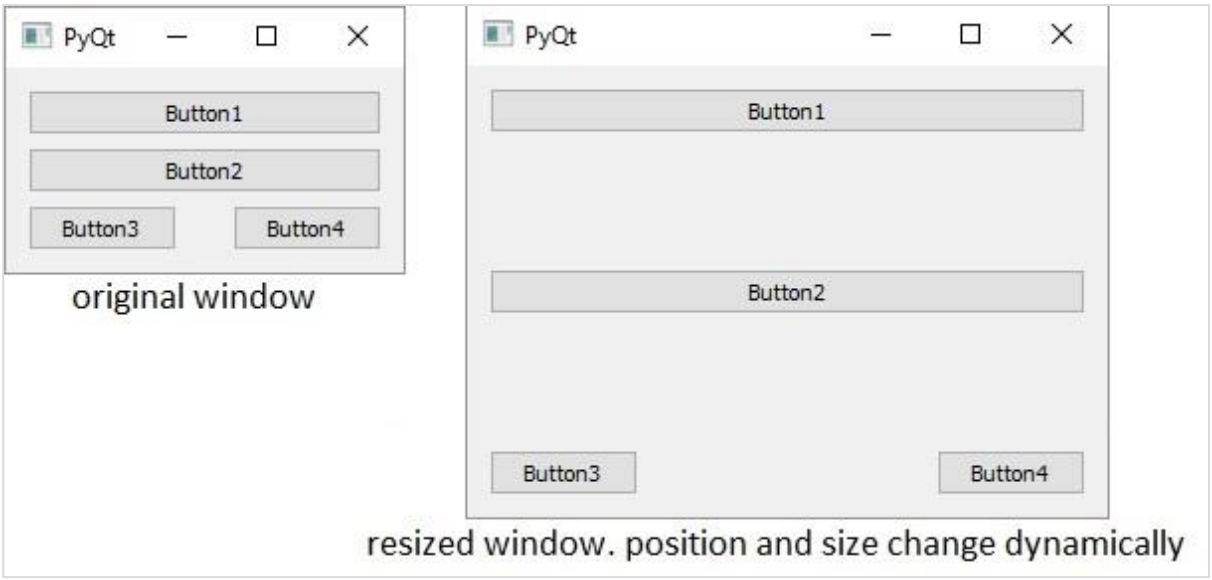
```
if __name__ == '__main__':
   window()
```

The above code produces the following output:



QGridLayout Class

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

Sr.No.	Methods & Description
1	**addWidget(QWidget, int r, int c)** Adds a widget at specified row and column
2	**addWidget(QWidget, int r, int c, int rowspan, int columnspan)** Adds a widget at specified row and column and having specified width and/or height

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

Sr.No.	Methods & Description
1	**addLayout(QLayout, int r, int c)** Adds a layout object at specified row and column

Example

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

```
import sys
from 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()
```

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

The above code produces the following output:

QFormLayout Class

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

Sr.No.	Methods & Description
1	**addRow(QLabel, QWidget)** Adds a row containing label and input field
2	**addRow(QLabel, QLayout)** Adds a child layout in the second column
3	**addRow(QWidget)** Adds a widget spanning both columns

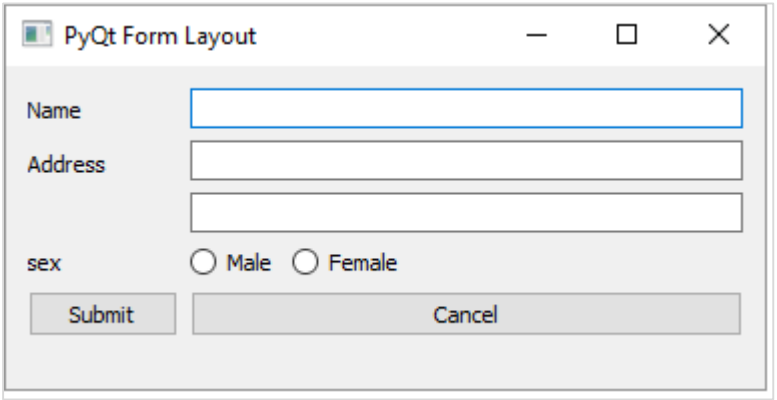
Example

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

```
import sys
from 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()
```

The above code produces the following output:



8. PyQt5 — Basic Widgets

Here is the list of Widgets which we will discuss one by one in this chapter.

Sr.No	Widgets & Description
1	**QLabel** 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.
2	**QLineEdit** 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.
3	**QPushButton** In PyQt API, the QPushButton class object presents a button which when clicked can be programmed to invoke a certain function.
4	**QRadioButton** 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.
5	**QCheckBox** 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.
6	**QComboBox** 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.
7	**QSpinBox** A QSpinBox object presents the user with a textbox which displays an integer with up/down button on its right.
8	**QSlider Widget & Signal** QSlider class object presents the user with a groove over which a handle can be moved. It is a classic widget to control a bounded value.
9	**QMenuBar, QMenu & QAction** A horizontal QMenuBar just below the title bar of a QMainWindow object is reserved for displaying QMenu objects.

QToolBar

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

QInputDialog

11 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.

QFontDialog

12 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.

QFileDialog

13 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.

QTab

14 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.

QStacked

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

QSplitter

16 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.

QDock

17 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.

QStatusBar

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

QList

19 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.

20	**QScrollBar** 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.
21	**QCalendar** 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.

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 = QLabel()
   l2 = QLabel()
   l3 = QLabel()
   l4 = QLabel()

   l1.setText("Hello World")
   l4.setText("TutorialsPoint")
   l2.setText("welcome to Python GUI Programming")

   l1.setAlignment(Qt.AlignCenter)
   l3.setAlignment(Qt.AlignCenter)
   l4.setAlignment(Qt.AlignRight)
```

```
   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.

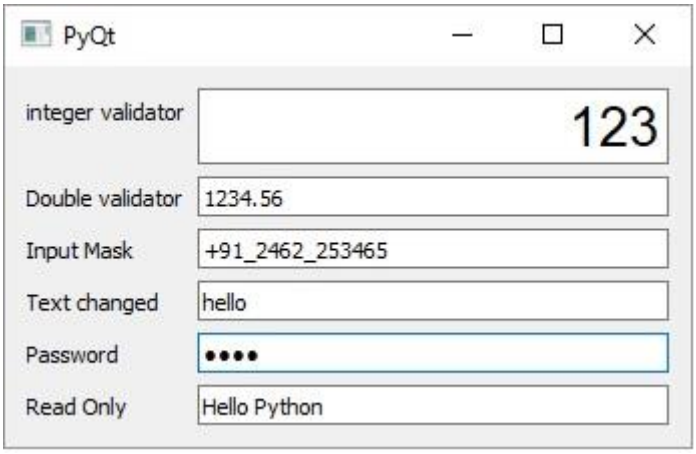
Contents of **e5** field are echoed in password form as its EchoMode property is set to Password. Its editingfinished() signal is connected to presenter() method. So, once the user presses the Enter key, the function will be executed. The field **e6** shows a default text, which cannot be edited as it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

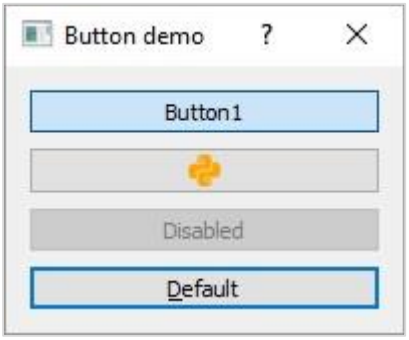
The complete code is given below:

```
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.

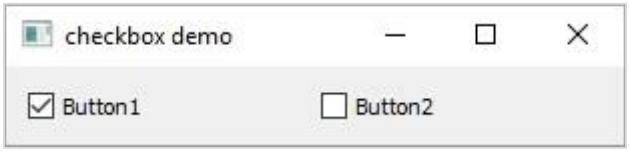
```
if b.text() == "Button1":
   if b.isChecked() == True:
      print b.text()+" is selected"
   else:
      print b.text()+" is deselected"
```

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)
```

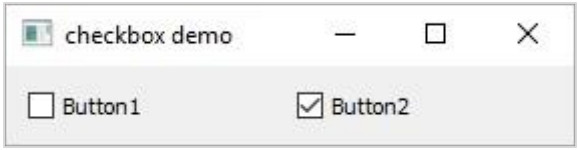
QButtonGroup object, provides abstract container for buttons and doesn't have a visual representation. It emits buttonCliked() signal and sends Button object's reference to the slot function btngroup().

```
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.

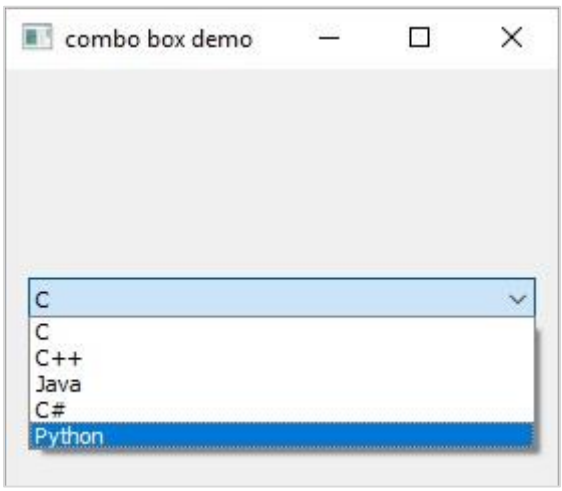
```
def selectionchange(self,i):
   print ("Items in the list are :")

   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)
```

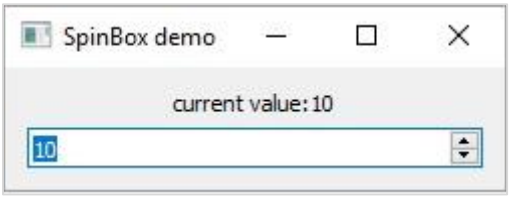
The valueChange() function displays the current value as caption of the label.

```
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.

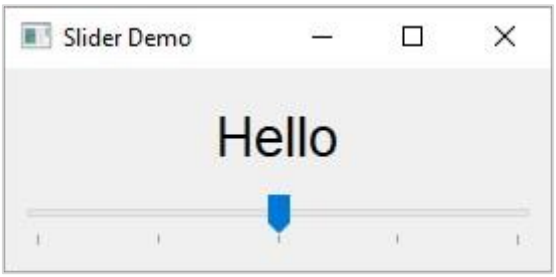
```
size = self.sl.value()
self.l1.setFont(QFont("Arial",size))
```

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)
```

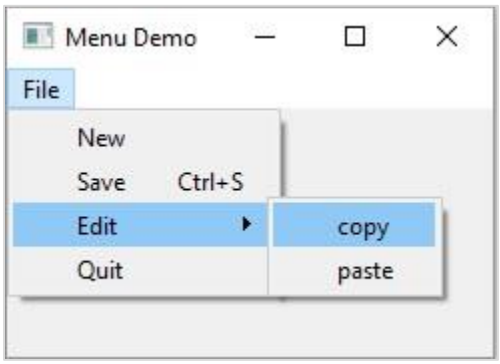
The complete code is as follows:

```
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().

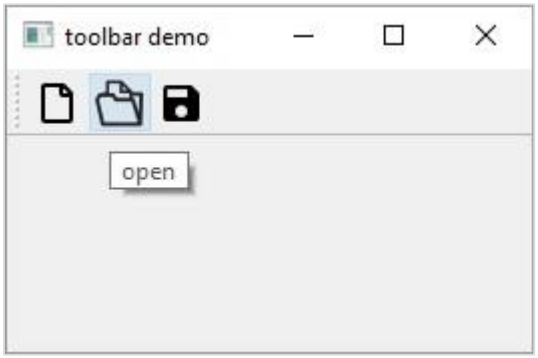
```
tb.actionTriggered[QAction].connect(self.toolbtnpressed)
```

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))
```

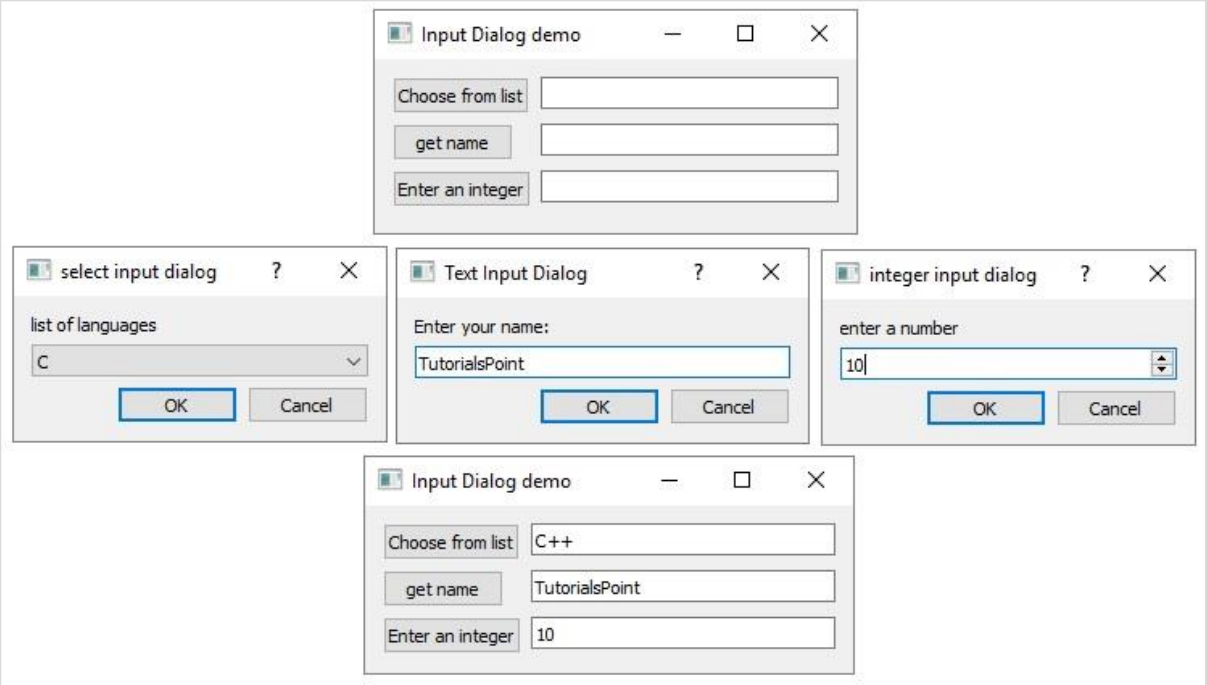
The complete code is as follows:

```
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

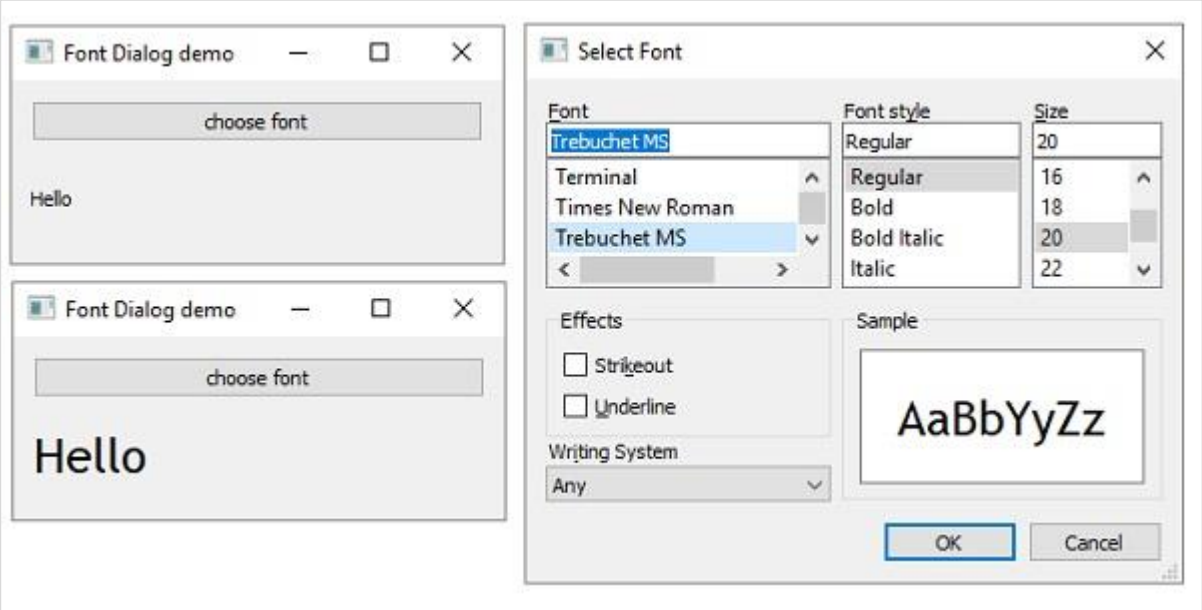
The following example has a button and a label. When the button is clicked, the font dialog pops up. The font chosen by the user (face, style and size) is applied to the text on the label.

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.

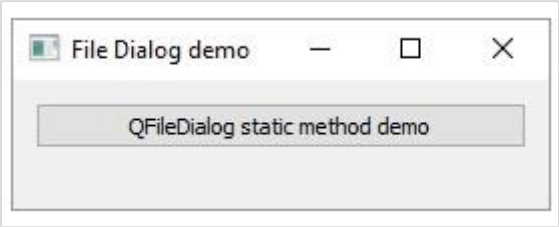
```
f = open(filenames[0], 'r')
   with f:
      data = f.read()
      self.contents.setText(data)
```

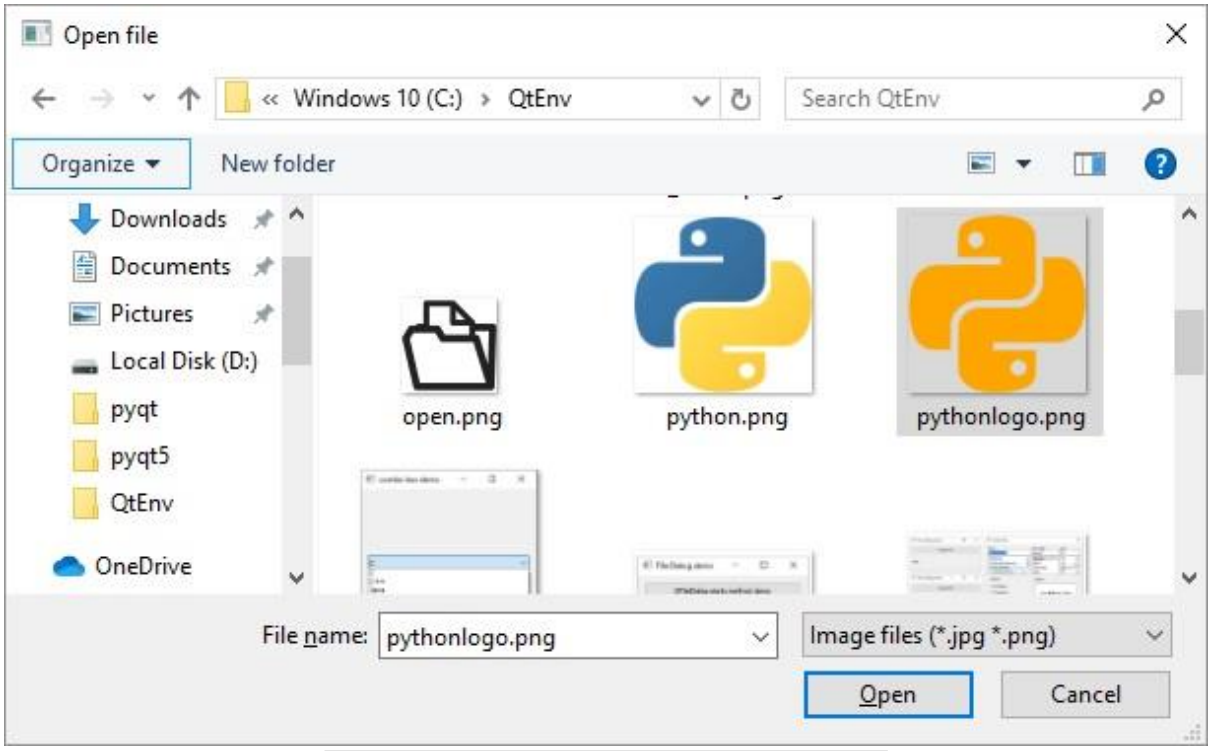
The complete code for static method is as follows:

```
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:





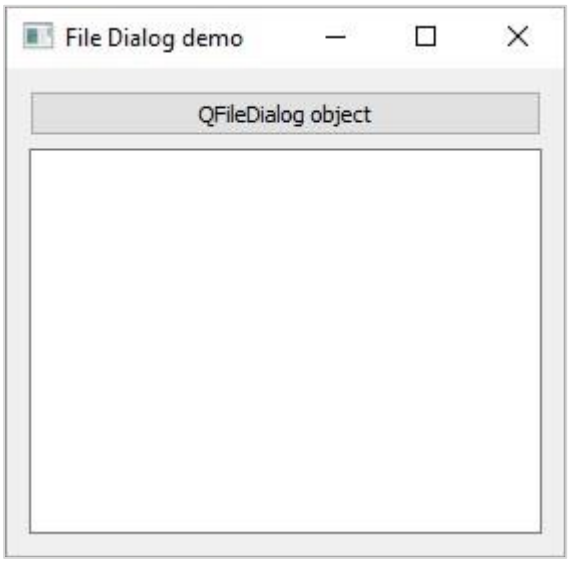
The complete code for exec_() method is as follows:

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
class CustomDialog(QFileDialog):
```

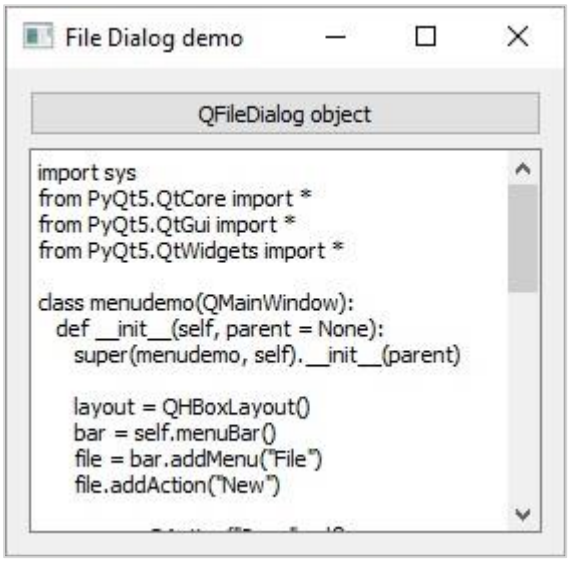
```
    def __init__(self, *args, **kwargs):
        super(CustomDialog, self).__init__(*args, **kwargs)

        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

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.

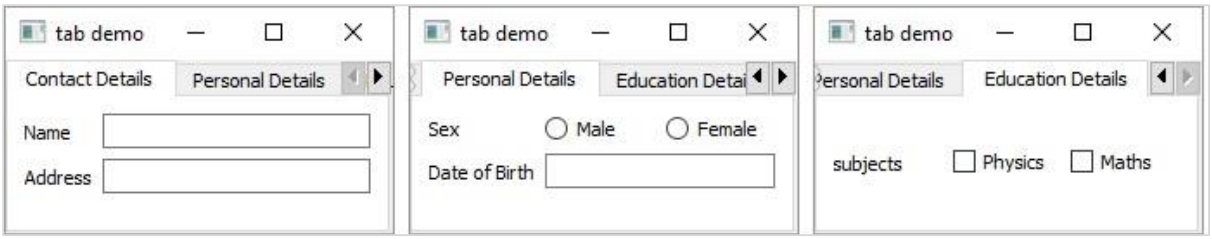
```
self.setTabText(0,"Contact Details")
self.setTabText(1,"Personal Details")
self.setTabText(2,"Education Details")
```

The complete code is as follows:

```
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")
```

```
   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)
```

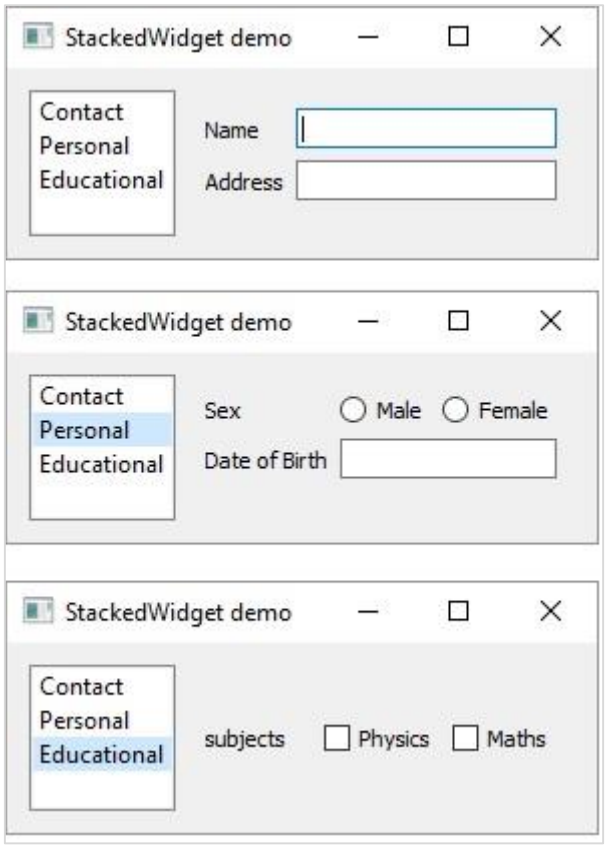
The complete code is as follows:

```
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()
```

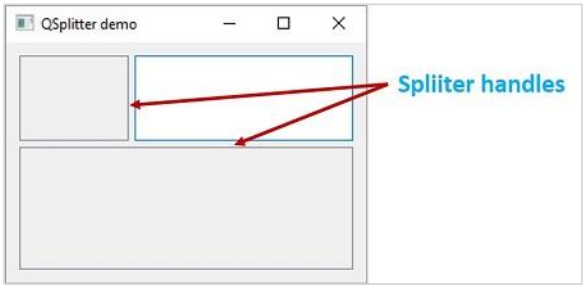
```
splitter2 = QSplitter(Qt.Vertical)
splitter2.addWidget(splitter1)
splitter2.addWidget(bottom)

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

Example

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

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

A dockable window is first created.

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

A QListWidget object is added as a dock window.

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

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

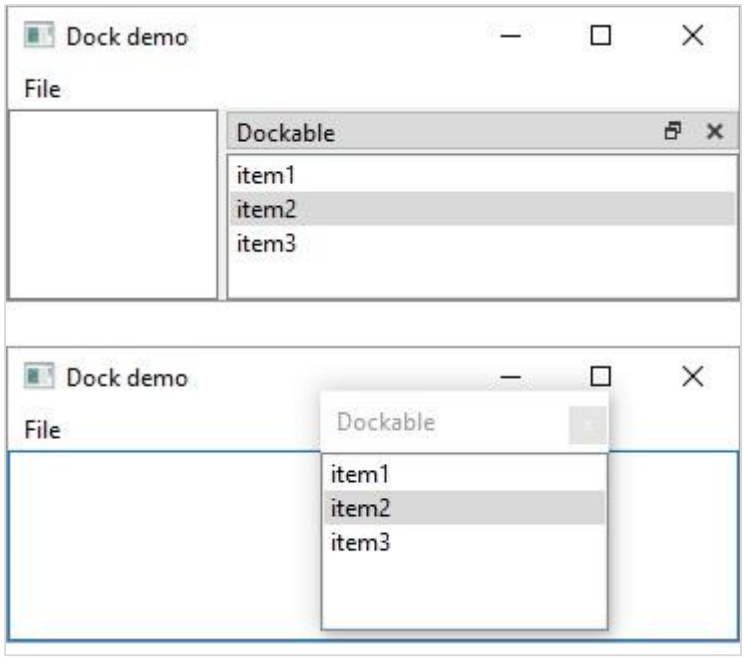
```
self.addDockWidget(Qt.RightDockWidgetArea, self.items)
```

The complete code is as follows:

```
import sys
from 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.

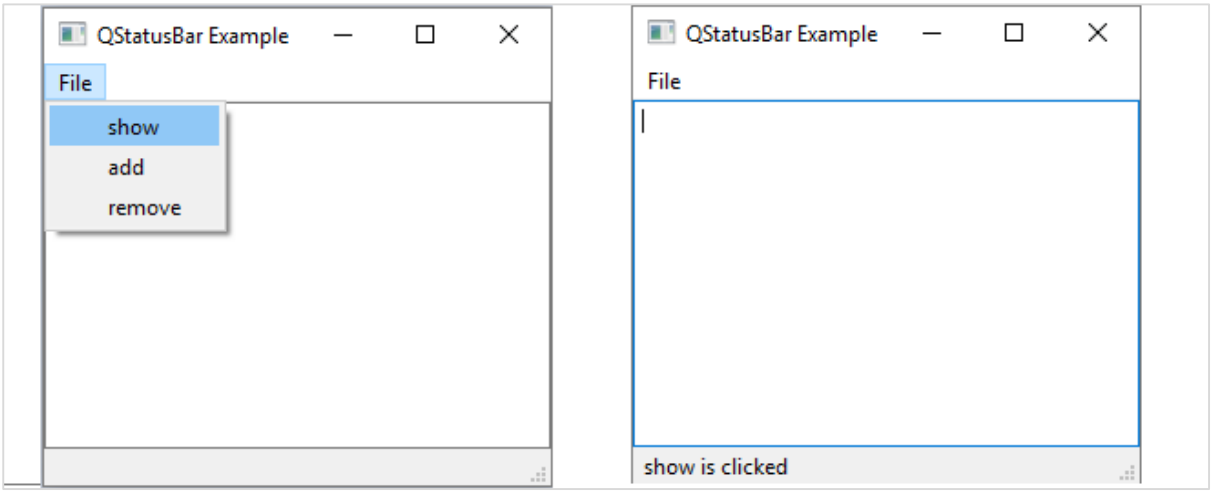
```
if q.text() == "remove":
   self.statusBar.removeWidget(self.b)
   self.statusBar.show()
```

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

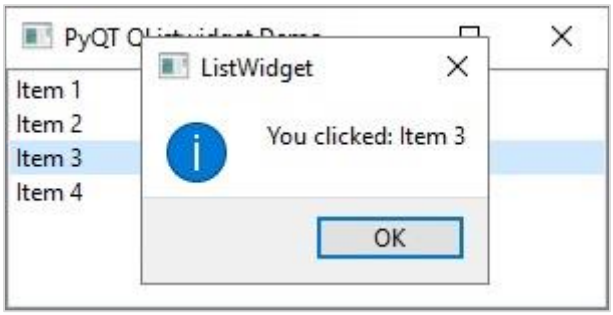
Example

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 Scroll Bar

c: Page control

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")
```