    win.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    window()
```

The above code produces the following output:



10. QLabel Widget

A **QLabel** object acts as a placeholder to display non-editable text or image, or a movie of animated GIF. It can also be used as a mnemonic key for other widgets. Plain text, hyperlink or rich text can be displayed on the label.

The following table lists the important methods defined in QLabel class:

setAlignment()	Aligns the text as per alignment constants Qt.AlignLeft Qt.AlignRight Qt.AlignCenter Qt.AlignJustify
setIndent()	Sets the labels text indent
setPixmap()	Displays an image
Text()	Displays the caption of the label
setText()	Programmatically sets the caption
selectedText()	Displays the selected text from the label (The textInteractionFlag must be set to TextSelectableByMouse)
setBuddy()	Associates the label with any input widget
setWordWrap()	Enables or disables wrapping text in the label

Signals of QLabel Class

linkActivated	If the label containing embedded hyperlink is clicked, the URL will open. setOpenExternalLinks feature must be set to true.
linkHovered	Slot method associated with this signal will be called when the label having embedded hyperlinked is hovered by the mouse.

Example

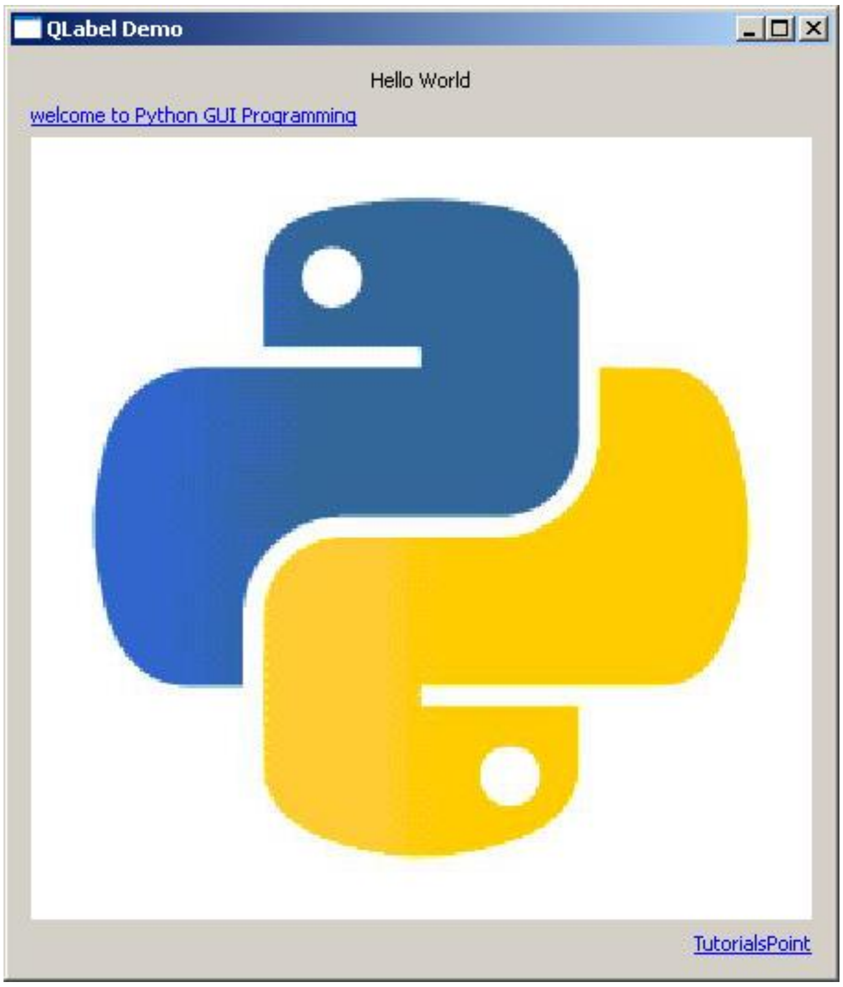
In this example, QLabel objects l2 and l4 have the caption containing hyperlink. setOpenExternalLinks for l2 is set to true. Hence, if this label is clicked, the associated URL will open in the browser. linkHovered signal of l4 is connected to hovered() function. So, whenever the mouse hovers over it, the function will be executed.

QPixmap object prepares offscreen image from python.jpg file. It is displayed as label l3 by using setPixmap() method.

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
def window():
    app = QApplication(sys.argv)
    win = QWidget()

    l1=QLabel()
    l2=QLabel()
    l3=QLabel()
    l4=QLabel()
    l1.setText("Hello World")
    l4.setText("<A href='www.TutorialsPoint.com'>TutorialsPoint</a>")
    l2.setText("<a href='#'>welcome to Python GUI Programming</a>")
    l1.setAlignment(Qt.AlignCenter)
    l3.setAlignment(Qt.AlignCenter)
    l4.setAlignment(Qt.AlignRight)
    l3.setPixmap(QPixmap("python.jpg"))
    vbox=QVBoxLayout()
    vbox.addWidget(l1)
    vbox.addStretch()
    vbox.addWidget(l2)
    vbox.addStretch()
    vbox.addWidget(l3)
    vbox.addStretch()
    vbox.addWidget(l4)

    l1.setOpenExternalLinks(True)
    l4.linkActivated.connect(clicked)
    l2.linkHovered.connect(hovered)
    l1.setTextInteractionFlags(Qt.TextSelectableByMouse)
    win.setLayout(vbox)

    win.setWindowTitle("QLabel Demo")
    win.show()
    sys.exit(app.exec_())
```

```

def hovered():
    print "hovering"
def clicked():
    print "clicked"

if __name__ == '__main__':
    window()
```

The above code produces the following output:


QLabel Demo

Hello World

welcome to Python GUI Programming

TutorialsPoint

11. QLineEdit Widget

QLineEdit object is the most commonly used input field. It provides a box in which one line of text can be entered. In order to enter multi-line text, **QTextEdit** object is required.

The following table lists a few important methods of QLineEdit class:

setAlignment()	Aligns the text as per alignment constants Qt.AlignLeft Qt.AlignRight Qt.AlignCenter Qt.AlignJustify
clear()	Erases the contents
setEchoMode()	Controls the appearance of the text inside the box. Echomode values are: QLineEdit.Normal QLineEdit.NoEcho QLineEdit.Password QLineEdit.PasswordEchoOnEdit
setMaxLength()	Sets the maximum number of characters for input
setReadOnly()	Makes the text box non-editable
setText()	Programmatically sets the text
text()	Retrives text in the field
setValidator()	Sets the validation rules. Available validators are QIntValidator: Restricts input to integer QDoubleValidator: Fraction part of number limited to specified decimals QRegexpValidator: Checks input against a Regex expression
setInputMask()	Applies mask of combination of characters for input
setFont()	Displays the contents QFont object

QLineEdit object emits the following signals:

cursorPositionChanged()	Whenever the cursor moves
editingFinished()	When you press 'Enter' or the field loses focus
returnPressed()	When you press 'Enter'
selectionChanged()	Whenever the selected text changes
textChanged()	As text in the box changes either by input or by programmatic means
textEdited()	Whenever the text is edited

Example

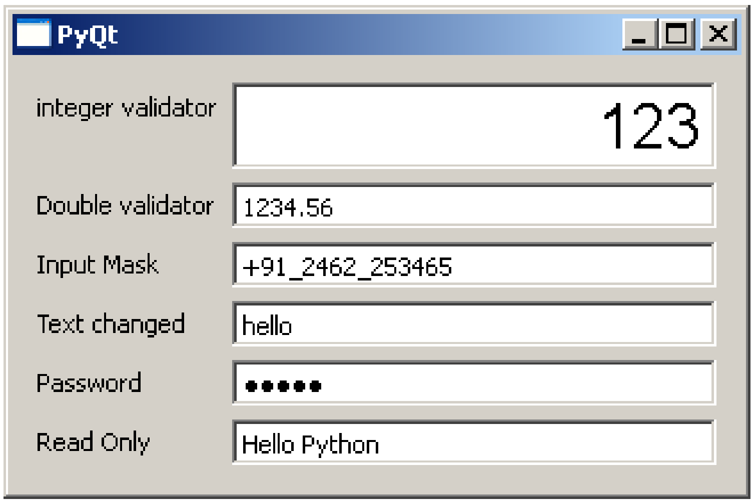
QLineEdit objects in this example demonstrate use of some of these methods.

First field **e1** shows text using a custom font, in right alignment and allows integer input. Second field restricts input to a number with 2 digits after decimal point. An input mask for entering the phone number is applied on the third field. textChanged() signal on the field **e4** is connected to textchanged() slot method.

Contents of **e5** field are echoed in password form as its EchoMode property is set to Password. Its editingfinished() signal is connected to presenter() method. So, once the user presses the Enter key, the function will be executed. The field **e6** shows a default text, which cannot be edited as it is set to read only.

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
def window():
    app = QApplication(sys.argv)
    win = QWidget()

    e1=QLineEdit()
    e1.setValidator(QIntValidator())
    e1.setMaxLength(4)
    e1.setAlignment(Qt.AlignRight)
    e1.setFont(QFont("Arial",20))
    e2=QLineEdit()
    e2.setValidator(QDoubleValidator(0.99,99.99,2))
    flo=QFormLayout()
    flo.addRow("integer validator", e1)
    flo.addRow("Double validator",e2)
    e3=QLineEdit()
    e3.setInputMask('+99_9999_999999')
    flo.addRow("Input Mask",e3)
    e4=QLineEdit()
    e4.textChanged.connect(textchanged)
    flo.addRow("Text changed",e4)
    e5=QLineEdit()
    e5.setEchoMode(QLineEdit.Password)

    flo.addRow("Password",e5)
    e6=QLineEdit("Hello Python")
    e6.setReadOnly(True)
    flo.addRow("Read Only",e6)
    e5.editingFinished.connect(enterPress)
    win.setLayout(flo)
    win.setWindowTitle("PyQt")
    win.show()
    sys.exit(app.exec_())

def textchanged(text):
    print "contents of text box: "+text
def enterPress():
    print "edited"

if __name__ == '__main__':
    window()
```

The above code produces the following output:



integer validator	123
Double validator	1234.56
Input Mask	+91_2462_253465
Text changed	hello
Password	●●●●●
Read Only	Hello Python

```
contents of text box: h
contents of text box: he
contents of text box: hel
contents of text box: hell
contents of text box: hello
editing finished
```

12. QPushButton Widget

In any GUI design, the command button is the most important and most often used control. Buttons with Save, Open, OK, Yes, No and Cancel etc. as caption are familiar to any computer user. In PyQt API, the **QPushButton** class object presents a button which when clicked can be programmed to invoke a certain function.

QPushButton class inherits its core functionality from **QAbstractButton** class. It is rectangular in shape and a text caption or icon can be displayed on its face.

Following are some of the most commonly used methods of QPushButton class:

setCheckable()	Recognizes pressed and released states of button if set to true
toggle()	Toggles between checkable states
setIcon()	Shows an icon formed out of pixmap of an image file
setEnabled()	When set to false, the button becomes disabled, hence clicking it doesn't emit a signal
isChecked()	Returns Boolean state of button
setDefault()	Sets the button as default
setText()	Programmatically sets buttons' caption
text()	Retrieves buttons' caption

Example

Four QPushButton objects are set with some of the above attributes. The example is written in object oriented form, because the source of the event is needed to be passed as an argument to slot function.

Four QPushButton objects are defined as instance variables in the class. First button **b1** is converted into toggle button by the statements:

```
self.b1.setCheckable(True)
self.b1.toggle()
```

Clicked signal of this button is connected to a member method btnstate() which identifies whether button is pressed or released by checking isChecked() property.

```
def btnstate(self):
        if self.b1.isChecked():
            print "button pressed"
        else:
            print "button released"
```

Second button **b2** displays an icon on the face. setIcon() method takes a pixmap object of any image file as argument.

```
b2.setIcon(QIcon(QPixmap("python.gif")))
```

Button **b3** is set to be disabled by using setEnabled() method:

```
b3.setEnabled(False)
```

PushButton **b4** is set to default button by setDefault() method. Shortcut to its caption is created by prefixing & to the caption (&Default). As a result, by using the keyboard combination Alt+D, connected slot method will be called.

Buttons b1 and b4 are connected to whichbtn() slot method. Since the function is intended to retrieve caption of the clicked button, the button object should be passed as an argument. This is achieved by the use of lambda function.

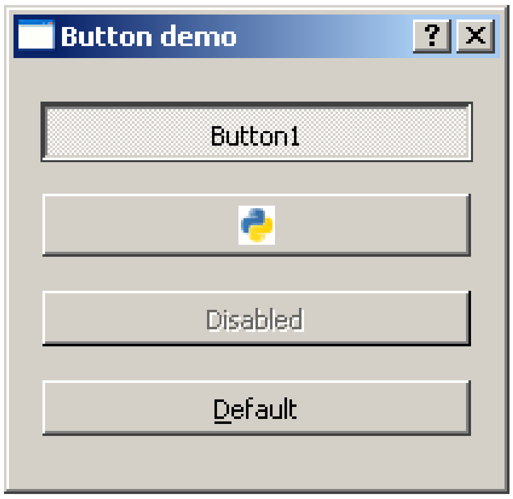
For example,

```
b4.clicked.connect(lambda:self.whichbtn(self.b4))
```

The complete code is given below:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class Form(QDialog):
    def __init__(self, parent=None):
        super(Form, self).__init__(parent)

        layout = QVBoxLayout()
        self.b1=QPushButton("Button1")
        self.b1.setCheckable(True)
        self.b1.toggle()
        self.b1.clicked.connect(lambda:self.whichbtn(self.b1))
        self.b1.clicked.connect(self.btnstate)
        layout.addWidget(self.b1)

        self.b2=QPushButton()
        self.b2.setIcon(QIcon(QPixmap("python.gif")))
        self.b2.clicked.connect(lambda:self.whichbtn(self.b2))
        layout.addWidget(self.b2)
        self.setLayout(layout)
        self.b3=QPushButton("Disabled")
        self.b3.setEnabled(False)
        layout.addWidget(self.b3)

        self.b4=QPushButton("&Default")
        self.b4.setDefault(True)
        self.b4.clicked.connect(lambda:self.whichbtn(self.b4))
        layout.addWidget(self.b4)

        self.setWindowTitle("Button demo")

    def btnstate(self):
        if self.b1.isChecked():
            print "button pressed"
        else:
            print "button released"
    def whichbtn(self,b):
        print "clicked button is "+b.text()

def main():
    app = QApplication(sys.argv)
    ex = Form()
    ex.show()
    sys.exit(app.exec_())



if __name__ == '__main__':
    main()
```

The above code produces the following output.



```
clicked button is Button1
button released
clicked button is Button1
button pressed
clicked button is &Default
```

13. QRadioButton Widget

A **QRadioButton** class object presents a selectable button with a text label. The user can select one of many options presented on the form. This class is derived from QAbstractButton class.

Radio buttons are autoexclusive by default. Hence, only one of the radio buttons in the parent window can be selected at a time. If one is selected, previously selected button is automatically deselected. Radio buttons can also be put in a **QGroupBox** or **QButtonGroup** to create more than one selectable fields on the parent window.

The following listed methods of QRadioButton class are most commonly used.

setChecked()	Changes the state of radio button
setText()	Sets the label associated with the button
text()	Retrieves the caption of button
isChecked()	Checks if the button is selected

Default signal associated with QRadioButton object is toggled(), although other signals inherited from QAbstractButton class can also be implemented.

Example

Here two mutually exclusive radio buttons are constructed on a top level window.

Default state of b1 is set to checked by the statement:

```
Self.b1.setChecked(True)
```

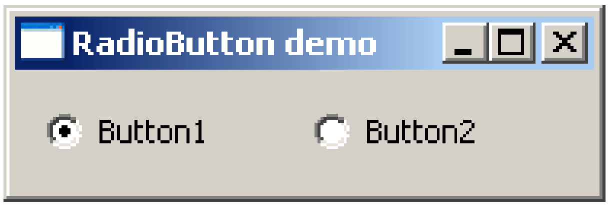
The toggled() signal of both the buttons is connected to btnstate() function. Use of lambda allows the source of signal to be passed to the function as an argument.

```
self.b1.toggled.connect(lambda:self.btnstate(self.b1))
self.b2.toggled.connect(lambda:self.btnstate(self.b2))
```

The btnstate() function checks state of button emitting toggled() signal.

```
if b.isChecked()==True:
                print b.text()+" is selected"
            else:
                print b.text()+" is deselected"
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class Radiodemo(QWidget):
    def __init__(self, parent=None):
        super(Radiodemo, self).__init__(parent)

        layout = QHBoxLayout()
        self.b1=QRadioButton("Button1")
        self.b1.setChecked(True)
        self.b1.toggled.connect(lambda:self.btnstate(self.b1))
        layout.addWidget(self.b1)

        self.b2=QRadioButton("Button2")
        self.b2.toggled.connect(lambda:self.btnstate(self.b2))

        layout.addWidget(self.b2)
        self.setLayout(layout)
        self.setWindowTitle("RadioButton demo")

    def btnstate(self,b):

        if b.text()=="Button1":
            if b.isChecked()==True:
                print b.text()+" is selected"
            else:
                print b.text()+" is deselected"
        if b.text()=="Button2":
            if b.isChecked()==True:
                print b.text()+" is selected"
            else:
                print b.text()+" is deselected"


def main():

    app = QApplication(sys.argv)
    ex = Radiodemo()
    ex.show()
    sys.exit(app.exec_())


if __name__ == '__main__':
    main()
```

The above code produces the following output:



```
Button1 is deselected
Button2 is selected
Button2 is deselected
Button1 is selected
```

14. QCheckBox Widget

A rectangular box before the text label appears when a **QCheckBox** object is added to the parent window. Just as QRadioButton, it is also a selectable button. Its common use is in a scenario when the user is asked to choose one or more of the available options.

Unlike Radio buttons, check boxes are not mutually exclusive by default. In order to restrict the choice to one of the available items, the check boxes must be added to QButtonGroup.

The following table lists commonly used QCheckBox class methods:

setChecked()	Changes the state of checkbox button
setText()	Sets the label associated with the button
text()	Retrieves the caption of the button
isChecked()	Checks if the button is selected
setTriState()	Provides no change state to checkbox

Each time a checkbox is either checked or cleared, the object emits stateChanged() signal.

Example

Here, two QCheckBox objects are added to a horizontal layout. Their stateChanged() signal is connected to btnstate() function. The source object of signal is passed to the function using lambda.

```
self.b1.stateChanged.connect(lambda:self.btnstate(self.b1))
self.b2.toggled.connect(lambda:self.btnstate(self.b2))
```

The isChecked() function is used to check if the button is checked or not.

```
if b.text()=="Button1":
            if b.isChecked()==True:
                print b.text()+" is selected"
            else:
                print b.text()+" is deselected"
```

The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class checkdemo(QWidget):
    def __init__(self, parent=None):
        super(checkdemo, self).__init__(parent)

        layout = QHBoxLayout()
        self.b1=QCheckBox("Button1")
        self.b1.setChecked(True)
        self.b1.stateChanged.connect(lambda:self.btnstate(self.b1))
        layout.addWidget(self.b1)

        self.b2=QCheckBox("Button2")
        self.b2.toggled.connect(lambda:self.btnstate(self.b2))

        layout.addWidget(self.b2)
        self.setLayout(layout)
        self.setWindowTitle("checkbox demo")

    def btnstate(self,b):
        if b.text()=="Button1":
            if b.isChecked()==True:
                print b.text()+" is selected"
            else:
                print b.text()+" is deselected"
        if b.text()=="Button2":
            if b.isChecked()==True:
                print b.text()+" is selected"
            else:
                print b.text()+" is deselected"

def main():

    app = QApplication(sys.argv)
    ex = checkdemo()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

As mentioned earlier, checkBox buttons can be made mutually exclusive by adding them in the QButtonGroup object.

```
self.bg=QButtonGroup()
self.bg.addButton(self.b1,1)
self.bg.addButton(self.b2,2)
```

QButtonGroup object, provides abstract container for buttons and doesn’t have a visual representation. It emits buttonCliked() signal and sends Button object’s reference to the slot function btngroup().

```
self.bg.buttonClicked[QAbstractButton].connect(self.btngroup)
```

The btngroup() function displays the caption of the clicked checkbox.

```
def btngroup(self,btn):
        print btn.text()+" is selected"
```

15. QComboBox Widget

A **QComboBox** object presents a dropdown list of items to select from. It takes minimum screen space on the form required to display only the currently selected item.

A Combo box can be set to be editable; it can also store pixmap objects. The following methods are commonly used:

addItem()	Adds string to collection
addItems()	Adds items in a list object
Clear()	Deletes all items in the collection
count()	Retrieves number of items in the collection
currentText()	Retrieves the text of currently selected item
itemText()	Displays text belonging to specific index
currentIndex()	Returns index of selected item
setItemText()	Changes text of specified index

QComboBox Signals

activated()	When the user chooses an item
currentIndexChanged()	Whenever the current index is changed either by the user or programmatically
highlighted()	When an item in the list is highlighted

Example

Let us see how some features of QComboBox widget are implemented in the following example.

Items are added in the collection individually by addItem() method or items in a List object by addItems() method.

```
self.cb.addItem("C++")
self.cb.addItems(["Java", "C#", "Python"])
```

QComboBox object emits currentIndexChanged() signal. It is connected to selectionchange() method.

Items in a combo box are listed using itemText() method for each item. Label belonging to the currently chosen item is accessed by currentText() method.

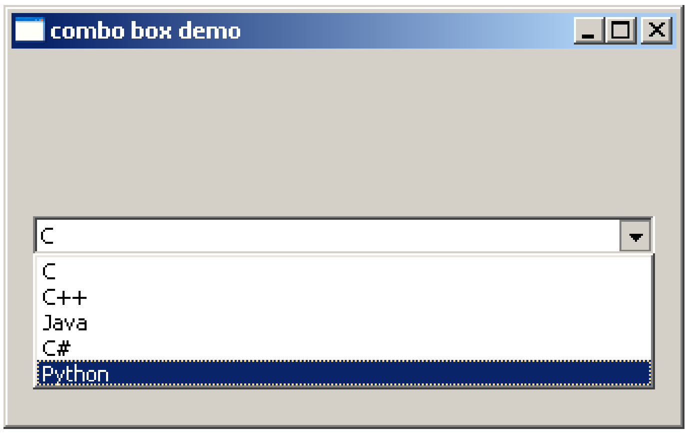
```
def selectionchange(self,i):
        print "Items in the list are :"
        for count in range(self.cb.count()):
            print self.cb.itemText(count)
        print "Current index",i,"selection changed ",self.cb.currentText()
```

The entire code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class combodemo(QWidget):
    def __init__(self, parent=None):
        super(combodemo, self).__init__(parent)

        layout = QHBoxLayout()
        self.cb = QComboBox()
        self.cb.addItem("C")
        self.cb.addItem("C++")
        self.cb.addItems(["Java", "C#", "Python"])
        self.cb.currentIndexChanged.connect(self.selectionchange)
        layout.addWidget(self.cb)
        self.setLayout(layout)
        self.setWindowTitle("combo box demo")

    def selectionchange(self,i):
        print "Items in the list are :"
        for count in range(self.cb.count()):
            print self.cb.itemText(count)
        print "Current index",i,"selection changed ",self.cb.currentText()

def main():
    app = QApplication(sys.argv)
    ex = combodemo()
    ex.show()
    sys.exit(app.exec_())
```

```
if __name__ == '__main__':
    main()
```

The above code produces the following output:



Items in the list are:

C

C++

Java

C#

Python

Current selection index 4 selection changed Python

16. QSpinBox Widget

A **QSpinBox** object presents the user with a textbox which displays an integer with up/down button on its right. The value in the textbox increases/decreases if the up/down button is pressed.

By default, the integer number in the box starts with 0, goes upto 99 and changes by step 1. Use QDoubleSpinBox for float values.

Important methods of QSpinBox class are listed in the following table:

setMinimum()	Sets the lower bound of counter
setMaximum()	Sets the upper bound of counter
setRange()	Sets the minimum, maximum and step value
setValue()	Sets the value of spin box programmatically
Value()	Returns the current value
singleStep()	Sets the step value of counter

QSpinBox object emits valueChanged() signal every time when up/own button is pressed. The associated slot function can retrieve current value of the widget by value() method.

Following example has a label (l1) and spinbox (sp) put in vertical layout of a top window. The valueChanged() signal is connected to valuechange() method.

```
self.sp.valueChanged.connect(self.valuechange)
```

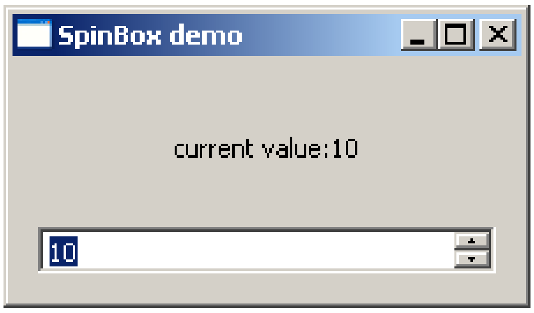
The valueChange() function displays the current value as caption of the label.

```
self.l1.setText("current value:"+str(self.sp.value()))
```

The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class spindemo(QWidget):
    def __init__(self, parent=None):
        super(spindemo, self).__init__(parent)

        layout = QVBoxLayout()
        self.l1=QLabel("current value:")
        self.l1.setAlignment(Qt.AlignCenter)
        layout.addWidget(self.l1)
        self.sp=QSpinBox()
        layout.addWidget(self.sp)
        self.sp.valueChanged.connect(self.valuechange)
        self.setLayout(layout)
        self.setWindowTitle("SpinBox demo")

    def valuechange(self):
        self.l1.setText("current value:"+str(self.sp.value()))

 def main():
    app = QApplication(sys.argv)
    ex = spindemo()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

The above code produces the following output:



current value:10

10

17. QSlider - Widget & Signal

QSlider class object presents the user with a groove over which a handle can be moved. It is a classic widget to control a bounded value. Position of the handle on the groove is equivalent to an integer between the lower and the upper bounds of the control.

A slider control can be displayed in horizontal or vertical manner by mentioning the orientation in the constructor.

```
self.sp=QSlider(Qt.Horizontal)
self.sp=QSlider(Qt.Vertical)
```

The following table lists some of the frequently used methods of QSlider class:

setMinimum()	Sets the lower bound of the slider
setMaximum()	Sets the upper bound of the slider
setSingleStep()	Sets the increment/decrement step
setValue()	Sets the value of the control programmatically
value()	Returns the current value
setTickInterval()	Puts the number of ticks on the groove
setTickPosition()	Places the ticks on the groove. Values are:

QSlider.NoTicks	No tick marks
QSlider.TicksBothSides	Tick marks on both sides
QSlider.TicksAbove	Tick marks above the slider
QSlider.TicksBelow	Tick marks below the slider
QSlider.TicksLeft	Tick marks to the left of the slider
QSlider.TicksRight	Tick marks to the right of the slider

QSlider Signals

Signal	Description
valueChanged()	When the slider's value has changed
sliderPressed()	When the user starts to drag the slider
sliderMoved()	When the user drags the slider

sliderReleased()	When the user releases the slider

valueChanged() signal is the one which is most frequently used.

Example

The following example demonstrates the above functionality. A Label and a horizontal slider is placed in a vertical layout. Slider's valueChanged() signal is connected to valuechange() method.

```
self.sl.valueChanged.connect(self.valuechange)
```

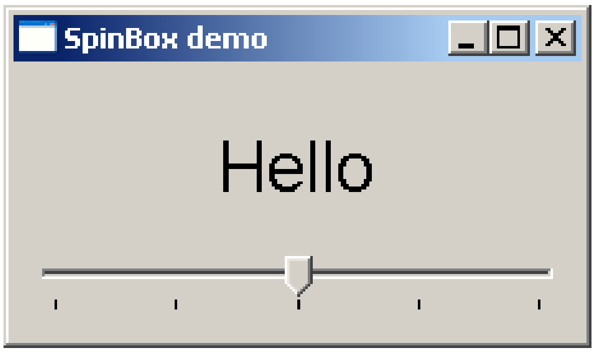
The slot function valuechange() reads current value of the slider and uses it as the size of font for label's caption.

```
size=self.sl.value()
self.l1.setFont(QFont("Arial",size))
```

The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class sliderdemo(QWidget):
    def __init__(self, parent=None):
        super(sliderdemo, self).__init__(parent)

        layout = QVBoxLayout()
        self.l1=QLabel("Hello")
        self.l1.setAlignment(Qt.AlignCenter)
        layout.addWidget(self.l1)
        self.sl=QSlider(Qt.Horizontal)
        self.sl.setMinimum(10)
        self.sl.setMaximum(30)
        self.sl.setValue(20)
        self.sl.setTickPosition(QSlider.TicksBelow)
        self.sl.setTickInterval(5)
        layout.addWidget(self.sl)
        self.sl.valueChanged.connect(self.valuechange)
        self.setLayout(layout)
        self.setWindowTitle("SpinBox demo")
    def valuechange(self):
        size=self.sl.value()
        self.l1.setFont(QFont("Arial",size))

def main():
    app = QApplication(sys.argv)
    ex = sliderdemo()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

The above code produces the following output:



The font size of the label changes as handle of the slider is moved across the handle.

18. QMenuBar, QMenu & QAction Widgets

A horizontal **QMenuBar** just below the title bar of a QMainWindow object is reserved for displaying QMenu objects.

QMenu class provides a widget which can be added to menu bar. It is also used to create context menu and popup menu. Each QMenu object may contain one or more **QAction** objects or cascaded QMenu objects.

To create a popup menu, PyQt API provides createPopupMenu() function. menuBar() function returns main window's QMenuBar object. addMenu() function lets addition of menu to the bar. In turn, actions are added in the menu by addAction() method.

Following table lists some of the important methods used in designing a menu system.

menuBar()	Returns main window's QMenuBar object
addMenu()	Adds a new QMenu object to menu bar
addAction()	Adds an action button to QMenu widget consisting of text or icon
setEnabled()	Sets state of action button to enabled/disabled
addSeperator()	Adds a separator line in the menu
Clear()	Removes contents of menu/menu bar
setShortcut()	Associates keyboard shortcut to action button
setText()	Assigns text to action button
setTitle()	Sets the title of QMenu widget
text()	Retrieves the text associated with QAction object
title()	Retrieves the text associated with QMenu object

QMenu object emits triggered() signal whenever any QAction button is clicked. Reference to the clicked QAction object is passed on to the connected slot function.

Example

In this example, first all reference to QMenuBar object of top level window (which has to be a QMainWindow object) is stored.

```
bar=self.menuBar()
```

File menu is added to the menu bar by addMenu() method.

```
file=bar.addMenu("File")
```

An action button in the menu may be a string or a QAction object.

```
file.addAction("New")
save=QAction("Save",self)
save.setShortcut("Ctrl+S")
file.addAction(save)
```

A submenu is added to top level menu.

```
edit=file.addMenu("Edit")
edit.addAction("copy")
edit.addAction("paste")
```

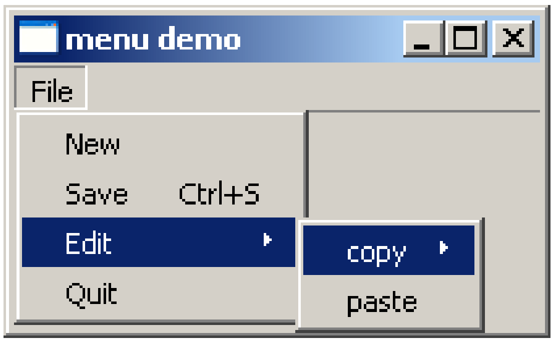
triggered() signal emitted by file menu is connected to processtrigger() method, which receives QAction object causing the signal.

```
file.triggered[QAction].connect(self.processtrigger)
```

The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class menudemo(QMainWindow):
    def __init__(self, parent=None):
        super(menudemo, self).__init__(parent)

        layout = QHBoxLayout()
        bar=self.menuBar()
        file=bar.addMenu("File")
        file.addAction("New")

        save=QAction("Save",self)
        save.setShortcut("Ctrl+S")
        file.addAction(save)

        edit=file.addMenu("Edit")
        edit.addAction("copy")
        edit.addAction("paste")

        quit=QAction("Quit",self)
        file.addAction(quit)
        file.triggered[QAction].connect(self.processtrigger)
        self.setLayout(layout)
        self.setWindowTitle("menu demo")

    def processtrigger(self,q):
        print q.text()+" is triggered"

def main():
    app = QApplication(sys.argv)
    ex = menudemo()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

The above code produces the following output:



19. QToolBar Widget

A **QToolBar** widget is a movable panel consisting of text buttons, buttons with icons or other widgets.

It is usually situated in a horizontal bar below menu bar, although it can be floating. Some useful methods of QToolBar class are as follows:

addAction()	Adds tool buttons having text or icon
addSeperator()	Shows tool buttons in groups
addWidget()	Adds controls other than button in the toolbar
addToolBar()	QMainWindow class method adds a new toolbar
setMovable()	Toolbar becomes movable
setOrientation()	Toolbar's orientation sets to Qt.Horizontal or Qt.vertical

Whenever a button on the toolbar is clicked, ActionTriggered() signal is emitted. Additionally, it sends reference to QAction object associated with the event to the connected function.

A File toolbar is added in the toolbar area by calling addToolBar() method.

```
tb = self.addToolBar("File")
```

Although tool buttons with text captions can be added, a toolbar usually contains graphic buttons. A QAction object with an icon and name is added to the toolbar.

```
new=QAction(QIcon("new.bmp"),"new",self)
 tb.addAction(new)
```

Similarly, open and save buttons are added.

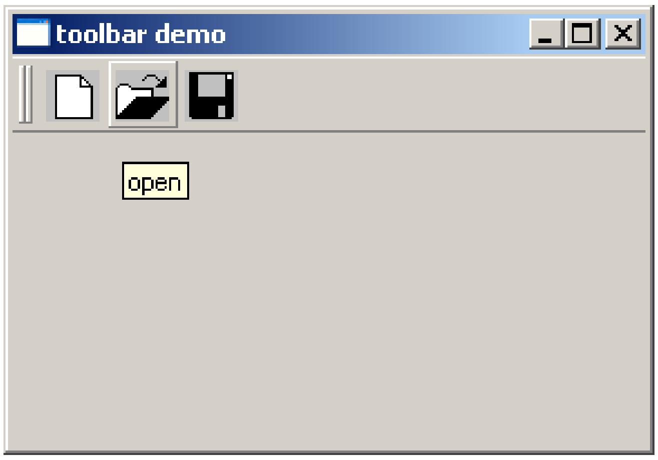
Finally, actionTriggered() signal is connected to a slot function toolbtnpressed()

```
tb.actionTriggered[QAction].connect(self.toolbtnpressed)
```

The complete code to execute the example is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class tooldemo(QMainWindow):
    def __init__(self, parent=None):
        super(tooldemo, self).__init__(parent)
        layout = QVBoxLayout()
        tb = self.addToolBar("File")

        new=QAction(QIcon("new.bmp"),"new",self)
        tb.addAction(new)

        open=QAction(QIcon("open.bmp"),"open",self)
        tb.addAction(open)
        save=QAction(QIcon("save.bmp"),"save",self)
        tb.addAction(save)
        tb.actionTriggered[QAction].connect(self.toolbtnpressed)
        self.setLayout(layout)
        self.setWindowTitle("toolbar demo")

    def toolbtnpressed(self,a):
        print "pressed tool button is",a.text()

def main():
     app = QApplication(sys.argv)
    ex = tooldemo()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

The above code produces the following output:


toolbar demo
open

20. QDialog Class

A **QDialog** widget presents a top level window mostly used to collect response from the user. It can be configured to be **Modal** (where it blocks its parent window) or **Modeless** (the dialog window can be bypassed).

PyQt API has a number of preconfigured Dialog widgets such as InputDialog, FileDialog, FontDialog, etc.

Example

In the following example, WindowModality attribute of Dialog window decides whether it is modal or modeless. Any one button on the dialog can be set to be default. The dialog is discarded by QDialog.reject() method when the user presses the Escape key.

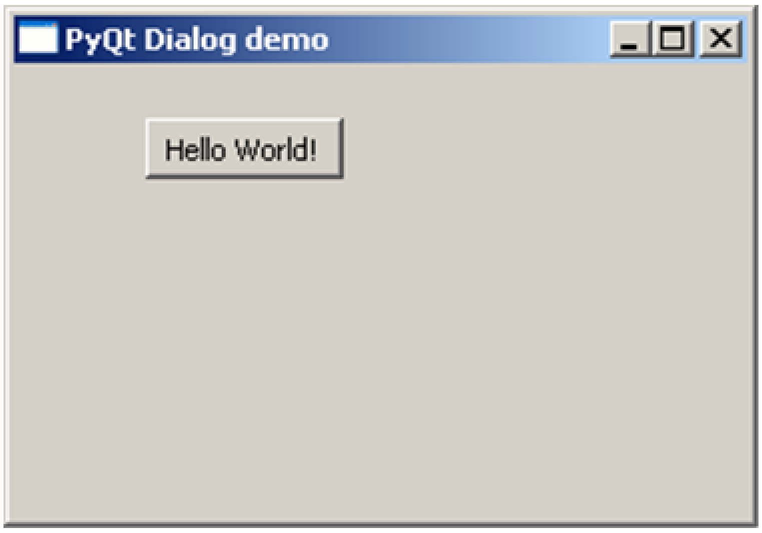
A PushButton on a top level QWidget window, when clicked, produces a Dialog window. A Dialog box doesn't have minimize and maximize controls on its title bar.

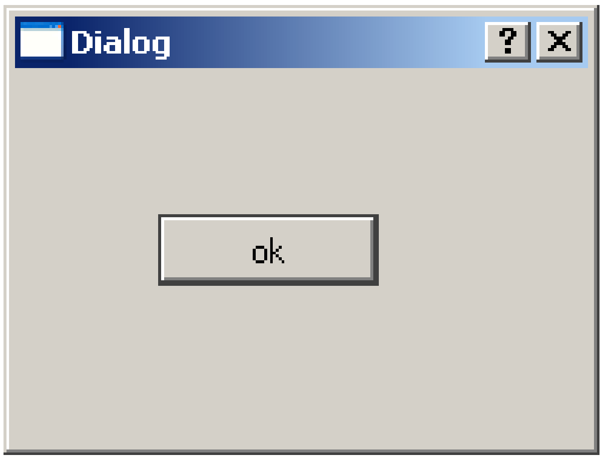
The user cannot relegate this dialog box in the background because its WindowModality is set to ApplicationModal.

```
import sys
from PyQt4.QtGui import *
from PyQt4.QtCore import *
def window():
    app = QApplication(sys.argv)
    w = QWidget()
    b= QPushButton(w)
    b.setText("Hello World!")
    b.move(50,50)
    b.clicked.connect(showdialog)
    w.setWindowTitle("PyQt Dialog demo")
    w.show()
    sys.exit(app.exec_())

def showdialog():
   d=QDialog()
    b1=QPushButton("ok",d)
    b1.move(50,50)
    d.setWindowTitle("Dialog")
    d.setWindowModality(Qt.ApplicationModal)
    d.exec_()
```

```
if __name__ == '__main__':
    window()
```

The above code produces the following output:





21. QMessageBox

QMessageBox is a commonly used modal dialog to display some informational message and optionally ask the user to respond by clicking any one of the standard buttons on it. Each standard button has a predefined caption, a role and returns a predefined hexadecimal number.

Important methods and enumerations associated with QMessageBox class are given in the following table:

setIcon()	Displays predefined icon corresponding to severity of the message Question Information Warning Critical
setText()	Sets the text of the main message to be displayed
setInformativeText()	Displays additional information
setDetailText()	Dialog shows a Details button. This text appears on clicking it
setTitle()	Displays the custom title of dialog
setStandardButtons()	List of standard buttons to be displayed. Each button is associated with QMessageBox.Ok 0x00000400 QMessageBox.Open 0x00002000 QMessageBox.Save 0x00000800 QMessageBox.Cancel 0x00400000 QMessageBox.Close 0x00200000 QMessageBox.Yes 0x00004000 QMessageBox.No 0x00010000 QMessageBox.Abort 0x00040000

	QMessageBox.Retry 0x00080000 QMessageBox.Ignore 0x00100000
setDefaultButton()	Sets the button as default. It emits the clicked signal if Enter is pressed
setEscapeButton()	Sets the button to be treated as clicked if the escape key is pressed

Example

In the following example, click signal of the button on the top level window, the connected function displays the messagebox dialog.

```
msg=QMessageBox()
msg.setIcon(QMessageBox.Information)
msg.setText("This is a message box")
msg.setInformativeText("This is additional information")
msg.setWindowTitle("MessageBox demo")
msg.setDetailedText("The details are as follows:")
```

setStandardButton() function displays desired buttons.

```
msg.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)
```

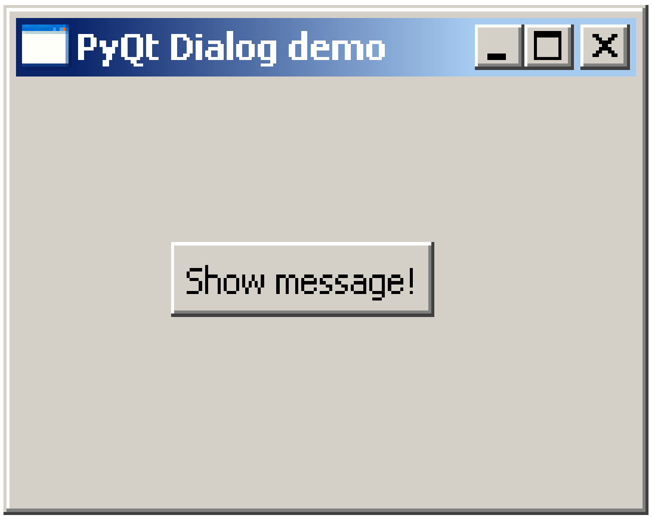
buttonClicked() signal is connected to a slot function, which identifies the caption of source of the signal.

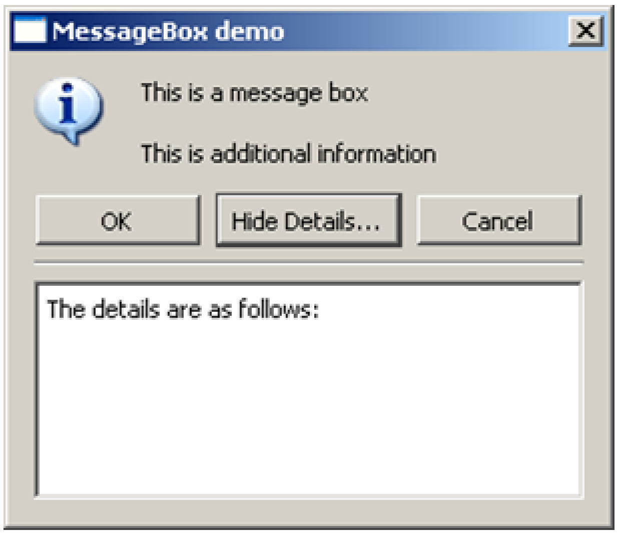
```
msg.buttonClicked.connect(msgbtn)
```

The complete code for the example is as follows:

```
import sys
from PyQt4.QtGui import *
from PyQt4.QtCore import *
def window():
    app = QApplication(sys.argv)
    w = QWidget()
    b= QPushButton(w)
    b.setText("Show message!")

    b.move(50,50)
    b.clicked.connect(showdialog)
    w.setWindowTitle("PyQt Dialog demo")
    w.show()
    sys.exit(app.exec_())

def showdialog():
    msg=QMessageBox()
    msg.setIcon(QMessageBox.Information)

    msg.setText("This is a message box")
    msg.setInformativeText("This is additional information")
    msg.setWindowTitle("MessageBox demo")
    msg.setDetailedText("The details are as follows:")
    msg.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)
    msg.buttonClicked.connect(msgbtn)
    retval=msg.exec_()
    print "value of pressed message box button:", retval

def msgbtn(i):
    print "Button pressed is:",i.text()
if __name__ == '__main__':
    window()
```

The above code produces the following output:





This is a message box

This is additional information

The details are as follows:

22. QInputDialog Widget

This is a preconfigured dialog with a text field and two buttons, OK and Cancel. The parent window collects the input in the text box after the user clicks on Ok button or presses Enter.

The user input can be a number, a string or an item from the list. A label prompting the user what he should do is also displayed.

The QInputDialog class has the following static methods to accept input from the user:

getInt()	Creates a spinner box for integer number
getDouble()	Spinner box with floating point number can be input
getText()	A simple line edit field to type text
getItem()	A combo box from which user can choose item

Example

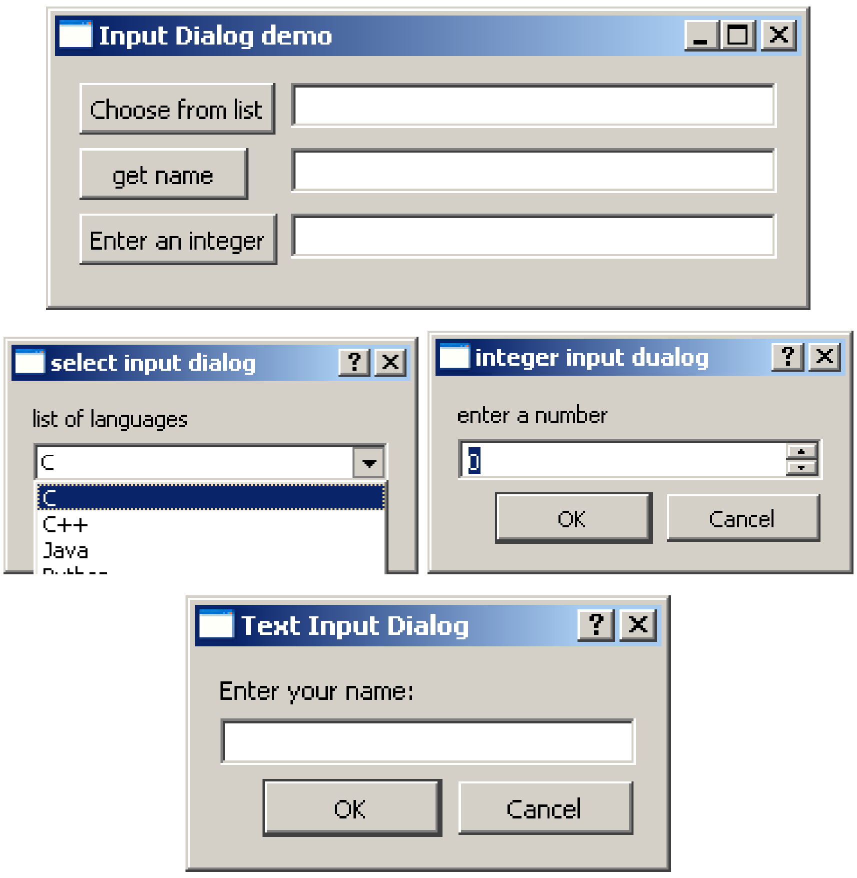
The following example implements the input dialog functionality. The top level window has three buttons. Their clicked() signal pops up InputDialog through connected slots.

```
items = ("C", "C++", "Java", "Python")


item, ok = QInputDialog.getItem(self, "select input dialog",
                "list of languages", items, 0, False)
      if ok and item:
            self.le.setText(item)
      def gettext(self):
        text, ok = QInputDialog.getText(self, 'Text Input Dialog', 'Enter your name:')
        if ok:
            self.le1.setText(str(text))
    def getint(self):
        num,ok=QInputDialog.getInt(self,"integer input dualog","enter a number")
        if ok:
            self.le2.setText(str(num))
```

The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class inputdialogdemo(QWidget):
    def __init__(self, parent=None):
        super(inputdialogdemo, self).__init__(parent)

        layout = QFormLayout()
        self.btn=QPushButton("Choose from list")
        self.btn.clicked.connect(self.getItem)

        self.le=QLineEdit()
        layout.addRow(self.btn,self.le)
        self.btn1=QPushButton("get name")
        self.btn1.clicked.connect(self.gettext)

        self.le1=QLineEdit()
        layout.addRow(self.btn1,self.le1)
        self.btn2=QPushButton("Enter an integer")
        self.btn2.clicked.connect(self.getint)

        self.le2=QLineEdit()
        layout.addRow(self.btn2,self.le2)
        self.setLayout(layout)
        self.setWindowTitle("Input Dialog demo")

    def getItem(self):
        items = ("C", "C++", "Java", "Python")

        item, ok = QInputDialog.getItem(self, "select input dialog",
                "list of languages", items, 0, False)
        if ok and item:
            self.le.setText(item)

    def gettext(self):
        text, ok = QInputDialog.getText(self, 'Text Input Dialog', 'Enter your name:')
        if ok:
            self.le1.setText(str(text))
    def getint(self):
        num,ok=QInputDialog.getInt(self,"integer input dualog","enter a number")
        if ok:
            self.le2.setText(str(num))

def main():
    app = QApplication(sys.argv)
    ex = inputdialogdemo()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

The above code produces the following output:



23. QFontDialog Widget

Another commonly used dialog, a font selector widget is the visual appearance of **QDialog** class. Result of this dialog is a Qfont object, which can be consumed by the parent window.

The class contains a static method getFont(). It displays the font selector dialog. setCurrentFont() method sets the default Font of the dialog.

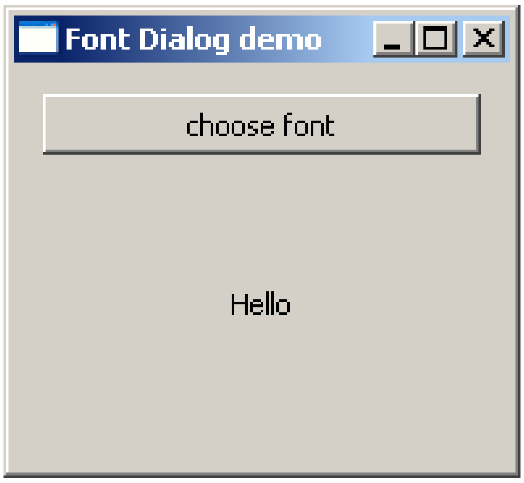
Example

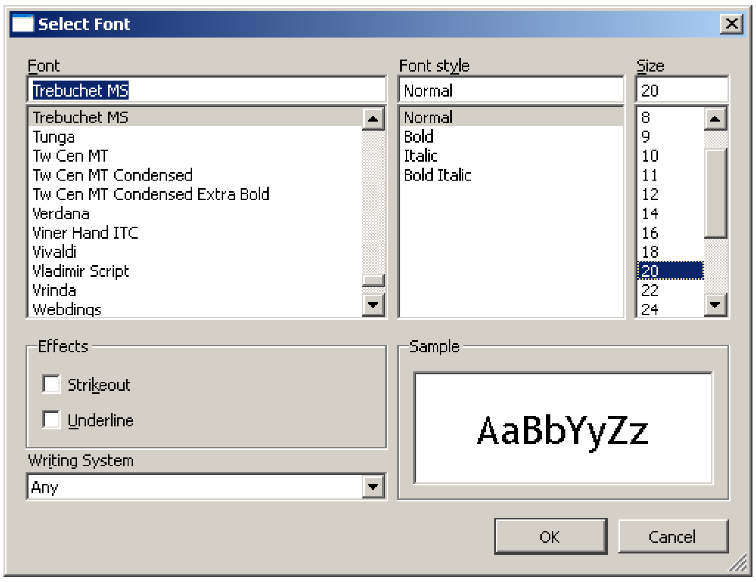
The following example has a button and a label. When the button is clicked, the font dialog pops up. The font chosen by the user (face, style and size) is applied to the text on the label.

The complete code is:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class fontdialogdemo(QWidget):
    def __init__(self, parent=None):
        super(fontdialogdemo, self).__init__(parent)

        layout = QVBoxLayout()
        self.btn=QPushButton("choose font")
        self.btn.clicked.connect(self.getfont)
        layout.addWidget(self.btn)
        self.le=QLabel("Hello")
        layout.addWidget(self.le)
        self.setLayout(layout)
        self.setWindowTitle("Font Dialog demo")

    def getfont(self):
        font, ok = QFontDialog.getFont()
        if ok:
            self.le.setFont(font)



def main():
    app = QApplication(sys.argv)
    ex = fontdialogdemo()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

The above code produces the following output:





24. QFileDialog Widget

This widget is a file selector dialog. It enables the user to navigate through the file system and select a file to open or save. The dialog is invoked either through static functions or by calling exec_() function on the dialog object.

Static functions of **QFileDialog** class (getOpenFileName() and getSaveFileName()) call the native file dialog of the current operating system.

A file filter can also applied to display only files of the specified extensions. The starting directory and default file name can also be set.

Important methods and enumerations of QFileDialog class are listed in the following table:

getOpenFileName()	Returns name of the file selected by the user to open it
getSaveFileName()	Uses the file name selected by the user to save the file
setacceptMode()	Determines whether the file box acts as open or save dialog QFileDialog.AcceptOpen QFileDialog.AcceptSave
setFileMode()	Type of selectable files. Enumerated constants are: QFileDialog.AnyFile QFileDialog.ExistingFile QFileDialog.Directory QFileDialog.ExistingFiles
setFilter()	Displays only those files having mentioned extensions

Example

Both methods of invoking the file dialog are demonstrated in the following example.

The first button invokes the file dialog by the static method.

```
fname = QFileDialog.getOpenFileName(self, 'Open file', 'c:\\',"Image files (*.jpg
*.gif)")
```

The selected image file is displayed on a label widget. The second button invokes the file dialog by calling exec_() method on QFileDialog object.

```
dlg=QFileDialog()
dlg.setFileMode(QFileDialog.AnyFile)
dlg.setFilter("Text files (*.txt)")
filenames=QStringList()
    if dlg.exec_():
       filenames=dlg.selectedFiles()
```

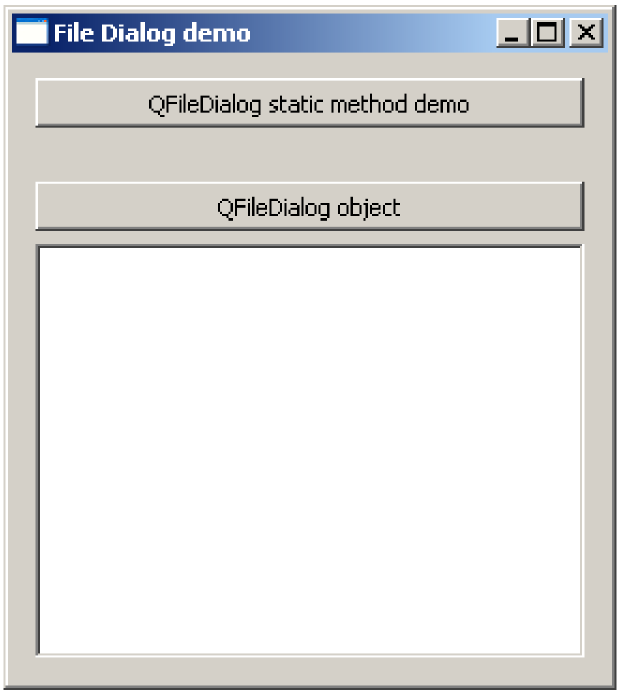
The contents of the selected file are displayed in the TextEdit widget.

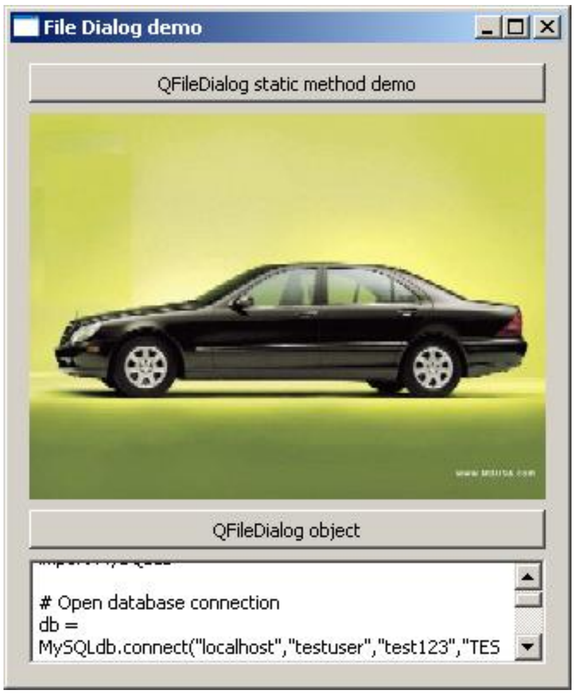
```
f = open(filenames[0], 'r')
            with f:
                data = f.read()
                self.contents.setText(data)
```

The complete code is as follows:

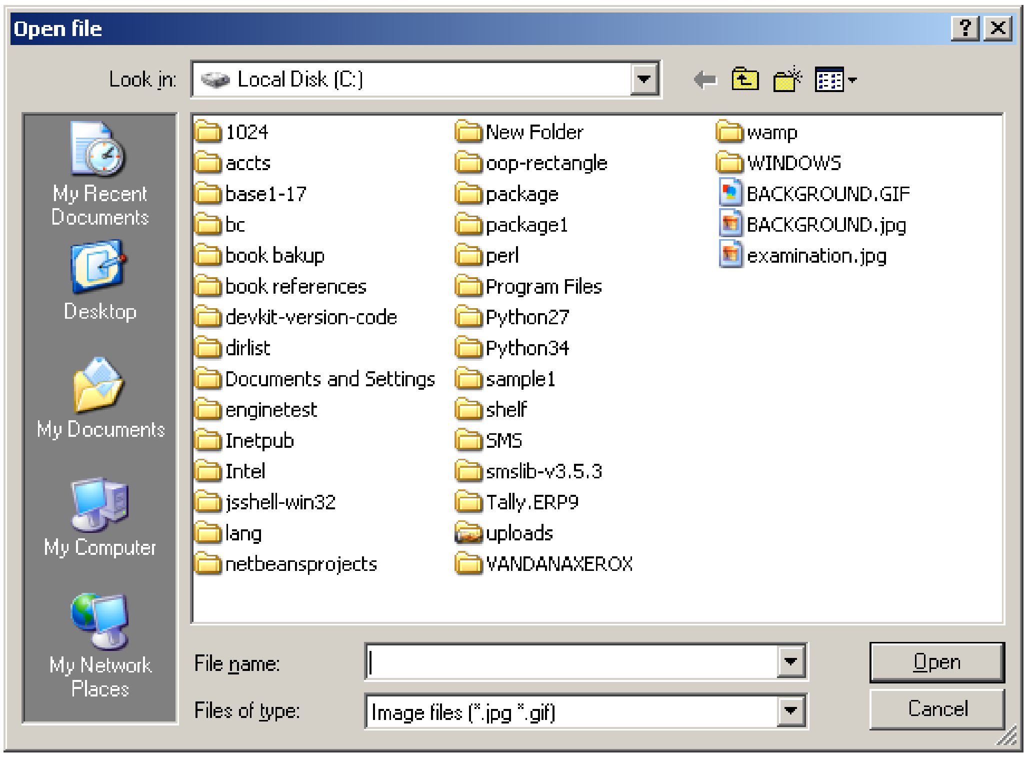
```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class filedialogdemo(QWidget):
    def __init__(self, parent=None):
        super(filedialogdemo, self).__init__(parent)

        layout = QVBoxLayout()
        self.btn=QPushButton("QFileDialog static method demo")
        self.btn.clicked.connect(self.getfile)
        layout.addWidget(self.btn)
        self.le=QLabel("Hello")
        layout.addWidget(self.le)
        self.btn1=QPushButton("QFileDialog object")
        self.btn1.clicked.connect(self.getfiles)
        layout.addWidget(self.btn1)

        self.contents=QTextEdit()
        layout.addWidget(self.contents)
        self.setLayout(layout)
        self.setWindowTitle("File Dialog demo")
    def getfile(self):
        fname = QFileDialog.getOpenFileName(self, 'Open file', 'c:\\',"Image files
(*.jpg *.gif)")
        self.le.setPixmap(QPixmap(fname))

    def getfiles(self):
        dlg=QFileDialog()
        dlg.setFileMode(QFileDialog.AnyFile)
        dlg.setFilter("Text files (*.txt)")
        filenames=QStringList()
        if dlg.exec_():
            filenames=dlg.selectedFiles()
            f = open(filenames[0], 'r')
            with f:
                data = f.read()
                self.contents.setText(data)

def main():

    app = QApplication(sys.argv)
    ex = filedialogdemo()
    ex.show()
    sys.exit(app.exec_())


if __name__ == '__main__':
    main()
```

The above code produces the following output:





```
# Open database connection
db =
MySQLdb.connect("localhost","testuser","test123","TES
```



25. QTabWidget

If a form has too many fields to be displayed simultaneously, they can be arranged in different pages placed under each tab of a Tabbed Widget. The QTabWidget provides a tab bar and a page area. The page under the first tab is displayed and others are hidden. The user can view any page by clicking on the desired tab.

Following are some of the frequently used methods of QTabWidget class:

addTab()	Adds a tab associated with a widget page
insertTab()	Inserts a tab with the page at the desired position
removeTab()	Removes tab at given index
setCurrentIndex()	Sets the index of the currently visible page as current
setCurrentWidget()	Makes the visible page as current
setTabBar()	Sets the tab bar of the widget
setTabPosition()	Position of the tabs are controlled by the values QTabWidget.North above the pages QTabWidget.South below the pages QTabWidget.West to the left of the pages QTabWidget.East to the right of the pages
setTabText()	Defines the label associated with the tab index

The following signals are associated with QTabWidget object:

currentChanged()	Whenever the current page index changes
tabClosedRequested()	When the close button on the tab is clicked

Example

In the following example, the contents of a form are grouped in three categories. Each group of widgets is displayed under a different tab.

Top level window itself is a QTabWidget. Three tabs are added into it.

```
self.addTab(self.tab1,"Tab 1")
self.addTab(self.tab2,"Tab 2")
self.addTab(self.tab3,"Tab 3")
```

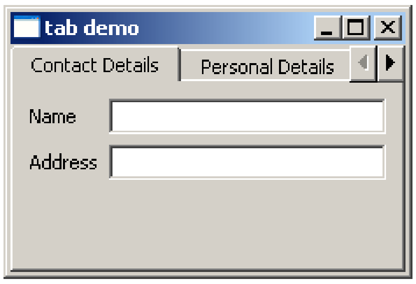
Each tab displays a sub form designed using a layout. Tab text is altered by the statement.

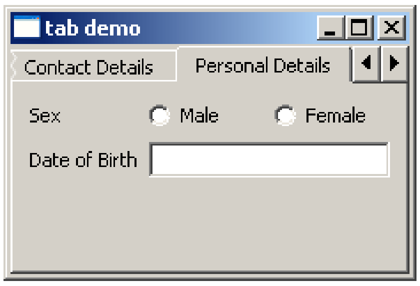
```
self.setTabText(0,"Contact Details")
self.setTabText(1,"Personal Details")
self.setTabText(2,"Education Details")
```

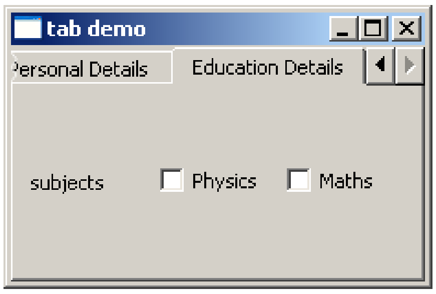
The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class tabdemo(QTabWidget):
    def __init__(self, parent=None):
        super(tabdemo, self).__init__(parent)
        self.tab1 = QWidget()
        self.tab2 = QWidget()
        self.tab3 = QWidget()

        self.addTab(self.tab1,"Tab 1")
        self.addTab(self.tab2,"Tab 2")
        self.addTab(self.tab3,"Tab 3")
        self.tab1UI()
        self.tab2UI()
        self.tab3UI()
        self.setWindowTitle("tab demo")

    def tab1UI(self):
        layout=QFormLayout()
        layout.addRow("Name",QLineEdit())
        layout.addRow("Address",QLineEdit())
        self.setTabText(0,"Contact Details")
        self.tab1.setLayout(layout)
    def tab2UI(self):
        layout=QFormLayout()
        sex=QHBoxLayout()
        sex.addWidget(QRadioButton("Male"))
        sex.addWidget(QRadioButton("Female"))
        layout.addRow(QLabel("Sex"),sex)
        layout.addRow("Date of Birth",QLineEdit())
        self.setTabText(1,"Personal Details")
        self.tab2.setLayout(layout)
    def tab3UI(self):
        layout=QHBoxLayout()
        layout.addWidget(QLabel("subjects"))
        layout.addWidget(QCheckBox("Physics"))
        layout.addWidget(QCheckBox("Maths"))
        self.setTabText(2,"Education Details")
        self.tab3.setLayout(layout)


def main():

    app = QApplication(sys.argv)
    ex = tabdemo()
    ex.show()
    sys.exit(app.exec_())


if __name__ == '__main__':
    main()
```

The above code produces the following output:







26. QStackedWidget

Functioning of **QStackedWidget** is similar to QTabWidget. It also helps in the efficient use of window's client area.

QStackedWidget provides a stack of widgets, only one of which can be viewed at a time. It is a useful layout built on top of QStackedLayout.

Example

A parent QStackedWidget object is populated with more than one child widget.

```
self.Stack = QStackedWidget (self)
self.stack1= QWidget()
self.stack2= QWidget()
self.stack3= QWidget()
self.Stack.addWidget (self.stack1)
self.Stack.addWidget (self.stack2)
self.Stack.addWidget (self.stack3)
```

Each child widget can have its own layout of form elements. QStackedWidget on its own cannot switch between the pages. It is linked with the currently selected index of QListWidget.

```
self.leftlist = QListWidget ()
self.leftlist.insertItem (0, 'Contact' )
self.leftlist.insertItem (1, 'Personal' )
self.leftlist.insertItem (2, 'Educational' )
self.leftlist.currentRowChanged.connect(self.display)
```

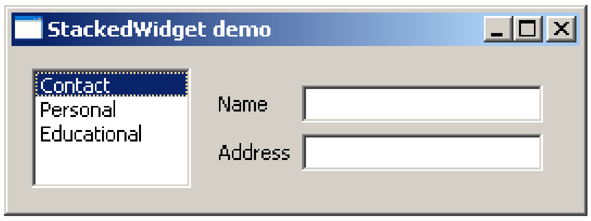
Here, the currentRowChanged() signal of QListWidget is connected to display() function, which changes the view of stacked widget.

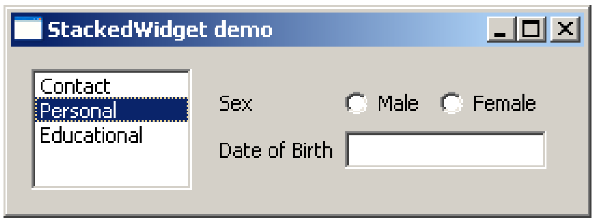
```
def display(self,i):
        self.Stack.setCurrentIndex(i)
```

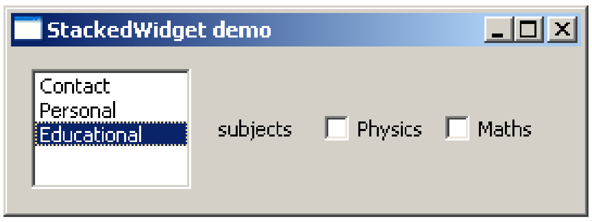
The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class stackedExample(QWidget):

    def __init__(self):
        super(stackedExample, self).__init__()
        self.leftlist = QListWidget ()
        self.leftlist.insertItem (0, 'Contact' )
        self.leftlist.insertItem (1, 'Personal' )
        self.leftlist.insertItem (2, 'Educational' )
        self.stack1= QWidget()
        self.stack2= QWidget()
        self.stack3= QWidget()
        self.stack1UI()
        self.stack2UI()
        self.stack3UI()
        self.Stack = QStackedWidget (self)
        self.Stack.addWidget (self.stack1)
        self.Stack.addWidget (self.stack2)
        self.Stack.addWidget (self.stack3)
        hbox = QHBoxLayout(self)
        hbox.addWidget(self.leftlist)
        hbox.addWidget(self.Stack)

        self.setLayout(hbox)
        self.leftlist.currentRowChanged.connect(self.display)
        self.setGeometry(300, 50, 10,10)
        self.setWindowTitle('StackedWidget demo')
        self.show()
    def stack1UI(self):
        layout=QFormLayout()
        layout.addRow("Name",QLineEdit())
        layout.addRow("Address",QLineEdit())
        #self.setTabText(0,"Contact Details")
        self.stack1.setLayout(layout)
    def stack2UI(self):
        layout=QFormLayout()
        sex=QHBoxLayout()
        sex.addWidget(QRadioButton("Male"))
        sex.addWidget(QRadioButton("Female"))
        layout.addRow(QLabel("Sex"),sex)
        layout.addRow("Date of Birth",QLineEdit())
        self.stack2.setLayout(layout)
    def stack3UI(self):
        layout=QHBoxLayout()
        layout.addWidget(QLabel("subjects"))
        layout.addWidget(QCheckBox("Physics"))
        layout.addWidget(QCheckBox("Maths"))
        self.stack3.setLayout(layout)
    def display(self,i):
        self.Stack.setCurrentIndex(i)

def main():
    app = QApplication(sys.argv)
    ex = stackedExample()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

The above code produces the following output:







27. QSplitter Widget

This is another advanced layout manager which allows the size of child widgets to be changed dynamically by dragging the boundaries between them. The Splitter control provides a handle that can be dragged to resize the controls.

The widgets in a **QSplitter** object are laid horizontally by default although the orientation can be changed to Qt.Vertical.

Following are the methods and signals of QSplitter class:

addWidget()	Adds the widget to splitter's layout
indexOf()	Returns the index of the widget in the layout
insetWidget()	Inserts a widget at the specified index
setOrientation()	Sets the layout of splitter to Qt.Horizontal or Qt.Vertical
setSizes()	Sets the initial size of each widget
count()	Returns the number of widgets in splitter widget

splitterMoved() is the only signal emitted by QSplitter object whenever the splitter handle is dragged.

Example

The following example has a splitter object, splitter1, in which a frame and QTextEdit object are horizontally added.

```
topleft = QFrame()
textedit=QTextEdit()
splitter1.addWidget(topleft)
splitter1.addWidget(textedit)
```

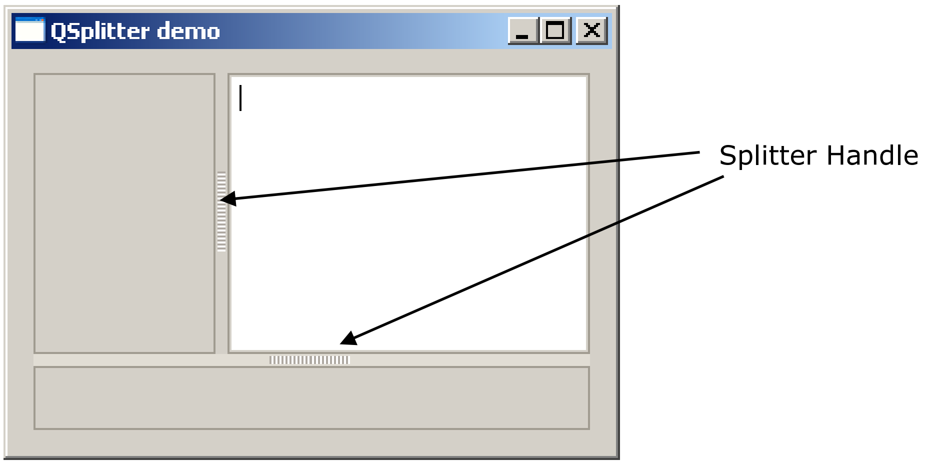
This splitter object splitter1 and a bottom frame object are added in another splitter, splitter2, vertically. The object splitters is finally added in the top level window.

```
bottom = QFrame()
splitter2 = QSplitter(Qt.Vertical)
splitter2.addWidget(splitter1)
splitter2.addWidget(bottom)
hbox.addWidget(splitter2)
self.setLayout(hbox)
```

The complete code is as follows:

```
import sys
from PyQt4.QtGui import *
from PyQt4.QtCore import *

class Example(QWidget):

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

        self.initUI()

    def initUI(self):

        hbox = QHBoxLayout(self)

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

        splitter1 = QSplitter(Qt.Horizontal)
        textedit=QTextEdit()
        splitter1.addWidget(topleft)
        splitter1.addWidget(textedit)
        splitter1.setSizes([100,200])

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

        hbox.addWidget(splitter2)

        self.setLayout(hbox)
        QApplication.setStyle(QStyleFactory.create('Cleanlooks'))

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

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

if __name__ == '__main__':
    main()
```

The above code produces the following output:



28. Multiple Document Interface

A typical GUI application may have multiple windows. Tabbed and stacked widgets allow to activate one such window at a time. However, many a times this approach may not be useful as view of other windows is hidden.

One way to display multiple windows simultaneously is to create them as independent windows. This is called as SDI (single Document Interface). This requires more memory resources as each window may have its own menu system, toolbar, etc.

MDI (Multiple Document Interface) applications consume lesser memory resources. The sub windows are laid down inside main container with relation to each other. The container widget is called **QMdiArea**.

QMdiArea widget generally occupies the central widget of QMainWondow object. Child windows in this area are instances of QMdiSubWindow class. It is possible to set any QWidget as the internal widget of subWindow object. Sub-windows in the MDI area can be arranged in cascaded or tile fashion.

The following table lists important methods of QMdiArea class and QMdiSubWindow class:

addSubWindow()	Adds a widget as a new subwindow in MDI area
removeSubWindow()	Removes a widget that is internal widget of a subwindow
setActiveSubWindow()	Activates a subwindow
cascadeSubWindows()	Arranges subwindows in MDiArea in a cascaded fashion
tileSubWindows()	Arranges subwindows in MDiArea in a tiled fashion
closeActiveSubWindow()	Closes the active subwindow
subWindowList()	Returns the list of subwindows in MDI Area
setWidget()	Sets a QWidget as an internal widget of a QMdiSubwindow instance

QMdiArea object emits subWindowActivated() signal whereas windowStateChanged() signal is emitted by QMdisubWindow object.

Example

In the following example, top level window comprising of QMainWindow has a menu and MdiArea.

```
self.mdi = QMdiArea()

self.setCentralWidget(self.mdi)
```

```
bar=self.menuBar()
file=bar.addMenu("File")
file.addAction("New")
file.addAction("cascade")
file.addAction("Tiled")
```

Triggered() signal of the menu is connected to windowaction() function.

```
file.triggered[QAction].connect(self.windowaction)
```

The new action of menu adds a subwindow in MDI area with a title having an incremental number to it.

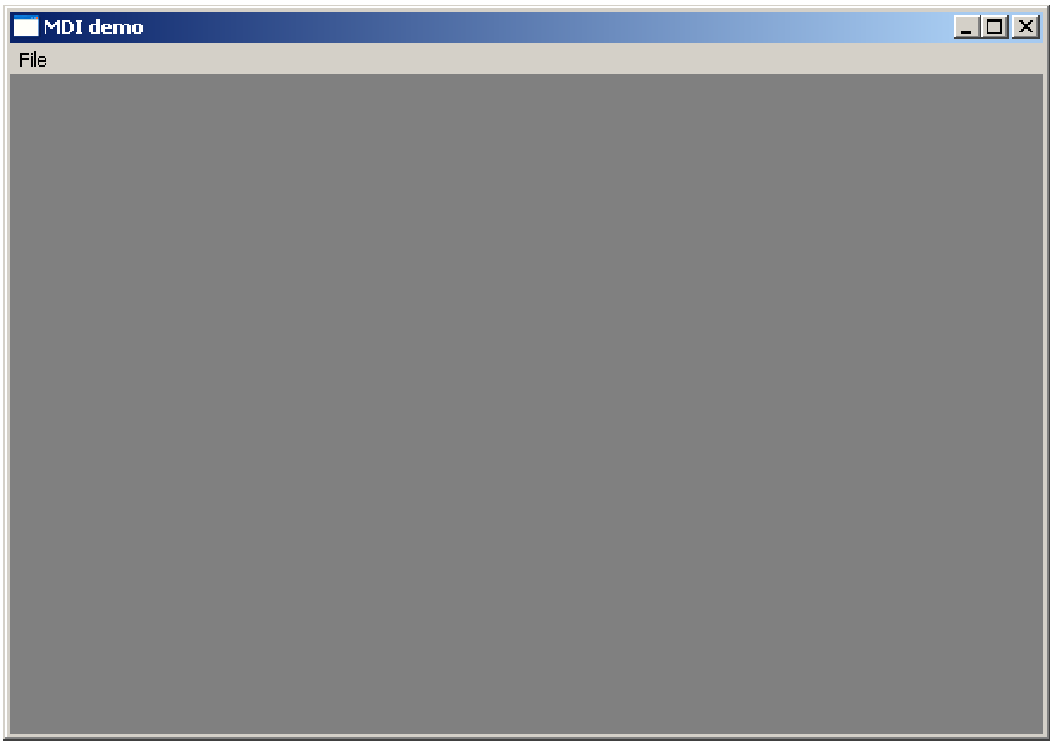
```
MainWindow.count=MainWindow.count+1
sub=QMdiSubWindow()
sub.setWidget(QTextEdit())
sub.setWindowTitle("subwindow"+str(MainWindow.count))
self.mdi.addSubWindow(sub)
sub.show()
```

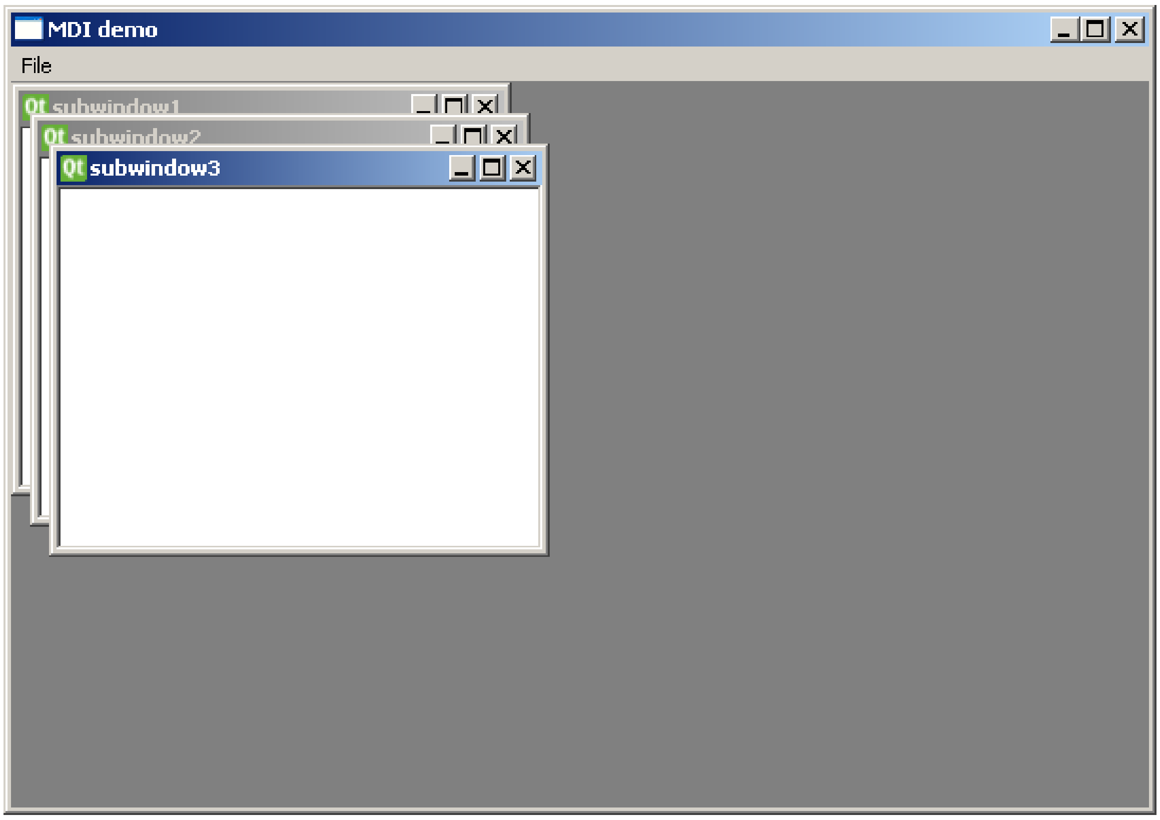
Cascaded and tiled buttons of the menu arrange currently displayed subwindows in cascaded and tiled fashion respectively.

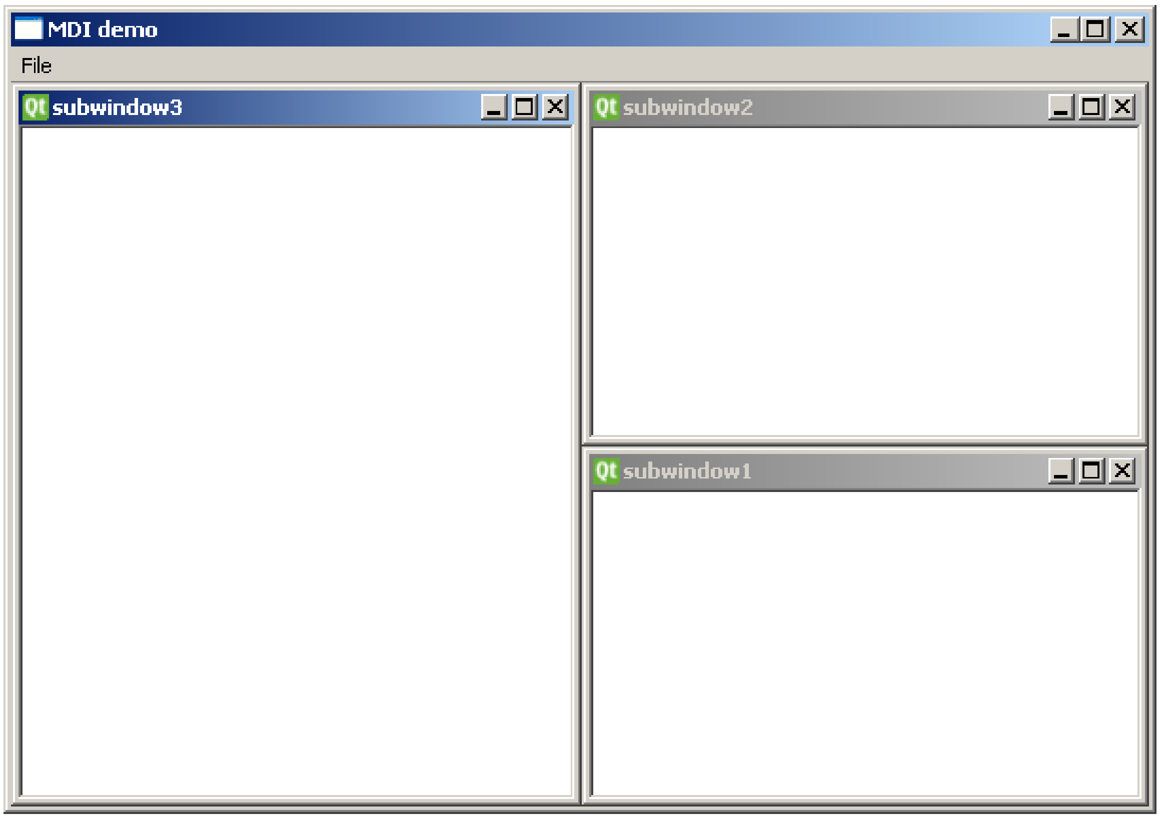
The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class MainWindow(QMainWindow):
    count=0
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.mdi = QMdiArea()
        self.setCentralWidget(self.mdi)
        bar=self.menuBar()
        file=bar.addMenu("File")
        file.addAction("New")
        file.addAction("cascade")
        file.addAction("Tiled")
        file.triggered[QAction].connect(self.windowaction)
        self.setWindowTitle("MDI demo")
    def windowaction(self, q):
        print "triggered"
        if q.text()=="New":
            MainWindow.count=MainWindow.count+1
            sub=QMdiSubWindow()
            sub.setWidget(QTextEdit())
            sub.setWindowTitle("subwindow"+str(MainWindow.count))
            self.mdi.addSubWindow(sub)
            sub.show()
        if q.text()=="cascade":
            self.mdi.cascadeSubWindows()
        if q.text()=="Tiled":
            self.mdi.tileSubWindows()

def main():
    app = QApplication(sys.argv)
    ex = MainWindow()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

The above code produces the following output:


MDI demo
File


MDI demo
File
subwindow1
subwindow2
subwindow3


MDI demo
File
subwindow3
subwindow2
subwindow1

29. Drag and Drop

The provision of **drag and drop** is very intuitive for the user. It is found in many desktop applications where the user can copy or move objects from one window to another.

MIME based drag and drop data transfer is based on QDrag class. **QMimeData** objects associate the data with their corresponding MIME type. It is stored on clipboard and then used in the drag and drop process.

The following QMimeData class functions allow the MIME type to be detected and used conveniently.

Tester	Getter	Setter	MIME Types
hasText()	text()	setText()	text/plain
hasHtml()	html()	setHtml()	text/html
hasUrls()	urls()	setUrls()	text/uri-list
hasImage()	imageData()	setImageData()	image/ *
hasColor()	colorData()	setColorData()	application/x-color

Many QWidget objects support the drag and drop activity. Those that allow their data to be dragged have setDragEnabled() which must be set to true. On the other hand, the widgets should respond to the drag and drop events in order to store the data dragged into them.

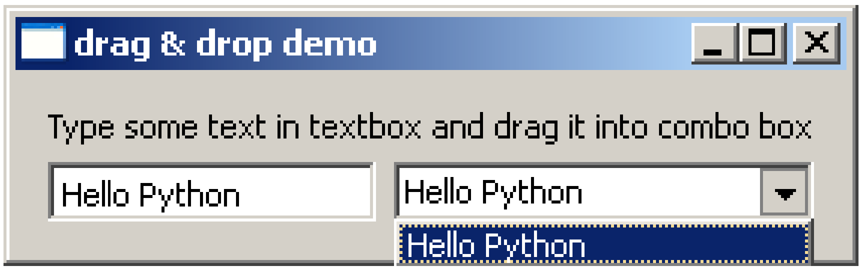
- **DragEnterEvent** provides an event which is sent to the target widget as dragging action enters it.
- **DragMoveEvent** is used when the drag and drop action is in progress.
- **DragLeaveEvent** is generated as the drag and drop action leaves the widget.
- **DropEvent**, on the other hand, occurs when the drop is completed. The event's proposed action can be accepted or rejected conditionally.

Example

In the following code, the DragEnterEvent verifies whether the MIME data of the event contains text. If yes, the event's proposed action is accepted and the text is added as a new item in the ComboBox.

```
import sys
from PyQt4.QtGui import *
from PyQt4.QtCore import *


class combo(QComboBox):
    def __init__(self, title, parent):
        super(combo, self).__init__( parent)

        self.setAcceptDrops(True)

    def dragEnterEvent(self, e):
        print e

        if e.mimeData().hasText():
            e.accept()
        else:
            e.ignore()

    def dropEvent(self, e):
        self.addItem(e.mimeData().text())


class Example(QWidget):

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

        self.initUI()

    def initUI(self):
        lo=QFormLayout()
        lo.addRow(QLabel("Type some text in textbox and drag it into combo box"))

        edit = QLineEdit()
        edit.setDragEnabled(True)
        com = combo("Button", self)
        lo.addRow(edit,com)
        self.setLayout(lo)
        self.setWindowTitle('Simple drag & drop')
def main():
    app = QApplication(sys.argv)
    ex = Example()
    ex.show()
    app.exec_()


if __name__ == '__main__':
    main()
```

The above code produces the following output:


drag & drop demo
Type some text in textbox and drag it into combo box
Hello Python
Hello Python
Hello Python

30. Database Handling

PyQt API contains an elaborate class system to communicate with many SQL based databases. Its QSqlDatabase provides access through a Connection object. Following is the list of currently available SQL drivers:

Driver Type	Description
QDB2	IBM DB2
QIBASE	Borland InterBase Driver
QMYSQL	MySQL Driver
QOCI	Oracle Call Interface Driver
QODBC	ODBC Driver (includes Microsoft SQL Server)
QPSQL	PostgreSQL Driver
QSQLITE	SQLite version 3 or above
QSQLITE2	SQLite version 2

Example

A connection with a SQLite database is established using the static method:

```
db = QtSql.QSqlDatabase.addDatabase('QSQLITE')
db.setDatabaseName('sports.db')
```

Other methods of QSqlDatabase class are as follows:

setDatabaseName()	Sets the name of the database with which connection is sought
setHostName()	Sets the name of the host on which the database is installed
setUserName()	Specifies the user name for connection
setPassword()	Sets the connection object's password if any
commit()	Commits the transactions and returns true if successful
rollback()	Rolls back the database transaction
close()	Closes the connection

QSqlQuery class has the functionality to execute and manipulate SQL commands. Both DDL and DML type of SQL queries can be executed. The most important method in the class is exec_(), which takes as an argument a string containing SQL statement to be executed.

```
query = QtSql.QSqlQuery()
 query.exec_("create table sportsmen(id int primary key, "
               "firstname varchar(20), lastname varchar(20))")
```

The following script creates a SQLite database sports.db with a table of sportsperson populated with five records.

```
from PyQt4 import QtSql, QtGui
def createDB():
    db = QtSql.QSqlDatabase.addDatabase('QSQLITE')
    db.setDatabaseName('sports.db')
    if not db.open():
        QtGui.QMessageBox.critical(None, QtGui.qApp.tr("Cannot open database"),
                QtGui.qApp.tr("Unable to establish a database connection.\n"
                             "This example needs SQLite support. Please read "
                           "the Qt SQL driver documentation for information "
                          "how to build it.\n\n"
                          "Click Cancel to exit."),
                QtGui.QMessageBox.Cancel)
        return False

    query = QtSql.QSqlQuery()
    query.exec_("create table sportsmen(id int primary key, "
                "firstname varchar(20), lastname varchar(20))")
    query.exec_("insert into sportsmen values(101, 'Roger', 'Federer')")
    query.exec_("insert into sportsmen values(102, 'Christiano', 'Ronaldo')")
    query.exec_("insert into sportsmen values(103, 'Ussain', 'Bolt')")
    query.exec_("insert into sportsmen values(104, 'Sachin', 'Tendulkar')")
    query.exec_("insert into sportsmen values(105, 'Saina', 'Nehwal')")
    return True

if __name__ == '__main__':
    import sys

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

QSqlTableModel class in PyQt is a high-level interface that provides editable data model for reading and writing records in a single table. This model is used to populate a QTableView object. It presents to the user a scrollable and editable view that can be put on any top level window.

A QTableModel object is declared in the following manner:

```
model = QtSql.QSqlTableModel()
```

Its editing strategy can be set to any of the following:

QSqlTableModel.OnFieldChange	All changes will be applied immediately
QSqlTableModel.OnRowChange	Changes will be applied when the user selects a different row
QSqlTableModel.OnManualSubmit	All changes will be cached until either submitAll() or revertAll() is called

Example

In the following example, sportsperson table is used as a model and the strategy is set as:

```
model.setTable('sportsmen')
model.setEditStrategy(QtSql.QSqlTableModel.OnFieldChange)

    model.select()
```

QTableView class is part of Model/View framework in PyQt. The QTableView object is created as follows:

```
view = QtGui.QTableView()

view.setModel(model)

view.setWindowTitle(title)

return view
```

This QTableView object and two QPushButton widgets are added to the top level QDialog window. Clicked() signal of add button is connected to addrow() which performs insertRow() on the model table.

```
button.clicked.connect(addrow)

def addrow():

    print model.rowCount()

    ret=model.insertRows(model.rowCount(), 1)

    print ret
```

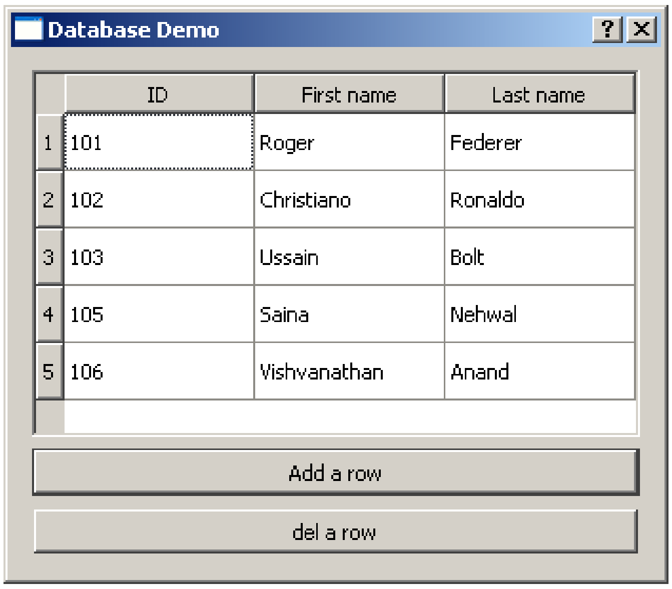
The Slot associated with the delete button executes a lambda function that deletes a row, which is selected by the user.

```
btn1.clicked.connect(lambda: model.removeRow(view1.currentIndex().row()))
```

The complete code is as follows:

```
import sys
from PyQt4 import QtCore, QtGui, QtSql
import sportsconnection

def initializeModel(model):
    model.setTable('sportsmen')
    model.setEditStrategy(QtSql.QSqlTableModel.OnFieldChange)
    model.select()
    model.setHeaderData(0, QtCore.Qt.Horizontal, "ID")
    model.setHeaderData(1, QtCore.Qt.Horizontal, "First name")
    model.setHeaderData(2, QtCore.Qt.Horizontal, "Last name")

def createView(title, model):
    view = QtGui.QTableView()
    view.setModel(model)
    view.setWindowTitle(title)
    return view
def addrow():
    print model.rowCount()
    ret=model.insertRows(model.rowCount(), 1)
    print ret

def findrow(i):
    delrow=i.row()

if __name__ == '__main__':

    app = QtGui.QApplication(sys.argv)

    db = QtSql.QSqlDatabase.addDatabase('QSQLITE')
    db.setDatabaseName('sports.db')
    model = QtSql.QSqlTableModel()
    delrow=-1
    initializeModel(model)

    view1 = createView("Table Model (View 1)", model)
    view1.clicked.connect(findrow)


    dlg=QtGui.QDialog()
    layout = QtGui.QVBoxLayout()
    layout.addWidget(view1)
    button = QtGui.QPushButton("Add a row")
    button.clicked.connect(addrow)
    layout.addWidget(button)
    btn1 = QtGui.QPushButton("del a row")
    btn1.clicked.connect(lambda: model.removeRow(view1.currentIndex().row()))
    layout.addWidget(btn1)
    dlg.setLayout(layout)
    dlg.setWindowTitle("Database Demo")
    dlg.show()
    sys.exit(app.exec_())
```

The above code produces the following output:



	ID	First name	Last name
1	101	Roger	Federer
2	102	Christiano	Ronaldo
3	103	Ussain	Bolt
4	105	Saina	Nehwal
5	106	Vishvanathan	Anand

31. Drawing API in PyQt

All the **QWidget** classes in PyQt are sub classed from QPaintDevice class. A **QPaintDevice** is an abstraction of two dimensional space that can be drawn upon using a QPainter. Dimensions of paint device are measured in pixels starting from the top-left corner.

QPainter class performs low level painting on widgets and other paintable devices such as printer. Normally, it is used in widget's paint event. The **QPaintEvent** occurs whenever the widget's appearance is updated.

The painter is activated by calling the begin() method, while the end() method deactivates it. In between, the desired pattern is painted by suitable methods as listed in the following table.

begin()	Starts painting on the target device
drawArc()	Draws an arc between the starting and the end angle
drawEllipse()	Draws an ellipse inside a rectangle
drawLine()	Draws a line with endpoint coordinates specified
drawPixmap()	Extracts pixmap from the image file and displays it at the specified position
drwaPolygon()	Draws a polygon using an array of coordinates
drawRect()	Draws a rectangle starting at the top-left coordinate with the given width and height
drawText()	Displays the text at given coordinates
fillRect()	Fills the rectangle with the QColor parameter
setBrush()	Sets a brush style for painting
setPen()	Sets the color, size and style of pen to be used for drawing

32. BrushStyle Constants

Predefined QColor Styles

Qt.NoBrush	No brush pattern
Qt.SolidPattern	Uniform color
Qt.Dense1Pattern	Extremely dense brush pattern
Qt.HorPattern	Horizontal lines
Qt.VerPattern	Vertical lines
Qt.CrossPattern	Crossing horizontal and vertical lines
Qt.BDiagPattern	Backward diagonal lines
Qt.FDiagPattern	Forward diagonal lines
Qt.DiagCrossPattern	Crossing diagonal lines

Predefined QColor Objects

Qt.white
Qt.black
Qt.red
Qt.darkRed
Qt.green
Qt.darkGreen
Qt.blue
Qt.cyan
Qt.magenta
Qt.yellow
Qt.darkYellow
Qt.gray

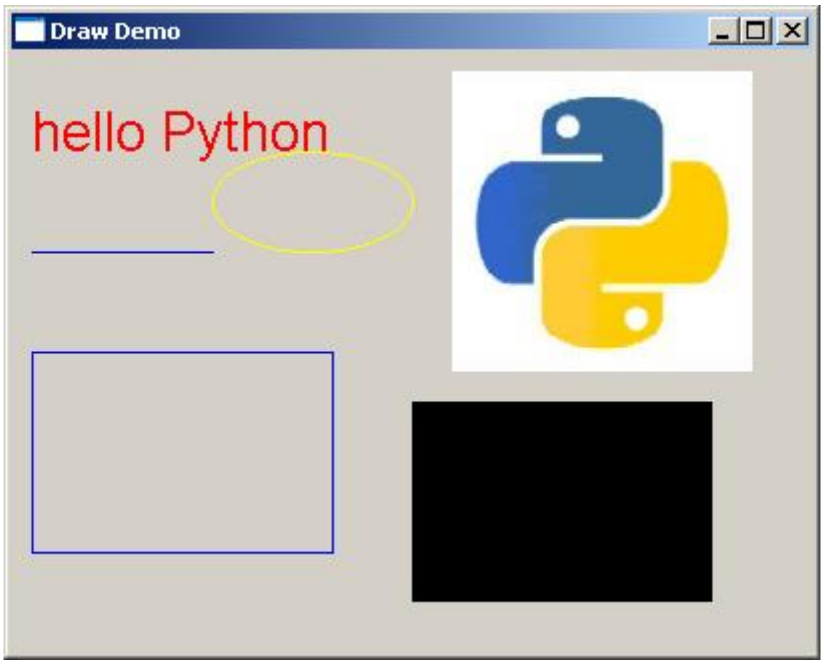
Custom color can be chosen by specifying RGB or CMYK or HSV values.

Example

The following example implements some of these methods.

```
import sys
from PyQt4.QtGui import *
from PyQt4.QtCore import *

class Example(QWidget):

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

    def initUI(self):
        self.text = "hello world"
        self.setGeometry(100,100, 400,300)
        self.setWindowTitle('Draw Demo')
         self.show()

    def paintEvent(self, event):
        qp = QPainter()
        qp.begin(self)
        qp.setPen(QColor(Qt.red))
        qp.setFont(QFont('Arial', 20))
        qp.drawText(10,50, "hello Python")
        qp.setPen(QColor(Qt.blue))
        qp.drawLine(10,100,100,100)
        qp.drawRect(10,150,150,100)
        qp.setPen(QColor(Qt.yellow))
        qp.drawEllipse(100,50,100,50)
        qp.drawPixmap(220,10,QPixmap("python.jpg"))
        qp.fillRect(200,175,150,100,QBrush(Qt.SolidPattern))
        qp.end()

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

if __name__ == '__main__':
    main()
```

The above code produces the following output:



33. QClipboard

The **QClipboard** class provides access to system-wide clipboard that offers a simple mechanism to copy and paste data between applications. Its action is similar to QDrag class and uses similar data types.

QApplication class has a static method clipboard() which returns reference to clipboard object. Any type of MimeData can be copied to or pasted from the clipboard.

Following are the clipboard class methods that are commonly used:

clear()	Clears clipboard contents
setImage()	Copies QImage into clipboard
setMimeData()	Sets MIME data into clipbopard
setPixmap()	Copies Pixmap object in clipboard
setText()	Copies QString in clipboard
text()	Retrieves text from clipboard

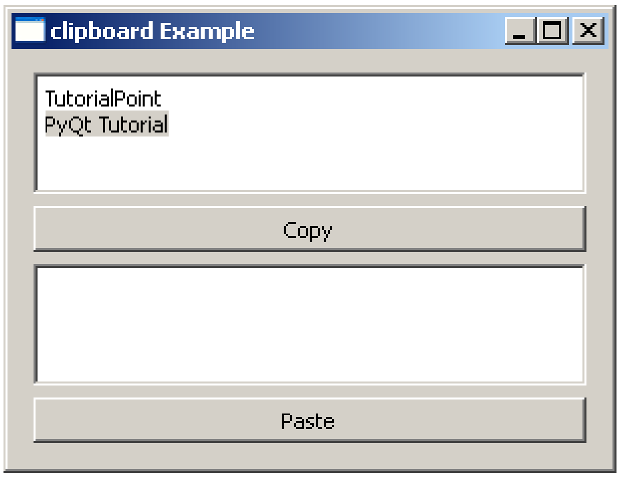
Signal associated with clipboard object is:

dataChanged()	Whenever clipboard data changes

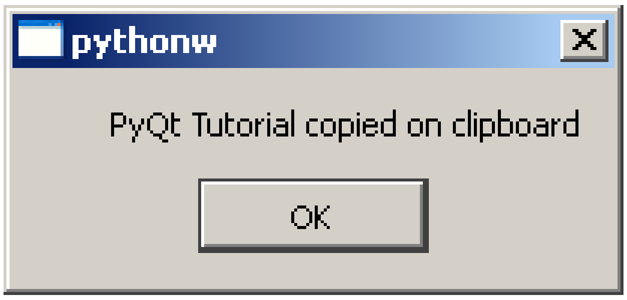
Example

In the following example, two TextEdit objects and two Pushbutons are added to a top level window.

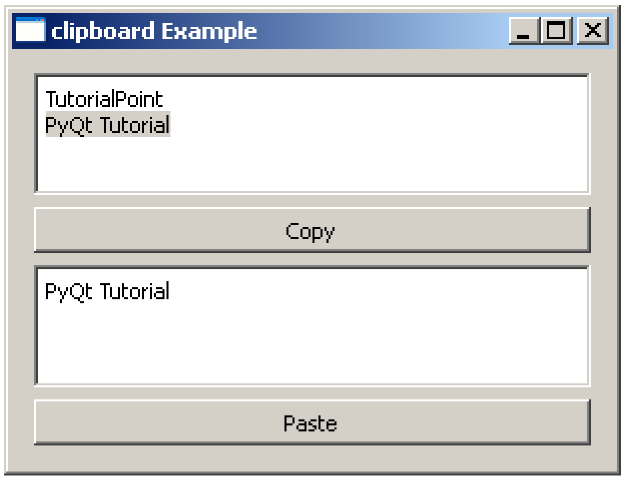
To begin with the clipboard object is instantiated. Copy() method of textedit object copies the data onto the system clipboard. When the Paste button is clicked, it fetches the clipboard data and pastes it in other textedit object.



TutorialPoint
PyQt Tutorial



PyQt Tutorial copied on clipboard



TutorialPoint
PyQt Tutorial

PyQt Tutorial

34. QDockWidget

A dockable window is a subwindow that can remain in floating state or can be attached to the main window at a specified position. Main window object of QMainWindow class has an area reserved for dockable windows. This area is around the central widget.

A dock window can be moved inside the main window, or they can be undocked to be moved into a new area by the user. These properties are controlled by the following **QDockWidget** class methods:

setWidget()	Sets any QWidget in the dock window's area
setFloating()	If set to true, the dock window can float
setAllowedAreas()	Sets the areas to which the window can be docked
	LeftDockWidgetArea
	RightDockWidgetArea
	TopDockWidgetArea
	BottomDockWidgetArea
	NoDockWidgetArea
setFeatures()	Sets the features of dock window
	DockWidgetClosable
	DockWidgetMovable
	DockWidgetFloatable
	DockWidgetVerticalTitleBar
	NoDockWidgetFeatures

Example

In the following example, top level window is a QMainWindow object. A QTextEdit object is its central widget.

```
self.setCentralWidget(QTextEdit())
```

A dockable window is first created.

```
self.items = QDockWidget("Dockable", self)
```

A QListWidget object is added as a dock window.

```
self.listWidget = QListWidget()
self.listWidget.addItem("item1")
self.listWidget.addItem("item2")
self.listWidget.addItem("item3")
self.items.setWidget(self.listWidget)
```

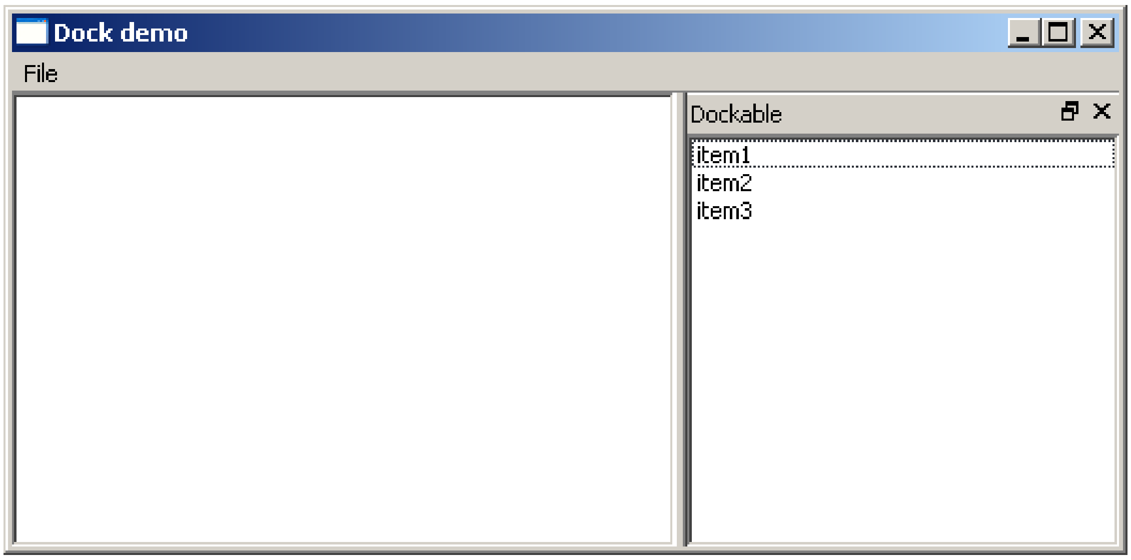
Dockable object is placed towards the right side of the central widget.

self.addDockWidget(Qt.RightDockWidgetArea, self.items)

The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class dockdemo(QMainWindow):
    def __init__(self, parent=None):
        super(dockdemo, self).__init__(parent)

        layout = QHBoxLayout()
        bar=self.menuBar()
        file=bar.addMenu("File")
        file.addAction("New")
        file.addAction("save")
        file.addAction("quit")

        self.items = QDockWidget("Dockable", self)
        self.listWidget = QListWidget()
        self.listWidget.addItem("item1")
        self.listWidget.addItem("item2")
        self.listWidget.addItem("item3")
        self.items.setWidget(self.listWidget)
        self.items.setFloating(False)
        self.setCentralWidget(QTextEdit())
        self.addDockWidget(Qt.RightDockWidgetArea, self.items)
        self.setLayout(layout)
        self.setWindowTitle("Dock demo")

def main():
    app = QApplication(sys.argv)
    ex = dockdemo()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

The above code produces the following output:



35. QStatusBar Widget

QMainWindow object reserves a horizontal bar at the bottom as the **status bar**. It is used to display either permanent or contextual status information.

There are three types of status indicators:

- **Temporary** – Briefly occupies most of the status bar. For example, used to explain tool tip texts or menu entries.
- **Normal** – Occupies part of the status bar and may be hidden by temporary messages. For example, used to display the page and line number in a word processor.
- **Permanent** – It is never hidden. Used for important mode indications. For example, some applications put a Caps Lock indicator in the status bar.

Status bar of QMainWindow is retrieved by statusBar() function. setStatusBar() function activates it.

```
self.statusBar= QStatusBar()

self.setStatusBar(self.statusBar)
```

Methods of QStatusBar Class

addWidget()	Adds the given widget object in the status bar
addPermanentWidget()	Adds the given widget object in the status bar permanently
showMessage()	Displays a temperory message in the status bar for a specified time interval
clearMessage()	Removes any temperory message being shown
removeWidget()	Removes specified widget from the status bar

Example

In the following example, a top level QMainWindow has a menu bar and a QTextEdit object as its central widget.

Window’s status bar is activated as explained above.

Menu’s triggered signal is passed to processtrigger() slot function. If ‘show’ action is triggered, it displays a temperory message in the status bar as:

```
if (q.text()=="show"):

     self.statusBar.showMessage(q.text()+" is clicked",2000)
```

The message will be erased after 2000 milliseconds (2 sec). If ‘add’ action is triggered, a button widget is added.

```
if q.text()=="add":
      self.statusBar.addWidget(self.b)
```

Remove action will remove the button from the status bar.

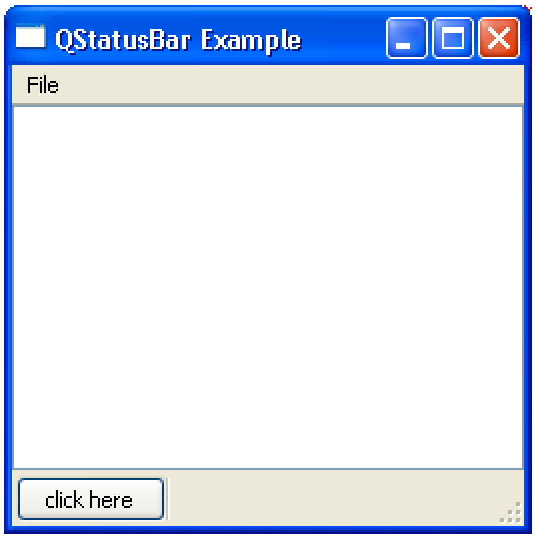
```
if q.text()=="remove":
       self.statusBar.removeWidget(self.b)
       self.statusBar.show()
```

The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class statusdemo(QMainWindow):
    def __init__(self, parent=None):
        super(statusdemo, self).__init__(parent)

        bar=self.menuBar()
        file=bar.addMenu("File")
        file.addAction("show")
        file.addAction("add")
        file.addAction("remove")
        file.triggered[QAction].connect(self.processtrigger)
        self.setCentralWidget(QTextEdit())

        self.statusBar= QStatusBar()
        self.b=QPushButton("click here")
        self.setWindowTitle("QStatusBar Example")
        self.setStatusBar(self.statusBar)

    def processtrigger(self,q):

            if (q.text()=="show"):
                self.statusBar.showMessage(q.text()+" is clicked",2000)
            if q.text()=="add":
                self.statusBar.addWidget(self.b)

            if q.text()=="remove":
                self.statusBar.removeWidget(self.b)
```

```
                 self.statusBar.show()

def main():
    app = QApplication(sys.argv)
    ex = statusdemo()
    ex.show()
    sys.exit(app.exec_())


if __name__ == '__main__':
    main()
```

The above code produces the following output:



36. QListWidget

QListWidget class is an item-based interface to add or remove items from a list. Each item in the list is a QListWidgetItem object. ListWidget can be set to be multiselectable.

Following are the frequently used methods of QListWidget class:

addItem()	Adds QListWidgetItem object or string in the list
addItems()	Adds each item in the list
insertItem()	Inserts item at the specified index
clear()	Removes contents of the list
setCurrentItem()	Sets currently selected item programmatically
sortItems()	Rearranges items in ascending order

Following are the signals emitted by QListWidget:

currentItemChanged()	Whenever current item changes
itemClicked()	Whenever an item in the list is clicked

Example

The following example shows the click event being captured to pop up a message box.

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys

class myListWidget(QListWidget):



    def Clicked(self,item):
      QMessageBox.information(self,   "ListWidget",   "You clicked: "+item.text())

def main():
    app      = QApplication(sys.argv)
    listWidget     = myListWidget()

    #Resize width and height
    listWidget.resize(300,120)

    listWidget.addItem("Item 1");
    listWidget.addItem("Item 2");
    listWidget.addItem("Item 3");
    listWidget.addItem("Item 4");

    listWidget.setWindowTitle('PyQT QListwidget Demo')
    listWidget.itemClicked.connect(listWidget.Clicked)

    listWidget.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

37. QPixmap Class

QPixmap class provides an off-screen representation of an image. It can be used as a QPaintDevice object or can be loaded into another widget, typically a label or button.

Qt API has another similar class QImage, which is optimized for I/O and other pixel manipulations. Pixmap, on the other hand, is optimized for showing it on screen. Both formats are interconvertible.

The types of image files that can be read into a QPixmap object are as follows:

BMP	Windows Bitmap
GIF	Graphic Interchange Format (optional)
JPG	Joint Photographic Experts Group
JPEG	Joint Photographic Experts Group
PNG	Portable Network Graphics
PBM	Portable Bitmap
PGM	Portable Graymap
PPM	Portable Pixmap
XBM	X11 Bitmap
XPM	X11 Pixmap

Following methods are useful in handling QPixmap object:

copy()	Copies pixmap data from a QRect object
fromImage()	Converts QImage object into QPixmap
grabWidget()	Creates a pixmap from the given widget
grabWindow()	Create pixmap of data in a window
Load()	Loads an image file as pixmap
save()	Saves the QPixmap object as a file
toImage	Converts a QPixmap to QImage

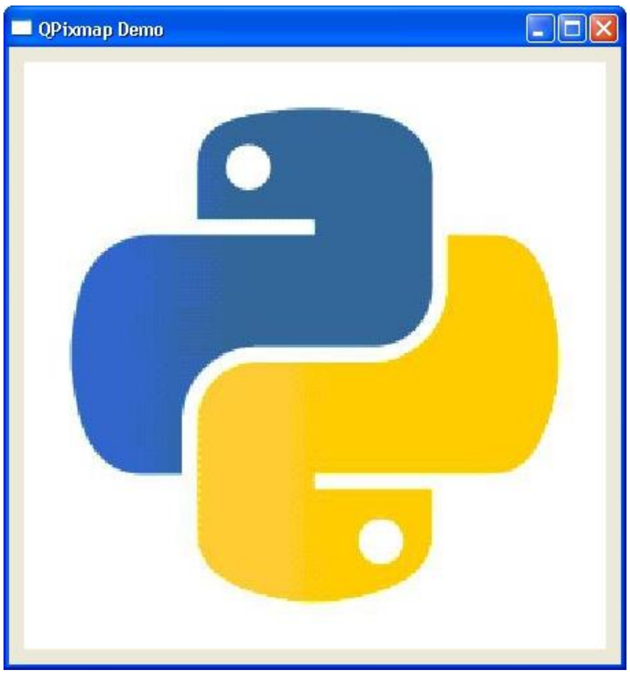
The most common use of QPixmap is to display image on a label/button.

Example

The following example shows an image displayed on a QLabel by using the setPixmap() method. The complete code is as follows:

```
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
def window():
    app = QApplication(sys.argv)
    win = QWidget()
    l1=QLabel()
    l1.setPixmap(QPixmap("python.jpg"))
    vbox=QVBoxLayout()
    vbox.addWidget(l1)
    win.setLayout(vbox)
    win.setWindowTitle("QPixmap Demo")
    win.show()
    sys.exit(app.exec_())


if __name__ == '__main__':
    window()
```

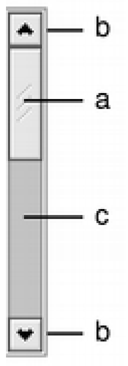
The above code produces the following output:



38. QScrollBar Widget

A scrollbar control enables the user to access parts of the document that is outside the viewable area. It provides visual indicator to the current position. It has a slider by which a value between a preset range is set in analogous fashion. This value is usually correlated to bring a hidden data inside the viewport.

The scrollbar control has four controls:

A slider Two Scroll arrows Page control	

Following signals of QScrollBar class are frequently used:

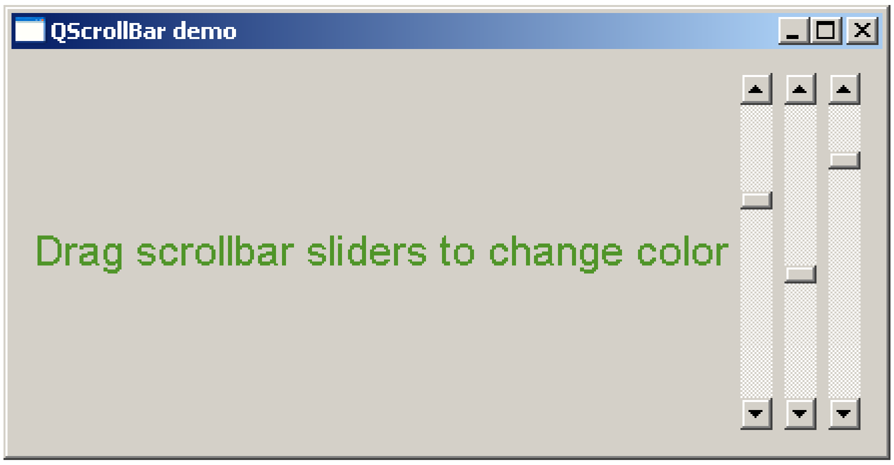
valueChanged()	When the scrollbar's value changes
sliderMoved()	When the user drags the slider

Example

In the following example, three scroll bars are placed to control RGB values of font color for the text displayed in a label. The complete code is as follows:

```
import sys
from PyQt4.QtGui import *
from PyQt4.QtCore import *

class Example(QWidget):

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

    def initUI(self):
        vbox=QVBoxLayout(self)
        hbox = QHBoxLayout()
        self.l1=QLabel("Drag scrollbar sliders to change color")
        self.l1.setFont(QFont("Arial",16))


        hbox.addWidget(self.l1)
        self.s1=QScrollBar()
        self.s1.setMaximum(255)
        self.s1.sliderMoved.connect(self.sliderval)
        self.s2=QScrollBar()
        self.s2.setMaximum(255)
        self.s2.sliderMoved.connect(self.sliderval)
        self.s3=QScrollBar()
        self.s3.setMaximum(255)
        self.s3.sliderMoved.connect(self.sliderval)
        hbox.addWidget(self.s1)
        hbox.addWidget(self.s2)
        hbox.addWidget(self.s3)

        self.setGeometry(300, 300, 300, 200)
        self.setWindowTitle('QSplitter demo')
        self.show()
    def sliderval(self):
        print self.s1.value(),self.s2.value(), self.s3.value()
        palette = QPalette()
        c=QColor(self.s1.value(),self.s2.value(), self.s3.value(),255)
        palette.setColor(QPalette.Foreground,c)
        self.l1.setPalette(palette)

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


if __name__ == '__main__':
    main()
```

The above code produces the following output:



Drag scrollbar sliders to change color

39. QCalendar Widget

QCalendar widget is a useful date picker control. It provides a month-based view. The user can select the date by the use of the mouse or the keyboard, the default being today's date. Calendar's date range can also be stipulated.

Following are some utility methods of this class:

setDateRange()	Sets the lower and upper date available for selection
setFirstDayOfWeek()	Determines the day of the first column in the calendar The predefined day constants are: • Qt.Monday • Qt.Tuesday • Qt.Wednesday • Qt.Thursday • Qt.Friday • Qt.Saturday • Qt.Sunday
setMinimumDate()	Sets the lower date for selection
setMaximumDate()	Sets the upper date for selection
setSelectedDate()	Sets a QDate object as the selected date
showToday()	Shows the month of today
selectedDate()	Retrieves the selected date
setGridvisible()	Turns the calendar grid on or off

Example

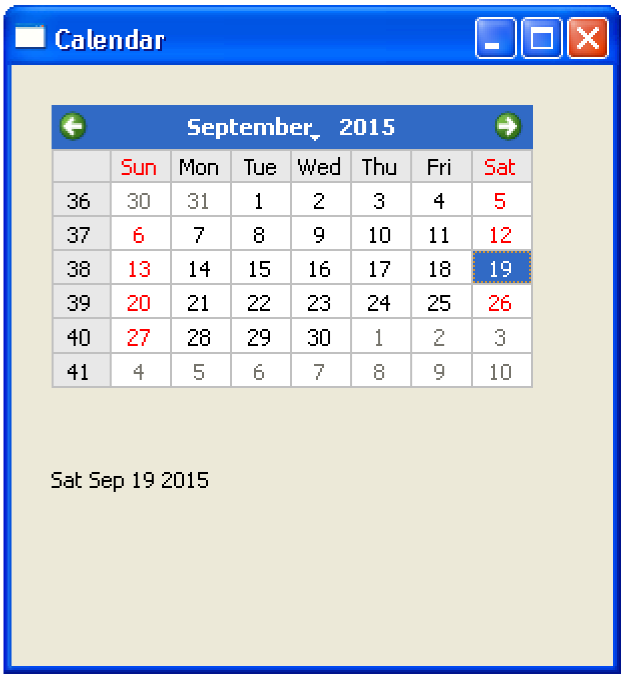
The following example has a calendar widget and a label which displays the currently selected date. The complete code is as follows:

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

```
class Example(QtGui.QWidget):

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

        self.initUI()


    def initUI(self):

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

        self.lbl = QtGui.QLabel(self)
        date = cal.selectedDate()
        self.lbl.setText(date.toString())
        self.lbl.move(20, 200)

        self.setGeometry(100,100,300,300)
        self.setWindowTitle('Calendar')
        self.show()

    def showDate(self, date):

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


def main():

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

```
if __name__ == '__main__':
    main()
```

The above code produces the following output:



	Sun	Mon	Tue	Wed	Thu	Fri	Sat
36	30	31	1	2	3	4	5
37	6	7	8	9	10	11	12
38	13	14	15	16	17	18	19
39	20	21	22	23	24	25	26
40	27	28	29	30	1	2	3
41	4	5	6	7	8	9	10

```
Sat Sep 19 2015
```