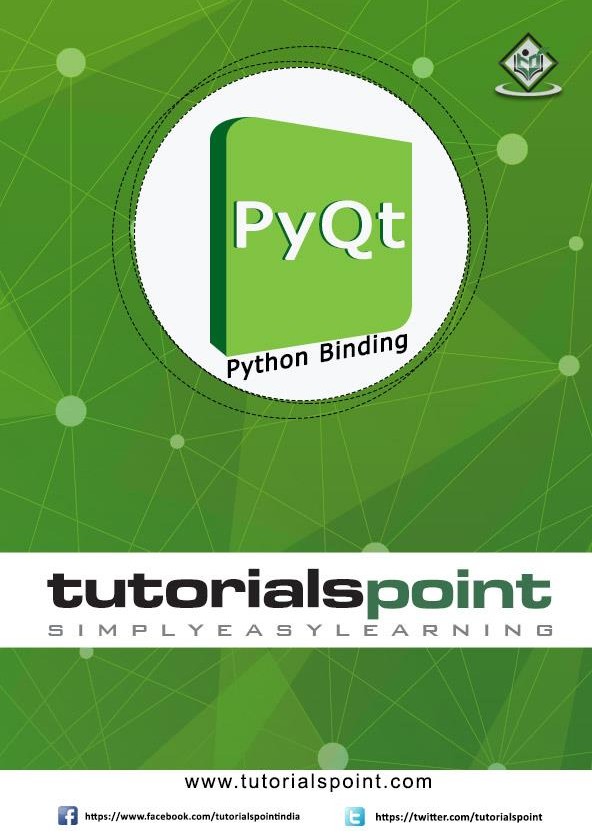
PyQt



i

## About the Tutorial

PyQt5 is the latest version of a GUI widgets toolkit developed by Riverbank Computing. It is a Python interface for **Qt**, one of the most powerful, and popular cross-platform GUI library. PyQt5 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. Our tutorial on earlier version - PyQt4 is available [here](https://www.tutorialspoint.com/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 knowledge of Python programming is a desirable.

## Copyright & Disclaimer

 Copyright 2020 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at [contact@tutorialspoint.com](mailto:contact@tutorialspoint.com)

## Table of Contents

[About the Tutorial ii](#_bookmark0)

[Audience ii](#_bookmark1)

[Prerequisites ii](#_bookmark2)

[Copyright & Disclaimer ii](#_bookmark3)

[Table of Contents iii](#_bookmark4)

1. [PyQt5 — Introduction 1](#_bookmark5)

[Supporting Environments 1](#_bookmark6)

1. [PyQt5 — What’s New 3](#_bookmark7)
2. [PyQt5 — Hello World 4](#_bookmark8)
3. [PyQt5 ― Major Classes 7](#_bookmark9)
4. [PyQt5 ― Using Qt Designer 11](#_bookmark10)
5. [PyQt5 ― Signals & Slots 15](#_bookmark11)

[Using Qt Designer's Signal/Slot Editor 15](#_bookmark12)

[Building Signal-slot Connection 17](#_bookmark13)

1. [PyQt5 ― Layout Management 20](#_bookmark14)

[setGeometry() syntax 20](#_bookmark15)

[QBoxLayout 22](#_bookmark16)

[QGridLayout Class 26](#_bookmark17)

[QFormLayout Class 27](#_bookmark18)

1. [PyQt5 — Basic Widgets 29](#_bookmark19)

[QLabel Widget 31](#_bookmark20)

[QLineEdit Widget 35](#_bookmark21)

[QPushButton Widget 39](#_bookmark22)

[QRadioButton Widget 43](#_bookmark23)

[QCheckBox Widget 46](#_bookmark24)

[QComboBox Widget 49](#_bookmark25)

[QSpinBox Widget 52](#_bookmark26)

[QSlider Widget 54](#_bookmark27)

[QMenuBar, QMenu & QAction Widgets 57](#_bookmark28)

[QToolBar Widget 60](#_bookmark29)

[QInputDialog Widget 62](#_bookmark30)

[QFontDialog Widget 65](#_bookmark31)

[QFileDialog Widget 68](#_bookmark32)

[QTab Widget 75](#_bookmark33)

[QStackedWidget 78](#_bookmark34)

[QSplitter Widget 82](#_bookmark35)

[QDock Widget 85](#_bookmark36)

[QStatusBar Widget 88](#_bookmark37)

[QList Widget 91](#_bookmark38)

[QScrollBar Widget 93](#_bookmark39)

[QCalendar Widget 95](#_bookmark40)

1. [PyQt5 — QDialog Class 98](#_bookmark41)
2. [PyQt5 — QMessageBox 100](#_bookmark42)
3. [PyQt5 — Multiple Document Interface 104](#_bookmark43)
4. [PyQt5 — Drag & Drop 109](#_bookmark44)
5. [PyQt5 — Database Handling 112](#_bookmark45)
6. [PyQt5 — Drawing API 119](#_bookmark46)
7. [PyQt5 ― Brush Style Constants 122](#_bookmark47)

[Brush Style Constants 122](#_bookmark48)

[Predefined QColor Styles 122](#_bookmark49)

[Predefined QColor Objects 123](#_bookmark50)

1. [PyQt5 — QClipboard 124](#_bookmark51)
2. [PyQt5 — QPixmap Class 128](#_bookmark52)

# PyQt5 — Introduction

PyQt

PyQt è un toolkit di widget GUI. Èun'interfaccia Python per **Qt**, una delle librerie GUI multipiattaforma più potenti e popolari. PyQt è stato sviluppato da RiverBank Computing Ltd. PyQt5 è l'ultima versione. Può essere scaricato dal suo sito ufficiale

1. [riverbankcomputing.com](https://www.riverbankcomputing.com/software/pyqt/download5).

PyQt API è un insieme di moduli contenenti un gran numero di classi e funzioni. Mentre il modulo **QtCore**  contiene funzionalità non GUI per lavorare con file e directory, ecc., Il modulo **QtGui**  contiene tutti i controlli grafici. Inoltre, ci sono moduli per lavorare con XML  **(QtXml)**, SVG **(QtSvg)** e SQL **(QtSql)**, ecc.

Di seguito è riportato un elenco dei moduli utilizzati di frequente:

* 1. **QtCore**: Classi core non GUI utilizzate da altri moduli
  2. **QtGui**: Componenti dell'interfaccia utente grafica
  3. **QtMultimedia**: Classi per la programmazione multimediale di basso livello
  4. **QtNetwork**: Classi per la programmazione di rete
  5. **QtOpenGL**: Classi di supporto OpenGL
  6. **QtScript**: Classi per la valutazione degli script Qt
  7. **QtSql**: Classi per l'integrazione di database tramite SQL
  8. **QtSvg**: Classi per la visualizzazione del contenuto dei file SVG
  9. **QtWebKit**: Classi per il rendering e la modifica di HTML
  10. **QtXml**: Classi per la gestione di XML
  11. **QtWidgets**: Classi per la creazione di interfacce utente classiche in stile desktop
  12. **QtDesigner**: Classi per estendere Qt Designer

## Ambienti di supporto

PyQt è compatibile con tutti i sistemi operativi più diffusi tra cui Windows, Linux e Mac OS. È con doppia licenza, disponibile sotto LICENZA GPL e licenza commerciale. L'ultima versione stabile è  **PyQt5-5.13.2.**

### Finestre

Vengono fornite ruote per architettura a 32 bit o 64 bit compatibili con Python versione 3.5 o successiva. Il modo consigliato per l'installazione consiste nell'utilizzare l'utilità  **PIP:**

pip3 install PyQt5

Per installare strumenti di sviluppo come Qt Designer per supportare le ruote PyQt5, di seguito è riportato il comando:

pip3 install pyqt5-tools

PyQt

Puoi anche creare PyQt5 su Linux / macOS dal codice sorgente:

[https://www.riverbankcomputing.com/static/Downloads/PyQt5/5.13.2/PyQt5-](https://www.riverbankcomputing.com/static/Downloads/PyQt5/5.13.2/PyQt5-5.13.2.tar.gz) [5.13.2.tar.gz](https://www.riverbankcomputing.com/static/Downloads/PyQt5/5.13.2/PyQt5-5.13.2.tar.gz)

L'API PyQt5 non è automaticamente compatibile con le versioni precedenti. Quindi, il codice Python che coinvolge i moduli PyQt4 dovrebbe essere aggiornato manualmente apportando modifiche pertinenti. In questo capitolo sono state elencate le principali differenze tra PyQt4 e PyQt5.

PyQt5 non è supportato nelle versioni di Python precedenti alla v2.6.

PyQt5 non supporta il metodo connect() della classe QObject per la connessione tra segnale e slot. Quindi l'utilizzo non può più essere implementato:

QObject.connect(widget, QtCore.SIGNAL(‘signalname’), slot\_function)

Viene definita solo la sintassi seguente:

widget.signal.connect(slot\_function)

Le classi definite nel modulo earler QtGui sono state distribuite nei moduli **QtGui,**  **QtPrintSupport** e  **QtWidgets.**

Nella nuova classe QFileDialog, il metodo **getOpenFileNameAndFilter**() viene sostituito da **getOpenFileName**(), **getOpenFileNamesAndFilter**() da **getOpenFileNames**() e **getSaveFileNameAndFilter**() da **getSaveFileName**(). Anche le vecchie firme di questi metodi sono cambiate.

PyQt5 non dispone di provisioning per definire una classe che è sottoclassificata da più di una classe Qt.

L'utilità pyuic5 (per generare codice Python dal file XML di Designer) non supporta il -

-pyqt3-wrapper flag.

**pyrcc5** non supporta i flag -py2 e -py3. L'output di pyrcc5 è compatibile con tutte le versioni di Python v2.6 in poi.

PyQt5 richiama sempre  **sip.setdestroyonexit**() automaticamente e chiama il distruttore C++ di tutte le istanze con wrapping di cui è proprietario.

La creazione di una semplice applicazione GUI utilizzando PyQt comporta i seguenti passaggi:

* 1. Importa i moduli QtCore, QtGui e QtWidgets dal pacchetto PyQt5.
  2. Creare un oggetto applicazione della classe QApplication.
  3. Un oggetto QWidget crea una finestra di primo livello. Aggiungere l'oggetto QLabel al suo interno.
  4. Imposta la didascalia dell'etichetta come "hello world".
  5. Definire le dimensioni e la posizione della finestra con il metodo setGeometry().
  6. Immettere il mainloop dell'applicazione con il metodo  **app.exec\_().**

Di seguito è riportato il codice per eseguire il programma Hello World in PyQt:

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \* def window():

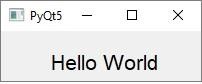
app = QApplication(sys.argv) w = QWidget()

b = QLabel(w) b.setText("Hello World!") w.setGeometry(100,100,200,50) b.move(50,20) w.setWindowTitle("PyQt5") w.show() sys.exit(app.exec\_())

if name == ' main ': window()

It is also possible to develop an object oriented solution of the above code.

* + Import QtCore, QtGui and QtWidgets modules from PyQt5 package.



* 1. Creare un oggetto applicazione della classe QApplication.
  2. Dichiarare la classe window in base alla classe QWidget
  3. Aggiungi un oggetto QLabel e imposta la didascalia dell'etichetta come "hello world".
  4. Definire le dimensioni e la posizione della finestra con il metodo setGeometry().
  5. Immettere il mainloop dell'applicazione con il metodo  **app.exec\_().**

Di seguito è riportato il codice completo della soluzione orientata agli oggetti:

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \* class window(QWidget):

def init (self, parent = None): super(window, self). init (parent) self.resize(200,50) self.setWindowTitle("PyQt5") self.label = QLabel(self) self.label.setText("Hello World") font = QFont() font.setFamily("Arial") font.setPointSize(16) self.label.setFont(font) self.label.move(50,20)

def main():

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

ex.show() sys.exit(app.exec\_())

if name == ' main ': main()

# PyQt5 ― Major Classes

PyQt

**PyQt API** è una vasta raccolta di classi e metodi. Queste classi sono definite in più di 20 moduli.

Di seguito sono riportati alcuni dei moduli utilizzati di frequente:

|  |  |
| --- | --- |
| **Sr.No.** | **Modules & Description** |
| 1 | **QtCore**  Core non-GUI classes used by other modules |
| 2 | **QtGui**  Graphical user interface components |
| 3 | **QtMultimedia**  Classes for low-level multimedia programming |
| 4 | **QtNetwork**  Classes for network programming |
| 5 | **QtOpenGL**  OpenGL support classes |
| 6 | **QtScript**  Classes for evaluating Qt Scripts |
| 7 | **QtSql**  Classes for database integration using SQL |
| 8 | **QtSvg**  Classes for displaying the contents of SVG files |
| 9 | **QtWebKit**  Classes for rendering and editing HTML |
| 10 | **QtXml**  Classes for handling XML |
| 11 | **QtWidgets**  Classes for creating classic desktop-style UIs. |
| 12 | **QtDesigner**  Classes for extending Qt Designer |

|  |  |
| --- | --- |
| 13 | **QtAssistant**  Support for online help |

Gli strumenti di sviluppo di PyQt5 sono una raccolta di utilità utili per lo sviluppo di Qt. Di seguito è riportato un elenco selezionato di tali utilità:

|  |  |
| --- | --- |
| **Tool Name** | **Description** |
| assistant | Qt Assistant documentation tool |
| pyqt5designer | Qt Designer GUI layout tool |
| linguist | Qt Linguist translation tool |
| lrelease | compile ts files to qm files |
| pylupdate5 | extract translation strings and generate or update ts files |
| qmake | Qt software build tool |
| pyqt5qmlscene | QML file viewer |
| pyqmlviewer | QML file viewer |
| pyrcc5 | Qt resource file compiler |
| pyuic5 | Qt User Interface Compiler for generating code from ui files |
| pyqmltestrunner | running unit tests on QML code |
| qdbus | command-line tool to list D-Bus services |
| QDoc | documentation genearator for software projects. |
| Qhelpgenerator | generating and viewing Qt help files. |
| qmlimportscanner | parses and reports on QML imports |

PyQt API contiene più di 400 classi. La classe **QObject**  è in cima alla gerarchia delle classi. È la classe base di tutti gli oggetti Qt. Inoltre, la classe **QPaintDevice**  è la classe base per tutti gli oggetti che possono essere dipinti.

**La classe QApplication** gestisce le impostazioni principali e il flusso di controllo di un'applicazione GUI. Contiene il ciclo di eventi principale all'interno del quale vengono elaborati e inviati eventi generati da elementi di finestra e altre origini. Gestisce anche le impostazioni a livello di sistema e di applicazione.

**La classe QWidget,** derivata dalle classi QObject e QPaintDevice, è la classe base per tutti gli oggetti dell'interfaccia utente. Anche le classi **QDialog** e  **QFrame** derivano dalla classe QWidget. Hanno il loro sistema di sottoclassi.

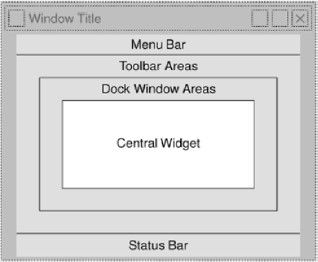
Ecco un elenco selezionato di widget utilizzati di frequente:

|  |  |
| --- | --- |
| **Sr.No.** | **Widgets & Description** |
| 1 | **QLabel**  Utilizzato per visualizzare testo o immagine |
| 2 | **QLineEdit**  Consente all'utente di immettere una riga di testo |
| 3 | **QTextEdit**  Consente all'utente di immettere testo su più righe |
| 4 | **QPushButton**  Un pulsante di comando per richiamare l'azione |
| 5 | **QRadioButton**  Consente di sceglierne una tra più opzioni |
| 6 | **QCheckBox**  Consente di scegliere tra più di una opzione |
| 7 | **QSpinBox**  Consente di aumentare/diminuire un valore intero |
| 8 | **QScrollBar**  Consente di accedere al contenuto di un widget oltre l'apertura dello schermo |
| 9 | **QSlider**  Consente di modificare il valore associato in modo lineare. |
| 10 | **QComboBox**  Fornisce un elenco a discesa di elementi tra cui selezionare |
| 11 | **QMenuBar**  Barra orizzontale che contiene oggetti QMenu |
| 12 | **QStatusBar**  Di solito nella parte inferiore di QMainWindow, fornisce informazioni sullo stato. |
| 13 | **QToolBar**  Di solito in cima a QMainWindow o galleggiante. Contiene pulsanti di azione |
| 14 | **QListView**  Fornisce un elenco selezionabile di elementi in ListMode o IconMode |

|  |  |
| --- | --- |
| 15 | **QPixmap**  Rappresentazione di immagini fuori schermo per la visualizzazione su oggetti QLabel o QPushButton |
| 16 | **QDialog**  Finestra modale o senza modalità che può restituire informazioni alla finestra padre |

La finestra di primo livello di un'applicazione tipica basata su GUI viene creata dall'oggetto widget **QMainWindow.**  Alcuni widget come sopra elencati prendono il loro posto in questa finestra principale, mentre altri sono posizionati nell'area centrale del widget utilizzando vari gestori di layout.

Il diagramma seguente mostra il framework QMainWindow:

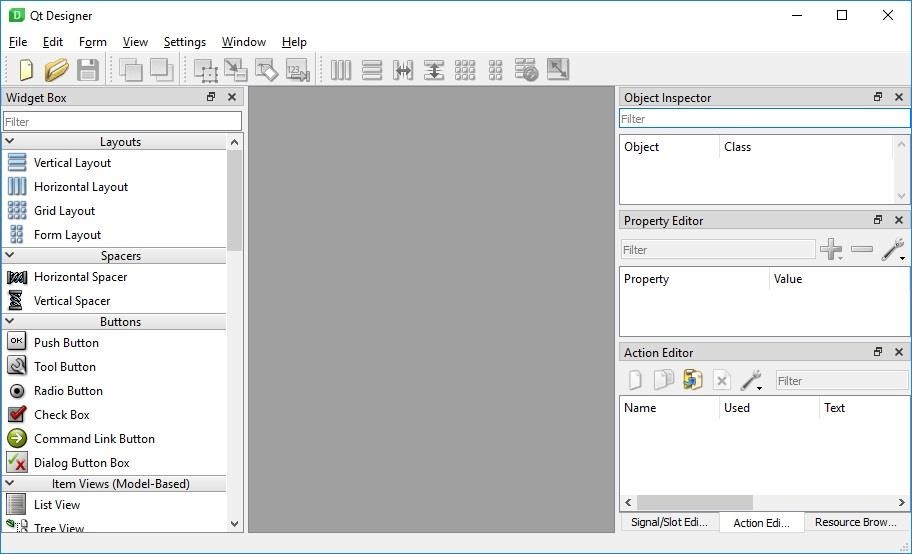


# PyQt5 ― Using Qt Designer

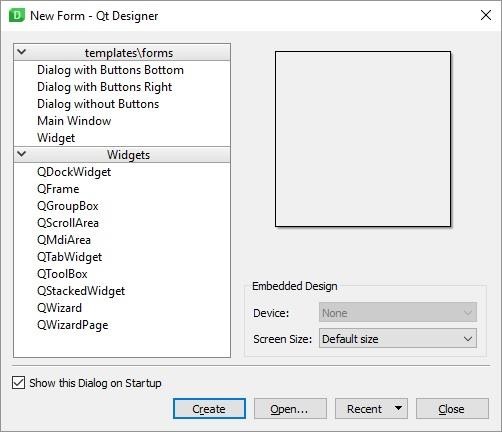
PyQt

Il programma di installazione di PyQt viene fornito con uno strumento di creazione di GUI chiamato  **Qt**  **Designer**. Utilizzando la sua semplice interfaccia drag and drop, un'interfaccia GUI può essere costruita rapidamente senza dover scrivere il codice. Tuttavia, non è un IDE come Visual Studio. Pertanto, Qt Designer non ha la possibilità di eseguire il debug e compilare l'applicazione.

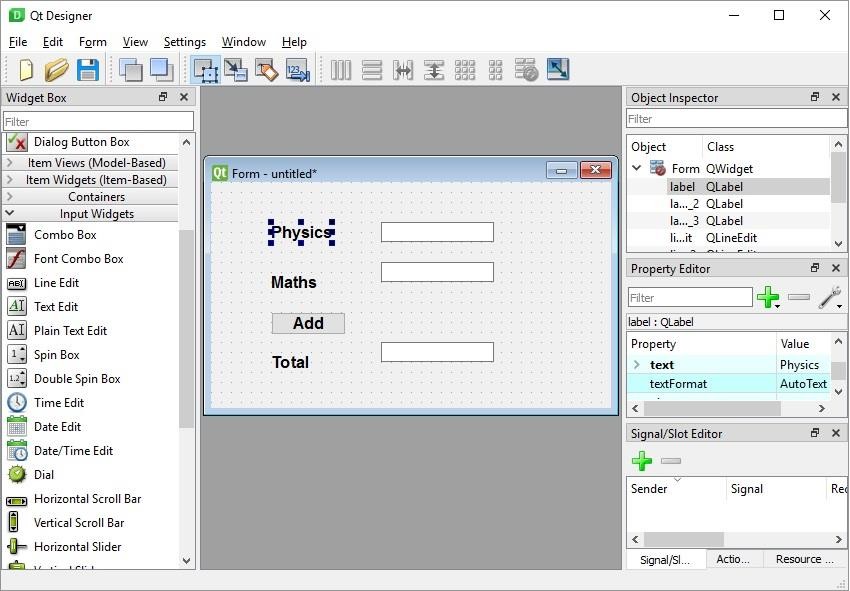
Avviare l'applicazione Qt Designer che fa parte degli strumenti di sviluppo e installata nella cartella script dell'ambiente virtuale.



Inizia a progettare l'interfaccia GUI scegliendo il menu File -> Nuovo.



È quindi possibile trascinare e rilasciare i widget richiesti dalla casella del widget nel riquadro di sinistra. È inoltre possibile assegnare valore alle proprietà del widget posato nel modulo.



Il modulo progettato viene salvato come demo.ui. Questo file dell'interfaccia utente contiene la rappresentazione XML dei widget e le relative proprietà nella progettazione. Questo design è tradotto in equivalente Python utilizzando l'utilità della riga di comando pyuic5. Questa utility è un wrapper per il modulo uic del toolkit Qt. L'uso di pyuic5 è il seguente:

pyuic5 -x demo.ui -o demo.py

Nel comando precedente, l'opzione -x aggiunge una piccola quantità di codice aggiuntivo allo script Python generato (da XML) in modo che diventi un'applicazione autonoma autoeseguibile.

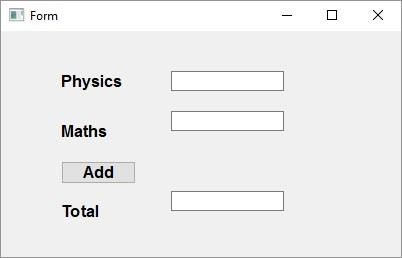
if name == " main ": import sys

app = QtGui.QApplication(sys.argv) Dialog = QtGui.QDialog()

ui = Ui\_Dialog() ui.setupUi(Dialog) Dialog.show() sys.exit(app.exec\_())

Lo script python risultante viene eseguito per visualizzare la seguente finestra di dialogo:

python demo.py



L'utente può inserire i dati nei campi di input, ma facendo clic sul pulsante Aggiungi non genererà alcuna azione in quanto non è associata ad alcuna funzione. La reazione alla risposta generata dall'utente è chiamata gestione **degli eventi.**

# PyQt5 ― Signals & Slots

PyQt

A differenza di un'applicazione in modalità console, che viene eseguita in modo sequenziale, un'applicazione basata su GUI è guidata da eventi. Funzioni o metodi vengono eseguiti in risposta alle azioni dell'utente come fare clic su un pulsante, selezionare un elemento da una raccolta o un clic del mouse ecc., Chiamato  **eventi**.

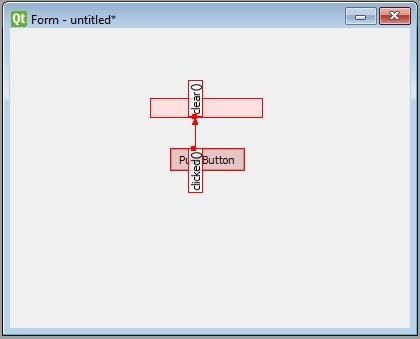
I widget utilizzati per costruire l'interfaccia GUI fungono da fonte di tali eventi. Ogni widget PyQt, che deriva dalla classe QObject, è progettato per emettere '**segnale**' in risposta auno o più eventi. Il segnale da solo non esegue alcuna azione. Invece, è "collegato" a uno "**slot**". Lo slot può essere qualsiasi  **funzione Python**  **richiamabile.**

## Utilizzo dell'editor di segnali/slot di Qt Designer

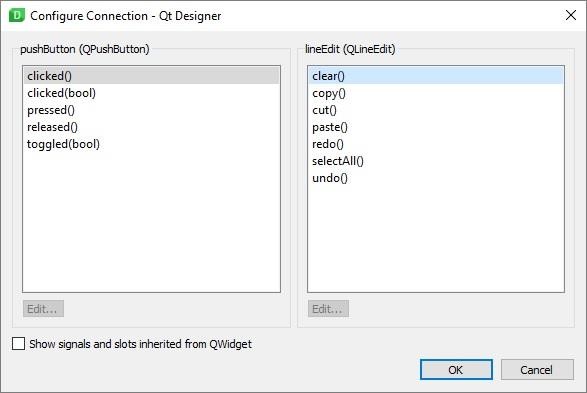
Per prima cosa progetta un form semplice con un controllo LineEdit e un PushButton.

Si desidera che se si preme il pulsante, il contenuto della casella di testo deve essere cancellato. Il widget QLineEdit ha un metodo clear() per questo scopo. Quindi, il segnale **cliccato** del pulsante deve essere collegato al metodo  **clear()** della casella di testo.

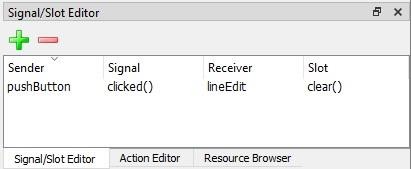
Per cominciare, scegli Modifica segnali/slot dal menu Modifica (o premi F4). Quindi evidenziare il pulsante con il mouse e trascinare il cursore verso la casella di testo



Quando il mouse viene rilasciato, verrà visualizzata una finestra di dialogo che mostra i segnali del pulsante e i metodi di slot. Seleziona il segnale cliccato e il metodo clear()



La finestra Signal/Slot Editor in basso a destra mostrerà il risultato:



Salva l'interfaccia utente e il codice Build e Python dal file ui come mostrato nel codice seguente:

pyuic5 -x signalslot.ui -o signalslot.py

Il codice Python generato avrà la connessione tra segnale e slot dalla seguente istruzione:

self.pushButton.clicked.connect(self.lineEdit.clear)

Eseguire signalslot.py e immettere del testo in LineEdit. Il testo verrà cancellato se si preme il pulsante.

## Costruzione della connessione slot di segnale

Invece di utilizzare Designer, è possibile stabilire direttamente la connessione segnale-slot con la seguente sintassi:

widget.signal.connect(slot\_function)

Supponiamo se una funzione deve essere chiamata quando si fa clic su un pulsante. Qui, il segnale cliccato deve essere collegato a una funzione richiamabile. Può essere ottenuto in una delle seguenti tecniche:

button.clicked.connect(slot\_function)

### Esempio

Nell'esempio seguente vengono aggiunti due oggetti QPushButton (b1 e b2) nella finestra QDialog. Vogliamo chiamare le funzioni b1\_clicked() e b2\_clicked() facendo clic rispettivamente su b1 e b2.

Quando si fa clic su b1, il segnale clicked() è collegato alla funzione b1\_clicked():

b1.clicked.connect(b1\_clicked())

Quando si fa clic su b2, il segnale clicked() è collegato alla funzione b2\_clicked().

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

Il codice precedente produce il seguente output:

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) b2.clicked.connect(b2\_clicked)

win.setGeometry(100,100,200,100)

win.setWindowTitle("PyQt5") win.show() sys.exit(app.exec\_())

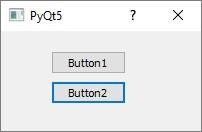
def b1\_clicked():

print ("Button 1 clicked")

def b2\_clicked():

print ("Button 2 clicked")

if name == ' main ': window()



#### Output

Button 1 clicked

Button 2 clicked

# PyQt5 ― Layout Management

PyQt

Un widget GUI può essere posizionato all'interno della finestra del contenitore specificando le sue coordinate assolute misurate in pixel. Le coordinate sono relative alle dimensioni della finestra definite dal metodo setGeometry().

## sintassi setGeometry()

Di seguito è riportata la sintassi di setGeometry():

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

Nel frammento di codice seguente, la finestra di livello superiore di 300 x 100 pixel viene visualizzata in posizione (50,20) sul monitor.

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

def window():

app = QApplication(sys.argv) w = QWidget()

b = QPushButton(w) b.setText("Hello World!") b.move(50,20)

w.setGeometry(100,100,300,100) w.setWindowTitle('PyQt5 Window') w.show()

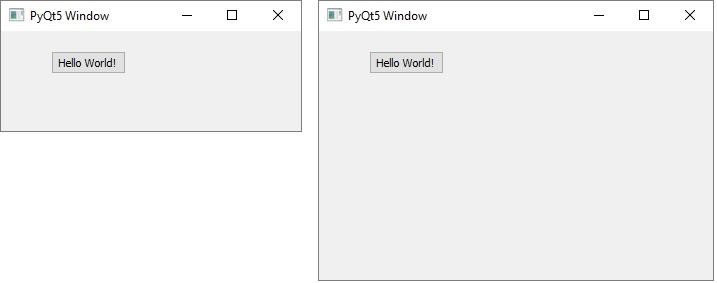
sys.exit(app.exec\_())

if name == ' main ': window()

Un widget  **PushButton** viene aggiunto nella finestra e posizionato in una posizione di 50 pixel verso destra e 20 pixel sotto la posizione in alto a sinistra della finestra.

Questo Posizionamento Assoluto, tuttavia, non è adatto per i seguenti motivi:

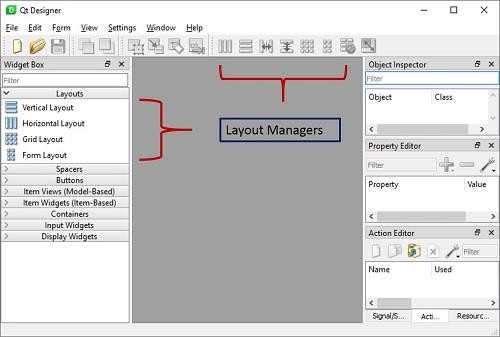
* 1. La posizione del widget non cambia anche se la finestra viene ridimensionata.
  2. L'aspetto potrebbe non essere uniforme su diversi dispositivi di visualizzazione con risoluzioni diverse.
  3. La modifica del layout è difficile in quanto potrebbe essere necessario riprogettare l'intero modulo.



PyQt API fornisce classi di layout per una gestione più elegante del posizionamento dei widget all'interno del contenitore. I vantaggi dei Layout manager rispetto al posizionamento assoluto sono:

* 1. I widget all'interno della finestra vengono ridimensionati automaticamente.
  2. Garantisce un aspetto uniforme su dispositivi di visualizzazione con risoluzioni diverse.
  3. L'aggiunta o la rimozione dinamica del widget è possibile senza dover riprogettare.

Il toolkit Qt definisce vari layout che possono essere utilizzati con l'utilità Qt Designer.



Ecco l'elenco delle classi di cui parleremo una per una in questo capitolo.

|  |  |
| --- | --- |
| **Sr.No.** | **Classes & Description** |
| 1 | **QBoxLayout**  La classe QBoxLayout allinea i widget verticalmente o orizzontalmente. Le sue classi derivate sono  **QVBoxLayout** (per disporre i widget verticalmente) e  **QHBoxLayout** (per disporre i widget orizzontalmente). |
| 2 | [**QGridLayout**](http://docs.google.com/pyqt5/pyqt_qgridlayout_class.htm)  Un oggetto classe GridLayout presenta una griglia di celle disposte in righe e colonne. La classe contiene il metodo **addWidget().**  Qualsiasi widget può essere aggiunto specificando il numero di righe e colonne della cella. |
| 3 | [**QFormLayout**](http://docs.google.com/pyqt5/pyqt_qformlayout_class.htm) |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  |  | | --- | --- | |  | QFormLayout è un modo pratico per creare un modulo a due colonne, in cui ogni riga è costituita da un campo di input associato a un'etichetta. Come convenzione, la colonna di sinistra contiene l'etichetta e la colonna di destra contiene un campo di input. | |

## [QBoxLayout](http://docs.google.com/pyqt5/pyqt_qboxlayout_class.htm)

**La classe QBoxLayout** allinea i widget verticalmente o orizzontalmente. Le sue classi derivate sono  **QVBoxLayout** (per disporre i widget verticalmente) e  **QHBoxLayout** (per disporre i widget orizzontalmente). Nella tabella seguente vengono illustrati i metodi importanti della classe QBoxLayout:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addWidget()**  Aggiunge un widget a BoxLayout |
| 2 | **addStretch()**  Crea una casella estensibile vuota |
| 3 | **addLayout()**  Aggiunge un altro layout nidificato |

### Esempio 1

Qui vengono aggiunti due pulsanti nel layout della casella verticale. Uno spazio vuoto estensibile viene aggiunto tra di loro con il metodo **addStretch().** Pertanto, se la finestra di livello superiore viene ridimensionata, la posizione dei pulsanti viene automaticamente spostata.

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

def window():

app = QApplication(sys.argv) win = QWidget()

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

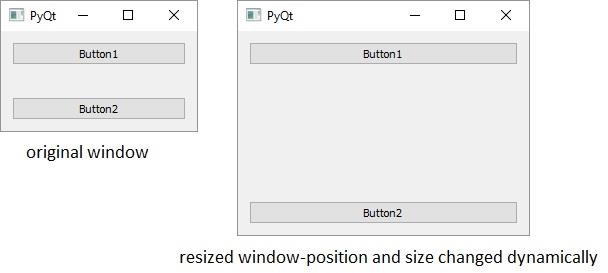
vbox = QVBoxLayout() vbox.addWidget(b1) vbox.addStretch() vbox.addWidget(b2) win.setLayout(vbox)

win.setWindowTitle("PyQt") win.show() sys.exit(app.exec\_())

if name == ' main ':

window()

Il codice precedente produce il seguente output:



### Esempio 2

In questo esempio viene utilizzato il layout a casella orizzontale. Il metodo **addStretch()** inserisce uno spazio vuoto estensibile tra i due oggetti pulsante. Quindi, quando la finestra viene ridimensionata, le dimensioni e la posizione del pulsante cambiano dinamicamente.

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

def window():

app = QApplication(sys.argv) win = QWidget()

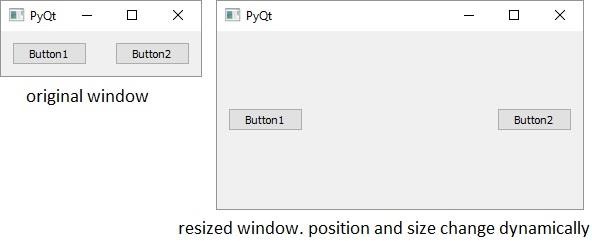
b1 = QPushButton("Button1") b2 = QPushButton("Button2")

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()

Il codice precedente produce il seguente output:



### Esempio 3

In questo esempio viene illustrato come nidificare i layout. Qui, due pulsanti vengono aggiunti al layout della casella verticale. Quindi, viene aggiunto un oggetto di layout casella orizzontale con due pulsanti e uno spazio vuoto estensibile. Infine, l'oggetto layout casella verticale viene applicato alla finestra di livello superiore con il metodo setLayout().

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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()

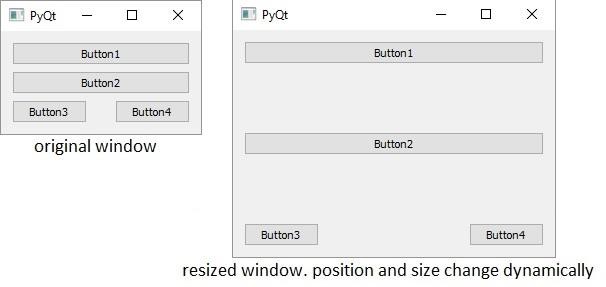
b3 = QPushButton("Button3") b4 = QPushButton("Button4") hbox.addWidget(b3) hbox.addStretch() hbox.addWidget(b4)

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

win.setWindowTitle("PyQt") win.show() sys.exit(app.exec\_())

if name == ' main ': window()

Il codice precedente produce il seguente output:



## QGridLayout Classe

Un oggetto classe  **GridLayout** presenta una griglia di celle disposte in righe e colonne. La classe contiene il metodo **addWidget().** Qualsiasi widget può essere aggiunto specificando il numero di righe e colonne della cella. Facoltativamente, un fattore di spanning per riga e colonna, se specificato, rende il widget più largo o più alto di una cella. Due overload del metodo addWidget() sono i seguenti:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addWidget(QWidget, int r, int c)**  Aggiunge un widget in corrispondenza di righe e colonne specificate |
| 2 | **addWidget(QWidget, int r, int c, int rowspan, int columnspan)**  Aggiunge un widget in corrispondenza di righe e colonne specificate e con larghezza e/o altezza specificate |

Un oggetto di layout figlio può anche essere aggiunto in qualsiasi cella della griglia.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addLayout(QLayout, int r, int c)**  Aggiunge un oggetto layout in corrispondenza di righe e colonne specificate |

### Esempio

Il codice seguente crea un layout a griglia di 16 pulsanti disposti in un layout a griglia di 4 righe e 4 colonne.

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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 Grid Layout") win.show()

sys.exit(app.exec\_())

if name == ' main ': window()

Il codice utilizza due cicli nidificati per i numeri di riga e colonna, indicati dalle variabili *i* e *j*. Vengono convertiti in stringa per concatenare la didascalia di ogni pulsante da aggiungere alla  *ia* riga e alla *jesima* colonna.

Il codice precedente produce il seguente output:



## QFormLayout Classe

**QFormLayout** è un modo pratico per creare un modulo a due colonne, in cui ogni riga è costituita da un campo di input associato a un'etichetta. Come convenzione, la colonna di sinistra contiene l'etichetta e la colonna di destra contiene un campo di input. Esistono principalmente tre versioni sovraccariche del metodo **addRow()** comunemente utilizzate.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addRow(QLabel, QWidget)**  Aggiunge una riga contenente etichetta e campo di input |
| 2 | **addRow(QLabel, QLayout)**  Aggiunge un layout figlio nella seconda colonna |
| 3 | **addRow(QWidget)**  Aggiunge un widget che copre entrambe le colonne |

### Esempio

Questo codice aggiunge un campo LineEdit al nome di input nella prima riga. Quindi aggiunge un layout di casella verticale per due campi indirizzo nella seconda colonna della riga successiva. Successivamente, un oggetto layout casella orizzontale contenente due campi pulsante di opzione viene aggiunto nella seconda colonna della terza riga. La quarta riga mostra due pulsanti 'Invia' e 'Annulla'.

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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"))

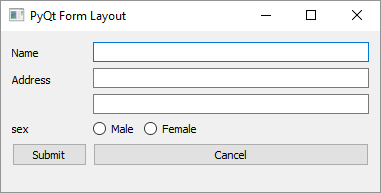
win.setLayout(fbox)

win.setWindowTitle("PyQt5 Form Layout") win.show()

sys.exit(app.exec\_())

if name == ' main ': window()

Il codice precedente produce il seguente output:



# PyQt5 — Basic Widgets

PyQt

Ecco l'elenco dei widget di cui parleremo uno per uno in questo capitolo.

|  |  |
| --- | --- |
| **Sr.No** | **Widget &**  **Descrizione** |
| 1 | [**QLabel**](file://localhost/C:/pyqt5/pyqt_qlabel_widget.htm)  Un oggetto QLabel funge da segnaposto per visualizzare testo o immagine non modificabili o un filmato di GIF animata. Può anche essere usato come chiave mnemonica per altri widget. |
| 2 | [**QLineModifica**](file://localhost/C:/pyqt5/pyqt_qlineedit_widget.htm)  L'oggetto QLineEdit è il campo di input più comunemente usato. Fornisce una casella in cui è possibile inserire una riga di testo. Per immettere testo su più righe, è necessario l'oggetto QTextEdit. |
| 3 | [**QPushButton**](file://localhost/C:/pyqt5/pyqt_qpushbutton_widget.htm)  Nell'API PyQt, l'oggetto classe QPushButton presenta un pulsante che, quando cliccato, può essere programmato per richiamare una determinata funzione. |
| 4 | [**QRadioButton**](file://localhost/C:/pyqt5/pyqt_qradiobutton_widget.htm)  Un oggetto classe QRadioButton presenta un pulsante selezionabile con un'etichetta di testo. L'utente può selezionare una delle tante opzioni presentate nel modulo. Questa classe è derivata dalla classe QAbstractButton. |
| 5 | [**QCheckBox**](file://localhost/C:/pyqt5/pyqt_qcheckbox_widget.htm)  Una casella rettangolare prima dell'etichetta di testo viene visualizzata quando un oggetto QCheckBox viene aggiunto alla finestra padre. Proprio come QRadioButton, è anche un pulsante selezionabile. |
| 6 | [**QComboBox**](file://localhost/C:/pyqt5/pyqt_qcombobox_widget.htm)  Un oggetto QComboBox presenta un elenco a discesa di elementi tra cui selezionare. Ci vuole uno spazio minimo sullo schermo nel modulo richiesto per visualizzare solo l'elemento attualmente selezionato. |
| 7 | [**QSpinBox**](file://localhost/C:/pyqt5/pyqt_qspinbox_widget.htm)  Un oggetto QSpinBox presenta all'utente una casella di testo che visualizza un numero intero con il pulsante su/giù alla sua destra. |
| 8 | [**QSlider Widget**](file://localhost/C:/pyqt5/pyqt_qslider_widget_signal.htm)  [**&**](file://localhost/C:/pyqt5/pyqt_qslider_widget_signal.htm)  [**Segnale**](file://localhost/C:/pyqt5/pyqt_qslider_widget_signal.htm)  L'oggetto classe QSlider presenta all'utente una scanalatura su cui è possibile spostare una maniglia. È un widget classico per controllare un valore limitato. |
| 9 | [**QMenuBar, QMenu**](file://localhost/C:/pyqt5/qmenubar_qmenu_qaction_widgets.htm)  [**e**](file://localhost/C:/pyqt5/qmenubar_qmenu_qaction_widgets.htm)  [**QAction**](file://localhost/C:/pyqt5/qmenubar_qmenu_qaction_widgets.htm)  Un QMenuBar orizzontale appena sotto la barra del titolo di un oggetto QMainWindow è riservato alla visualizzazione degli oggetti QMenu. |

|  |  |
| --- | --- |
| 10 | [**QToolBar**](file://localhost/C:/pyqt5/pyqt_qtoolbar_widget.htm)  Un widget QToolBar è un pannello mobile composto da pulsanti di testo, pulsanti con icone o altri widget. |
| 11 | [**QInputDialog**](file://localhost/C:/pyqt5/pyqt_qinputdialog_widget.htm)  Questa è una finestra di dialogo preconfigurata con un campo di testo e due pulsanti, OK e Annulla. La finestra principale raccoglie l'input nella casella di testo dopo che l'utente ha fatto clic sul pulsante OK o ha premuto Invio. |
| 12 | [**QFontDialogo**](file://localhost/C:/pyqt5/pyqt_qfontdialog_widget.htm)  Un'altra finestra di dialogo comunemente usata, un widget di selezione dei caratteri, è l'aspetto visivo della classe QDialog. Il risultato di questa finestra di dialogo è un oggetto Qfont, che può essere utilizzato dalla finestra padre. |
| 13 | [**QFileDialog**](file://localhost/C:/pyqt5/pyqt_qfiledialog_widget.htm)  Questo widget è una finestra di dialogo di selezione file. Consente all'utente di navigare attraverso il file system e selezionare un file da aprire o salvare. La finestra di dialogo viene richiamata tramite funzioni statiche o chiamando exec\_ funzione() sull'oggetto finestra di dialogo. |
| 14 | [**QTab**](file://localhost/C:/pyqt5/pyqt_qtabwidget.htm)  Se un modulo ha troppi campi da visualizzare contemporaneamente, possono essere disposti in pagine diverse posizionate sotto ogni scheda di un widget a schede. QTabWidget fornisce una barra delle schede e un'area di pagina. |
| 15 | [**QStacked**](file://localhost/C:/pyqt5/pyqt_qstackedwidget.htm)  Il funzionamento di QStackedWidget è simile a QTabWidget. Aiuta anche nell'uso efficiente dell'area clienti della finestra. |
| 16 | [**QSplitter**](file://localhost/C:/pyqt5/pyqt_qsplitter_widget.htm)  Questo è un altro gestore di layout avanzato che consente di modificare dinamicamente le dimensioni dei widget figlio trascinando i confini tra di loro. Il controllo Splitter fornisce un handle che può essere trascinato per ridimensionare i controlli. |
| 17 | [**QDock ·**](file://localhost/C:/pyqt5/pyqt_qdockwidget.htm)  Una finestra agganciabile è una sottofinestra che può rimanere in stato mobile o può essere collegata alla finestra principale in una posizione specificata. L'oggetto finestra principale della classe QMainWindow ha un'area riservata alle finestre agganciabili. |
| 18 | [**QStatusBar**](file://localhost/C:/pyqt5/pyqt_qstatusbar_widget.htm)  L'oggetto QMainWindow riserva una barra orizzontale nella parte inferiore come barra di stato. Viene utilizzato per visualizzare informazioni sullo stato permanente o contestuale. |
| 19 | [**QList**](file://localhost/C:/pyqt5/pyqt_qlistwidget.htm)  La classe QListWidget è un'interfaccia basata su elementi per aggiungere o rimuovere elementi da un elenco.  Ogni elemento nell'elenco è un oggetto QListWidgetItem. ListWidget può essere impostato su multiseleggibile. |

|  |  |
| --- | --- |
| 20 | [**QScrollBar**](file://localhost/C:/pyqt5/pyqt_qscrollbar_widget.htm)  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. |
| |  |  | | --- | --- | | 20 | [**QScrollBar**](file://localhost/C:/pyqt5/pyqt_qscrollbar_widget.htm)  Un controllo della barra di scorrimento consente all'utente di accedere a parti del documento che si trovano all'esterno dell'area visualizzabile. Fornisce un indicatore visivo della posizione corrente. | | |  |  | | --- | --- | | 20 | [**QScrollBar**](file://localhost/C:/pyqt5/pyqt_qscrollbar_widget.htm)  Un controllo della barra di scorrimento consente all'utente di accedere a parti del documento che si trovano all'esterno dell'area visualizzabile. Fornisce un indicatore visivo della posizione corrente. | | 21 | [**QCalendario**](file://localhost/C:/pyqt5/pyqt_qcalender_widget.htm)  Il widget QCalendar è un utile controllo di selezione data. Fornisce una visualizzazione mensile.  L'utente può selezionare la data utilizzando il mouse o la tastiera, la data predefinita è la data di oggi. |   21 | [**QCalendario**](file://localhost/C:/pyqt5/pyqt_qcalender_widget.htm)  Il widget QCalendar è un utile controllo di selezione data. Fornisce una visualizzazione mensile.  L'utente può selezionare la data utilizzando il mouse o la tastiera, la data predefinita è la data di oggi. | |

## 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **setAlignment()**  Aligns the text as per alignment constants Qt.AlignLeft  Qt.AlignRight Qt.AlignCenter  Qt.AlignJustify |
| 2 | **setIndent()**  Sets the labels text indent |
| 3 | **setPixmap()**  Displays an image |
| 4 | **Text()**  Displays the caption of the label |
| 5 | **setText()**  Programmatically sets the caption |
| 6 | **selectedText()**  Displays the selected text from the label (The textInteractionFlag must be set to TextSelectableByMouse) |
| 7 | **setBuddy()** |

|  |  |
| --- | --- |
|  | Associates the label with any input widget |
| 8 | **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 hyperlink 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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \* def window():

app = QApplication(sys.argv)

win = QWidget()

l1.setText("Hello World")

l4.setText("[TutorialsPoint](https://www.tutorialspoint.com/)") l2.setText("[welcome to Python GUI Programming](file://localhost/C:/Users/User/Desktop/delete/pyqt5/pyqt_qlabel_widget.htm)")

l1.setAlignment(Qt.AlignCenter)

l3.setAlignment(Qt.AlignCenter) l4.setAlignment(Qt.AlignRight)

|  |  |  |
| --- | --- | --- |
| l1 | = | QLabel() |
| l2 | = | QLabel() |
| l3 | = | QLabel() |
| l4 | = | QLabel() |

l3.setPixmap(QPixmap("python.png"))

vbox = QVBoxLayout() vbox.addWidget(l1) vbox.addStretch() vbox.addWidget(l2) vbox.addStretch() vbox.addWidget(l3) vbox.addStretch() vbox.addWidget(l4)

l4.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\_())

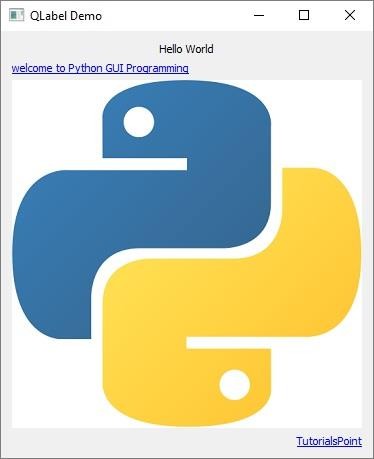
def hovered():

print ("hovering") def clicked():

print ("clicked")

if name == ' main ': window()

The above code produces the following output:



## 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **setAlignment()**  Aligns the text as per alignment constants Qt.AlignLeft  Qt.AlignRight Qt.AlignCenter  Qt.AlignJustify |
| 2 | **clear()**  Erases the contents |
| 3 | **setEchoMode()**  Controls the appearance of the text inside the box. Echomode values are − QLineEdit.Normal  QLineEdit.NoEcho QLineEdit.Password  QLineEdit.PasswordEchoOnEdit |
| 4 | **setMaxLength()**  Sets the maximum number of characters for input |
| 5 | **setReadOnly()**  Makes the text box non-editable |
| 6 | **setText()**  Programmatically sets the text |
| 7 | **text()**  Retrieves text in the field |
| 8 | **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 |
| 9 | **setInputMask()**  Applies mask of combination of characters for input |
| 10 | **setFont()**  Displays the contents QFont object |

QLineEdit object emits the following signals:

Given below are the most commonly used methods of signals.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **cursorPositionChanged()**  Whenever the cursor moves |
| 2 | **editingFinished()**  When you press 'Enter' or the field loses focus |
| 3 | **returnPressed()**  When you press 'Enter' |
| 4 | **selectionChanged()**  Whenever the selected text changes |
| 5 | **textChanged()**  As text in the box changes either by input or by programmatic means |
| 6 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \*

from PyQt5.QtWidgets 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):

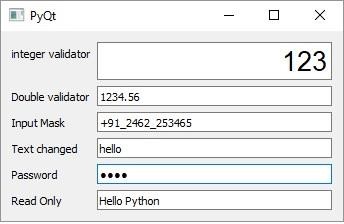
print ("contents of text box: {}".format(text))

def enterPress():

print ("editeing finished")

if name == ' main ': window()

The above code produces the following output:



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

## 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **setCheckable()**  Recognizes pressed and released states of button if set to true |
| 2 | **toggle()**  Toggles between checkable states |
| 3 | **setIcon()**  Shows an icon formed out of pixmap of an image file |
| 4 | **setEnabled()**  When set to false, the button becomes disabled, hence clicking it doesn’t emit a signal |
| 5 | **isChecked()**  Returns Boolean state of button |
| 6 | **setDefault()**  Sets the button as default |
| 7 | **setText()**  Programmatically sets buttons’ caption |
| 8 | **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.

Following is the example of the above statement:

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

The complete code is given below:

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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("pythonlogo.png"))) 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():

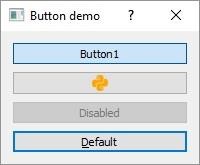
app = QApplication(sys.argv) ex = Form()

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

## 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.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | setChecked()  Changes the state of radio button |
| 2 | setText()  Sets the label associated with the button |
| 3 | text()  Retrieves the caption of button |
| 4 | 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"

Complete code for QRadioButton example is as below:

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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")

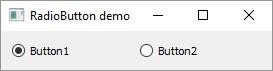
def main():

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

## 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **setChecked()**  Changes the state of checkbox button |
| 2 | **setText()**  Sets the label associated with the button |
| 3 | **text()**  Retrieves the caption of the button |
| 4 | **isChecked()**  Checks if the button is selected |
| 5 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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()

The above code produces the following output:



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

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().

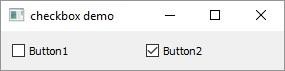
self.bg.buttonClicked[QAbstractButton].connect(self.btngroup)

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

def btngroup(self,btn):

print (btn.text()+" is selected")

After above changes, the application window appears as follows:



Button1 is selected Button2 is selected Button1 is selected

## 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addItem()**  Adds string to collection |
| 2 | **addItems()**  Adds items in a list object |
| 3 | **Clear()**  Deletes all items in the collection |
| 4 | **count()**  Retrieves number of items in the collection |
| 5 | **currentText()**  Retrieves the text of currently selected item |
| 6 | **itemText()**  Displays text belonging to specific index |
| 7 | **currentIndex()**  Returns index of selected item |
| 8 | **setItemText()**  Changes text of specified index |

### QComboBox Signals

The following methods are commonly used in QComboBox Signals:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **activated()**  When the user chooses an item |

|  |  |
| --- | --- |
| 2 | **currentIndexChanged()**  Whenever the current index is changed either by the user or programmatically |
| 3 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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())

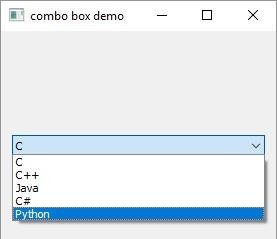
def main():

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 as follows:

C C++

Java C#

Python

Current selection index 4 selection changed Python

## 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. You can use QDoubleSpinBox for float values.

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

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **setMinimum()**  Sets the lower bound of counter |
| 2 | **setMaximum()**  Sets the upper bound of counter |
| 3 | **setRange()**  Sets the minimum, maximum and step value |
| 4 | **setValue()**  Sets the value of spin box programmatically |
| 5 | **Value()**  Returns the current value |
| 6 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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()))

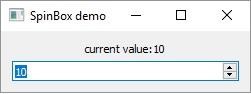
def main():

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

ex.show() sys.exit(app.exec\_())

if name == ' main ': main()

The above code produces the following output:



## QSlider Widget

**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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **setMinimum()**  Sets the lower bound of the slider |
| 2 | **setMaximum()**  Sets the upper bound of the slider |
| 3 | **setSingleStep()**  Sets the increment/decrement step |
| 4 | **setValue()**  Sets the value of the control programmatically |
| 5 | **value()**  Returns the current value |
| 6 | **setTickInterval()**  Puts the number of ticks on the groove |
| 7 | **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

The following are the methods in QSlider Signals:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **valueChanged()**  When the slider's value has changed |
| 2 | **sliderPressed()**  When the user starts to drag the slider |
| 3 | **sliderMoved()**  When the user drags the slider |
| 4 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \*

from PyQt5.QtWidgets 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("Slider 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.

## 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.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **menuBar()**  Returns main window's QMenuBar object |
| 2 | **addMenu()**  Adds a new QMenu object to menu bar |
| 3 | **addAction()**  Adds an action button to QMenu widget consisting of text or icon |
| 4 | **setEnabled()**  Sets state of action button to enabled/disabled |
| 5 | **addSeperator()**  Adds a separator line in the menu |
| 6 | **Clear()**  Removes contents of menu/menu bar |
| 7 | **setShortcut()**  Associates keyboard shortcut to action button |
| 8 | **setText()** |

|  |  |
| --- | --- |
|  | Assigns text to action button |
| 9 | **setTitle()**  Sets the title of QMenu widget |
| 10 | **text()**  Retrieves the text associated with QAction object |
| 11 | **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 PyQt5.QtCore import \*

from PyQt5.QtGui import \* from PyQt5.QtWidgets 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():

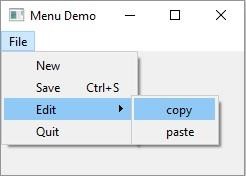
app = QApplication(sys.argv) ex = menudemo()

ex.show() sys.exit(app.exec\_())

if name == ' main ':

main()

The above code produces the following output:



## 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addAction()**  Adds tool buttons having text or icon |
| 2 | **addSeperator()**  Shows tool buttons in groups |
| 3 | **addWidget()**  Adds controls other than button in the toolbar |
| 4 | **addToolBar()**  QMainWindow class method adds a new toolbar |
| 5 | **setMovable()**  Toolbar becomes movable |
| 6 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

class tooldemo(QMainWindow):

def init (self, parent = None): super(tooldemo, self). init (parent) layout = QVBoxLayout()

tb = self.addToolBar("File")

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

open = QAction(QIcon("open.png"),"open",self) tb.addAction(open)

save = QAction(QIcon("save.png"),"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())

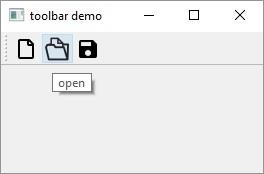
def main():

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

ex.show() sys.exit(app.exec\_())

if name == ' main ': main()

The above code produces the following output:



## 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | getInt()  Creates a spinner box for integer number |
| 2 | getDouble()  Spinner box with floating point number can be input |
| 3 | getText()  A simple line edit field to type text |
| 4 | 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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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 dialog","enter a number")

if ok:

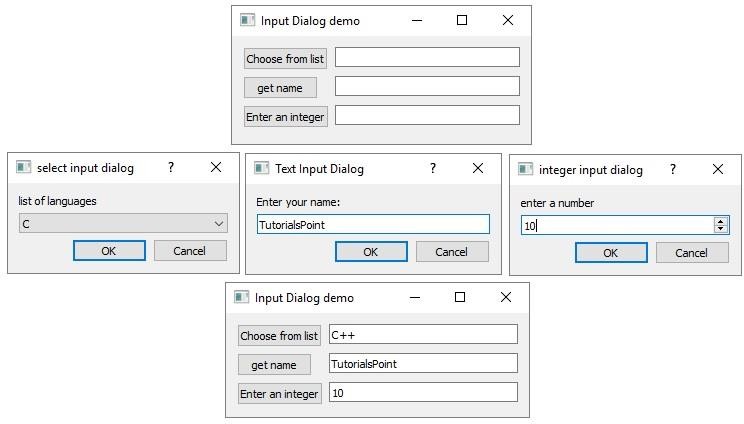
self.le2.setText(str(num))

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:



## 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 as follows:

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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)

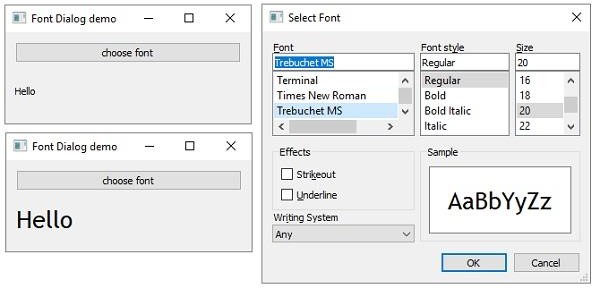
def main():

app = QApplication(sys.argv) ex = fontdialogdemo() ex.show() sys.exit(app.exec\_())

if name == ' main ':

main()

The above code produces the following output:



## 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **getOpenFileName()**  Returns name of the file selected by the user to open it |
| 2 | **getSaveFileName()**  Uses the file name selected by the user to save the file |
| 3 | **setacceptMode()**  Determines whether the file box acts as open or save dialog QFileDialog.AcceptOpen  QFileDialog.AcceptSave |
| 4 | **setFileMode()**  Type of selectable files. Enumerated constants are − QFileDialog.AnyFile  QFileDialog.ExistingFile  QFileDialog.Directory QFileDialog.ExistingFiles |
| 5 | **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 for static method is as follows:

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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("")

layout.addWidget(self.le)

self.setLayout(layout) self.setWindowTitle("File Dialog demo")

def getfile(self):

fname = QFileDialog.getOpenFileName(self, 'Open file', 'd:\\',"Image files (\*.jpg \*.png)")

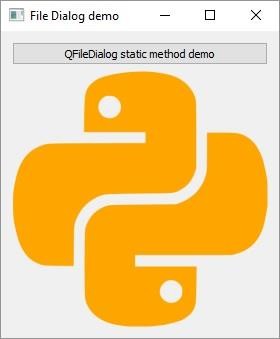
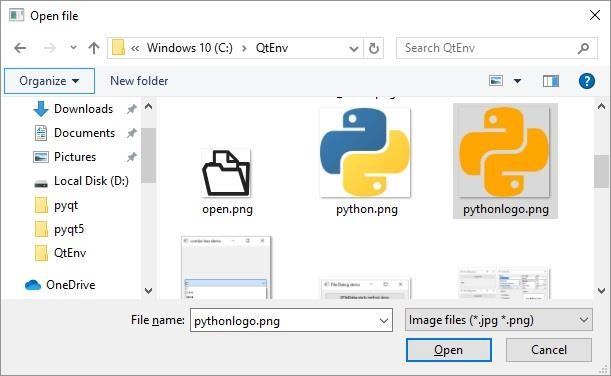
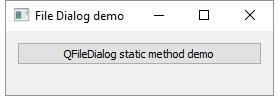
self.le.setPixmap(QPixmap(fname[0]))

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:



The complete code for exec\_() method is as follows:

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

class CustomDialog(QFileDialog):

def init (self, \*args, \*\*kwargs): super(CustomDialog, self). init (\*args, \*\*kwargs)

self.setWindowTitle("HELLO!")

QBtn = QDialogButtonBox.Ok | QDialogButtonBox.Cancel

self.buttonBox = QDialogButtonBox(QBtn) self.buttonBox.accepted.connect(self.accept) self.buttonBox.rejected.connect(self.reject)

self.layout = QVBoxLayout() self.layout.addWidget(self.buttonBox) self.setLayout(self.layout)

class filedialogdemo(QWidget):

def init (self, parent = None):

super(filedialogdemo, self). init (parent)

layout = QVBoxLayout()

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 getfiles(self, s): print("click", s)

dlg = CustomDialog(self) if dlg.exec\_():

filenames = dlg.selectedFiles() f = open(filenames[0], 'r')

with f:

data = f.read() self.contents.setText(data)

def main():

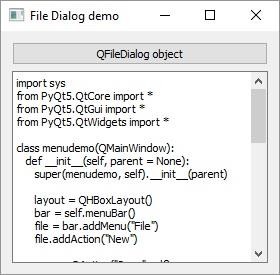
app = QApplication(sys.argv) ex = filedialogdemo() ex.show() sys.exit(app.exec\_())

if name == ' main ': main()

The above code produces the following output:



Text in selected file will be displayed in TextEdit control.



## QTab Widget

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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addTab()**  Adds a tab associated with a widget page |
| 2 | **insertTab()**  Inserts a tab with the page at the desired position |
| 3 | **removeTab()**  Removes tab at given index |
| 4 | **setCurrentIndex()**  Sets the index of the currently visible page as current |
| 5 | **setCurrentWidget()**  Makes the visible page as current |
| 6 | **setTabBar()**  Sets the tab bar of the widget |
| 7 | **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 |
| 8 | **setTabText()**  Defines the label associated with the tab index |

The following signals are associated with QTabWidget object:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **currentChanged()**  Whenever the current page index changes |

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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")

|  |  |
| --- | --- |
| 2 | **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:

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)

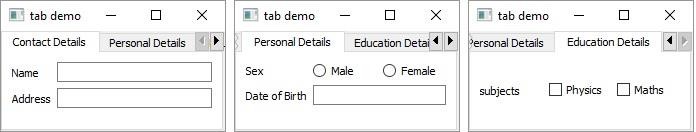
def main():

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

ex.show() sys.exit(app.exec\_())

if name == ' main ': main()

All three tabs are shown in the following output:



## 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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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)

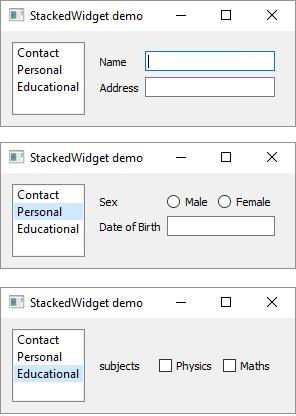
def display(self,i): self.Stack.setCurrentIndex(i)

def main():

app = QApplication(sys.argv) ex = stackedExample() sys.exit(app.exec\_())

if name == ' main ': main()

The above code produces the following output:



## 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addWidget()**  Adds the widget to splitter’s layout |
| 2 | **indexOf()**  Returns the index of the widget in the layout |
| 3 | **insetWidget()**  Inserts a widget at the specified index |
| 4 | **setOrientation()**  Sets the layout of splitter to Qt.Horizontal or Qt.Vertical |
| 5 | **setSizes()**  Sets the initial size of each widget |
| 6 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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'))

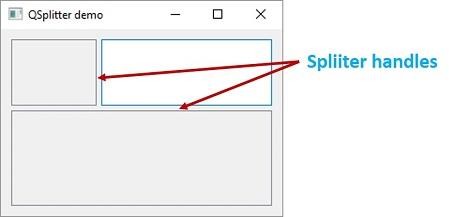
self.setGeometry(300, 300, 300, 200) self.setWindowTitle('QSplitter demo') self.show()

def main():

app = QApplication(sys.argv) ex = Example() sys.exit(app.exec\_())

if name == ' main ': main()

The above code produces the following output:



## QDock Widget

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:

**Sr.No. Methods & Description**

#### setWidget()

1

Sets any QWidget in the dock window’s area

#### setFloating()

2

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 |

3

#### setFeatures()

Sets the features of dock window

|  |
| --- |
| DockWidgetClosable |
| DockWidgetMovable |
| DockWidgetFloatable |
| DockWidgetVerticalTitleBar |
| NoDockWidgetFeatures |

4

### 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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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")

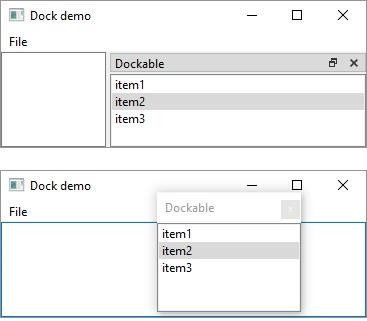
def main():

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

ex.show() sys.exit(app.exec\_())

if name == ' main ': main()

The above code produces the following output. Click on Dock icon to undock the ListWidget window. Double click to dock again:



## 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

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addWidget()**  Adds the given widget object in the status bar |
| 2 | **addPermanentWidget()**  Adds the given widget object in the status bar permanently |
| 3 | **showMessage()**  Displays a temporary message in the status bar for a specified time interval |
| 4 | **clearMessage()**  Removes any temporary message being shown |
| 5 | **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 temporary 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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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():

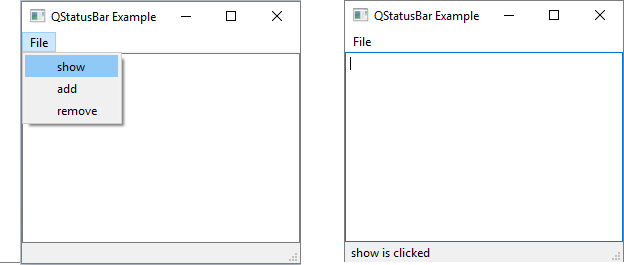
app = QApplication(sys.argv) ex = statusdemo()

ex.show() sys.exit(app.exec\_())

if name == ' main ':

main()

The above code produces the following output. Status bar shows caption of selected menu button:



## QList Widget

**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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addItem()**  Adds **QListWidgetItem** object or string in the list |
| 2 | **addItems()**  Adds each item in the list |
| 3 | **insertItem()**  Inserts item at the specified index |
| 4 | **clear()**  Removes contents of the list |
| 5 | **setCurrentItem()**  Sets currently selected item programmatically |
| 6 | **sortItems()**  Rearranges items in ascending order |

Following are the signals emitted by QListWidget:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **currentItemChanged()**  Whenever current item changes |
| 2 | **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()

The above code produces the following output. Status bar shows caption of selected menu button:



## QScrollBar Widget

A QScrollbar 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 **QScrollbar** has four controls:

|  |  |
| --- | --- |
| a: slider  b: Two Scroll arrows c: Page control | Scroll Bar |

Following signals of QScrollBar class are frequently used:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **valueChanged()**  When the scrollbar’s value changes |
| 2 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

class Example(QWidget):

def init (self):

super(Example, self). init () self.initUI()

def initUI(self):

vbox = QVBoxLayout(self) self.setLayout(vbox) 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) vbox.addLayout(hbox)

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

def sliderval(self): 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:



## 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.

Given below are the most commonly used methods of QCalendar:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **setDateRange()**  Sets the lower and upper date available for selection |
| 2 | **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 |
| 3 | **setMinimumDate()**  Sets the lower date for selection |

|  |  |
| --- | --- |
| 4 | **setMaximumDate()**  Sets the upper date for selection |
| 5 | **setSelectedDate()**  Sets a QDate object as the selected date |
| 6 | **showToday()**  Shows the month of today |
| 7 | **selectedDate()**  Retrieves the selected date |
| 8 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

class Example(QWidget):

def init (self):

super(Example, self). init ()

self.initUI()

def initUI(self):

cal = QCalendarWidget(self) cal.setGridVisible(True) cal.move(20, 20)

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

self.lbl = QLabel(self) date = cal.selectedDate()

self.lbl.setText(date.toString()) self.lbl.move(100, 250)

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

def showDate(self, date):

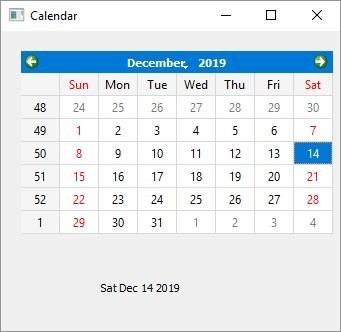
self.lbl.setText(date.toString())

def main():

app = QApplication(sys.argv) ex = Example() sys.exit(app.exec\_())

if name == ' main ': main()

The above code produces the following output. Selected date is displayed on the label:



# PyQt5 — QDialog Class

PyQt

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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

def window():

app = QApplication(sys.argv) w = QWidget()

btn = QPushButton(w) btn.setText("Hello World!") btn.move(100,50) btn.clicked.connect(showdialog) w.setWindowTitle("PyQt Dialog demo") w.show()

sys.exit(app.exec\_())

def showdialog(): dlg = QDialog()

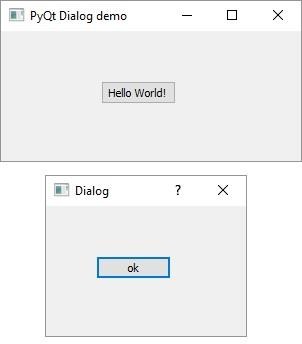
b1 = QPushButton("ok",dlg) b1.move(50,50) dlg.setWindowTitle("Dialog")

PyQt

dlg.setWindowModality(Qt.ApplicationModal) dlg.exec\_()

if name == ' main ': window()

The above code produces the following output. Click on button in main window and dialog box pops up:



# PyQt5 — QMessageBox

PyQt

**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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **setIcon()**  Displays predefined icon corresponding to severity of the message Question  Information  Warning Critical |
| 2 | **setText()**  Sets the text of the main message to be displayed |
| 3 | **setInformativeText()**  Displays additional information |
| 4 | **setDetailText()**  Dialog shows a Details button. This text appears on clicking it |
| 5 | **setTitle()**  Displays the custom title of dialog |
| 6 | **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 |
| 7 | **setDefaultButton()**  Sets the button as default. It emits the clicked signal if Enter is pressed |
| 8 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

def window():

app = QApplication(sys.argv) w = QWidget()

b = QPushButton(w) b.setText("Show message!")

b.move(100,50) b.clicked.connect(showdialog) w.setWindowTitle("PyQt MessageBox 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)

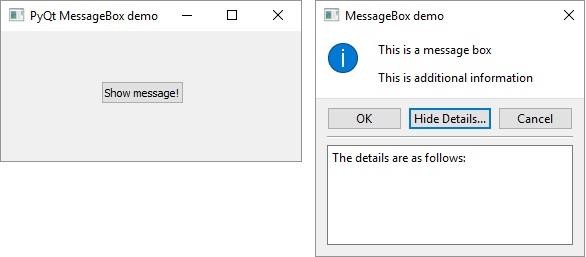
retval = msg.exec\_()

def msgbtn(i):

print ("Button pressed is:",i.text())

if name == ' main ': window()

The above code produces the following output. Message Box pops up when main windows’ button is clicked:



If you click on Ok or Cancel button on MessageBox, the following output is produced on the console:

Button pressed is: OK Button pressed is: Cancel

# PyQt5 — Multiple Document Interface

PyQt

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 less 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **addSubWindow()**  Adds a widget as a new subwindow in MDI area |
| 2 | **removeSubWindow()**  Removes a widget that is internal widget of a subwindow |
| 3 | **setActiveSubWindow()**  Activates a subwindow |
| 4 | **cascadeSubWindows()**  Arranges subwindows in MDiArea in a cascaded fashion |
| 5 | **tileSubWindows()**  Arranges subwindows in MDiArea in a tiled fashion |
| 6 | **closeActiveSubWindow()**  Closes the active subwindow |
| 7 | **subWindowList()**  Returns the list of subwindows in MDI Area |
| 8 | **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, the 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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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():

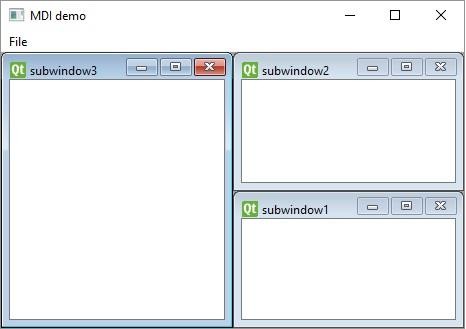
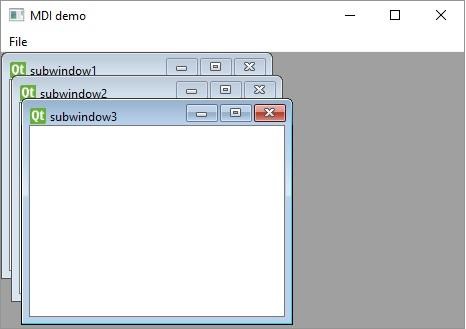
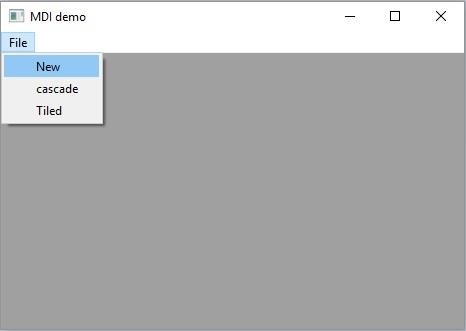
app = QApplication(sys.argv) ex = MainWindow()

ex.show() sys.exit(app.exec\_())

if name == ' main ':

main()

Run above code and three windows in cascased and tiled formation:



# PyQt5 — Drag & Drop

PyQt

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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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 and drop')

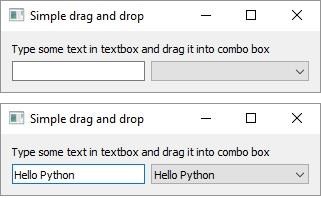
def main():

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

ex.show() app.exec\_()

if name == ' main ': main()

The above code produces the following output:



# PyQt5 — Database Handling

PyQt

PyQt5 library contains **QtSql** module. It is 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Driver Type & Description** |
| 1 | QDB2  IBM DB2 |
| 2 | QIBASE  Borland InterBase Driver |
| 3 | QMYSQL  MySQL Driver |
| 4 | QOCI  Oracle Call Interface Driver |
| 5 | QODBC  ODBC Driver (includes Microsoft SQL Server) |
| 6 | QPSQL  PostgreSQL Driver |
| 7 | QSQLITE  SQLite version 3 or above |
| 8 | QSQLITE2  SQLite version 2 |

### Example

For this chapter, 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **setDatabaseName()**  Sets the name of the database with which connection is sought |
| 2 | **setHostName()**  Sets the name of the host on which the database is installed |
| 3 | **setUserName()**  Specifies the user name for connection |
| 4 | **setPassword()**  Sets the connection object’s password if any |
| 5 | **commit()**  Commits the transactions and returns true if successful |
| 6 | **rollback()**  Rolls back the database transaction |
| 7 | **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. First step is to create SQlite database using the following statements:

db = QSqlDatabase.addDatabase('QSQLITE') db.setDatabaseName('sportsdatabase.db')

Next, obtain Query object with **QSqlQuery**() method and call its most important method **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.

import sys

from PyQt5.QtSql import \* from PyQt5.QtCore import \* from PyQt5.QtGui import \*

from PyQt5.QtWidgets import \*

def createDB():

db = QSqlDatabase.addDatabase('QSQLITE') db.setDatabaseName('sportsdatabase.db')

if not db.open():

msg = QMessageBox() msg.setIcon(QMessageBox.Critical) msg.setText("Error in Database Creation") retval = msg.exec\_()

return False

query = QSqlQuery()

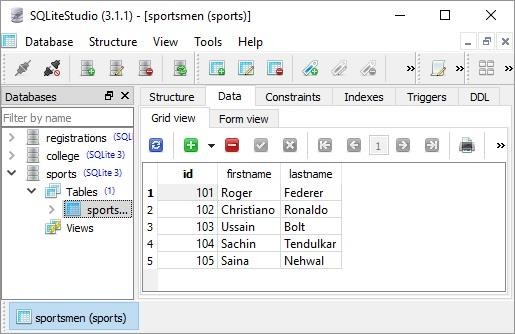
query.exec\_("create table sportsmen(id int primary key, " "firstname varchar(20), lastname varchar(20))")

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 ':

app = QApplication(sys.argv) createDB()

To confirm that the SQLite datbase is created with above records added in sportsmen table in it, use a SQLite Gui utility called **SQLiteStudio.**



**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 QSqlTableModel 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 PyQt5.QtSql import \* from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

def initializeModel(model): model.setTable('sportsmen') model.setEditStrategy(QSqlTableModel.OnFieldChange) model.select()

model.setHeaderData(0, Qt.Horizontal, "ID") model.setHeaderData(1, Qt.Horizontal, "First name") model.setHeaderData(2, Qt.Horizontal, "Last name")

def createView(title, model): view = 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 = QApplication(sys.argv)

db = QSqlDatabase.addDatabase('QSQLITE') db.setDatabaseName('sportsdatabase.db') model = QSqlTableModel()

delrow = -1 initializeModel(model)

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

dlg = QDialog() layout = QVBoxLayout()

layout.addWidget(view1)

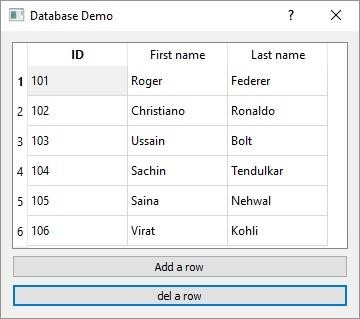
button = QPushButton("Add a row") button.clicked.connect(addrow) layout.addWidget(button)

btn1 = 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:

Try adding and deleting a few records and go back to SQLiteStudio to confirm the transactions.

# PyQt5 — Drawing API

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.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **begin()**  Starts painting on the target device |
| 2 | **drawArc()**  Draws an arc between the starting and the end angle |
| 3 | **drawEllipse()**  Draws an ellipse inside a rectangle |
| 4 | **drawLine()**  Draws a line with endpoint coordinates specified |
| 5 | **drawPixmap()**  Extracts pixmap from the image file and displays it at the specified position |
| 6 | **drawPolygon()**  Draws a polygon using an array of coordinates |
| 7 | **drawRect()**  Draws a rectangle starting at the top-left coordinate with the given width and height |
| 8 | **drawText()**  Displays the text at given coordinates |
| 9 | **fillRect()**  Fills the rectangle with the QColor parameter |

|  |  |
| --- | --- |
| 10 | **setBrush()**  Sets a brush style for painting |
| 11 | **setPen()**  Sets the color, size and style of pen to be used for drawing |

### Example

In the following code, various methods of PyQt's drawing methods are used.

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets 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("pythonlogo.png"))

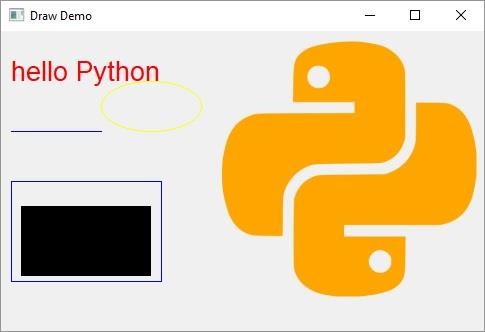
qp.fillRect(20,175,130,70,QBrush(Qt.SolidPattern)) qp.end()

def main():

app = QApplication(sys.argv) ex = Example() sys.exit(app.exec\_())

if name == ' main ': main()

The above code produces the following output:



# PyQt5 ― Brush Style Constants

PyQt

In this chapter, we shall learn Brush Style Constants.

## Brush Style Constants

Given below are the Brush Style Constants:

|  |  |
| --- | --- |
| Qt.NoBrush | No brush pattern. |
| Qt.SolidPattern | Uniform color. |
| Qt.Dense1Pattern | Extremely dense pattern. Denisty 1-7 |
| 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 Styles

Given below are the 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 |

PyQt

## Predefined QColor Objects

Given below are the 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.

# PyQt5 — QClipboard

PyQt

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 **clipboard()** method 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **clear()**  Clears clipboard contents |
| 2 | **setImage()**  Copies QImage into clipboard |
| 3 | **setMimeData()**  Sets MIME data into clipboard |
| 4 | **setPixmap()**  Copies Pixmap object in clipboard |
| 5 | **setText()**  Copies QString in clipboard |
| 6 | **text()**  Retrieves text from clipboard |

Signal associated with clipboard object is:

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **dataChanged()**  Whenever clipboard data changes |

### Example

In the following example, two TextEdit objects and two Pushbuttons 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.

import sys

from PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

class Example(QWidget):

def init (self):

super(Example, self). init ()

self.initUI()

def initUI(self):

hbox = QVBoxLayout() self.edit1=QTextEdit() hbox.addWidget(self.edit1) self.btn1=QPushButton("Copy") hbox.addWidget(self.btn1) self.edit2=QTextEdit() self.btn2=QPushButton("Paste") hbox.addWidget(self.edit2) hbox.addWidget(self.btn2) self.btn1.clicked.connect(self.copytext) self.btn2.clicked.connect(self.pastetext) self.setLayout(hbox)

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

def copytext(self):

#clipboard.setText(self.edit1.copy()) self.edit1.copy()

print (clipboard.text())

msg=QMessageBox()

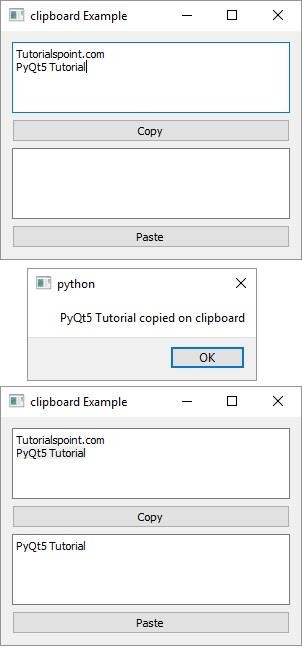
msg.setText(clipboard.text()+" copied on clipboard") msg.exec\_()

def pastetext(self): self.edit2.setText(clipboard.text())

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

ex = Example() ex.setWindowTitle("clipboard Example") sys.exit(app.exec\_())

The above code produces the following output:



# PyQt5 — QPixmap Class

PyQt

**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:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **copy()**  Copies pixmap data from a QRect object |
| 2 | **fromImage()**  Converts QImage object into QPixmap |
| 3 | **grabWidget()**  Creates a pixmap from the given widget |
| 4 | **grabWindow()**  Create pixmap of data in a window |
| 5 | **Load()** |

|  |  |
| --- | --- |
|  | Loads an image file as pixmap |
| 6 | **save()**  Saves the QPixmap object as a file |
| 7 | **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 PyQt5.QtCore import \* from PyQt5.QtGui import \* from PyQt5.QtWidgets import \*

def window():

app = QApplication(sys.argv) win = QWidget()

l1 = QLabel() l1.setPixmap(QPixmap("python.png"))

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:

