Introduction à PySide

Loïc Gouarin

Laboratoire de Mathématiques d'Orsay

2 au 6 décembre 2013 - Biarritz

Les interfaces graphiques avec Python

- Tkinter pour Tk
- WxPython pour WxWidget
- pyGtk pour Gtk
- PyQt et PySide pour Qt
- Kivy

Différences entre PyQt et PySide

- PyQt utilise l'API 1 pour les versions Python 2.x et l'API 2 pour les versions Python 3.x.
- PySide utilise toujours l'API 2.
- Les licences sont différentes : PyQt -> GPL, PySide -> LGPL.
- Quelques différences de syntaxe : http://qt-project.org/wiki/Differences_Between_PySide_and_PyQt

Votre première application

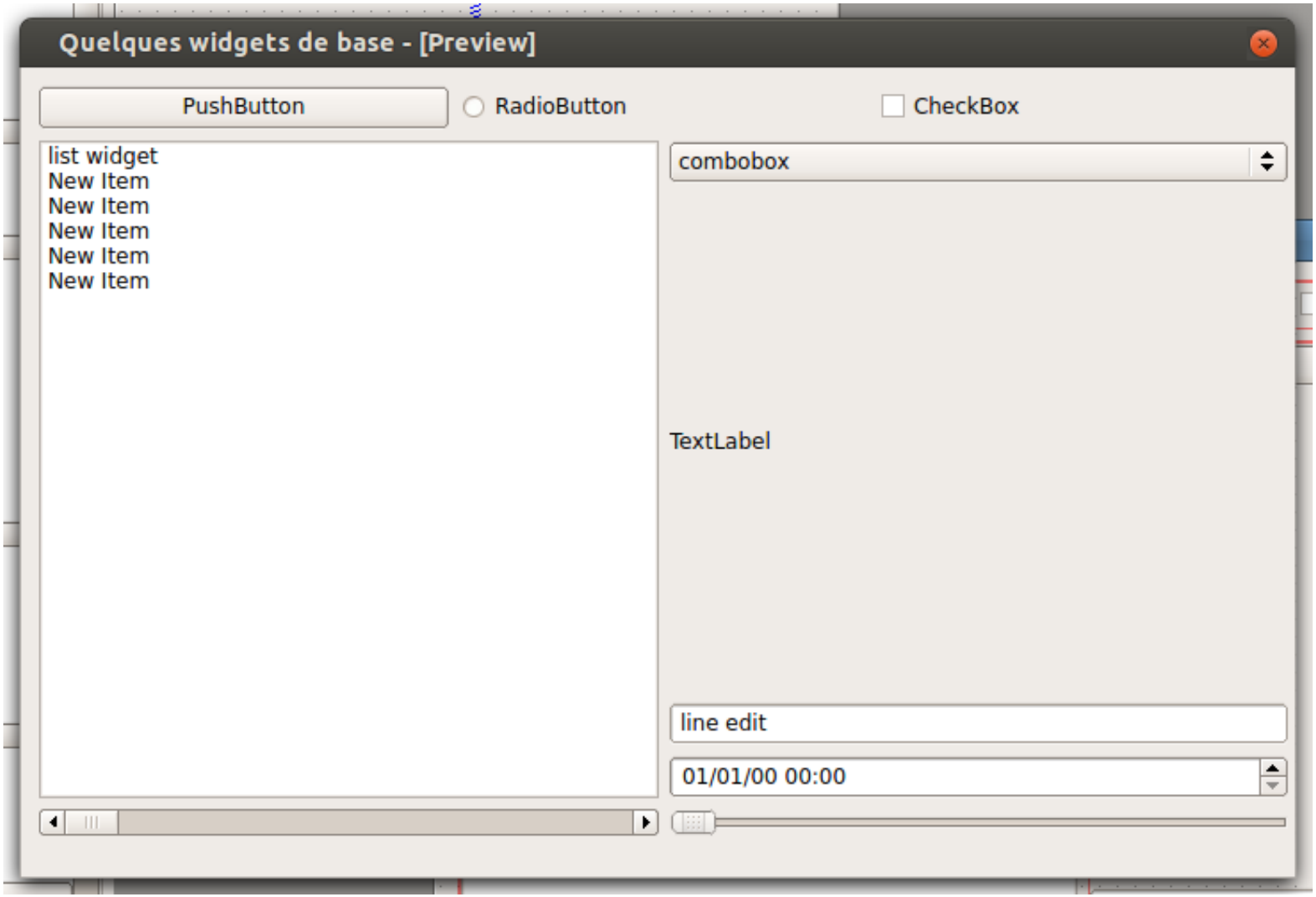
```
import sys
from PySide import QtGui, QtCore

app = QtGui.QApplication(sys.argv)
data = ['square', 'rectangle', 'cube', 'parallelepiped']

comboBox = QtGui.QComboBox()
comboBox.addItems(data)
comboBox.show()

listWidget = QtGui.QListWidget()
listWidget.addItems(data)
listWidget.show()
app.exec_()
```

Votre première application

- `QtCore` : toutes les fonctionnalités non-GUI.
- `QtGui` : toutes les fonctionnalités GUI.
- `QApplication` : permet, entre autre, de gérer les événements.

Les widgets de base

Dans Qt, tous les éléments qui peuvent composer l'interface graphique (bouton, éditeur de texte, liste, label, ...) sont des classes dérivées de la classe `QWidget`.

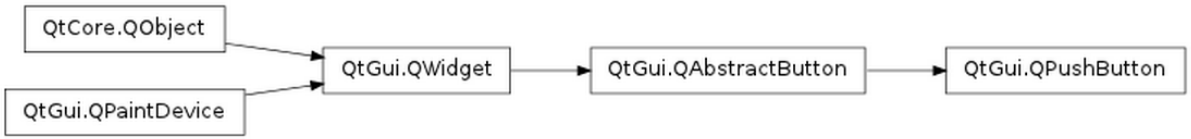


Documentation PySide

Pour connaître les méthodes de chacune des classes de PySide et le diagramme des classes, il suffit de consulter les documentations

- PySide 1.0.7 API Reference
- PySide 1.1.0 API Reference

QPushButton



Inherited by: *QCommandLinkButton*

Synopsis

Functions

def **autoDefault** ()
def **initStyleOption** (option)
def **isDefault** ()
def **isFlat** ()
def **menu** ()
def **setAutoDefault** (arg__1)
def **setDefault** (arg__1)
def **setFlat** (arg__1)
def **setMenu** (menu)

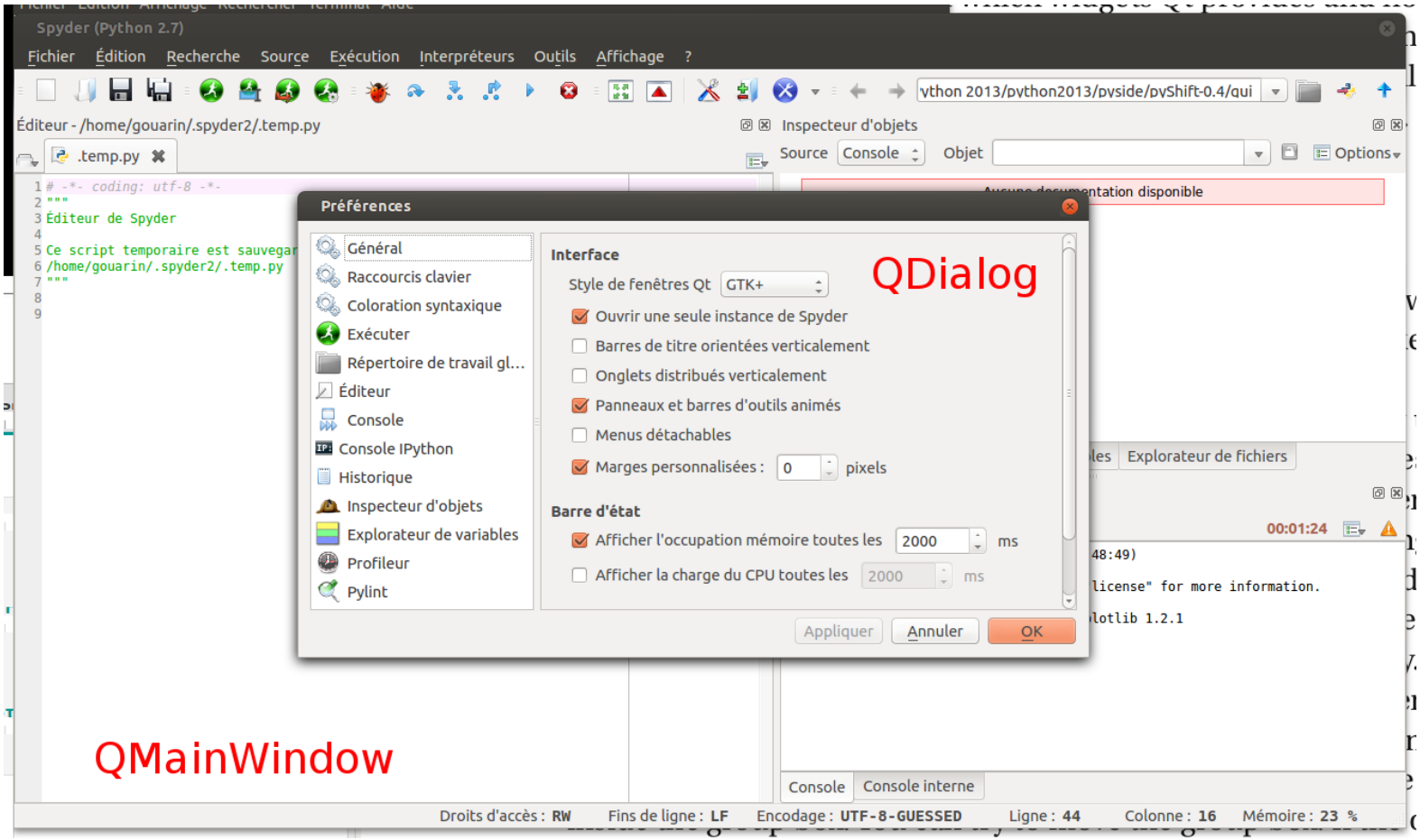
Slots

def **showMenu** ()

Detailed Description

Applications

La plupart des interfaces graphiques sont construites en utilisant des fenêtres de dialogue et une fenêtre principale.

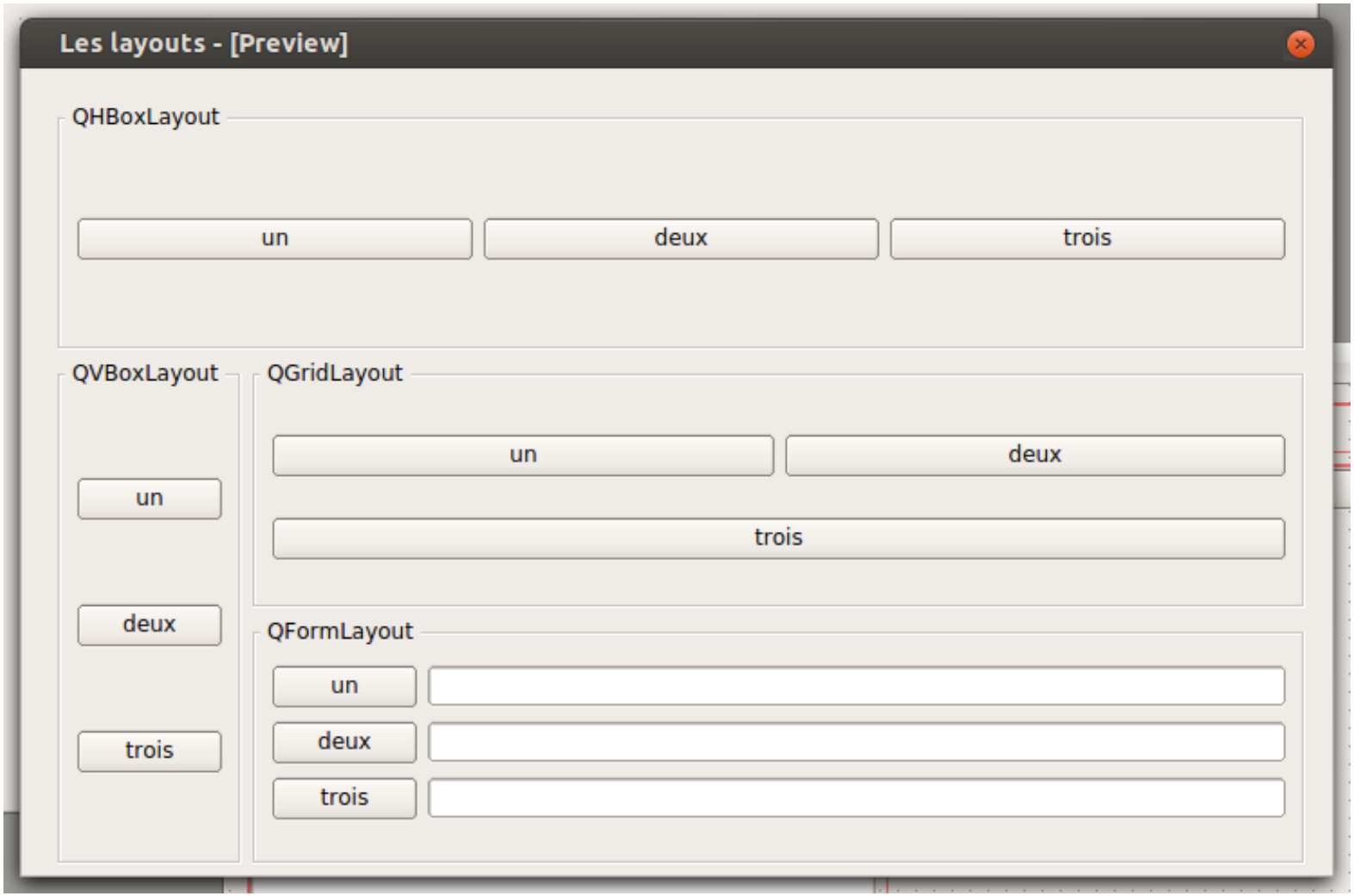


Exemple de création d'une fenêtre de dialogue

```
import sys
from PySide import QtCore, QtGui

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

        data = ['square', 'rectangle',
                'cube', 'parallelepiped']

        self.setWindowTitle("exemple de dialogue")
        self.setFixedSize(300, 100)
        listWidget = QtGui.QListWidget(self)
        listWidget.addItems(data)

if __name__ == '__main__':
    app = QtGui.QApplication(sys.argv)
    d = monDialog()
    d.show()
    app.exec_()
```

Les layouts

Les layouts permettent de placer correctement les widgets entre eux.



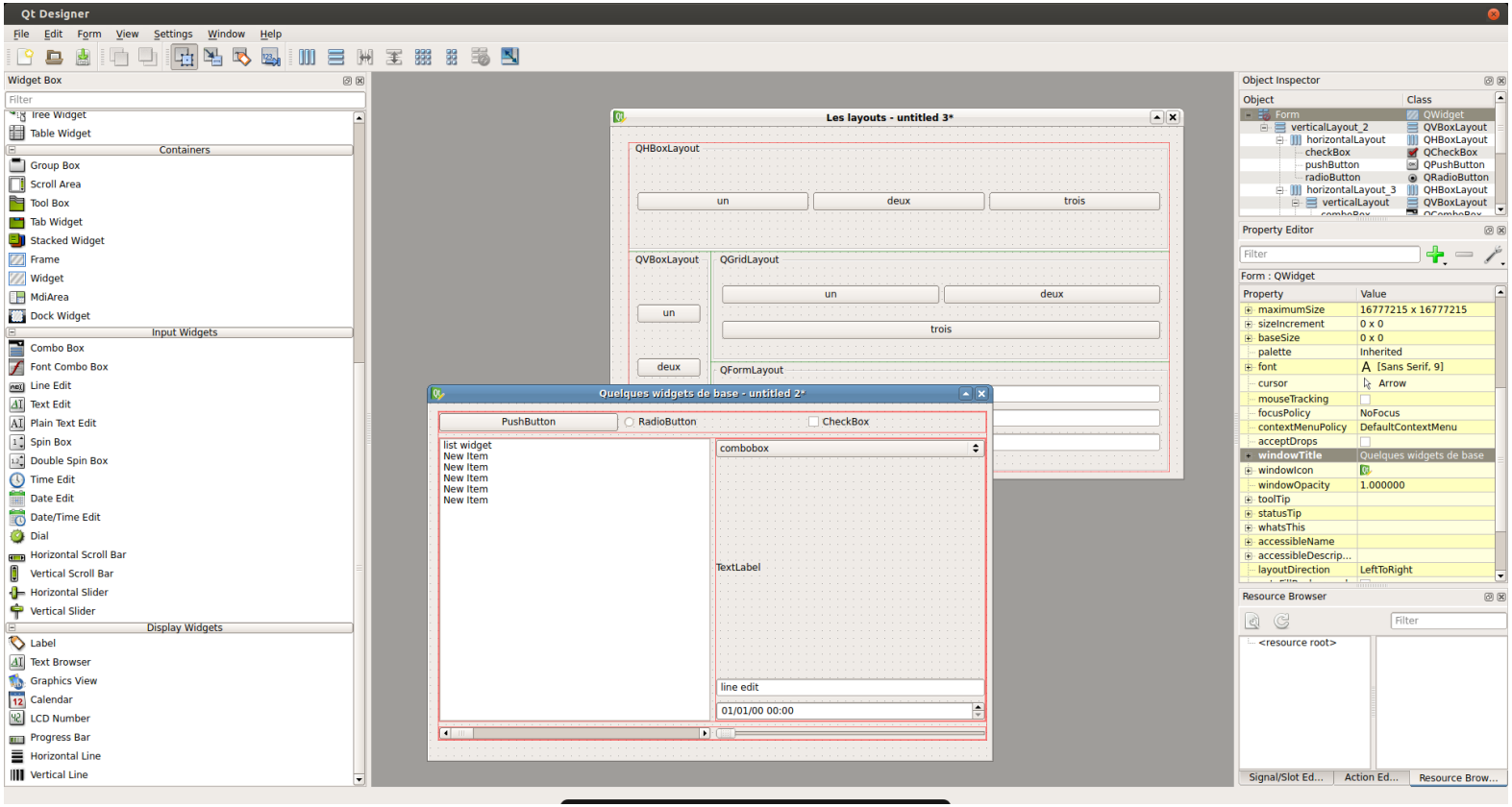
Exemple d'utilisation de layout

```
import sys
from PySide import QtCore, QtGui

class monDialog(QtGui.QDialog):
    def __init__(self, parent = None):
        super(monDialog, self).__init__(parent)
        btn1 = QtGui.QPushButton("un")
        btn2 = QtGui.QPushButton("deux")
        btn3 = QtGui.QPushButton("trois")

        layout = QtGui.QHBoxLayout()
        layout.addWidget(btn1)
        layout.addWidget(btn2)
        layout.addWidget(btn3)
        self.setLayout(layout)

if __name__ == '__main__':
    app = QtGui.QApplication(sys.argv)
    d = monDialog()
    d.show()
    app.exec_()
```

Exercice

Ecrire le code Python permettant d'avoir la fenêtre de dialogue suivante

Utilisation du designer



Politiques sur le redimensionnement des widgets

Politique	peut grandir	peut rétrécir	veut grandir
Fixed	non	non	non
Minimum	oui	non	non
Maximum	non	oui	non
Preferred	oui	oui	non
Expanding	oui	oui	oui
MinimumExpanding	oui	non	oui
Ignored	oui	oui	oui

Comment lire un fichier `ui` en Python ?

- utiliser la commande `pyside-uic` pour transformer le fichier en une classe Python,
- utiliser le module `QtUiTools` pour importer directement le fichier dans son code Python.

La commande `pyside-uic`

- Créer son fichier à partir du designer.
- Créer le fichier Python associé en exécutant la commande

```
pyside-uic mondialog.ui > mondialog.py
```

- Importer la classe ainsi créée dans votre application.
- Créer une instance de la classe ainsi créée.
- Appeler la méthode `setupUi`.

La commande `pyside-uic`

```
import sys
from PySide import QtCore, QtGui
from forms.dialog_ui import Ui_Form

class monDialog(QtGui.QDialog):
    def __init__(self, parent = None):
        super(monDialog, self).__init__(parent)
        self.ui = Ui_Form()
        self.ui.setupUi(self)

if __name__ == '__main__':
    app = QtGui.QApplication(sys.argv)
    d = monDialog()
    d.show()
    app.exec_()
```

Le module QtUiTools

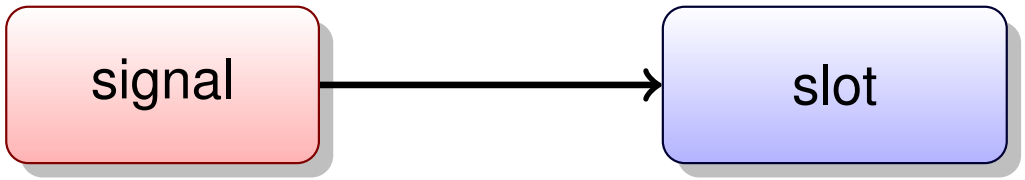
Le module QtUiTools permet de charger directement le fichier ui dans son code Python sans passer par une opération en ligne de commandes.

```
import sys
from PySide import QtCore, QtGui, QtUiTools

class monDialog(QtGui.QDialog):
    def __init__(self, parent = None):
        super(monDialog, self).__init__(parent)
        loader = QtUiTools.QUiLoader()
        self.ui = loader.load("./forms/dialog.ui")

    def show(self):
        self.ui.show();

if __name__ == '__main__':
    app = QtGui.QApplication(sys.argv)
    d = monDialog()
    d.show()
    app.exec_()
```

Les signaux et les slots

Il existe deux façons de gérer les événements

- méthode bas niveau : gérer à la main les événements (clavier, souris, ...)
- méthode haut niveau : utiliser les signaux et les slots



Les signaux et les slots

Exemple avec un QPushButton (QAbstractButton)

Slots

```
def animateClick ([msec=100])
def click ()
def setChecked (arg__1)
def setIconSize (size)
def toggle ()
```

Signals

```
def clicked ([checked=false])
def pressed ()
def released ()
def toggled (checked)
```

Les signaux et les slots

Exemple avec un QPushButton (QAbstractButton)

```
import sys
from PySide import QtCore, QtGui

def cliquer():
    print "click"

app = QtGui.QApplication(sys.argv)

button = QtGui.QPushButton("mon bouton")
button.clicked.connect(cliquer)
button.show()

app.exec_()
```

Les signaux et les slots

Créer ses propres signaux

```
import sys
from PySide import QtCore, QtGui

class Bouton(QtGui.QPushButton):
    myclicked = QtCore.Signal()

    def __init__(self, text, parent=None):
        super(Bouton, self).__init__(parent)
        self.setText(text)
        self.clicked.connect(self.myclicked)
        self.myclicked.connect(self.cliquer)

    @QtCore.Slot()
    def cliquer(self):
        print "click"

app = QtGui.QApplication(sys.argv)
button = Bouton("mon bouton")
button.show()
app.exec_()
```

Les signaux et les slots

Créer ses propres signaux

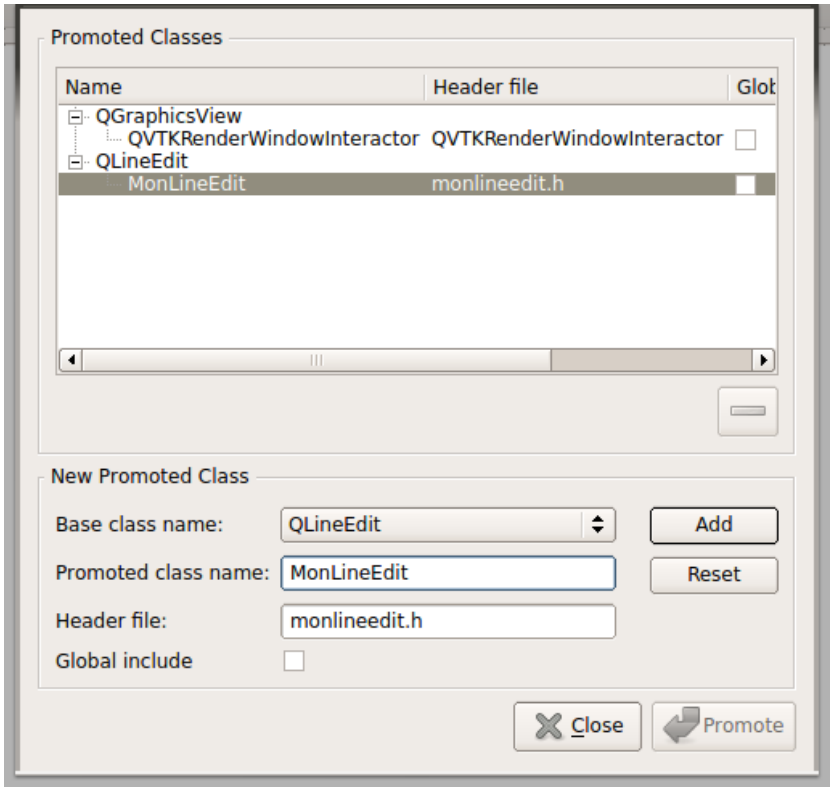
```
import sys
from PySide import QtGui, QtCore

class TestSignal(QtGui.QDialog):
    mysignal = QtCore.Signal(int)

    def __init__(self, parent=None):
        super(TestSignal, self).__init__(parent)
        self.mysignal.connect(self.myslot)

    @QtCore.Slot(int)
    def myslot(self, value):
        print "value:", value

if __name__ == '__main__':
    app = QtGui.QApplication(sys.argv)
    t = TestSignal()
    t.show()
    t.mysignal.emit(6)
    app.exec_()
```

Utiliser ses propres widgets avec designer

On écrit sa classe pour son widget

```
class MonLineEdit(QtGui.QLineEdit):
    def __init__(self, parent=None):
        super(MonLineEdit, self).__init__(parent)
        self.setText("Entrer votre code")
```

On fait son interface avec designer et on utilise `promote` pour le widget que l'on veut customiser.



Utiliser ses propres widgets avec designer

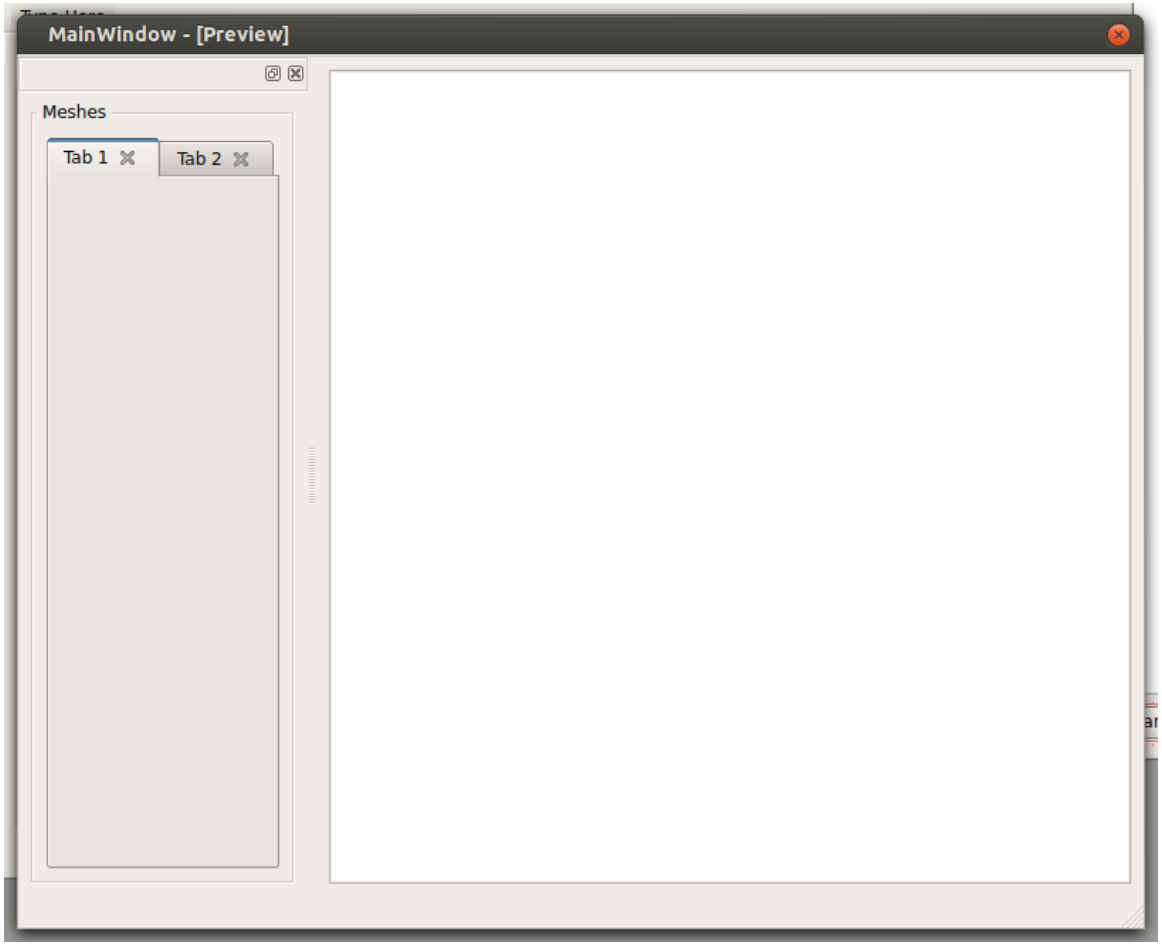
Il faut indiquer à Qt comment trouver ce nouveau widget si on utilise `QUiTools`.

```
class Test(QtGui.QDialog):
    def __init__(self, parent=None):
        super(Test, self).__init__(parent)
        loader = QtUiTools.QUiLoader()
        loader.registerCustomWidget(MonLineEdit)
        self.ui = loader.load("./forms/test.ui")

    def show(self):
        self.ui.show()
```

Nous allons créer une interface graphique pour pyShift en ayant à la fin l'arborescence suivante

```
pyShiftGui/
├── forms
│   ├── dialog.ui
│   ├── meshWindow.ui
│   └── tabMesh.ui
├── gui.py
├── model.py
├── QVTKRenderWindowInteractor.py
└── vtkPlot.py
```

Créer dans un designer les 2 fenêtres suivantes



que vous enregistrerez respectivement dans `forms/meshWindow.ui` et `forms/tabMesh.ui`.

- Créer une classe `MainWindow` qui initialise la fenêtre de dialogue permettant de créer une nouvelle forme.
- Connecter le bouton `add` à une fonction qui ajoute un maillage.
- Ecrire le slot associé qui construit pour le moment un maillage avec pyShift avec les arguments entrés dans les différentes cases.

- Créer une classe `MeshWindow` qui initialise la fenêtre pour visualiser les maillages avec VTK en utilisant le `meshWindow.ui` créé précédemment.
- Mettre dans ses attributs une instance de la classe
  - `pyShiftModel`,
  - `VtkPlot`
- Ajouter une instance de `MeshWindow` dans `MainWindow`
- Modifier la méthode qui ajoute un maillage dans la classe `MainWindow` pour qu'elle appelle une méthode de la classe `MeshWindow` qui ajoute un maillage en prenant en paramètres d'entrée `nx, ny, nz` et la forme.

- Créer une méthode qui ajoute un maillage dans la classe `MeshWindow`. Cette méthode vérifie que les paramètres sont exacts et affiche le maillage dans la fenêtre VTK. Sinon, émet un signal qui demande à afficher un message disant qu'un des paramètres n'est pas correct.
- Créer une méthode qui ajoute un onglet dans le dock widget. Dans cet onglet, on mettra le widget `forms/tabMesh.ui` créé précédemment.
- Mettre dans la `TableWidget` de cet onglet les coordonnées des points (on ira voir dans la doc pour plus de précision).

- Connecter un signal de `checkBox` display à un slot qui permet d'afficher ou non le maillage de l'onglet sélectionné dans la fenêtre VTK en fonction de l'état de la case (cochée ou pas).
- Connecter un signal d'une `tabWidget` à un slot lui indiquant ce qu'il faut faire si l'utilisateur ferme un onglet. On enlèvera des différentes listes ce maillage et on rafraîchira la fenêtre VTK.
- Connecter un signal de la classe `MainWindow` à un slot qui permet de quitter l'application si on ferme cette fenêtre.