

## PyQt5 - Guide rapide

### PyQt5 - Présentation

PyQt est une boîte à outils de widgets GUI. Il s'agit d'une interface Python pour **Qt**, l'une des bibliothèques GUI multiplateformes les plus puissantes et les plus populaires. PyQt a été développé par RiverBank Computing Ltd. La dernière version de PyQt peut être téléchargée depuis son site officiel – [riverbankcomputing.com](http://riverbankcomputing.com)

L'API PyQt est un ensemble de modules contenant un grand nombre de classes et de fonctions. Alors que le module **QtCore** contient des fonctionnalités non graphiques pour travailler avec des fichiers et des répertoires, etc., le module **QtGui** contient tous les contrôles graphiques. De plus, il existe des modules pour travailler avec XML (**QtXml**) , SVG (**QtSvg**) et SQL (**QtSql**) , etc.

Une liste des modules fréquemment utilisés est donnée ci-dessous -

- **QtCore** - Classes principales non GUI utilisées par d'autres modules
- **QtGui** - Composants d'interface utilisateur graphique
- **QtMultimedia** - Classes pour la programmation multimédia de bas niveau
- **QtNetwork** – Classes pour la programmation réseau
- **QtOpenGL** - Classes de support OpenGL
- **QtScript** - Classes pour évaluer les scripts Qt
- **QtSql** - Classes pour l'intégration de bases de données à l'aide de SQL
- **QtSvg** - Classes pour afficher le contenu des fichiers SVG
- **QtWebKit** - Classes pour le rendu et l'édition HTML
- **QtXml** - Classes pour gérer XML
- **QtWidgets** - Classes pour créer des interfaces utilisateur classiques de style bureau
- **QtDesigner** - Classes pour étendre Qt Designer

### Environnements de support

PyQt est compatible avec tous les systèmes d'exploitation populaires, y compris Windows, Linux et Mac OS. Il s'agit d'une double licence, disponible sous licence GPL ainsi que sous licence commerciale. La dernière version stable est **PyQt5-5.13.2**.

### les fenêtres

Des roues pour l'architecture 32 bits ou 64 bits sont fournies qui sont compatibles avec Python version 3.5 ou ultérieure. La méthode d'installation recommandée consiste à utiliser l' utilitaire **PIP** –

```
pip3 install PyQt5
```

Pour installer des outils de développement tels que Qt Designer pour prendre en charge les roues PyQt5, voici la commande -

```
pip3 install pyqt5-tools
```

Vous pouvez également compiler PyQt5 sur Linux/macOS à partir du code source [www.riverbankcomputing.com/static/Downloads/PyQt5](http://www.riverbankcomputing.com/static/Downloads/PyQt5)

### PyQt5 - Quoi de neuf

L'API PyQt5 n'est pas automatiquement compatible avec les versions antérieures. Par conséquent, le code Python impliquant des modules PyQt4 doit être mis à jour manuellement en apportant les modifications pertinentes. Dans ce chapitre, les principales différences entre PyQt4 et PyQt5 ont été répertoriées.

PyQt5 n'est pas pris en charge sur les versions de Python antérieures à la v2.6.

PyQt5 ne prend pas en charge la méthode `connect()` de la classe `QObject` pour la connexion entre le signal et le slot. Par conséquent, l'utilisation ne peut plus être mise en œuvre -

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

Seule la syntaxe suivante est définie -

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

Les classes définies dans le module QtGui précédent ont été distribuées dans les modules **QtGui**, **QtPrintSupport** et **QtWidgets**.

Dans la nouvelle classe `QFileDialog`, la méthode `getOpenFileNameAndFilter()` est remplacée par `getOpenFileName()`, `getOpenFileNamesAndFilter()` par `getOpenFileNames()` et `getSaveFileNameAndFilter()` par `getSaveFileName()`. Les anciennes signatures de ces méthodes ont également changé.

PyQt5 n'a pas de disposition pour définir une classe qui est sous-classée à partir de plus d'une classe Qt.

**L'utilitaire pyuic5** (pour générer du code Python à partir du fichier XML de Designer) ne prend pas en charge l'indicateur `--pyqt3-wrapper`.

**pyrcc5** ne prend pas en charge les drapeaux `-py2` et `-py3`. La sortie de `pyrcc5` est compatible avec toutes les versions de Python v2.6 et suivantes.

PyQt5 invoque toujours automatiquement `sip.setdestroyonexit()` et appelle le destructeur C++ de toutes les instances enveloppées qu'il possède.

## PyQt5 - Bonjour le monde

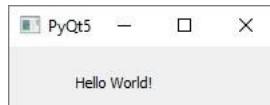
La création d'une application graphique simple à l'aide de PyQt implique les étapes suivantes :

- Importez les modules QtCore, QtGui et QtWidgets du package PyQt5.
- Créez un objet d'application de la classe QApplication.
- Un objet QWidget crée une fenêtre de niveau supérieur. Ajoutez-y un objet QLabel.
- Définissez la légende de l'étiquette sur "hello world".
- Définissez la taille et la position de la fenêtre par la méthode setGeometry().
- Entrez la boucle principale de l'application par la méthode `app.exec_()`.

Voici le code pour exécuter le programme Hello World dans 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()
```

Le code ci-dessus produit la sortie suivante -



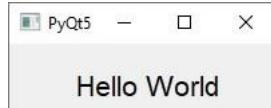
Il est également possible de développer une solution orientée objet du code ci-dessus.

- Importez les modules QtCore, QtGui et QtWidgets du package PyQt5.
- Créez un objet d'application de la classe QApplication.
- Déclarer la classe de fenêtre basée sur la classe QWidget
- Ajoutez un objet QLabel et définissez la légende de l'étiquette sur "hello world".
- Définissez la taille et la position de la fenêtre par la méthode setGeometry().
- Entrez la boucle principale de l'application par la méthode `app.exec_()`.

Voici le code complet de la solution orientée objet -

```
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 - Classes principales

L'API **PyQt** est une grande collection de classes et de méthodes. Ces classes sont définies dans plus de 20 modules.

Voici quelques-uns des modules fréquemment utilisés -

Sr.No.	Modules & Descriptif
1	<b>QtCoreComment</b> Classes principales non GUI utilisées par d'autres modules
2	<b>QtGuiName</b> Composants de l'interface utilisateur graphique
3	<b>QtMultimédia</b> Classes de programmation multimédia de bas niveau
4	<b>QtNetworkName</b> Cours de programmation réseau
5	<b>QtOpenGLName</b> Classes de support OpenGL
6	<b>QtScriptComment</b> Classes pour évaluer les scripts Qt
7	<b>QtSqlName</b> Classes pour l'intégration de bases de données à l'aide de SQL
8	<b>QtSvg</b> Classes pour afficher le contenu des fichiers SVG
9	<b>QtWebKitName</b> Classes pour le rendu et l'édition HTML
dix	<b>QtXmlName</b> Classes de gestion XML
11	<b>QtWidgets</b> Classes pour créer des interfaces utilisateur classiques de style bureau.
12	<b>QtDesignerName</b> Classes pour étendre Qt Designer
13	<b>Assistant Qt</b> Prise en charge de l'aide en ligne

Les outils de développement PyQt5 sont une collection d'utilitaires utiles pour le développement Qt. Voici une liste de sélection de ces utilitaires -

Sr.No.	Nom et description de l'outil
1	<b>assistant</b> Outil de documentation Qt Assistant
2	<b>pyqt5designer</b> Outil de mise en page de l'interface graphique de Qt Designer
3	<b>linguiste</b> Outil de traduction Qt Linguist
4	<b>lrelease</b> compiler les fichiers ts en fichiers qm
5	<b>pylupdate5</b> extraire les chaînes de traduction et générer ou mettre à jour les fichiers ts
6	<b>qmake</b> Outil de création de logiciel Qt
7	<b>pyqt5qmlscene</b> Visualiseur de fichiers QML
8	<b>pyqmlviewer</b> Visualiseur de fichiers QML
9	<b>pyrcc5</b> Compilateur de fichiers de ressources Qt
dix	<b>pyuic5</b> Compilateur d'interface utilisateur Qt pour générer du code à partir de fichiers ui
11	<b>pyqmlltestrunner</b> exécuter des tests unitaires sur du code QML
12	<b>qdbus</b> outil en ligne de commande pour lister les services D-Bus
13	<b>QDoc</b> générateur de documentation pour les projets logiciels.
14	<b>Qhelpgenerator</b> générer et afficher les fichiers d'aide de Qt.
15	<b>qmlimportscanner</b> analyse et rapporte les importations QML

L'API PyQt contient plus de 400 classes. La classe **QObject** est au sommet de la hiérarchie des classes. C'est la classe de base de tous les objets Qt. De plus, la classe **QPaintDevice** est la classe de base pour tous les objets pouvant être peints.

**La classe QApplication** gère les paramètres principaux et le flux de contrôle d'une application graphique. Il contient la boucle d'événements principale à l'intérieur de laquelle les événements générés par les éléments de fenêtre et d'autres sources sont traités et distribués. Il gère également les paramètres à l'échelle du système et de l'application.

**La classe QWidget**, dérivée des classes QObject et QPaintDevice, est la classe de base pour tous les objets de l'interface utilisateur. Les classes QDialog et QFrame sont également dérivées de la classe QWidget. Ils ont leur propre système de sous-classes.

Voici une liste de sélection des widgets fréquemment utilisés

Sr.No.	Widgets et description
1	<b>QLabel</b> Utilisé pour afficher du texte ou une image
2	<b>QLineModifier</b> Permet à l'utilisateur de saisir une ligne de texte
3	<b>QTextEdit</b> Permet à l'utilisateur de saisir du texte multiligne
4	<b>QPushButton</b> Un bouton de commande pour invoquer une action
5	<b>QRadioButton</b> Permet de choisir une parmi plusieurs options
6	<b>QCheckBox</b> Permet de choisir plusieurs options
7	<b>QSpinBox</b> Permet d'augmenter/diminuer une valeur entière
8	<b>QScrollBar</b> Permet d'accéder au contenu d'un widget au-delà de l'ouverture d'affichage
9	<b>QSlider</b> Permet de changer la valeur liée linéairement.
dix	<b>QComboBox</b> Fournit une liste déroulante d'éléments à sélectionner
11	<b>QMenuBar</b> Barre horizontale contenant des objets QMenu
12	<b>QStatusBar</b> Habituellement en bas de QMainWindow, fournit des informations d'état.
13	<b>QToolBarComment</b> Généralement en haut de QMainWindow ou flottant. Contient des boutons d'action
14	<b>QListView</b> Fournit une liste sélectionnable d'éléments dans ListMode ou IconMode
15	<b>QPixmap</b> Représentation d'image hors écran pour affichage sur un objet QLabel ou QPushButton
16	<b>QDialogName</b> Fenêtre modale ou non modale pouvant renvoyer des informations à la fenêtre mère

La fenêtre de niveau supérieur d'une application typique basée sur l'interface graphique est créée par l'objet widget **QMainWindow**. Certains widgets répertoriés ci-dessus prennent leur place dans cette fenêtre principale, tandis que d'autres sont placés dans la zone centrale des widgets à l'aide de divers gestionnaires de mise en page.

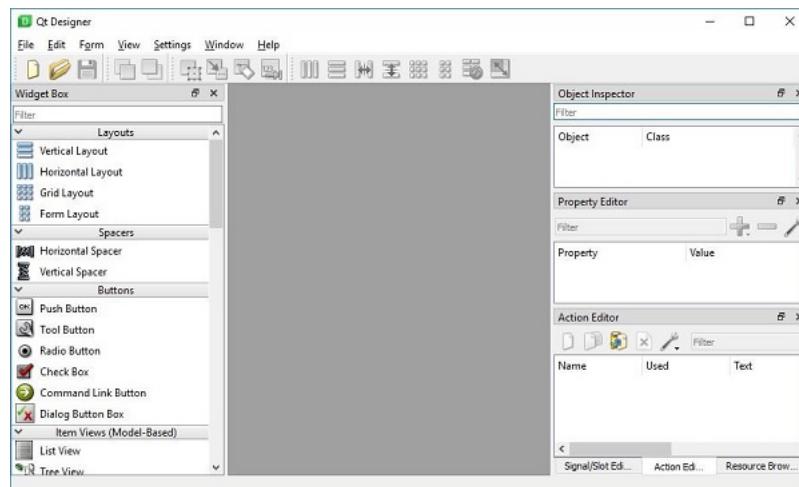
Le diagramme suivant montre le framework QMainWindow -



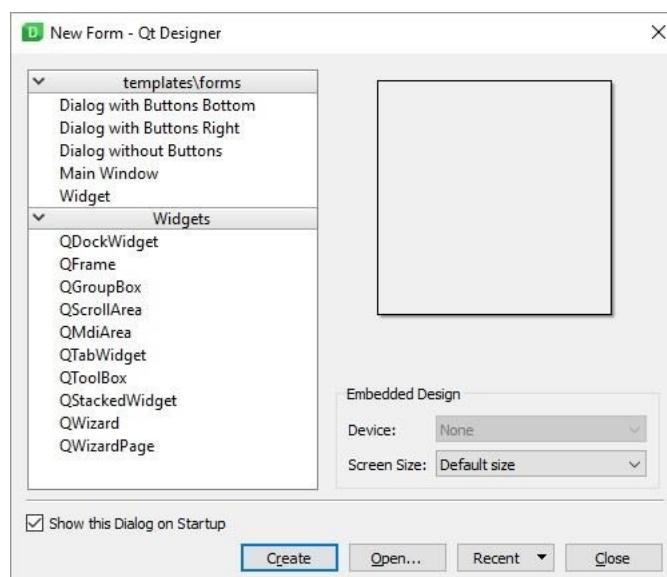
## PyQt5 - Utilisation de Qt Designer

Le programme d'installation de PyQt est fourni avec un outil de création d'interface graphique appelé **Qt Designer**. Grâce à sa simple interface glisser-déposer, une interface graphique peut être rapidement construite sans avoir à écrire le code. Ce n'est cependant pas un IDE tel que Visual Studio. Par conséquent, Qt Designer n'a pas la possibilité de déboguer et de construire l'application.

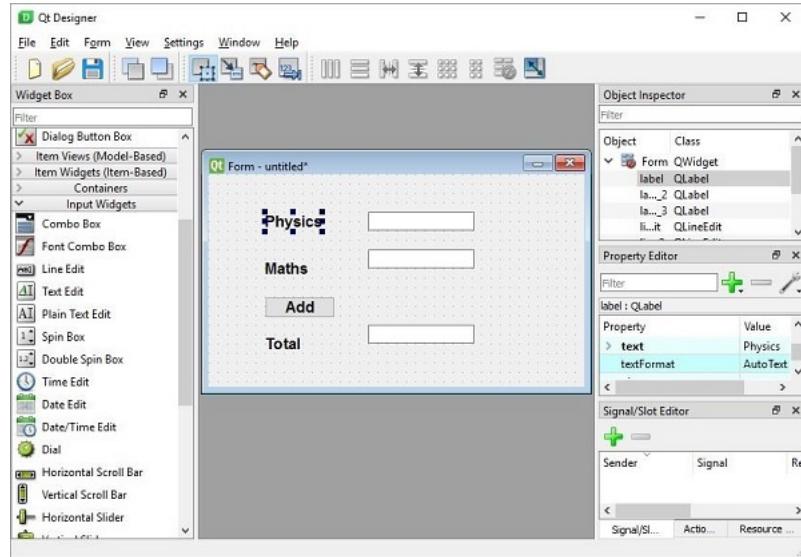
Démarrez l'application Qt Designer qui fait partie des outils de développement et est installée dans le dossier des scripts de l'environnement virtuel.



Commencez à concevoir l'interface graphique en choisissant Fichier → Nouveau menu.



Vous pouvez ensuite faire glisser et déposer les widgets requis à partir de la zone de widgets dans le volet de gauche. Vous pouvez également attribuer une valeur aux propriétés du widget posées sur le formulaire.



Le formulaire conçu est enregistré sous demo.ui. Ce fichier ui contient une représentation XML des widgets et de leurs propriétés dans la conception. Cette conception est traduite en équivalent Python à l'aide de l'utilitaire de ligne de commande pyuic5. Cet utilitaire est un wrapper pour le module uic de la boîte à outils Qt. L'utilisation de pyuic5 est la suivante -

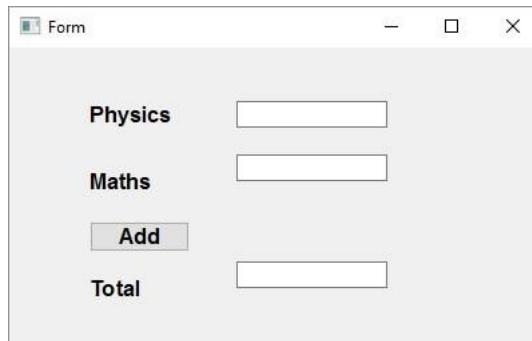
```
pyuic5 -x demo.ui -o demo.py
```

Dans la commande ci-dessus, le commutateur -x ajoute une petite quantité de code supplémentaire au script Python généré (à partir de XML) afin qu'il devienne une application autonome auto-exécutable.

```
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_())
```

Le script python résultant est exécuté pour afficher la boîte de dialogue suivante -

```
python demo.py
```



L'utilisateur peut saisir des données dans les champs de saisie, mais cliquer sur le bouton Ajouter ne générera aucune action car il n'est associé à aucune fonction. La réaction à la réponse générée par l'utilisateur est appelée **gestion d'événement**.

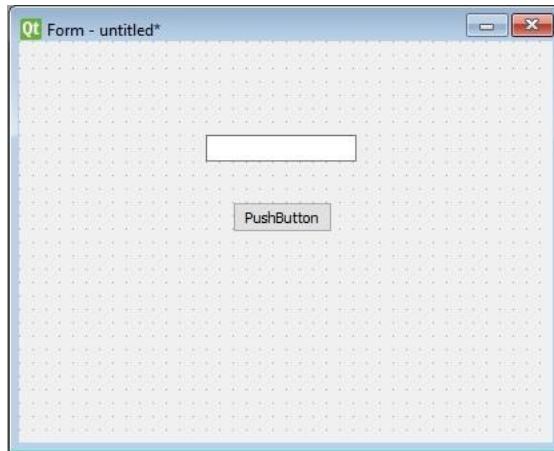
## PyQt5 - Signaux et emplacements

Contrairement à une application en mode console, qui est exécutée de manière séquentielle, une application basée sur une interface graphique est pilotée par des événements. Les fonctions ou les méthodes sont exécutées en réponse aux actions de l'utilisateur comme un clic sur un bouton, la sélection d'un élément dans une collection ou un clic de souris, etc., appelées **événements**.

Les widgets utilisés pour créer l'interface graphique agissent comme source de tels événements. Chaque widget PyQt, dérivé de la classe QObject, est conçu pour émettre un "signal" en réponse à un ou plusieurs événements. Le signal seul n'effectue aucune action. Au lieu de cela, il est 'connecté' à un 'slot'. L'emplacement peut être n'importe quelle **fonction Python appelleable**.

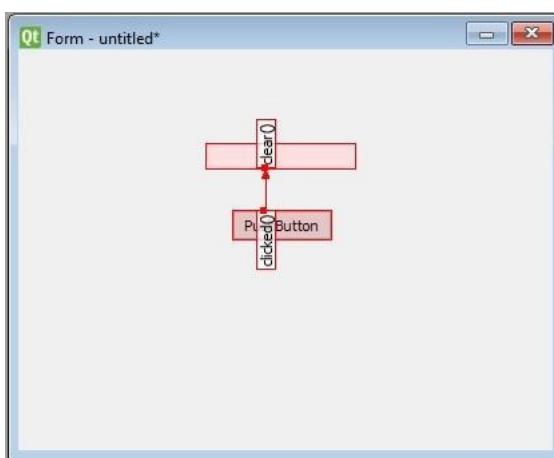
### Utilisation de l'éditeur de signal/slot de Qt Designer

Concevez d'abord un formulaire simple avec un contrôle LineEdit et un PushButton.

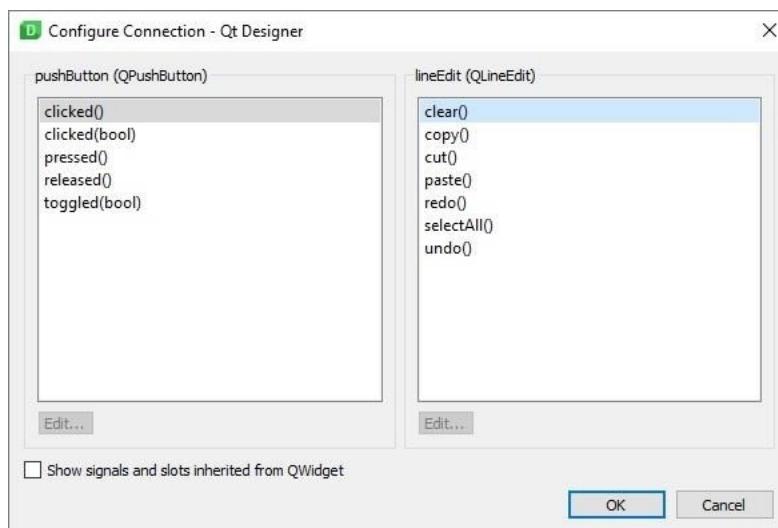


Il est souhaitable que si le bouton est enfoncé, le contenu de la zone de texte soit effacé. Le widget QLineEdit a une méthode clear() à cet effet. Par conséquent, le signal **cliqué** du bouton doit être connecté à la méthode **clear()** de la zone de texte.

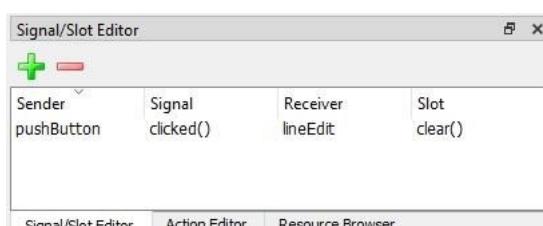
Pour commencer, choisissez Editer signaux/slots dans le menu Edition (ou appuyez sur F4). Sélectionnez ensuite le bouton avec la souris et faites glisser le curseur vers la zone de texte



Lorsque la souris est relâchée, une boîte de dialogue montrant les signaux du bouton et les méthodes de slot s'affiche. Sélectionnez le signal cliqué et la méthode clear()



La fenêtre Signal/Slot Editor en bas à droite affichera le résultat -



Enregistrez le code ui et Build et Python à partir du fichier ui comme indiqué dans le code ci-dessous -

```
pyuic5 -x signalslot.ui -o signalslot.py
```

Le code Python généré aura la connexion entre le signal et l'emplacement par l'instruction suivante -

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

Exécutez signauxlot.py et entrez du texte dans LineEdit. Le texte sera effacé si le bouton est enfoncé.

## Connexion signal-fente du bâtiment

Au lieu d'utiliser Designer, vous pouvez directement établir une connexion signal-slot en suivant la syntaxe -

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

Supposons qu'une fonction soit appelée lorsqu'un bouton est cliqué. Ici, le signal cliqué doit être connecté à une fonction appelleable. Il peut être réalisé dans l'une des techniques suivantes -

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

## Exemple

Dans l'exemple suivant, deux objets QPushButton (b1 et b2) sont ajoutés dans la fenêtre QDialog. Nous voulons appeler les fonctions b1\_clicked() et b2\_clicked() en cliquant respectivement sur b1 et b2.

Lorsque b1 est cliqué, le signal clicked() est connecté à la fonction b1\_clicked() -

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

Lorsque b2 est cliqué, le signal clicked() est connecté à la fonction b2\_clicked().

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

def window():
    app = QApplication(sys.argv)
    win = QDialog()
    b1 = QPushButton(win)
    b1.setText("Button1")
    b1.move(50,20)
    b1.clicked.connect(b1_clicked)

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

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

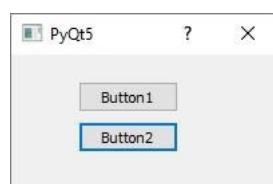
    win.setWindowTitle("PyQt5")
    win.show()
    sys.exit(app.exec_())

def b1_clicked():
    print ("Button 1 clicked")

def b2_clicked():
    print ("Button 2 clicked")

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

Le code ci-dessus produit la sortie suivante -



**Sortir**

```
Button 1 clicked
Button 2 clicked
```

## PyQt5 - Gestion de la mise en page

Un widget GUI peut être placé à l'intérieur de la fenêtre du conteneur en spécifiant ses coordonnées absolues mesurées en pixels. Les coordonnées sont relatives aux dimensions de la fenêtre définies par la méthode `setGeometry()`.

### Syntaxe `setGeometry()`

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

Dans l'extrait de code suivant, la fenêtre de niveau supérieur de dimensions 300 par 200 pixels est affichée à la position (10, 10) sur le moniteur.

```
import sys
from PyQt4 import QtGui

def window():
    app = QtGui.QApplication(sys.argv)
    w = QtGui.QWidget()

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

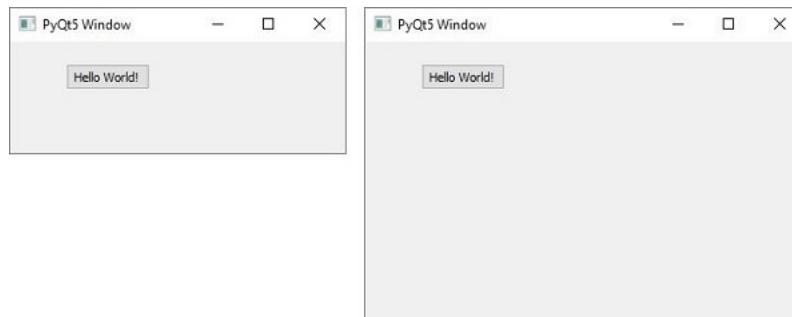
    w.setGeometry(10,10,300,200)
    w.setWindowTitle("PyQt")
    w.show()
    sys.exit(app.exec_())

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

Un widget **PushButton** est ajouté dans la fenêtre et placé à une position 50 pixels vers la droite et 20 pixels en dessous de la position en haut à gauche de la fenêtre.

Ce positionnement absolu, cependant, n'est pas approprié pour les raisons suivantes -

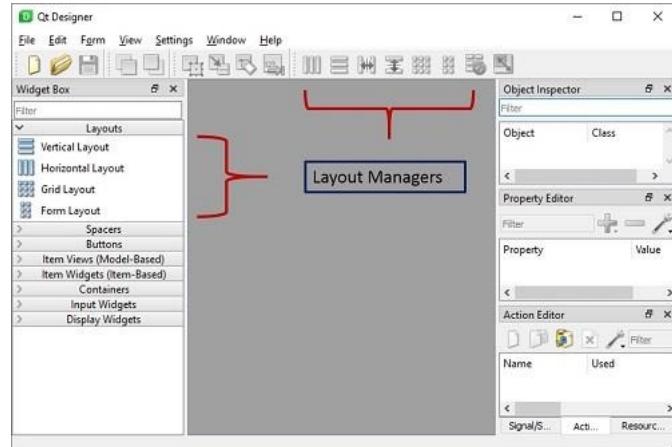
- La position du widget ne change pas même si la fenêtre est redimensionnée.
- L'apparence peut ne pas être uniforme sur différents appareils d'affichage avec différentes résolutions.
- La modification de la mise en page est difficile car il peut être nécessaire de reconcevoir l'ensemble du formulaire.



L'API PyQt fournit des classes de mise en page pour une gestion plus élégante du positionnement des widgets à l'intérieur du conteneur. Les avantages des gestionnaires de mise en page par rapport au positionnement absolu sont :

- Les widgets à l'intérieur de la fenêtre sont automatiquement redimensionnés.
- Assure une apparence uniforme sur les appareils d'affichage avec différentes résolutions.
- L'ajout ou la suppression dynamique d'un widget est possible sans avoir à reconcevoir.

La boîte à outils Qt définit diverses dispositions pouvant être utilisées avec l'utilitaire Qt Designer.



Voici la liste des classes dont nous parlerons une par une dans ce chapitre.

Sr.No.	Cours et description
1	<b>QBoxLayout</b> La classe QBoxLayout aligne les widgets verticalement ou horizontalement. Ses classes dérivées sont QVBoxLayout (pour organiser les widgets verticalement) et QHBoxLayout (pour organiser les widgets horizontalement).
2	<b>QGridLayout</b> Un objet de classe GridLayout présente une grille de cellules disposées en lignes et en colonnes. La classe contient la méthode addWidget(). Tout widget peut être ajouté en spécifiant le nombre de lignes et de colonnes de la cellule.
3	<b>QFormLayout</b> QFormLayout est un moyen pratique de créer un formulaire à deux colonnes, où chaque ligne est constituée d'un champ de saisie associé à une étiquette. Par convention, la colonne de gauche contient l'étiquette et la colonne de droite contient un champ de saisie.

## PyQt5 - Widgets de base

Voici la liste des Widgets dont nous parlerons un par un dans ce chapitre.

Sr. Non	Widgets et description
1	QLabel  Un objet QLabel agit comme un espace réservé pour afficher du texte ou une image non modifiable, ou un film de GIF animé. Il peut également être utilisé comme clé mnémonique pour d'autres widgets.
2	QLineModifier  L'objet QLineEdit est le champ de saisie le plus couramment utilisé. Il fournit une zone dans laquelle une ligne de texte peut être saisie. Pour saisir du texte multiligne, l'objet QTextEdit est requis.
3	QPushButton  Dans l'API PyQt, l'objet de classe QPushButton présente un bouton qui, lorsqu'il est cliqué, peut être programmé pour invoquer une certaine fonction.
4	QRadioButton  Un objet de classe QRadioButton présente un bouton sélectionnable avec une étiquette de texte. L'utilisateur peut sélectionner l'une des nombreuses options présentées sur le formulaire. Cette classe est dérivée de la classe QAbstractButton.
5	QCheckBox  Une boîte rectangulaire devant l'étiquette de texte apparaît lorsqu'un objet QCheckBox est ajouté à la fenêtre parent. Tout comme QRadioButton, c'est aussi un bouton sélectionnable.
6	QComboBox  Un objet QComboBox présente une liste déroulante d'éléments à sélectionner. Il faut un minimum d'espace à l'écran sur le formulaire requis pour afficher uniquement l'élément actuellement sélectionné.
7	QSpinBox  Un objet QSpinBox présente à l'utilisateur une zone de texte qui affiche un entier avec un bouton haut/bas à sa droite.
8	Widget et signal QSlider  L'objet de classe QSlider présente à l'utilisateur une rainure sur laquelle une poignée peut être déplacée. C'est un widget classique pour contrôler une valeur bornée.
9	QMenuBar, QMenu et QAction  Un QMenuBar horizontal juste en dessous de la barre de titre d'un objet QMainWindow est réservé à l'affichage des objets QMenu.
dix	QToolBarComment  Un widget QToolBar est un panneau mobile composé de boutons de texte, de boutons avec des icônes ou d'autres widgets.
11	QInputDialog  Il s'agit d'une boîte de dialogue préconfigurée avec un champ de texte et deux boutons, OK et Annuler. La fenêtre parent recueille l'entrée dans la zone de texte après que l'utilisateur a cliqué sur le bouton OK ou appuyé sur Entrée.
12	QFontDialog  Une autre boîte de dialogue couramment utilisée, un widget de sélection de polices, est l'apparence visuelle de la classe QDialog. Le résultat de cette boîte de dialogue est un objet QFont, qui peut être consommé par la fenêtre parent.
13	QFileDialog  Ce widget est une boîte de dialogue de sélection de fichiers. Il permet à l'utilisateur de naviguer dans le système de fichiers et de sélectionner un fichier à ouvrir ou à enregistrer. La boîte de dialogue est invoquée soit via des fonctions statiques, soit en appelant la fonction exec_() sur l'objet de la boîte de dialogue.
14	QTab  Si un formulaire comporte trop de champs pour être affichés simultanément, ils peuvent être organisés en différentes pages placées sous chaque onglet d'un widget à onglets. Le QTabWidget fournit une barre d'onglets et une zone de page.
15	QStacked  Le fonctionnement de QStackedWidget est similaire à QTabWidget. Il contribue également à l'utilisation efficace de la zone client de Windows.

16	QSplitter	Il s'agit d'un autre gestionnaire de disposition avancé qui permet de modifier dynamiquement la taille des widgets enfants en faisant glisser les limites entre eux. Le contrôle Splitter fournit une poignée qui peut être déplacée pour redimensionner les contrôles.
17	QDockComment	Une fenêtre ancrable est une sous-fenêtre qui peut rester à l'état flottant ou peut être attachée à la fenêtre principale à une position spécifiée. L'objet fenêtre principale de la classe QMainWindow a une zone réservée aux fenêtres ancrables.
18	QStatusBar	L'objet QMainWindow réserve une barre horizontale en bas comme barre d'état. Il est utilisé pour afficher des informations d'état permanentes ou contextuelles.
19	QListe	La classe QListWidget est une interface basée sur des éléments pour ajouter ou supprimer des éléments d'une liste. Chaque élément de la liste est un objet QListWidgetItem. QListWidget peut être défini pour être multisélectionnable.
20	QScrollBar	Un contrôle de barre de défilement permet à l'utilisateur d'accéder à des parties du document situées en dehors de la zone visible. Il fournit un indicateur visuel de la position actuelle.
21	QCalendrier	Le widget QCalendar est un contrôle de sélection de date utile. Il fournit une vue mensuelle. L'utilisateur peut sélectionner la date à l'aide de la souris ou du clavier, la date par défaut étant la date du jour.

## PyQt5 - Classe QDialog

Un widget **QDialog** présente une fenêtre de niveau supérieur principalement utilisée pour collecter la réponse de l'utilisateur. Il peut être configuré pour être **modal** (où il bloque sa fenêtre parente) ou **non modal** (la fenêtre de dialogue peut être contournée).

L'API PyQt possède un certain nombre de widgets Dialog préconfigurés tels que InputDialog, FileDialog, FontDialog, etc.

### Exemple

Dans l'exemple suivant, l'attribut **WindowModality** de la fenêtre Dialog décide s'il est modal ou non modal. N'importe quel bouton de la boîte de dialogue peut être défini par défaut. La boîte de dialogue est rejetée par la méthode **QDialog.reject()** lorsque l'utilisateur appuie sur la touche Échap.

Un bouton poussoir sur une fenêtre QWidget de niveau supérieur, lorsqu'il est cliqué, produit une fenêtre de dialogue. Une boîte de dialogue n'a pas de contrôles de réduction et d'agrandissement sur sa barre de titre.

L'utilisateur ne peut pas reléguer cette boîte de dialogue en arrière-plan car son WindowModality est défini sur **ApplicationModal**.

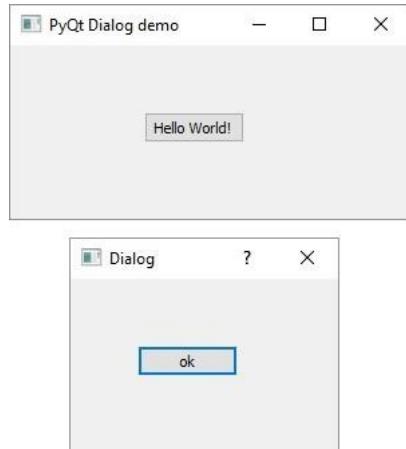
```
import sys
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

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") 9. PyQt5 – QDialog Class
    dlg.setWindowModality(Qt.ApplicationModal)
    dlg.exec_()
```

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

Le code ci-dessus produit la sortie suivante. Cliquez sur le bouton dans la fenêtre principale et la boîte de dialogue apparaît –



### PyQt5 - QMessageBox

**QMessageBox** est une boîte de dialogue modale couramment utilisée pour afficher un message d'information et éventuellement demander à l'utilisateur de répondre en cliquant sur l'un des boutons standard dessus. Chaque bouton standard a une légende prédéfinie, un rôle et renvoie un nombre hexadécimal prédéfini.

Les méthodes et énumérations importantes associées à la classe QMessageBox sont données dans le tableau suivant -

Sr.No.	Méthodes et description
1	<b>setIcon()</b> Affiche une icône prédéfinie correspondant à la gravité du message <ul style="list-style-type: none"> <li>• Question</li> <li>• Information</li> <li>• Avertissement</li> <li>• Critique</li> </ul>
2	<b>Définir le texte()</b> Définit le texte du message principal à afficher
3	<b>setInformativeText()</b> Affiche des informations supplémentaires
4	<b>setDetailText()</b> La boîte de dialogue affiche un bouton Détails. Ce texte apparaît en cliquant dessus
5	<b>définirTitre()</b> Affiche le titre personnalisé de la boîte de dialogue
6	<b>setStandardButtons()</b> Liste des boutons standard à afficher. Chaque bouton est associé à QMessageBox.Ok 0x00000400 QMessageBox.Ouvrir 0x00002000 QMessageBox.Save 0x00000800 QMessageBox.Annuler 0x00400000 QMessageBox.Fermer 0x00200000 QMessageBox.Oui 0x00004000 QMessageBox.No 0x00010000 QMessageBox.Abort 0x00040000 QMessageBox.Réessayer 0x00080000 QMessageBox.Ignorer 0x00100000
7	<b>setDefaultButton()</b> Définit le bouton par défaut. Il émet le signal cliqué si Enter est pressé
8	<b>setEscapeButton()</b> Définit le bouton pour qu'il soit traité comme cliqué si la touche d'échappement est enfoncee

## Exemple

Dans l'exemple suivant, cliquez sur le signal du bouton dans la fenêtre de niveau supérieur, la fonction connectée affiche la boîte de dialogue de la boîte de message.

```
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:")
```

La fonction setStandardButton() affiche les boutons souhaités.

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

Le signal buttonClicked() est connecté à une fonction slot, qui identifie la légende de la source du signal.

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

Le code complet de l'exemple est le suivant -

```
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)

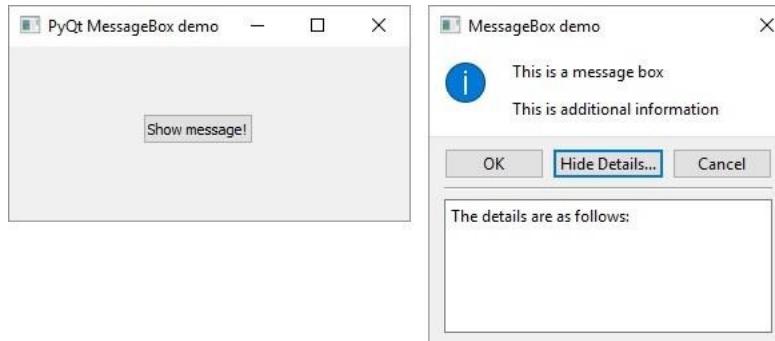
    msg.setText("This is a message box")
    msg.setInformativeText("This is additional information")
    msg.setWindowTitle("MessageBox demo")
    msg.setDetailedText("The details are as follows:")
    msg.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)
    msg.buttonClicked.connect(msgbtn)

    retval = msg.exec_()

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

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

Le code ci-dessus produit la sortie suivante. La boîte de message apparaît lorsque le bouton des fenêtres principales est cliqué -



Si vous cliquez sur le bouton OK ou Annuler sur MessageBox, la sortie suivante est produite sur la console -

```
Button pressed is: OK
Button pressed is: Cancel
```

## PyQt5 - Interface de documents multiples

Une application GUI typique peut avoir plusieurs fenêtres. Les widgets à onglets et empilés permettent d'activer une telle fenêtre à la fois. Cependant, cette approche peut souvent ne pas être utile car la vue des autres fenêtres est masquée.

Une façon d'afficher plusieurs fenêtres simultanément consiste à les créer en tant que fenêtres indépendantes. C'est ce qu'on appelle SDI (**interface de document unique**). Cela nécessite plus de ressources mémoire car chaque fenêtre peut avoir son propre système de menus, sa propre barre d'outils, etc.

Les applications MDI (**Multiple Document Interface**) consomment moins de ressources mémoire. Les sous-fenêtres sont disposées à l'intérieur du conteneur principal les unes par rapport aux autres. Le widget conteneur s'appelle **QMdiArea** .

Le widget QMdiArea occupe généralement le widget central de l'objet QMainWondow. Les fenêtres enfants de cette zone sont des instances de la classe **QMdiSubWindow** . Il est possible de définir n'importe quel QWidget comme widget interne de l'objet subWindow. Les sous-fenêtres de la zone MDI peuvent être disposées en cascade ou en mosaïque.

Le tableau suivant répertorie les méthodes importantes de la classe QMdiArea et de la classe QMdiSubWindow -

Sr.No.	Méthodes et description
1	<b>addSubWindow()</b> Ajoute un widget en tant que nouvelle sous-fenêtre dans la zone MDI
2	<b>removeSubWindow()</b> Supprime un widget qui est un widget interne d'une sous-fenêtre
3	<b>setActiveSubWindow()</b> Active une sous-fenêtre
4	<b>cascadeSousFenêtres()</b> Organise les sous-fenêtres dans MDiArea en cascade
5	<b>tileSubWindows()</b> Organise les sous-fenêtres dans MDiArea en mosaïque
6	<b>closeActiveSubWindow()</b> Ferme la sous-fenêtre active
7	<b>subWindowList()</b> Renvoie la liste des sous-fenêtres dans la zone MDI
8	<b>setWidget()</b> Définit un QWidget comme un widget interne d'une instance de QMdiSubwindow

L'objet QMdiArea émet le signal subWindowActivated() alors que le signal windowStateChanged() est émis par l'objet QMdisubWindow.

## Exemple

Dans l'exemple suivant, la fenêtre de niveau supérieur comprenant QMainWindow a un menu et MdiArea.

```
self.mdi = QMdiArea()
self.setCentralWidget(self.mdi)
bar = self.menuBar()
file = bar.addMenu("File")

file.addAction("New")
file.addAction("cascade")
file.addAction("Tiled")
```

Le signal Triggered() du menu est connecté à la fonction windowaction().

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

La nouvelle action du menu ajoute une sous-fenêtre dans la zone MDI avec un titre ayant un numéro incrémentiel.

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

Les boutons en cascade et en mosaïque du menu organisent les sous-fenêtres actuellement affichées respectivement en cascade et en mosaïque.

Le code complet est le suivant -

```

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()

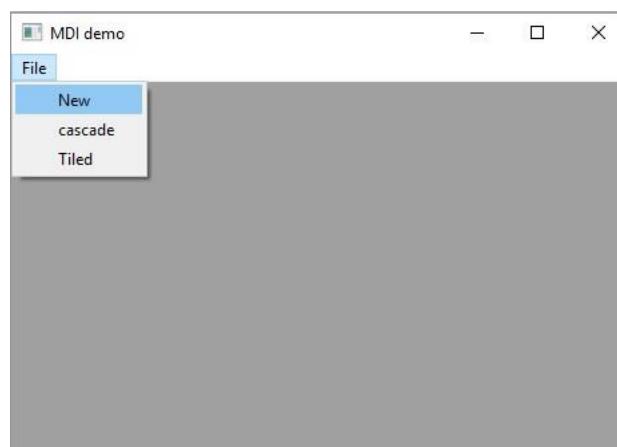
        if q.text() == "Tiled":
            self.mdi.tileSubWindows()

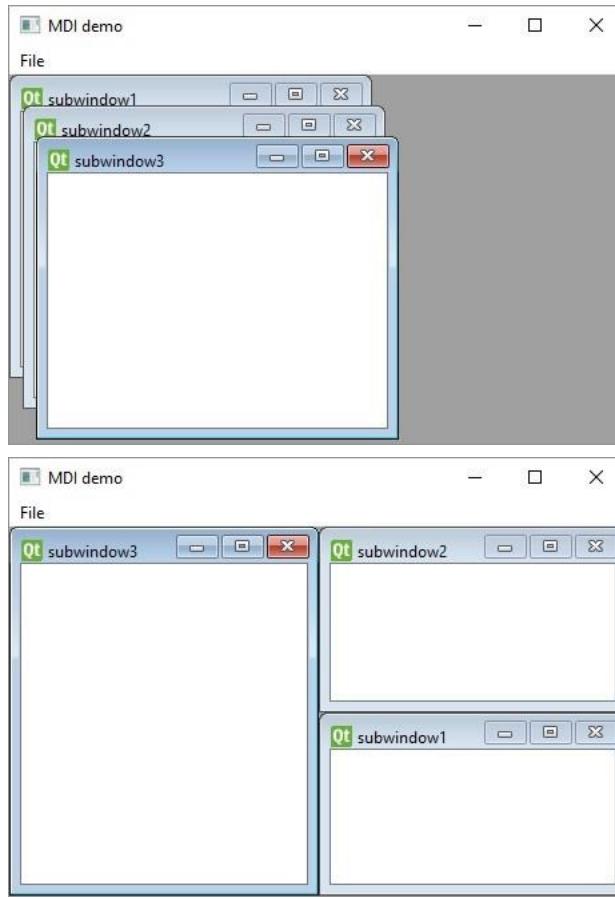
    def main():
        app = QApplication(sys.argv)
        ex = MainWindow()
        ex.show()
        sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

Exécuter au-dessus du code et trois fenêtres en cascade et en mosaïque -





## PyQt5 - Glisser-déposer

La mise à disposition du **glisser-déposer** est très intuitive pour l'utilisateur. On le trouve dans de nombreuses applications de bureau où l'utilisateur peut copier ou déplacer des objets d'une fenêtre à une autre.

Le transfert de données par glisser-déposer basé sur MIME est basé sur la classe **QDrag**. Les objets **QMimeData** associent les données à leur type MIME correspondant. Il est stocké dans le presse-papiers puis utilisé dans le processus de glisser-déposer.

Les fonctions de classe **QMimeData** suivantes permettent de détecter et d'utiliser facilement le type MIME.

Testeur	Getter	Setter	Type MIME
aTexte()	texte()	Définir le texte()	texte simple
hasHtml()	html()	setHtml()	texte/html
aUrls()	URL()	setUrls()	texte/uri-liste
alimage()	imageData()	setImageData()	image/ *
aCouleur()	colorData()	setColorData()	application/x-color

De nombreux objets **QWidget** prennent en charge l'activité de glisser-déposer. Ceux qui permettent de faire glisser leurs données ont **setDragEnabled()** qui doit être défini sur **true**. D'autre part, les widgets doivent répondre aux événements de glisser-déposer afin de stocker les données qui y sont glissées.

- **DragEnterEvent** fournit un événement qui est envoyé au widget cible lorsque l'action de glissement y entre.
- **DragMoveEvent** est utilisé lorsque l'action glisser-déposer est en cours.
- **DragLeaveEvent** est généré lorsque l'action glisser-déposer quitte le widget.
- **DropEvent**, d'autre part, se produit lorsque la suppression est terminée. L'action proposée pour l'événement peut être acceptée ou rejetée conditionnellement.

### Exemple

Dans le code suivant, **DragEnterEvent** vérifie si les données MIME de l'événement contiennent du texte. Si oui, l'action proposée de l'événement est acceptée et le texte est ajouté en tant que nouvel élément dans le **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"))

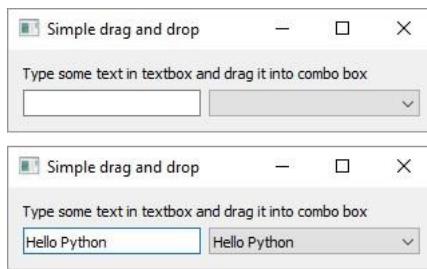
        edit = QLineEdit()
        edit.setDragEnabled(True)
        com = combo("Button", self)
        lo.addRow(edit, com)
        self.setLayout(lo)
        self.setWindowTitle('Simple drag and drop')

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

if __name__ == '__main__':
    main()

```

Le code ci-dessus produit la sortie suivante -



## PyQt5 - Gestion de la base de données

La bibliothèque PyQt5 contient le module **QtSql**. C'est un système de classe élaboré pour communiquer avec de nombreuses bases de données basées sur SQL. Sa **QSqlDatabase** fournit un accès via un objet Connection. Voici la liste des pilotes SQL actuellement disponibles -

Sr.No.	Type de pilote et description
1	<b>QDB2</b> IBMDB2
2	<b>QIBASE</b> Pilote Borland InterBase
3	<b>QMYSQL</b> Pilote MySQL
4	<b>QOCI</b> Pilote d'interface d'appel Oracle
5	<b>QODBC</b> Pilote ODBC (inclus Microsoft SQL Server)
6	<b>QPSQLComment</b> Pilote PostgreSQL
7	<b>QSQLITEName</b> SQLite version 3 ou supérieure
8	<b>QSQLITE2</b> SQLite version 2

### Exemple

Pour ce chapitre, une connexion avec une base de données SQLite est établie en utilisant la méthode statique -

```
db = QSqlDatabase::addDatabase("QSQLITE")
db.setDatabaseName("sports.db")
```

Les autres méthodes de la classe QSqlDatabase sont les suivantes -

Sr.No.	Méthodes et description
1	<b>setDatabaseName()</b> Définit le nom de la base de données avec laquelle la connexion est recherchée
2	<b>setNomHôte()</b> Définit le nom de l'hôte sur lequel la base de données est installée
3	<b>setUserName()</b> Spécifie le nom d'utilisateur pour la connexion
4	<b>setPassword()</b> Définit le mot de passe de l'objet de connexion, le cas échéant
5	<b>s'engager()</b> Valide les transactions et renvoie true en cas de succès
6	<b>retour en arrière()</b> Annule la transaction de base de données
7	<b>Fermer()</b> Ferme la connexion

La classe **QSqlQuery** a la fonctionnalité d'exécuter et de manipuler des commandes SQL. Les requêtes SQL de type DDL et DML peuvent être exécutées. La première étape consiste à créer une base de données SQLite en utilisant les instructions suivantes -

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

Ensuite, obtenez l'objet Query avec la méthode **QSqlQuery()** et appelez sa méthode la plus importante **exec\_()**, qui prend comme argument une chaîne contenant l'instruction SQL à exécuter.

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

Le script suivant crée une base de données SQLite sports.db avec une table de sportsperson remplie de cinq enregistrements.

```
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')")
```

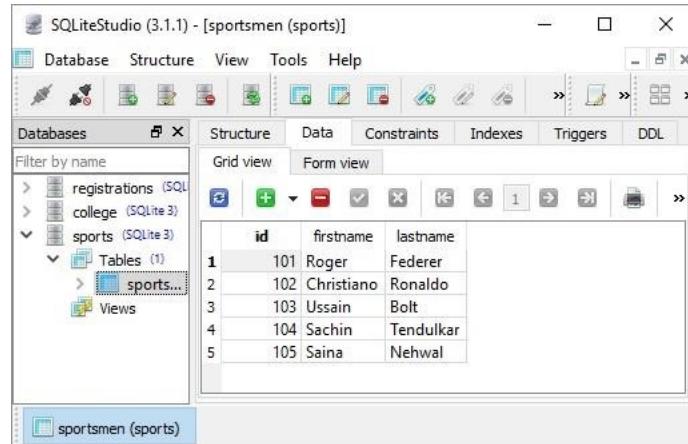
```

    return True

if __name__ == '__main__':
    app = QApplication(sys.argv)
    createDB()

```

Pour confirmer que la base de données SQLite est créée avec les enregistrements ci-dessus ajoutés dans la table des sportifs, utilisez un utilitaire SQLite Gui appelé **SQLiteStudio** .



La classe **QSqlTableModel** dans PyQt est une interface de haut niveau qui fournit un modèle de données modifiable pour lire et écrire des enregistrements dans une seule table. Ce modèle est utilisé pour remplir un objet **QTableView** . Il présente à l'utilisateur une vue déroulante et modifiable qui peut être placée sur n'importe quelle fenêtre de niveau supérieur.

Un objet QSqlTableModel est déclaré de la manière suivante -

```
model = QSqlTableModel()
```

Sa stratégie d'édition peut être définie sur l'une des valeurs suivantes -

<b>QSqlTableModel.OnFieldChange</b>	Toutes les modifications seront appliquées immédiatement
<b>QSqlTableModel.OnRowChange</b>	Les modifications seront appliquées lorsque l'utilisateur sélectionnera une ligne différente
<b>QSqlTableModel.OnManualSubmit</b>	Toutes les modifications seront mises en cache jusqu'à ce que submitAll() ou revertAll() soit appelé

## Exemple

Dans l'exemple suivant, la table des sportifs est utilisée comme modèle et la stratégie est définie comme -

```

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

```

La classe QTableView fait partie du framework Model/View dans PyQt. L'objet QTableView est créé comme suit -

```

view = QtGui.QTableView()
view.setModel(model)
view.setWindowTitle(title)
return view

```

Cet objet QTableView et deux widgets QPushButton sont ajoutés à la fenêtre QDialog de niveau supérieur. Le signal clicked() du bouton d'ajout est connecté à addrow() qui exécute insertRow() sur la table modèle.

```

button.clicked.connect(addrow)
def addrow():
    print model.rowCount()
    ret = model.insertRows(model.rowCount(), 1)
    print ret

```

Le Slot associé au bouton de suppression exécute une fonction lambda qui supprime une ligne, qui est sélectionnée par l'utilisateur.

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

Le code complet est le suivant -

```
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)

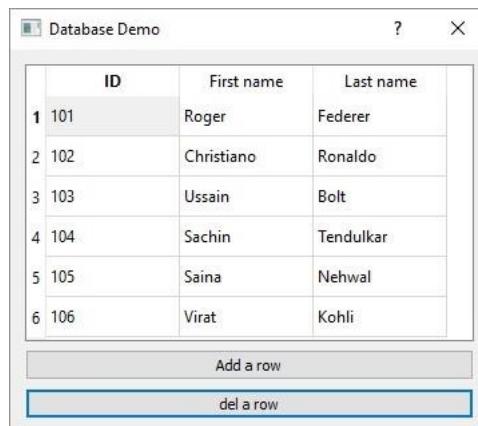
    button = QPushButton("Add a row")
    button.clicked.connect(addrow)
    layout.addWidget(button)

    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_())

```

Le code ci-dessus produit la sortie suivante -



Essayez d'ajouter et de supprimer quelques enregistrements et revenez à SQLiteStudio pour confirmer les transactions.

## PyQt5 - API de dessin

Toutes les classes **QWidget** dans PyQt sont des sous-classes de la classe **QPaintDevice**. Un **QPaintDevice** est une abstraction d'un espace bidimensionnel qui peut être dessiné à l'aide d'un **Painter**. Les dimensions du dispositif de peinture sont mesurées en pixels à partir du coin supérieur gauche.

**La classe QPainter** effectue une peinture de bas niveau sur des widgets et d'autres périphériques pouvant être peints tels qu'une imprimante. Normalement, il est utilisé dans l'événement de peinture du widget. Le **QPaintEvent** se produit chaque fois que l'apparence du widget est mise à jour.

Le peintre est activé en appelant la méthode **begin()**, tandis que la méthode **end()** le désactive. Entre les deux, le motif souhaité est peint par des procédés appropriés tels qu'énumérés dans le tableau suivant.

Sr.No.	Méthodes et description
1	<b>commencer()</b> Commence à peindre sur l'appareil cible
2	<b>dessinArc()</b> Dessine un arc entre l'angle de départ et l'angle de fin
3	<b>dessinerEllipse()</b> Dessine une ellipse à l'intérieur d'un rectangle
4	<b>dessiner une ligne()</b> Dessine une ligne avec les coordonnées du point final spécifiées
5	<b>drawPixmap()</b> Extrait le pixmap du fichier image et l'affiche à la position spécifiée
6	<b>drwaPolygon()</b> Dessine un polygone à l'aide d'un tableau de coordonnées
7	<b>drawRect()</b> Dessine un rectangle commençant à la coordonnée en haut à gauche avec la largeur et la hauteur données
8	<b>dessinerTexte()</b> Affiche le texte aux coordonnées données
9	<b>fillRect()</b> Remplit le rectangle avec le paramètre <b>QColor</b>
dix	<b>setBrush()</b> Définit un style de pinceau pour la peinture
11	<b>setPen()</b> Définit la couleur, la taille et le style du stylet à utiliser pour le dessin

### Exemple

Dans le code suivant, différentes méthodes de dessin de PyQt sont utilisées.

```
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()

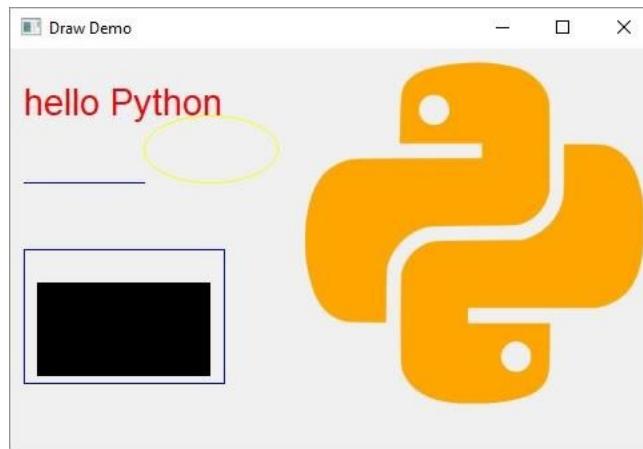
    def paintEvent(self, event):
        qp = QPainter()
        qp.begin(self)
        qp.setPen(QColor(Qt.red))
        qp.setFont(QFont('Arial', 20))
        qp.drawText(10,50, "hello Python")
        qp.setPen(QColor(Qt.blue))
        qp.drawLine(10,100,100,100)
        qp.drawRect(10,150,150,100)
        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()

```

Le code ci-dessus produit la sortie suivante -



PyQt5 - Constantes BrushStyle

Dans ce chapitre, nous apprendrons les constantes de style de pinceau.

### Constantes de style de pinceau

Ci-dessous sont les constantes de style de pinceau -

Qt.NoBrush	Pas de motif de brosse
Qt.SolidPattern	Couleur uniforme
Qt.Dense1Pattern	Motif de brosse extrêmement dense
Qt.HorPattern	Lignes horizontales
Qt.VerPattern	Lignes verticales
Qt.CrossPatternQt.CrossPattern	Croisement des lignes horizontales et verticales
Qt.BDiagPattern	Lignes diagonales vers l'arrière
Qt.FDiagPattern	Lignes diagonales vers l'avant
Qt.DiagCrossPatternQt.DiagCrossPattern	Traverser des lignes diagonales

### Styles QColor prédéfinis

Ci-dessous sont les styles QColor prédéfinis -

Qt.NoBrush	Pas de motif de brosse
Qt.SolidPattern	Couleur uniforme
Qt.Dense1Pattern	Motif de brosse extrêmement dense
Qt.HorPattern	Lignes horizontales
Qt.VerPattern	Lignes verticales
Qt.CrossPatternQt.CrossPattern	Croisement des lignes horizontales et verticales
Qt.BDiagPattern	Lignes diagonales vers l'arrière
Qt.FDiagPattern	Lignes diagonales vers l'avant
Qt.DiagCrossPatternQt.DiagCrossPattern	Traverser des lignes diagonales

### Objets QColor prédéfinis

Ci-dessous sont les objets QColor prédéfinis -

Qt.blanc
Qt.noir
Qt.rouge
Qt.darkRed
Qt.vert
Qt. vert foncé
Qt.bleu
Qt.cyan
Qt.magenta
Qt.jaune
Qt.jaunefoncé
Qt.gris

## PyQt5 - QClipboard

La classe **QClipboard** donne accès au presse-papiers à l'échelle du système qui offre un mécanisme simple pour copier et coller des données entre les applications. Son action est similaire à la classe **QDrag** et utilise des types de données similaires.

La classe QApplication a une méthode statique **clipboard()** qui renvoie la référence à l'objet presse-papiers. Tout type de MimeData peut être copié ou collé depuis le presse-papiers.

Voici les méthodes de classe de presse-papiers qui sont couramment utilisées -

Sr.No.	Méthodes et description
1	<b>dégager()</b> Efface le contenu du presse-papiers
2	<b>setImage()</b> Copie QImage dans le presse-papiers
3	<b>setMimeData()</b> Définit les données MIME dans le presse-papiers
4	<b>setPixmap()</b> Copie l'objet QPixmap dans le presse-papiers
5	<b>Définir le texte()</b> Copie QString dans le presse-papiers
6	<b>texte()</b> Récupère le texte du presse-papiers

Le signal associé à l'objet presse-papiers est -

Sr.No.	Méthode et description
1	<b>dataChanged()</b> Chaque fois que les données du presse-papiers changent

## Exemple

Dans l'exemple suivant, deux objets QTextEdit et deux Pushbuttons sont ajoutés à une fenêtre de niveau supérieur.

Pour commencer, l'objet presse-papiers est instancié. La méthode Copy() de l'objet textedit copie les données dans le presse-papiers du système. Lorsque le bouton Coller est cliqué, il récupère les données du presse-papiers et les colle dans un autre objet textedit.

```
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())

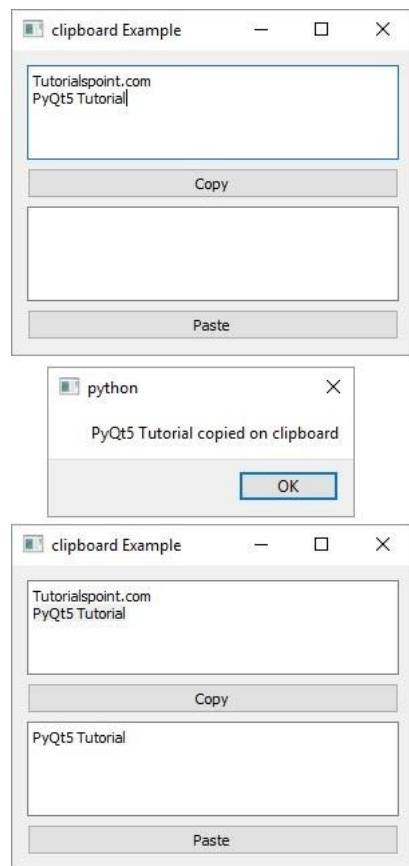
msg=QMessageBox()
msg.setText(clipboard.text()+" copied on clipboard")
msg.exec_()

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_())

```

Le code ci-dessus produit la sortie suivante -



## PyQt5 - Classe QPixmap

**La classe QPixmap** fournit une représentation hors écran d'une image. Il peut être utilisé comme objet QPainterDevice ou peut être chargé dans un autre widget, généralement une étiquette ou un bouton.

L'API Qt a une autre classe similaire **QImage**, qui est optimisée pour les E/S et autres manipulations de pixels. QPixmap, d'autre part, est optimisé pour l'afficher à l'écran. Les deux formats sont interconvertibles.

Les types de fichiers image pouvant être lus dans un objet QPixmap sont les suivants :

BMP	Bitmap Windows
GIF	Format d'échange graphique (facultatif)
JPG	Groupe mixte d'experts photographiques
JPEG	Groupe mixte d'experts photographiques
PNG	Portable Network Graphics
PBM	Bitmap portable
PGM	Carte grise portable
PPM	Table de pixels portable
XBM	Bitmap X11
XPM	Plan de pixels X11

Les méthodes suivantes sont utiles pour gérer l'objet QPixmap -

Sr.No.	Méthodes et description
1	<b>copie()</b> Copie les données pixmap d'un objet QRect
2	<b>de l'image()</b> Convertit l'objet QImage en QPixmap
3	<b>saisirWidget()</b> Crée un pixmap à partir du widget donné
4	<b>saisirFenêtre()</b> Créer une pixmap de données dans une fenêtre
5	<b>Charge()</b> Charge un fichier image en tant que pixmap
6	<b>enregistrer()</b> Enregistre l'objet QPixmap en tant que fichier
7	<b>àImage</b> Convertit un QPixmap en QImage

L'utilisation la plus courante de QPixmap consiste à afficher une image sur une étiquette/un bouton.

## Exemple

L'exemple suivant montre une image affichée sur un QLabel en utilisant la méthode **setPixmap()** .

Le code complet est le suivant -

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

def window():
    app = QApplication(sys.argv)
    win = QWidget()
    ll = QLabel()
    ll.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()
```

Le code ci-dessus produit la sortie suivante -

