Un livre de Wikilivres.

PyQt

Avant-propos

Ce rapport constitue un projet de trimestre effectué en section d'informatique logiciel à la Haute École d'ingénierie et de Gestion du Canton de Vaud. Il a été réalisé sur une période de huit semaines, à raison de 4 périodes par semaine, durant l'été 2006.

Problématique

Le langage Python comporte un grand nombre d'avantages : c'est un langage orienté objet, interprété, typé dynamiquement et bénéficiant d'une gestion de la mémoire via un ramasse-miette.

Pour toutes ces raisons, Python est un langage très simple à utiliser. Cependant la création d'interface en Python à l'aide des outils de base s'avère quelque peu fastidieuse. C'est pourquoi il existe de nombreux modules Python permettant de réaliser des interfaces graphiques, PyQt étant l'un d'eux.

PyQt est un module qui permet de lier le langage Python à la librairie Qt. Cette librairie offre une panoplie de composants graphiques et est extrêmement bien documentée, la société Trolltech qui développe Qt proposant de nombreux tutoriaux ainsi qu'un manuel de référence sur son site. Qt a été développée en C++, par conséquent toute la documentation de référence est en C++.

Le but de ce projet consiste en une étude de la libraire PyQt du point de vue d'un développeur Python. Ce rapport ne se veut ni comme un tutoriel ni une traduction du manuel de référence, son principal objectif étant d'introduire ce module à l'aide d'exemples codé en Python et non en C++. Dans cette optique, un petit guide de traduction de C++ à Python est fourni à la fin de ce rapport afin de permettre aux développeurs Python de comprendre la documentation de référence.

Présentation de la librairie

La librairie Qt

Qt est une bibliothèque logicielle offrant essentiellement des composants d'interface graphique (communément appelés widgets), mais également d'autres composants non-graphiques permettant entre autre l'accès aux données, les connexions réseaux, la gestion des files d'exécution, etc. Elle a été développée en C++ par la société Trolltech et est disponible pour de multiples environnements Unix utilisant X11 (dont Linux), Windows et Mac OS. Qt est un toolkit qui présente de nombreux avantages. Il est intéressant de les souligner puisque ces avantages se retrouvent dans PyQt.

- **Toolkit graphique complet en C++**
  - il est relativement simple à utiliser ;
  - il offre de nombreux outils et extensions ;
  - ce toolkit est disponible gratuitement pour Unix ;
  - il constitue la base de l'environnement KDE.
- **Multiplateformes**
  - disponible pour Unix, Windows et Mac OS X;
  - *look and feel* natif
  - les applications présentent l'apparence des systèmes d'exploitations sur lesquels elles tournent par défaut, mais il est possible d'installer un autre *look and feel*, même lors de l'exécution ;
  - il est implémenté sur les couches basses des systèmes graphiques.
- **Caractéristiques, extensions, outils**
  - internationalisation ;
  - OpenGL multiplateformes ;
  - base de données SQL ;
  - générateur d'interfaces Qt Designer.

PyQt

PyQt est un module qui permet de lier le langage Python avec la bibliothèque Qt. Il permet ainsi de créer des interfaces graphiques en python. Une extension de QtDesigner (utilitaire graphique de création d'interfaces Qt) permet de gérer le code python d'interfaces graphiques. PyQt dispose de tous les avantages lié à Qt.

Versions utilisées dans ce guide

À l'heure où nous terminons d'écrire ce guide la librairie PyQt permettant d'utiliser Qt en version 4 vient de sortir d'une longue phase de développement. Nous avons donc rédigé ce guide en utilisant la version 3.15 de la librairie, permettant d'utiliser Qt en version 3. Qt 4 apporte quelques changements importants dans la gestion des éléments graphiques, avec de nouvelles possibilités fort intéressantes. Nous vous laissons consulter la documentation de référence de Qt ainsi que de PyQt pour voir les différences avec la version 3 de ce guide.

Installation

Dans cette partie nous allons essayer de détailler l'installation de la librairie PyQt sous différents systèmes d'exploitations. Nous partons du point de vue que vous avez déjà la librairie Qt d'installée sur votre système, de même qu'un interpréteur Python. Si ce n'est pas le cas, référez-vous aux sites officiels pour trouver de la documentation d'installation sur ces deux librairies.

Linux • MacosX • Windows

Installation/Linux

Installation sous Gentoo Linux

L'installation sous Gentoo Linux est très aisée. Dans un terminal, connectez-vous avec le compte root, puis tapez la commande suivante :

```
wiki@books~$: emerge PyQt
```

Cela devrait installer l'ensemble des librairies et programmes pré-requis, les librairies Qt et Python comprises. Une fois la compilation terminée, vous pouvez lancer des programmes utilisant PyQt en appelant simplement l'interpréteur python sans paramètres supplémentaires.

Installation sous Ubuntu Linux & Debian Linux

Ubuntu Linux étant basé sur la distribution Debian Linux, l'installation de PyQt sous ces deux systèmes est identique. Pour cela, il vous suffit d'installer le paquetage python-qt3, soit en utilisant l'utilitaire de gestion de paquetages Synaptics soit en ligne de commande avec l'instruction suivante :

```
wiki@books~$: apt-get install python-qt3
```

Le gestionnaire de paquetages va ensuite se charger d'installer la librairie ainsi que tout ses prérequis pour qu'elle fonctionne correctement. Une fois l'installation terminée, vous pouvez lancer des programmes utilisant PyQt en appelant simplement l'interpréteur Python sans paramètres supplémentaires.

Installation/MacosX

À l'heure actuelle il n'existe pas de version pré-compilée de PyQt pour Mac, il vous faudra donc compiler cette librairie vous même. Mais cette étape n'est pas très compliquée.

Librairie SIP

Il vous faut en premier lieu installer SIP, qui est disponible en téléchargement depuis le même site web que la librairie PyQt. Nous installons ici la version 4.4.5. Tapez ensuite les commandes suivantes dans un terminal :

```
cd sip-4.4.5/
python configure.py
make
sudo make install
```

Librairie PyQt

Une fois la librairie SIP installée, vous pouvez passer à l'installation de PyQt proprement dite. Récupérez l'archive dénommée PyQt-mac-gpl-3.16.tar.gz, décompressez là et tapez les commandes suivantes dans un terminal :

```
cd PyQt-mac-gpl-3.16/
python configure.py
make
sudo make install
```

Installation/Windows

La librairie Qt 4.9 est Open-source.

Téléchargements

Il faut déjà avoir Python (http://www.python.org/getit/) [archive], connaitre sa version et savoir s'il est 32 bits ou 64 bits.

Ensuite il y a deux solutions, soit télécharger le .exe correspondant à la version de Python de l'ordinateur (http://www.riverbankcomputing.co.uk/software/pyqt/download) [archive], en l'installant dans le répertoire Python existant, soit compiler les sources soi-même (plus dur), avec :

1. QT (http://qt.nokia.com/downloads) [archive]
2. SIP (http://www.riverbankcomputing.com/software/sip/download) [archive].
3. PyQt (http://www.riverbankcomputing.co.uk/software/pyqt/download) [archive] (.zip ou .tar.gz).

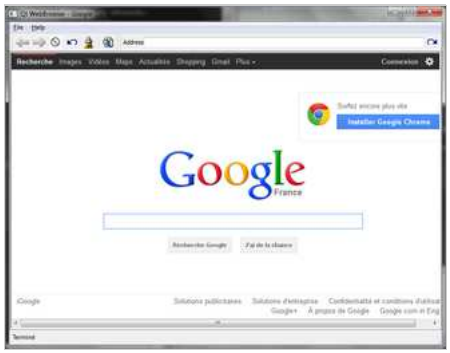
Après l'installation, on peut constater la présence de nouveaux raccourcis dans la liste de tous les programmes du menu démarrer : *PyQt GPL v4.9.1 for Python v2.7 (x64)*.

Test

Le programme est fournit avec quelques exemples.

Navigateur Web

```
C:\Program Files (x86)\Python\Lib\site-packages\PyQt4\examples\activeqt\webbrowser>python
```



Résultat de la commande "python webbrowser.pyw"

Premier exemple : Hello World !

Comme tout bon guide de programmation qui se respecte, nous allons commencer par vous montrer un exemple d'implémentation d'un Hello World. Ce programme affiche une fenêtre contenant un bouton avec le texte `Hello World !`. Si vous cliquez sur le bouton, la fenêtre se ferme et le programme se termine.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# helloworld.py
# Un simple exemple de traditionnel ”Hello World”

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

def main(args) :
        #chaque programme doit disposer d'une instance de QApplication gérant l'ensemble
        app=QApplication(args)
        #un nouveau bouton
        button=QPushButton("Hello World !", None)
        #qu'on affiche
        button.show()
        #fin de l'application lorsque toutes les fenêtres sont fermées
        app.connect(app,SIGNAL("lastWindowClosed()"),app,SLOT("quit()"))
        #fin de l'application lorsque l'utilisateur clique sur le bouton
        app.connect(button, SIGNAL("clicked()"),app,SLOT("quit()"))
        #boucle principale de traitement des évènements
        app.exec_()

if __name__ == "__main__" :
```

```
    main(sys.argv)
```

ou plus court :

```
#!/usr/bin/env python
# coding: utf-8

from PyQt4 import QtGui, QtCore
import sys

app = QtGui.QApplication(sys.argv)
hello = QtGui.QPushButton("Hello World!", None)
hello.show()
app.exec_()
```

La légende détaillée est disponible dans les pages suivantes.

Quelques explications

Ce premier exemple permet d'introduire quelques concepts de base.

Importation des librairies

En premier lieu, il faut indiquer à Python d'importer les librairies PyQt et sys. Ceci est fait avec les lignes suivantes :

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

La méthode `main`

Nous définissons ensuite une méthode main qui s'occupera de gérer l'ensemble du programme. Cette méthode crée en premier lieu une instance de `QApplication`. Cette instance représente l'ensemble de notre application, par conséquent elle est essentielle et doit rester unique pour assurer un fonctionnement correct. Toute application PyQt est constituée d'un objet de type `QApplication`. Par conséquent, une application débute toujours par la création d'un objet de la classe `QApplication` :

```
app=QApplication(args)
```

Veuillez noter toutefois que si vous développez une classe dans laquelle la référence vers `QApplication` n'est pas disponible (typiquement lors de la création de votre propre widget), Qt met à disposition la variable `qApp` fournissant une référence vers l'application Qt. Vous pouvez donc sans autre utiliser cette variable dans votre code.

Création d'un bouton

Un bouton est un objet de la classe `QPushButton` créé comme tout autre objet par l'appel de son constructeur :

```
button=QPushButton("Hello World !", None)
```

Les widgets ne sont pas visibles lors de leur création, c'est pourquoi la méthode `show` de la classe `QWidget` est appliquée à l'objet `button` afin de le rendre visible. Néanmoins, si vous déclarez un widget comme enfant d'un autre, un appel à show sur le parent affichera automatiquement tous ses enfants.

```
button.show()
```

Gestion des évènements

L'exemple utilise deux événements : l'application quitte lorsque l'utilisateur clique sur le bouton précédemment défini ou lorsqu'il ferme la fenêtre. La méthode statique `connect` de la classe `QObject` établit une communication unidirectionnelle entre deux objets. À chaque objet de la classe `QObject` ou de ses classes dérivées, peuvent être associés des signaux permettant d'envoyer des messages et des slots permettant de recevoir des messages. De manière plus rigoureuse, un slot correspond à tout élément pouvant être appelé tandis qu'un signal est le message qui est passé entre objets. Le fonctionnement est le suivant : les objets héritant de `QObject` émettent des signaux quand ils reçoivent des événements. Ces signaux peuvent être connectés à des slots, qui sont donc appelés automatiquement quand l'objet reçoit un événement donné. L'objet émetteur du signal ne se soucie pas de l'existence d'un objet susceptible de recevoir son signal. Cela assure une bonne encapsulation ainsi qu'un mécanisme modulaire. Un même signal peut être connecté à plusieurs slots, tout comme plusieurs signaux peuvent viser le même slot. Dans le cas de notre exemple, deux signaux sont connectés au même slot :

```
#fin de l'application lorsque toutes les fenêtres sont fermées
app.connect(app,SIGNAL("lastWindowClosed()"),app,SLOT("quit()"))
#fin de l'application lorsque l'utilisateur clique sur le bouton
app.connect(button, SIGNAL("clicked()"),app,SLOT("quit()"))
```

Les signaux « `lastWindowClosed()` » de l'objet app et « `clicked()` » de l'objet `button` sont tous deux connectés au slot « `quit()` » par l'appel à la méthode connect. Lorsque les widgets sont définis, la méthode `exec_` de la classe `QApplication` met en place la boucle d'événements. C'est grâce à cette boucle (infinie) que l'application va fonctionner et que les divers événements auront le comportement voulu.

```
#boucle principale de traitement des évènements
app.exec_()
```

Approche orientée objet

Dans cette version du très classique Hello World, une approche plus orientée objet est implémentée. La classe `HelloApplication` encapsule les détails de l'utilisation d'une fenêtre de base. La classe `HelloBouton` construit un bouton nommé Hello World !. Le fonctionnement est totalement identique à l'exemple précédent.

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# helloworld2.py
# Un traditionnel ”Hello World” avec une approche objet plus propre

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

# Classe définissant un bouton avec le texte Hello World !
class HelloButton(QPushButton) :
   def __init__ (self, args) :
      QPushButton.__init__ (self,None)
      self.setText("Hello World !")

class HelloApplication(QApplication) :
   def __init__ (self, args) :
      QApplication.__init__ (self,args)
      # Creation et affichage d'un objet HelloButton
      self.button=HelloButton(self)
      self.button.show()
      # Traitement des divers evenements
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.connect(self.button,SIGNAL("clicked()"),self,SLOT("quit()"))
      #boucle principale de traitement des evenements
      self.exec_()

if __name__ == "__main__" :
   app=HelloApplication(sys.argv)
```

Signaux et slots

Les signaux et les slots sont utilisés pour la communication entre les objets. Le mécanisme de signaux/slots est l’une des caractéristiques principales de Qt et PyQt qui les différencie des autres outils de développement.

Dans Qt comme dans PyQt, un signal est émis par un widget lorsqu’un événement se produit. Ces événements sont généralement le fait d’un utilisateur qui cliquerait par exemple sur un bouton ou qui remplirait un champ. Les slots sont des fonctions appelées en réponse à un signal particulier.

Ainsi en résumé, un objet héritant de `QObject` peut émettre des signaux quand il reçoit des événements comme un clic sur un bouton. Les signaux peuvent être connectés à des slots. Un slot est une fonction membre (« méthode ») appelée automatiquement.

Pour plus d’informations, la page de référence fournit de plus amples détails : http://doc.trolltech.com /3.3/signalsandslots.html.

Positionnement des éléments graphiques

Qt et PyQt mettent à disposition plusieurs méthodes permettant de disposer les widgets et de spécifier leurs tailles. Voici un petit aperçu de ces méthodes. La meilleure façon de disposer les éléments graphiques consiste à les placer dans un widget de type QHBox, QVBox et QGrid. Un widget déclaré enfant de l'une de ces classes sera automatiquement disposé dans son widget parent. Ainsi le placement est entièrement géré par le parent. Chacun des widgets énoncé plus haut possède sa spécificité dans le placement de ses widgets enfants.

`QHBox` dispose ses enfants horizontalement de la gauche vers la droite.

un	deux	trois	quatre	cinq

`QVBox` aligne les widgets enfants sur une colonne verticale depuis le haut vers le bas.

un
deux
trois
quatre
cinq

`QGrid` permet de mettre les enfants sur une grille à deux dimensions dont on peut spécifier le nombre de colonnes, les widgets étant disposés sur la ligne suivante lorsque la ligne précédente est pleine. L'alignement sur cette grille assure que les widgets ne se chevauchent pas lors d'un redimensionnement.

un	deux
trois	quatre
cinq	

Cependant, vous remarquerez que placer ces objets de la sorte est très laborieux et demande de multiples essais afin de voir ce qui est modifié lorsque l'utilisateur change la taille de la fenêtre.

C'est pourquoi nous vous recommandons plutôt d'utiliser Qt Designer, qui est un outil de développement d'interfaces Qt fourni avec la librairie. Cet outil vous permet de placer facilement vos objets, de régler leurs paramètres, d'introduire des objets invisibles élastiques afin de garder une mise en page cohérente, et tout cela de manière graphique. Consultez la section *Utilisation de Qt Designer* pour plus d'informations concernant cet outil.

Gestion du focus

La gestion du focus clavier sous Qt est assez similaire aux autres environnements graphiques. Elle s'effectue lors de différentes actions, listées ci-dessous avec une courte explication de leur fonctionnement et comment les gérer en Qt. Il est possible, pour chaque widget, de définir à quelles actions il doit réagir. Cela s'effectue par l'appel à la méthode `setFocusPolicy()`, prenant en paramètre le type de focus pour lequel le widget doit réagir.

L'utilisateur se déplace d'un widget à l'autre avec les touches tab et shift + tab

Il s'agit de l'action la plus courante, l'utilisateur désirant par exemple se déplacer d'un champ de saisie à l'autre dans un formulaire. Le principe de ces combinaisons de touches est de se déplacer circulairement entre les widgets, il y a donc une relation d'ordre. Bien évidemment, en tant que développeur vous pouvez choisir quels widgets répondent à ces actions ainsi que l'ordre de déplacement. Pour que votre widget réponde à ce type de focus, définissez la politique de focus à `QWidget.TabFocus`. De plus, l'ordre est modifiable par la méthode statique `QWidget.setTabOrder()`.

L'utilisateur clique sur un widget

Le clic pour obtenir le focus est une autre action très courante pour l'utilisateur, il est donc important que votre application le supporte. Cependant il faut bien paramétrer cela faute d'avoir un comportement étrange. Prenons l'exemple d'un éditeur de texte comportant un bouton permettant de mettre un texte en gras. Il faut donc que le bouton réponde au clic, mais pas au focus, faute de quoi le focus reste sur le bouton et l'utilisateur de votre application est quelque peu dérouté. Par conséquent, la recommandation de la documentation de référence est de désactiver le focus par un clic sur les widgets tels que les boutons. L'activation de ce focus s'effectue par la politique `QWidget.ClickFocus`. Remarque Il est possible d'activer le focus avec un clic et le focus avec les touches Tab et Shift + Tab en définissant la politique de focus à `QWidget.StrongFocus`.

L'utilisateur utilise un raccourci clavier

Il est aussi fréquent que le focus se déplace lors de l'utilisation d'un raccourci clavier, si par exemple une fenêtre de dialogue s'ouvre. La documentation de référence suggère donc fortement de gérer le focus lors de raccourcis claviers aussi.

L'utilisateur utilise la roulette de défilement de sa souris

Cette méthode est moins commune, surtout vu son fonctionnement différent entre les diverses plateformes supportées. Sous Microsoft Windows c'est le widget ayant le focus clavier qui reçoit les événements de la souris, tandis que sous X11 et Mac OS X c'est le widget gérant tous les événements de la souris qui reçoit ces informations. Qt gère le focus de la roulette en permettant au focus du clavier de se déplacer lorsque l'utilisateur utilise la roulette de défilement de sa souris. En mettant une politique correcte sur les widgets, le fonctionnement de l'application sur les plateformes supportées sera identique. Pour activer ce focus, il faut lui mettre QWidget.WheelFocus, qui active par la même occasion le focus défini par QWidget.StrongFocus.

L'utilisateur active la fenêtre de l'application

Une autre question à se poser lors du développement d'une application est de savoir quel widget va recevoir le focus lors de l'ouverture de l'application, mais aussi lorsque l'utilisateur met la fenêtre en avant-plan. Cependant, cette partie est assez simple à mettre en place. Si l'utilisateur met la fenêtre en avant-plan, le dernier élément à avoir le focus avant la mise en arrière-plan récupère le focus, car Qt gère cela automatiquement. Pour définir quel widget doit obtenir le focus lors de l'ouverture de la fenêtre, il suffit d'utiliser la méthode setFocus() avec comme parmètre le widget voulu. Celui ci aura automatiquement le focus lors de l'appel à show().

Aucun focus pour le widget

Il est aussi possible de définir un widget de telle sorte à ce qu'il ne puisse pas obtenir le focus. Cela est obtenu avec la politique de gestion de focus QWidget.NoFocus. Pour plus de détails quant à la gestion de focus nous vous invitons à consulter le guide de la documentation de référence à l'adresse

http://doc.trolltech.com/3.3/focus.html.

Quelques widgets

Ce chapitre nous permet d'introduire quelques widgets que nous considérons comme important de connaître[1][2].

L'ensemble des classes disponibles dans Qt peut être divisé en trois grands groupes[3] :

1. Les classes permettant d'obtenir des widgets graphiques.
2. Les classes abstraites telles que les actions utilisateurs, les minuteries, etc.
3. Les classes d'utilitaires permettant par exemple la connexion à une base de données, le traitement de documents XML, etc.

Nous allons commencer cette documentation par trois widgets relativement important, QApplication, QDialog et QMainWindow. Ces widgets représentent respectivement une application Qt complète, une boîte de dialogue et une fenêtre principale. Dans la plupart des guides existants on ne vous parle pas de ces widgets malgré leur importance dans la conception d'une application Qt. C'est pourquoi nous avons décidé de les documenter ici avec des widgets plus graphiques.

Références

1. anglais http://web.archive.org/web/20040306175723/http://doc.trolltech.com/3.3/
2. anglais http://www.riverbankcomputing.co.uk/software/pyqt/intro
3. anglais http://web.archive.org/web/20031226134522/http://doc.trolltech.com/3.3/classchart.html

Voir aussi

- Liste des classes (http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/classes.html) [archive]

QApplication

Description

La classe `QApplication` s'occupe de gérer l'ensemble des paramètres et des affichages d'une application Qt. Elle contient entre autre la boucle principale de traitement des événements.

Cette boucle est une boucle infinie à mettre à la fin de votre code afin que votre programme fonctionne et traite les événements de l'utilisateur ou des objets internes. Il est important de noter que pour chaque application Qt il y a une et une seule instance de QApplication, qu'il y ait 0, 1, 2 ou plus de fenêtres dans votre application. De plus, l'objet QApplication s'occupant d'initialiser l'ensemble de l'application (voir le paragraphe suivant pour plus de détails), il est important qu'il soit créé avant tout autre objet ayant un rapport avec l'interface. Voici les responsabilités de QApplication :

- initialisation de l'application avec les préférences du bureau ; entre autres, palette de couleurs, polices de caractère, intervalle du double clic ; ces éléments sont mis à jour automatiquement lorsque l'utilisateur

change les paramètres de son bureau ;
- gestion des événements venant du système d'exploitation et envoi de ces événements aux bons widgets (par exemple un clic de souris sur un bouton) ;
- parcours des arguments passés à l'exécutable et changement des variables internes en accord avec ces paramètres (par exemple style graphique de l'application) ;
- définition du look de l'application (en rapport avec le point précédent) ;
- localisation des chaînes de caractères (si le programme gère la traduction des éléments) ;
- connaissance de l'ensemble des éléments graphiques de l'application ; utile si vous désirez récupérer la position d'un widget ou connaître le widget qui est situé à une certaine position ;
- gestion du curseur de la souris en fonction de l'élément sur lequel il se trouve ;
- sur le système X-Window, met à disposition des méthodes permettant de gérer la communication avec le serveur d'affichage ;
- gestion des sessions ; permet de terminer correctement l'application lorsqu'un utilisateur se déconnecte du système, d'annuler un arrêt du système si l'application ne peut pas se terminer, et de conserver l'état de l'application pour une future utilisation.

L'ensemble des ces fonctions peuvent être modifiées ou utilisées par l'appel à des méthodes sur l'objet de type QApplication. Pour cela nous vous laissons consulter la documentation de référence de la classe à l'adresse http://qt-project.org/doc/qt-4.8/qapplication.html.

Constructeur

```
# Constructeurs pour la classe QApplication
QApplication ( int argc, str argv )
QApplication ( int argc, str argv, bool GUIenabled )
```

Exemple

Nous vous laissons consulter les autres exemples de ce guide pour voir comment utiliser cette classe. En effet, vu son caractère obligatoire pour obtenir une application elle est utilisée dans tous nos exemples.

QDialog

Description

La classe QDialog représente une fenêtre de dialogue. Une telle fenêtre est généralement utilisée pour de courtes actions et des communications brèves avec l'utilisateur. Cette fenêtre peut être modale ou non, avoir un bouton par défaut et retourner une valeur.

Une fenêtre modale est une fenêtre bloquant toute interaction de l'utilisateur sur d'autres éléments graphiques de l'application pendant tout le temps ou la fenêtre est active. À l'inverse, une fenêtre non-modale permet à l'utilisateur de continuer à interagir avec le reste des éléments graphiques. Cette propriété peut être modifiée par l'appel à une méthode ou bien être mise en place directement dans le constructeur de la classe. Le bouton par défaut est le bouton qui sera activé lorsque l'utilisateur appuie sur la touche « Entrée » du clavier. De même, lorsque l'utilisateur appuie sur la touche « Echap », la fenêtre se ferme et met le code Rejected dans sa valeur de retour. Il existe plusieurs sous-classes à QDialog, permettant de créer des types de dialogues prédéfinis[1] :

- QColorDialog : fenêtre de sélection d'une couleur ;
- QErrorMessage : fenêtre pour un message d'erreur ;
- QFileDialog : dialogue de sélection de fichier :
- QFontDialog : dialogue de sélection de police de caractères ;
- QInputDialog : fenêtre permettant de demander une valeur à l'utilisateur ;
- QMessageBox : boîte de message modale avec une icône, un message et des boutons ;
- QProgressDialog : fenêtre contenant une barre de progression ;
- QTabDialog : fenêtre avec de multiples onglets ;
- QWizard : fenêtre permettant de mettre en place facilement un assistant pour aider l'utilisateur.

Constructeur

```
# Constructeur pour la classe QDialog
QDialog ( QWidget parent = None, str name = "", bool modal = False, WFlags f = 0 )
```

Exemple

PyQt3.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qdialog.py
# Programme exemple pour la classe QDialog
from qt import *
import sys
class Demo(QApplication) :
   def init (self,args) :
      QApplication. init (self, args)
      # dialogue de sélection de fichier
      s = QFileDialog.getOpenFileName("", "", None, "Boite d'ouverture de fichier", "Choisiss
   # conversions nécessaires entre QString et le type str de python
   h = QString("<h1>Vous avez choisi le fichier</h1>\n<i>")
   h.append(s)
   h.append("</i>")
   # message d'information sur le fichier séléctionné
   QMessageBox.information(None, "Fichier choisi", h)
if name == "__main__" :
   x = Demo(sys.argv)
```

PyQt4.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qdialog.py
# Programme exemple pour la classe QDialog
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import sys

class Demo(QApplication) :
   def __init__ (self,args) :
```

```
        QApplication.__init__(self, args)
        # dialogue de sélection de fichier
        s = QFileDialog.getOpenFileName(None, "Boite d'ouverture de fichier")
        # conversions nécessaires entre QString et le type str de python
        h = QString("<h1>Vous avez choisi le fichier</h1>\n<i>")
        h.append(s)
        h.append("</i>")
        # message d'information sur le fichier sélectionné
        QMessageBox.information(None, "Fichier choisi", h)

if __name__ == "__main__" :
    x = Demo(sys.argv)
```

Références

1. anglais http://qt-project.org/doc/qt-4.8/qdialog.html

QMainWindow

Description

Cette classe représente une fenêtre principale d'une application, pouvant contenir entre autre une barre de menus, une zone pour les barres d'outils et une barre de statuts. Cette fenêtre est généralement utilisée pour fournir l'accès à un widget central comme par exemple une zone d'édition de texte, une zone de dessin ou un espace de travail (QWorkspace) pour les applications comportant plusieurs fenêtres. Afin d'ajouter simplement des menus et barres d'outils à une fenêtre, il suffit d'appeler le constructeur du widget voulu en lui passant la fenêtre comme paramètre pour le parent. Ce widget supporte aussi les « fenêtres crochantes », à savoir que vous pouvez avoir plusieurs éléments graphiques dans la fenêtre principale et les positionner à votre guise. Vu la multitude de possibilités offertes par ce widget nous ne pouvons que vous conseiller de consulter la documentation de référence à l'adresse http://qt-project.org/doc/qt-4.8/qmainwindow.html. Vous trouverez aussi sur cette page quelques extraits de code (C++) vous montrant la mise en œuvre de certaines fonctionnalités.

Constructeur

```
# Constructeurs pour la classe QMainWindow
QMainWindow ( QWidget parent = None, str name = "", WFlags f = WType TopLevel)
```

Exemple

PyQt3.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qmainwindow.py
```

```
# Programme exemple pour la classe QMainWindow
from qt import *
import sys
class Demo(QApplication) :
   def init (self, args) :
      QApplication. init (self,args)
      # widget principal, il s'agit d'une fenêtre de dialogue
      self.main = QMainWindow(None, "mainwindow")
      self.main.setCaption("Demo QMainWindow")
      self.main.resize(640,480)
      #widget central : QTextEdit
      self.edit = QTextEdit(self.main, "editor")
      self.edit.setFocus()
      self.main.setCentralWidget(self.edit)
      # ajout facile d'éléments dans la barre de menu
      # NOTE : crée la barre s'il n'existe pas
      self.main.menuBar().insertItem("New")
      self.main.menuBar().insertItem("Quit")
      # changement du message de status
      # NOTE : crée la zone de status si inexistante
      self.main.statusBar().message("Demo en cours", 200000)
      self.main.show()
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.exec loop()
if name == "__main__" :
   app=Demo(sys.argv)
```

PyQt4.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qmainwindow.py
# Programme exemple pour la classe QMainWindow
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
class Demo(QApplication) :
   def __init__(self, args) :
      QApplication.__init__(self,args)
      # widget principal, il s'agit d'une fenêtre de dialogue
      self.main = QMainWindow()
      self.main.setWindowTitle("Demo QMainWindow")
      self.main.resize(640,480)
      #widget central : QTextEdit
      self.edit = QTextEdit("editor", self.main)
      self.edit.setFocus()
      self.main.setCentralWidget(self.edit)
      # ajout facile d'éléments dans la barre de menu
      # NOTE : crée la barre s'il n'existe pas
      self.main.menuBar().addAction("New")
      self.main.menuBar().addAction("Quit")
      # changement du message de status
      # NOTE : crée la zone de status si inexistante
      self.main.statusBar().showMessage("Demo en cours", 10000)
      self.main.show()
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.exec_()
if __name__ == "__main__" :
   app=Demo(sys.argv)
```

QAction & QActionGroup

Description

La classe QAction définit un modèle abstrait permettant de représenter une action sur l'interface. Elle peut être utilisée dans une entrée d'un menu ou par l'intermédiaire d'une barre d'outils. La classe QActionGroup permet quant à elle de regrouper un ensemble d'actions qui doivent être exclusives entre elles.

Dans les interfaces utilisateur les commandes peuvent être invoquées par des menus déroulants, des barres d'outils ou des raccourcis clavier. Il n'est pas rare qu'il existe des actions identiques pouvant être invoquées par différents éléments de l'interface, et que ces éléments doivent rester synchronisés. Il est donc utile de représenter ces commandes comme des actions, puis d'ajouter ces actions aux divers éléments de l'interface.

Prenons un exemple : dans un éditeur de texte, l'utilisateur a la possibilité de mettre du texte en gras. Pour cela, il dispose d'une action dans un menu déroulant, d'un bouton dans une barre d'outils et d'un raccourci clavier. En utilisant une action représentant la mise en gras du texte, non seulement on factorise le code, mais en plus cela nous permet lorsque l'utilisateur utilise le raccourci clavier de mettre le bouton de la barre d'outils dans l'état activé. Un objet QAction peut contenir une icône, du texte, une touche accélératrice (raccourci clavier), un texte d'aide (WhatsThis) et un texte pour l'info-bulle. La plupart de ces données peuvent être chargées par l'intermédiaire du constructeur, mais il est aussi possible d'utiliser des méthodes spécifiques pour les modifier.

Les actions peuvent être de deux types : les actions à état (« toggle action »), et les commandes. Une action à état est par exemple le changement d'un texte en gras ou en normal, tandis qu'une commande peut par exemple représenter l'ouverture d'une fenêtre de sélection de fichier. Seules les actions à état émettent le signal toggled() tandis que les deux types d'actions émettent le signal activated(). Les actions peuvent être ajoutées aux éléments graphiques avec la méthode addTo() tandis qu'elles peuvent être supprimées par l'intermédiaire de l'appel à removeFrom().

Constructeur

```
# Constructeurs pour les classes QAction et QActionGroup
QAction ( QObject parent, str name = "" )
QAction ( Qstr menuText, QKeySequence accel, QObject parent, str name = "" )
QAction ( QIconSet icon, Qstr menuText, QKeySequence accel, QObject parent, str name
= "" )
QActionGroup ( QObject parent, str name = "" )
QActionGroup ( QObject parent, char name, bool exclusive )
```

Exemple

Il est très difficile de trouver un exemple mettant en œuvre ces deux classes. Cependant, vous pouvez consulter le code de la section Qt Designer dans laquelle, à partir d'une interface générée avec l'utilitaire Qt Designer, nous créons une petite application. Au sein de cette application nous avons utilisé la classe QAction.

QLabel

Description

Cette classe permet d'afficher au choix du texte ou une image, tout en spécifiant plusieurs paramètres concernant l'apparence visuelle de l'objet. Il est possible d'afficher du texte simple, du texte contenant des informations de mise en page, une image (pixmap), une vidéo, un nombre (converti en chaîne de caractères automatiquement) ou bien mettre un QLabel vide. Les changements concernant l'apparence de l'objet portent sur l'alignement dans son widget parent et l'indentation du texte. De plus, il est possible de définir des touches accélératrices permettant de transférer le focus du clavier.

Il est possible de mettre simplement en forme le texte. Qt utilise pour cela un sous-ensemble de balises HTML. Pour de plus amples informations consultez la page http://qt-project.org/doc/qt-4.8/q3stylesheet.html. html qui défini l'ensemble des balises disponibles et leurs effets sur le texte.

Constructeur

```
# Constructeurs pour la classe QLabel
QLabel ( QWidget parent, str name = "", WFlags f = 0 )
QLabel ( Qstr text, QWidget parent, str name = "", WFlags f = 0 )
QLabel ( QWidget buddy, Qstr text, QWidget parent, str name = "", WFlags f = 0 )
```

Exemple

PyQt3.*

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# qlabel.py
# Programme exemple pour la classe QLabel
from qt import *
import sys
class Demo(QApplication) :
   def init (self,args) :
      QApplication. init (self, args)
      # widget principal, il s'agit d'une fen^etre de dialogue
      self.dialog = QDialog(None, "Dialog")
      # premier élément textuel, nom de l'objet = ”text1”
      self.text1 = QLabel(self.dialog,"text1")
      # taille de l'objet et positionnement absolu
      self.text1.setGeometry(QRect(120,80,181,71))
      # le texte `a afficher
      self.text1.setText("Le texte \ntient sur \nplusieurs lignes !")
      # mise en page : très facile !
      self.text2 = QLabel(self.dialog,"text2")
      self.text2.setText("<h1><font color='red'>Et la mise en page\n"+ " est possible</font><
      self.text2.setGeometry(QRect(120,220,191,101))
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.dialog.show()
      self.exec loop()
if name == "__main__" :
   x = Demo(sys.argv)
```

PyQt4.*

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# qlabel.py
# Programme exemple pour la classe QLabel
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
class Demo(QApplication) :
   def __init__(self,args) :
      QApplication.__init__(self, args)
      # widget principal, il s'agit d'une fen^etre de dialogue
      self.dialog = QDialog()
      self.dialog.setWindowTitle("Dialog")
      # premier élément textuel, nom de l'objet = ”text1”
      self.text1 = QLabel(self.dialog)
      # taille de l'objet et positionnement absolu
      self.text1.setGeometry(QRect(120,80,180,70))
      # le texte `a afficher
      self.text1.setText("Le texte \ntient sur \nplusieurs lignes !")
      # mise en page : très facile !
      self.text2 = QLabel(self.dialog)
      self.text2.setText("<h1><font color='red'>Et la mise en page<br>"+ " est possible</font
      self.text2.setGeometry(QRect(120,220,250,100))
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.dialog.show()
      self.exec_()
if __name__ == "__main__" :
   x = Demo(sys.argv)
```

QCheckBox

Description

La classe QCheckBox permet de mettre en place une case à cocher avec soit un label textuel soit une image (pixmap) à côté. Le label textuel peut être paramétré par le constructeur et par un appel à la méthode setText(), tandis que l'image peut être ajoutée par setPixmap(). Cette case peut avoir deux états, soit coché (checked), soit décoché (unchecked). Le changement d'état peut être récupéré avec le signal toggled(), tandis qu'il est possible de récupérer l'état de la case avec la méthode isChecked().

En plus des deux états standards de ce type d'élément, Qt met à disposition un troisième état, appelé « no change ». Cet état est activable grâce à la méthode setTristate(). Cet état, que l'on peut traduire par « état de non changement », permet à l'utilisateur d'indiquer par exemple dans une fenêtre de configuration que la valeur stockée ne doit pas être modifiée. Néanmoins, cette fonctionnalité est très peu souvent utilisée, certainement car elle est difficilement compréhensible pour un utilisateur novice, et que l'on préfère mettre l'état de la case à coché si la valeur précédente l'était déjà. Il est possible de regrouper plusieurs boutons en utilisant la classe QButtonGroup.

Constructeur

```
# Constructeurs pour la classe QCheckBox
QCheckBox ( QWidget parent, str name = "" )
QCheckBox ( Qstr text, QWidget parent, str name = "" )
```

Exemple

PyQt3.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qcheckbox.py
# Programme exemple pour la classe QCheckBox#
from qt import *
import sys
class Demo(QApplication) :
   def init (self, args) :
      QApplication. init (self,args)
      # widget principal, il s'agit d'une fenêtre de dialogue
      self.dialog = QDialog(None, "Dialog")
      #Astuce : on peut grouper les boutons grâce à un
      # QButtonGroup ; positionnement horizontal des boutons
      self.group1=QButtonGroup(1, Qt.Horizontal, None, "group1")
      self.group1.setTitle("Essai QCheckBox")
      # définition de 3 checkboxes
      # NOTE : le widget QButtonGroup se redimensionne automatiquement
      self.check1=QCheckBox("checkbox 1", self.group1, "check1")
      self.check2=QCheckBox("checkbox 2", self.group1, "check2")
      self.check3=QCheckBox("checkbox 3", self.group1, "check3")
      # le bouton check1 est de type ”tristate”
      self.check1.setTristate()
      # le bouton check2 est par défaut coché
      self.check2.setChecked(True) ;
      # le bouton check3 n'est pas modifiable
      self.check3.setDisabled(True) ;
      # espace entre les boutons
      self.group1.addSpace(3)
      self.group1.show()
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.exec loop()
if name == "__main__" :
   app=Demo(sys.argv)
```

PyQt4.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qcheckbox.py
# Programme exemple pour la classe QCheckBox#
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
class Demo(QApplication) :
   def __init__(self, args) :
      QApplication.__init__(self,args)
      # widget principal, il s'agit d'une fenêtre de dialogue
      self.dialog = QDialog()
      self.dialog.setWindowTitle("Dialog")
      #Astuce : on peut grouper les boutons grâce à un
      # QGroupBox
      self.group1=QGroupBox("Essai QCheckBox", self.dialog)
      self.group1.setObjectName("group1")
      # définition de 3 checkboxes
      self.check1=QCheckBox("checkbox 1")
```

```
        self.check2=QCheckBox("checkbox 2")
        self.check3=QCheckBox("checkbox 3")
        # le bouton check1 est de type ”tristate”
        self.check1.setTristate()
        # le bouton check2 est par défaut coché
        self.check2.setChecked(True) ;
        # le bouton check3 n'est pas modifiable
        self.check3.setDisabled(True) ;
        # Layout vertical pour les checkboxes
        self.vbox = QVBoxLayout()
        self.vbox.addWidget(self.check1)
        self.vbox.addWidget(self.check2)
        self.vbox.addWidget(self.check3)
        self.vbox.addStretch(1)
        self.group1.setLayout(self.vbox)
        # Layout pour le groupe
        self.vbox2 = QVBoxLayout()
        self.vbox2.addWidget(self.group1)
        self.dialog.setLayout(self.vbox2)
        # afficher
        self.dialog.show()
        self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
        self.exec_()
if __name__ == "__main__" :
    app=Demo(sys.argv)
```

QRadioButton

Description

La classe QRadioButton permet de mettre en place un bouton radio avec soit un label textuel soit une image (pixmap) à côté. Le label textuel peut être paramétré par le constructeur et par un appel à la méthode setText(), tandis que l'image peut être ajoutée par setPixmap(). Ce bouton peut avoir deux états, soit coché (checked), soit décoché (unchecked). Le changement d'état peut être récupéré avec le signal toggled(), tandis qu'il est possible de récupérer l'état de la case avec la méthode isChecked().

La différence principale entre QRadioButton et QCheckBox est que QRadioButton ne permet qu'un seul et unique choix tandis que QCheckBox permet de sélectionner plusieurs cases. C'est-à dire que dans le cadre d'un groupe de boutons (représenté avec la classe QButtonGroup) un seul bouton radio pourra être sélectionné.

Constructeur

```
# Constructeurs pour la classe QRadioButton
QRadioButton ( QWidget parent, str name = "" )
QRadioButton ( Qstr text, QWidget parent, str name = "" )
```

Exemple

PyQt3.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qradiobutton.py
# Programme exemple pour la classe QRadioButton
from qt import *
import sys
class Demo(QApplication) :
   def init (self, args) :
      QApplication. init (self,args)
      # widget principal, il s'agit d'une fenêtre de dialogue
      self.dialog = QDialog(None, "Dialog")
      #Astuce : on peut grouper les boutons gr^ace à un
      # QButtonGroup ; positionnement vertical des boutons
      self.group1=QButtonGroup(1, Qt.Vertical, None, "group1")
      self.group1.setTitle("Essai QRadioButton")
      # ajout de trois boutons
      # NOTE : le widget QButtonGroup se redimensionne automatiquement
      self.radio1=QRadioButton("radio 1", self.group1, "radio1")
      self.radio2=QRadioButton("radio 2", self.group1, "radio2")
      self.radio3=QRadioButton("radio 3", self.group1, "radio3")
      self.radio2.setChecked(True)
      self.group1.show()
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.exec loop()
if name == "__main__" :
   app=Demo(sys.argv)
```

PyQt4.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qRadiobox.py
# Programme exemple pour la classe QRadioButton#
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
class Demo(QApplication) :
   def __init__(self, args) :
      QApplication.__init__(self,args)
      # widget principal, il s'agit d'une fenêtre de dialogue
      self.dialog = QDialog()
      self.dialog.setWindowTitle("Dialog")
      #Astuce : on peut grouper les boutons grâce à un
      # QGroupBox
      self.group1=QGroupBox("Essai QRadioButton", self.dialog)
      self.group1.setObjectName("group1")
      # définition de 3 Radioboxes
      # NOTE : le widget Q3ButtonGroup se redimensionne automatiquement
      self.Radio1=QRadioButton("Radiobox 1")
      self.Radio2=QRadioButton("Radiobox 2")
      self.Radio3=QRadioButton("Radiobox 3")
      # le bouton Radio3 n'est pas modifiable
      self.Radio3.setDisabled(True) ;
      # espace entre les boutons
      self.vbox = QVBoxLayout()
      self.vbox.addWidget(self.Radio1)
      self.vbox.addWidget(self.Radio2)
      self.vbox.addWidget(self.Radio3)
      self.vbox.addStretch(1)
      self.group1.setLayout(self.vbox)
      self.vbox2 = QVBoxLayout()
      self.vbox2.addWidget(self.group1)
```

```
      self.dialog.setLayout(self.vbox2)
      self.dialog.show()
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.exec_()
if __name__ == "__main__" :
   app=Demo(sys.argv)
```

QComboBox

Description

Cette classe permet d'afficher une liste d'éléments sous forme de liste déroulante avec, lorsque la liste n'est pas déroulée, uniquement l'élément sélectionné affiché. Il est possible de construire un objet permettant l'édition des éléments ou non (l'utilisateur peut introduire ses propres éléments) en spécifiant un paramètre au constructeur. De plus, un tel objet peut contenir des éléments textuels ou des images (pixmap). Un objet QComboBox émet deux signaux : `activated()` lorsqu'un élément a été sélectionné et `highlighted()` lorsqu'un élément est mis en évidence dans la liste.

Constructeur

```
# Constructeurs pour la classe QComboBox
QComboBox ( QWidget parent = None, str name = "" )
QComboBox ( bool rw, QWidget parent = None, str name = "" )
```

Exemple

PyQt3.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qcombobox.py
# Programme exemple pour la classe QComboBox
from qt import *
import sys
class Demo(QApplication) :
   def init (self,args) :
      QApplication. init (self, args)
      # widget principal, il s'agit d'une fen^etre de dialogue
      self.dialog = QDialog(None, "Dialog")
      self.cbox1 = QComboBox(self.dialog, "cbox1")
      self.cbox1.insertItem("choix 1")
      self.cbox1.insertItem("choix 2")
      self.cbox1.insertItem("choix 3")
      # l'utilisateur peut entrer son propre texte
      self.cbox1.setEditable(True)
      # choix par défaut : le troisième ( ! offset by one !)
      self.cbox1.setCurrentItem(2)
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.dialog.show()
```

```
      self.exec_loop()
if __name__ == "__main__" :
   x = Demo(sys.argv)
```

PyQt4.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qCombobox.py
# Programme exemple pour la classe QComboBox#
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
class Demo(QApplication) :
        def __init__(self, args) :
                QApplication.__init__(self,args)
                # widget principal, il s'agit d'une fenêtre de dialogue
                self.dialog = QDialog()
                self.dialog.setWindowTitle("Dialog")
                self.Combo1=QComboBox()
                self.Combo1.setObjectName("Liste 1")
                self.Combo2=QComboBox()
                self.Combo1.addItem ("choix 1")
                self.Combo1.addItem ("choix 2")
                self.Combo1.addItem ("choix 3")
                self.Combo2.setObjectName("Liste 2")
                self.Combo3=QComboBox()
                self.Combo3.setObjectName("Liste 3")
                # le bouton Combo3 n'est pas modifiable
                self.Combo3.setDisabled(True) ;
                # espace entre les boutons
                self.vbox = QVBoxLayout()
                self.vbox.addWidget(self.Combo1)
                self.vbox.addWidget(self.Combo2)
                self.vbox.addWidget(self.Combo3)
                self.vbox.addStretch(1)
                self.dialog.setLayout(self.vbox)
                self.dialog.show()
                self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
                self.exec_()
if __name__ == "__main__" :
   app=Demo(sys.argv)
```

QPushButton

Description

La classe QPushButton met à disposition un bouton de commande. C'est certainement l'un des élément graphique le plus utilisé dans une interface pour réaliser des actions de la part de l'utilisateur.

Cet élément permet d'afficher du texte ou une image, ainsi qu'optionnellement une petite icône. Le texte peut afficher la touche accélératrice grâce à l'utilisation du caractère & devant la lettre symbolisant la touche. Toutes ces propriétés peuvent soit être mises en place grâce au constructeur de la classe, soit par l'intermédiaire d'appel à des méthodes sur l'objet. Il est aussi possible de désactiver le bouton. L'utilisateur

ne pourra alors plus cliquer dessus, et l'aspect graphique du bouton sera tel que l'utilisateur est informé de la non-activation du bouton (généralement le texte du bouton est grisé, mais cela dépend du système de fenêtrage utilisé).

Un bouton poussoir émet le signal `clicked())` quand il est activé par un clic de la souris ou bien par le clavier. Mais les boutons mettent aussi à disposition les signaux pressed() (pressé) et released() (relaché), qui sont nettement moins utilisé en pratique.

Les boutons contenus dans des boîtes de dialogue sont définis comme boutons par défaut, à savoir qu'ils obtiennent automatiquement le focus clavier lors de l'ouverture de la boîte de dialogue. L'utilisateur n'a alors qu'à presser sur « Entrée » pour activer le bouton. Ce comportement est modifiable par la méthode setAutoDefault(). Il est aussi possible de modifier le comportement du bouton pour qu'il reste activé lorsque l'utilisateur clique dessus. Ce comportement est mis en place grâce à la méthode setToggleButton(). Il est aussi possible de faire en sorte que le bouton répète son action tant que l'utilisateur le maintient cliqué ou activé par le clavier. Pour cela nous vous laissons consulter la documentation de référence de la classe.

Une variante d'un bouton de commande est un bouton menu. Il met alors à disposition plusieurs actions, qui sont choisies grâce à un menu déroulant affiché lorsque l'utilisateur clique sur le bouton. La méthode setPopup()) permet d'associer un menu déroulant à un bouton. Pour la création d'un menu déroulant, référez-vous à la section parlant de la classe QPopupMenu.

Constructeur

```
# Constructeurs pour la classe QPushButton
QPushButton ( QWidget parent, str name = "" )
QPushButton ( Qstr text, QWidget parent, str name = "" )
QPushButton ( QIconSet icon, Qstr text, QWidget parent, str name = "" )
```

Exemple

PyQt3.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qpushbutton.py
# Programme exemple pour la classe QPushButton
from qt import *
import sys
class Demo(QApplication) :
   def init (self, args) :
      QApplication. init (self,args)
      # widget principal, il s'agit d'une fenêtre de dialogue
      self.dialog = QDialog(None, "Dialog")
      # Astuce : on peut grouper les boutons grâce à un
      # QButtonGroup ; positionnement horizontal des boutons
      self.group1=QButtonGroup(1, Qt.Horizontal, None, "group1")
      self.group1.setTitle("Essai QPushButton")
      # définition de 3 boutons
      # NOTE : le widget QButtonGroup se redimensionne automatiquement
      # NOTE 2 : & permet de définir la touche accélératrice (Alt+B pour
      # le premier bouton)
      self.push1=QPushButton("&Bouton 1", self.group1, "push1")
      self.push2=QPushButton("Bou&ton 2", self.group1, "push2")
      self.push3=QPushButton("Bouton &3", self.group1, "push3")
```

```
      # le bouton push1 est de type ”toggle”
      self.push1.setToggleButton(True)
      # le bouton push2 est un bouton menu
      self.menu = QPopupMenu(None, "menu1")
      self.menu.insertItem("Menu 1")
      self.menu.insertItem("Menu 2")
      self.push2.setPopup(self.menu)
      # le bouton push3 n'est pas modifiable
      self.push3.setDisabled(True)
      # espace entre les boutons
      self.group1.addSpace(3)
      self.group1.show()
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.exec loop()
if name == "__main__" :
   app=Demo(sys.argv)
```

PyQt4.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qpushbox.py
# Programme exemple pour la classe QpushBox#
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
class Demo(QApplication) :
   def __init__(self, args) :
      QApplication.__init__(self,args)
      # widget principal, il s'agit d'une fenêtre de dialogue
      self.dialog = QDialog()
      self.dialog.setWindowTitle("Dialog")
      #Astuce : on peut grouper les boutons grâce à un
      # QGroupBox
      self.group1=QGroupBox("Essai QpushBox", self.dialog)
      self.group1.setObjectName("group1")
      # NOTE : le widget Q3ButtonGroup se redimensionne automatiquement
      self.push1=QPushButton("push 1")
      self.push2=QPushButton("push 2")
      self.push3=QPushButton("push 3")
      # le bouton push3 n'est pas modifiable
      self.push3.setDisabled(True) ;
      # espace entre les boutons
      self.vbox = QVBoxLayout()
      self.vbox.addWidget(self.push1)
      self.vbox.addWidget(self.push2)
      self.vbox.addWidget(self.push3)
      self.vbox.addStretch(1)
      self.group1.setLayout(self.vbox)
      self.vbox2 = QVBoxLayout()
      self.vbox2.addWidget(self.group1)
      self.dialog.setLayout(self.vbox2)
      self.dialog.show()
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.exec_()
if __name__ == "__main__" :
   app=Demo(sys.argv)
```

QPopupMenu

Description

La classe QPopupMenu met à disposition un menu déroulant. Un tel menu peut être ajouté soit à un bouton (QPushButton), soit à une barre de menu (QMenuBar), soit être un menu de contexte affiché lorsque l'utilisateur clique avec le bouton droit de la souris dans une zone définie. Il est possible d'ajouter facilement des éléments à un menu par l'appel de la méthode insertItem(), qui permet d'ajouter un texte, une image, ou n'importe quel élément mettant à disposition des méthodes d'affichage (voir la classe QCustomMenuItem pour cela). De plus, tous ces éléments peuvent avoir une icône et une touche accélératrice.

Il est aussi possible d'ajouter des séparateurs (`insertSeparator()`) ou bien des sous-menus en passant à `insertItem()` un objet définissant lui même un menu déroulant. Cette classe offre de multiples possibilités. Il est possible d'ajouter des éléments cochables dans un menu (la méthode `setCheckable()` sur un élément du menu), mais un menu peut aussi contenir un widget entier, comme par exemple un sélecteur de couleur. Une autre fonctionnalité intéressante est la possibilité de déclarer le menu comme tear-off, à savoir qu'il est possible de détacher le menu de son emplacement de départ pour le déplacer à un autre endroit à l'écran et le laisser ouvert tout le temps pour accéder rapidement à ses fonctions. Toutes ces fonctionnalités sont là pour fournir un maximum de flexibilité. Pour en savoir plus, consultez la page de référence à l'adresse http://doc.trolltech.com/3.3/qpopupmenu.html.

Constructeur

```
# Constructeur pour la classe QPopupMenu
QPopupMenu ( QWidget parent = None, str name = "" )
```

Exemple

PyQt3.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qpopupmenu.py
# Programme exemple pour la classe QPopupMenu
from qt import *
import sys
class Demo(QApplication) :
   def init (self, args) :
      QApplication. init (self,args)
      # widget principal, il s'agit d'une fenêtre de dialogue
      self.dialog = QDialog(None, "Dialog")
      #Barre de menu
      self.bar=QMenuBar(self.dialog, "Menubar")
      #définition d'un premier menu
      self.menu1 = QPopupMenu(None, "menu1")
      self.menu1.insertItem("&Menu 1")
      self.menu1.insertItem("M&enu 2")
      self.menu1.insertSeparator()
      ret = self.menu1.insertItem("Me&nu 3")
      self.menu1.setItemEnabled(ret, False)
      #définition d'un second menu avec sous menu
```

```
        self.menu2 = QPopupMenu(None, "menu2")
        self.menu2.insertTearOffHandle()
        self.menu2.insertItem("Menu 1")
        self.menu2.insertItem("Sous menu", self.menu1 )
        #Il est possible d'ajouter n'importe quel widget au menu !
        self.menu2.insertItem(QCheckBox("checkbox 1", self.menu2, "check1"))
        #ajout des menus à la barre
        self.bar.insertItem("&Bar 1", self.menu1)
        self.bar.insertItem("B&ar 2", self.menu2)
        self.dialog.show()
        self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
        self.exec_loop()
if __name__ == "__main__" :
    app=Demo(sys.argv)
```

PyQt4.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qmenu.py
# Programme exemple pour la classe QMenu
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
class Demo(QApplication) :
        def __init__(self, args) :
                QApplication.__init__(self,args)
                # widget principal, il s'agit d'une fenêtre de dialogue
                self.dialog = QDialog()
                self.dialog.setWindowTitle("Dialog")
                #Barre de menu
                self.bar=QMenuBar()
                #définition d'un premier menu
                self.menu1 = QMenu("menu1")
                self.menu1.addMenu("&Menu 1")
                self.menu1.addMenu("M&enu 2")
                ret = self.menu1.addMenu("Me&nu 3")
                self.vbox = QVBoxLayout()
                self.vbox.addWidget(self.menu1)
                self.vbox.addStretch(1)
                self.dialog.setLayout(self.vbox)
                self.dialog.show()
                self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
                self.exec_()
if __name__ == "__main__" :
    app=Demo(sys.argv)
```

QLineEdit

Description

QLineEdit est un widget représentant une zone d'édition de texte sur une seule ligne. Il permet à l'utilisateur de saisir et modifier du texte avec des fonctions d'édition utiles telles que la possibilité d'annuler et refaire une action, la gestion du copier-coller et du glisser-déposer, ainsi qu'une multitude de raccourcis clavier facilitant l'édition de texte.

Il est possible de transformer la zone de saisie de texte en zone lecture seule grâce à la méthode setReadOnly() ou bien de modifier la manière dont le texte introduit est affiché par un appel à la méthode echoMode() (par exemple ne rien afficher si on demande un mot de passe à l'utilisateur). Ce widget gère aussi les contraintes sur le texte entré, comme par exemple une longueur maximale (méthode maxLength()), ou des contraintes plus générales comme la validation. Elle est mise en place par l'intermédiaire de la méthode setValidator()) (prenant en paramètre un objet de la classe QValidator) ou par l'utilisation d'un masque de saisie avec la méthode setInputMask(). Le texte peut être modifié grâce aux méthodes setText() ou insert(), tandis qu'il peut être récupéré par la méthode text(). Le texte affiché, qui peut être différent du texte entré en fonction du mode d'écho, est quant à lui récupéré grâce à displayText(). Vous pouvez effectuer des sélections sur le contenu avec les méthodes setSelection() et selectAll(), sur lesquelles il est possible d'appliquer les traditionnels couper/copier/coller avec cut(), copy() et paste(). Le texte peut être aligné avec la méthode setAlignment().

Lorsque le texte est modifié par l'utilisateur, le signal textChanged() est émis. Lorsqu'il appuie sur la touche Entrée, c'est returnPressed() qui est émis. Cependant, si une validation du texte est active le signal est émis uniquement si le texte est valide.

Comme expliqué ci-dessus il y a une multitude de raccourcis clavier disponibles sur cette zone de texte. Pour en obtenir la liste, ainsi que tous les détails concernant ce widget, consultez la documentation de référence à l'adresse http://doc.trolltech.com/3.3/qlineedit.html. Un widget relatif à cette zone de saisie est QTextEdit, qui fournit une zone de saisie de texte sur plusieurs lignes et avec des possibilités de mise en page plus importantes. Consultez la section suivante pour obtenir plus d'informations à son sujet.

Constructeur

```
# Constructeurs pour la classe QLineEdit
QLineEdit ( QWidget parent, str name = "" )
QLineEdit ( QString contents, QWidget parent, str name = "" )
QLineEdit ( QString contents, QString inputMask, QWidget parent, str name = "" )
```

Exemple

PyQt3.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qlineedit.py
# Programme exemple pour la classe QLineEdit
from qt import *
import sys
class Demo(QApplication) :
   def init (self,args) :
      QApplication. init (self, args)
      # widget principal, il s'agit d'une fenêtre de dialogue
      self.dialog = QDialog(None, "Dialog")
      # Grille de 2 lignes pour l'affichage
      self.grid = QGrid(2, Qt.Vertical, self.dialog)
      self.grid.setMinimumWidth(400)
      self.grid.setMinimumHeight(100)
      # première zone de saisie de texte
      self.line1 = QLineEdit(self.grid, "line1")
      self.line1.setMinimumWidth(400)
      self.line1.setMaxLength(50)
```

```
        self.line1.setText("Entrez votre texte ici")
        # seconde zone, en lecture seule
        self.line2 = QLineEdit(self.grid, "line2")
        self.line2.setReadOnly(True)
        self.line2.setAlignment(Qt.AlignHCenter)
        # le texte entré dans la première zone est copié dans la seconde
        self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
        self.connect(self.line1,SIGNAL("textChanged(const QString&)"),
        self.line2,SLOT("setText(const QString &)"))
        self.dialog.show()
        self.exec loop()
if name == "__main__" :
    x = Demo(sys.argv)
```

PyQt4.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qlineedit.py
# Programme exemple pour la classe QLineEdit
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import sys
class Demo(QApplication) :
        def __init__(self, args) :
                QApplication.__init__(self,args)
                # widget principal, il s'agit d'une fenêtre de dialogue
                self.dialog = QDialog()
                self.dialog.setWindowTitle("Dialog")

                self.grid = QGridLayout(self.dialog)
                self.grid.setColumnMinimumWidth(400, 400)
                self.grid.setRowMinimumHeight(100, 100)

                # première zone de saisie de texte
                self.line1 = QLineEdit("line1")
                self.line1.setMaxLength(50)
                self.line1.setText("Entrez votre texte ici")

                # seconde zone, en lecture seule
                self.line2 = QLineEdit("line2")
                self.line2.setReadOnly(True)
                self.line2.setAlignment(Qt.AlignHCenter)

                self.grid.addWidget(self.line1)
                self.grid.addWidget(self.line2)
                self.dialog.setLayout(self.grid)
                self.dialog.show()
                self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
                self.exec_()

if __name__ == "__main__" :
    app=Demo(sys.argv)
```

QTextEdit

Description

QTextEdit est un widget très puissant mettant à disposition une zone d'édition de texte d'une page supportant du contenu mis en forme. Il a été créé dans le but d'être le plus efficace possible, même avec beaucoup de contenu, et répondre le plus rapidement possible aux actions de l'utilisateur. Il fonctionne principalement selon 4 modes :

- éditeur de texte standard : texte standard sans mise en forme ; même si le texte contient des balises de mise en forme, lors de la lecture du contenu du widget les balises sont supprimées ;
- éditeur de texte avec mise en forme : texte supportant la mise en forme ;
- afficheur de texte : dans ce mode il n'est pas possible de modifier le texte, seulement de le consulter ;
- afficheur de fichiers logs : ce mode est identique à l'affichage de texte, seule la mise en forme du texte est désactivée.

Vu la quantité assez importante de méthodes et de fonctionnements nous ne pouvons que vous conseiller de consulter la documentation de référence disponible à l'adresse http://doc.trolltech.com/3.3/qtextedit.html.

Constructeur

```
# Constructeurs pour la classe QTextEdit
QTextEdit ( QString text, QString context = QString.null, QWidget parent = None, str
name = "" )
QTextEdit ( QWidget parent = None, str name = "" )
```

Exemple

PyQt3.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qtextedit.py
# Programme exemple pour la classe QTextEdit
from qt import *
import sys
class Demo(QApplication) :
   def init (self, args) :
      QApplication. init (self,args)
      # zone d'édition de texte comme widget principal
      self.textedit = QTextEdit()
      self.textedit.resize(640,480)
      # dialogue d'ouverture de fichier
      s = QFileDialog.getOpenFileName("", "", None,
      "Boite d'ouverture de fichier", "Choisissez un fichier")
      self.file = QFile (s)
      # essaie d'ouvrir le fichier et de le charger
      if self.file.open(IO ReadOnly) :
         self.stream = QTextStream(self.file)
         self.textedit.setText(self.stream.read())
      else :
         print "Impossible d'ouvrir le fichier " + self.file
         b = QMessageDialog.warning(None, "Erreur",
         "Impossible d'ouvrir le fichier sélectionné", QMessageBox.Ok)
         exit(-1)
      self.textedit.show()
      self.connect(self,SIGNAL("lastWindowClosed()"),self,SLOT("quit()"))
      self.exec loop()
if name == "__main__" :
```

```
    app=Demo(sys.argv)
```

PyQt4.*

```
#!/usr/bin/python
# -*- coding : utf-8 -*-
#
# qtextedit.py
# Programme exemple pour la classe QTextEdit
import os, sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *

class MyWindow(QWidget):
    def __init__(self, *args):
        QWidget.__init__(self, *args)
        label = QLabel(self.tr(u'Entrer une commande Shell et presser Entrée'))
        self.LineEdit = QLineEdit()
        self.TextEdit = QTextEdit()
        vbox = QVBoxLayout(self)
        vbox.addWidget(label)
        vbox.addWidget(self.LineEdit)
        vbox.addWidget(self.TextEdit)
        self.setLayout(vbox)

        # create connection
        self.connect(self.LineEdit, SIGNAL("returnPressed(void)"),self.run_command)

    def run_command(self):
        cmd = str(self.LineEdit.text())
        stdouterr = os.popen4(cmd)[1].read()
        self.TextEdit.setText(stdouterr)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    w = MyWindow()
    w.show()
    sys.exit(app.exec_())
```

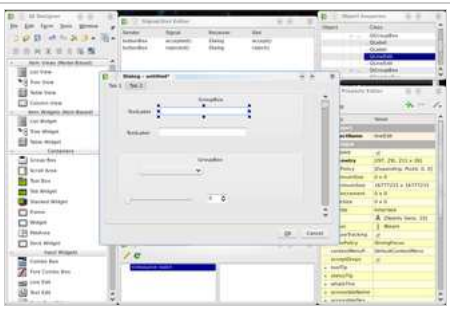
Références

- http://www.saltycrane.com/blog/2007/12/pyqt-example-how-to-run-command-and/

Utilisation de Qt Designer

La librairie Qt vient aussi avec une application graphique permettant de définir rapidement et simplement son interface. La plupart des propriétés des éléments sont directement modifiables avec cette application, ainsi que la définition des signaux et des slots. Il est ensuite très simple de générer du code C++ pour l'interface. Comme nous le verrons ci-dessous, il est aussi tout à fait possible de générer du code Python.

Nous ne pouvons que vous conseiller d'utiliser cet outil. En effet, non seulement il offre une vue d'ensemble de tous les widgets disponibles, mais il permet en plus de modifier facilement leurs propriétés et leur mise


Qt Designer 4.4.3

en page. Comme nous l'avons vu précédemment, effectuer le placement des éléments à la main est quelque peu laborieux.

Utilisation de Qt Designer/Création d'une liste de commission

Afin de se familiariser avec l'application Qt Designer nous allons vous montrer comment créer une petite application de gestion d'une liste de commission. Ceci constitue un bon exemple d'utilisation de l'interface créée à l'aide de Qt Designer.

Création de l'interface

L'entrée New... du menu Edit permet de créer un nouveau fichier. Le but étant de créer la fenêtre principale de l'application, il faut choisir l'entrée Main Window.

En choisissant les options par défaut, une barre de menu standard est créée, cependant pour cette petite application seules les entrées de menu Fichier et Édition seront nécessaires, par conséquent toutes les autres options par défaut sont décochées. Le menu d'accès aux propriétés des composants permet de renommer ces composants d'une manière plus explicite afin de pouvoir y faire référence dans le code de l'application. Le champ name contient le nom de votre composant en tant que nom d'objet tandis que le champ caption contient le nom de l'objet qui s'affichera dans l'interface.

L'interface de l'application se construit en sélectionnant les composants dans la barre des composants.

La liste de commissions se compose des widgets suivants :

Widget	Name	Caption
ListBox	lstPanier	-
ListEdit	élément	-
PushButton	bntAjouter	Ajouter
PushButton	bntModifier	Modifier
PushButton	bntSupprimer	Supprimer
PushButton	bntQuitter	Quitter

Le placement des widgets s'effectue très facilement à l'aide de la souris ou du clavier.

Gestion du redimensionnement

Qt Designer permet de gérer le positionnement des éléments de la fenêtre lors d'un redimensionnement.

Le menu Layout permet de sélectionner un alignement vertical, horizontal ou sur toute la grille. Dans notre cas, sélectionnez les widgets lneCourse et btnAjouter et sélectionnez l'entrée Lay Out Horizontally.

Pour les boutons btnModifier, btnSupprimer, BtnQuitter, choisir Lay Out Vertically. Les boutons

précédemment énumérés et l'élément lstPanier sont alignés horizontalement. Pour finir, en choisissant Lay Out in a grid lorsque la fenêtre principale est sélectionnée, permet d'aligner l'ensemble des widgets afin qu'ils occupent toute la fenêtre lors d'un redimensionnement.

Désactiver certains boutons

Qt Designer permet de désactiver les boutons lors du démarrage de l'application, l'utilisateur ne pouvant donc pas cliquer dessus. Cela permet de créer une interface cohérente et intuitive, les boutons étant réactivés en fonctions des interactions de l'utilisateur avec l'interface. La fenêtre Property Editor de Qt Designer permet de désactiver un bouton en sélectionnant False pour la valeur de enabled.

L'interface est achevée, les boutons btnAjouter, btnModifier et btnSupprimer sont désactivés dans le menu propriété.

Connexion d'un slot

Qt Designer permet de connecter des widgets de manière simple et rapide. Une connexion s'effectue en sélectionnant dans le menu Edit l'entrée Connexions... ou en sélectionnant le bouton Connect Signal/Slots dans la barre d'outil. Si cette dernière option est choisie il suffit de choisir le widget émetteur du signal et de le relier au widget récepteur, en cliquant sur l'émetteur sans relâcher le clic tout en se dirigeant vers le récepteur. Dans le cadre de notre liste de commission, le widget émetteur est le bouton Quitter tandis que le widget récepteur est la fenêtre principale. Une boîte de dialogue permet de choisir les slots correspondants, ainsi dans le cas de notre application, lorsque l'utilisateur clique sur le bouton Quitter, la fenêtre principale se ferme. Cela permet par exemple de connecter la LineEdit avec la ListBox permettant ainsi de remplir la ListBox avec les champs introduits dans la LineEdit lors d'un clique sur la touche « Return ».

Création d'un QAction dans Qt Designer

Les boutons peuvent être connectés directement à un slot comme c'est le cas du bouton quitter, mais il est également possible de définir des groupes d'action sous la forme de QAction. Ce widget fournit un signal permettant de manipuler des événements multiples comme des boutons, des entrées de menus. Quatre QAction seront définies pour l'application permettant de gérer les boutons et entrées de menus Ajouter, Modifier, Supprimer et Quitter. Ainsi, la même action s'effectuera lorsque l'utilisateur cliquera sur le bouton Ajouter ou choisira le menu correspondant. Une action se crée à l'aide de la fenêtre Action Editor qui se trouve dans le menu Window > View. Une nouvelle action s'affiche en cliquant sur le bouton New. Les paramètres de ces actions se règlent dans la fenêtre Property Editor.

Remarque : Lors de la création du document, les menus Fichier et Edition ont été créés par Qt Designer, créant également des actions. Ces actions n'étant pas nécessaires, elles sont supprim ées de la liste des actions et par conséquent les entrées des menus sont également supprimées. Les nouvelles entrées des menus sont définies par des actions, l'action aQuitter est placée dans le menu Fichier tandis que les trois autres sont dans le menu Edition. Le placement s'effectue par un glissé-déposé de l'action dans le menu souhaité. Les QAction doivent être connectées à la fenêtre principale. Des slots particuliers sont créés pour ces actions.

Création de ses propres slots avec les QAction

Il peut être utile de créer ses propres slots, dans notre cas, nous avons besoins de quatre slots afin d'ajouter un élément, de le modifier, de le supprimer ou de quitter l'application.

Le menu Edit > Slots permet d'écrire ses propres slots. Les slots privés ajouter(), modifier(), supprimer(),

quitter() sont ajoutés.

Connexion des slots avec les QAction

Les connexions entre les widgets ont déjà été abordées, il suffit d'ouvrir la fenêtre d'édition des connexions (menu Edit) et de créer les connexions suivantes :

Sender	Signal	Receiver	Slot
aJouter	actived()	frmMain	ajouter()
aModifier	actived()	frmMain	modifier()
aSupprimer	actived()	frmMain	supprimer()
aQuitter	actived()	frmMain	quitter()

Générer du code Python avec pyuic

Le logiciel pyuic, fourni avec PyQt, permet de traduire en langage Python les informations du fichier xml *.ui généré par Qt Designer.

```
pyuic frmMain.ui > frmMain.py
```

Le fichier Python contient le canevas des slots définis précédemment, il ne reste plus qu'à écrire le corps de ces méthodes. Un fichier 1 fourni avec ce rapport contient l'implémentation des slots.

Traduction de Qt à PyQt

À ce stade du guide vous devriez déjà avoir un bon aperçu des possibilités de la librairie PyQt. Cependant, il existe encore une multitude de classes Qt disponibles, permettant entre autre de gérer les dates, de se connecter à une base de données, de faire du traitement d'images, etc. Néanmoins, l'explication de ces classes sort du cadre de ce document.

Nous vous proposons ici une méthode vous permettant d'appréhender rapidement la documentation Qt afin d'écrire du code python. En effet, il est relativement aisé d'interpréter la documentation de référence en Python étant donné que la librairie Qt utilise toutes les possibilités de l'orienté objet mis à disposition par le langage C++. Avec les quelques explications qui suivent vous devriez être en mesure de comprendre l'ensemble de la documentation de référence Qt (http://web.archive.org/web/20031226145340/http://doc.trolltech.com/3.3/groups.html et http://qt-project.org/doc/qt-4.8/classes.html) et créer ainsi vos propres exemples.

S'y retrouver dans la documentation de référence

La documentation de référence contient une multitude de documents, comme entre autre la documentation de référence sur l'API (regroupée par groupes, annotations, héritages, etc.), quelques tutoriaux de base, des explications plus avancées sur les concepts de la librairie, les utilitaires disponibles, etc. La partie qui nous intéresse ici pour obtenir de la documentation sur les modules disponibles est la section intitulée API Reference. Elle contient les sections suivantes :

- All Classes : Contient la liste de l'ensemble des classes disponibles dans Qt, avec pour chaque classe un lien vers sa documentation. Utile si vous connaissez le nom de la classe qui vous intéresse.
- Main Classes : Liste des classes les plus utilisées.
- Grouped Classes : Permet de naviguer dans les classes par regroupement de classes ayant un domaine d'action similaire (groupes pour les widgets de base, la gestion du temps, etc.).
- Annotated Classes : Permet d'obtenir l'ensemble des classes avec pour chacune une brève description.
- Inheritance Hierarchy : Liste représentant les relations d'héritage C++ entre les diverses classes.
- Class Chart : Graphique représentant la hiérarchie de l'ensemble des classes disponibles ; possibilité de cliquer sur le nom d'une classe pour obtenir directement sa documentation.
- All Functions : Ensemble des fonctions membres de toutes les classes de la librairie, classées par ordre alphabétique.
- Header File Index : Liens vers les fichiers d'entêtes C++; peu utile dans notre cas car la plupart des informations à connaître pour une classe sont indiquées dans la documentation.
- FAQs : Foire aux questions sur la librairie.
- Change History Document : Indique les modifications effectuées à la librairie Qt à chaque nouvelle version.

Importer les librairies

Comme vous avez sûrement dû le remarquer, la première étape consiste en l'importation des librairies Qt. Elle est effectuée par l'intermédiaire des lignes suivantes :

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

Il est certes peu performant pour le démarrage de votre application d'importer l'ensemble des librairies Qt, cependant, pour le côté pratique de la chose nous le recommandons tout de même. Libre à vous de chercher après coup les modules utilisés et d'inclure seulement ceux nécessaires. Soyez néanmoins conscient que l'ensemble de la librairie Qt sera chargée en mémoire, car la version 3 est encore une librairie monolithique. La version 4 ne souffre plus de ce « problème ».

Comprendre la documentation d'une classe

La page de documentation contient beaucoup d'informations concernant la classe. Elle commence par une petite explication sur ce que fait la classe sélectionnée. Vient ensuite une ligne indiquant le fichier d'entête C++ à inclure. Par exemple :

```
#include <QCheckbox>
```

Cette ligne est simplement à ignorer en Python. Elle indique en effet au compilateur C++ dans quel (s) fichier (s) sont définis les prototypes des classes utilisées. Vous pouvez par contre vous baser sur cette information pour créer les lignes d'importation des modules Python.

Viennent ensuite les méthodes disponibles pour la classe. Les méthodes qui ne retournent aucune valeur et dont le nom est identique à la classe sont les constructeurs. Il en existe plusieurs, permettant de mettre des paramètres à l'objet nouvellement créé. Afin de les utiliser en Python il suffit de les appeler avec les bons paramètres. Ne prenez pas garde au mot clé const devant un paramètre, de même que les caractères & et * qui n'ont pas de sens en Python. Ce qui est important est uniquement le type du paramètre.

Lorsqu'un paramètre contient le caractère = cela veut dire qu'il a une valeur par défaut si elle n'est pas spécifiée dans l'appel au constructeur. Si vous désirez prendre la valeur par défaut d'un paramètre il faudra le spécifier avec le mot clé réservé None. Cependant, vous pouvez ne pas spécifier les paramètres par défaut après le dernier paramètre vous intéressant. Un exemple est le suivant :

```
grp = QCheckBox("checkbox 1", None, "check1")
```

Le reste des méthodes sont des méthodes applicables à la classe, à appeler soit en préfixant la méthode par l'objet sur lequel vous désirez l'appliquer, soit en utilisant le mot réservé self qui indique l'objet courant. La documentation de référence liste ensuite les signaux et les slots disponibles. Il y a peu de choses à retenir à ce sujet à part le fait que pour leur utilisation il faut obligatoirement utiliser le prototype C++ et non pas le convertir en python. Par exemple, si vous avez un signal défini comme tel :

```
void dockWindowPositionChanged ( QDockWindow * dockWindow )
```

Son utilisation dans une méthode `connect()` se fera de la manière suivante :

```
app.connect(qApp, SIGNAL("dockWindowPositionChanged(QDockWindow *)"),
qApp,SLOT("quit()"))
```

Code C++ en Python

Dans la documentation de référence il se trouve quelques exemples mettant en oeuvre des classes Qt. Afin de traduire le code C++ en Python, vous pouvez appliquer les quelques règles suivantes :

- les commentaires `/* ... */` et `// ...` sont à remplacer par des commentaires python `# ...` placés après les premières lignes (shebang line et encodage du fichier) ;
- les doubles deux-points (`::`) ainsi que les flèches (`->`) sont à remplacer par un point (`.`) ;
- supprimer les mots clés new ;
- remplacer | par or et & par and ;
- adapter les appels de méthodes à la syntaxe python.

Sous-classe implantant une classe Qt

La méthode la plus simple pour utiliser un objet d'une classe spécifique est de créer directement un objet de cette classe. Il suffit de faire appel au constructeur de la classe en question en lui passant les paramètres corrects pour obtenir un objet. Cependant, il peut être utile de définir vos propres classes qui héritent d'une ou de plusieurs classes Qt afin de spécialiser le comportement de certains widgets, voire de définir votre propre widget. Pour cela, l'étape la plus important est l'appel au (x) constructeur (s) de la (des) classe (s) parente (s). Une fois ces appels effectués vous aurez alors l'ensemble des méthodes des parents à disposition. L'utilisation de votre widget n'est ensuite pas différente de l'utilisation de n'importe quel objet Python.

Définir ses propres signaux et slots

Comme vu dans la partie concepts de base, les signaux et les slots sont extrêmement puissants et simples à utiliser. À l'instar de C++ où il faut passer par un compilateur méta-objet pour transformer les signaux et slots en fonctions C++, en Python il n'en est rien. Les signaux sont de simples chaînes de caractères, tandis que pour définir un slot il suffit de définir une méthode sur la classe voulue. Notez tout de même un point

important : si vous désirez dans un de vos propre widget connecter un des signaux et des slots mais que vous n'avez pas de référence vers l'objet QApplication définissant l'application dans son ensemble, vous pouvez utiliser la variable qApp, qui est une référence automatiquement convertie vers QApplication.

Bibliographie

Autres livres

- Programmation Python
- Programmation Qt pour C++

Liens externes

- Site web des paquetages PyQt et SIP (http://www.riverbankcomputing.co.uk/software/pyqt/) [archive]
- Site web de la société Nokia qui développe Qt (http://qt.nokia.com) [archive]
  - Documentation sur Qt (http://doc.qt.nokia.com) [archive]
- Le tutoriel indépendant de Qt (pour Qt version 3) (http://www.digitalfanatics.org/projects/qt_tutorial/fr/index.html) [archive]
- Quelques articles en français publiés dans le magazine GNU/Linux Magazine France (http://chl.be/glmf/kafka.fr.free.fr/articles/) [archive]
- Qtfr - la communauté francophone autour de Qt (http://qtfr.org/) [archive]
- Tutoriel complet (http://www.siteduzero.com/tutoriel-3-11240-introduction-a-qt.html) [archive]
- Télécharger un fichier sur le Web avec PyQt4 (http://pyqt.developpez.com/tutoriels/reseau/telecharger-fichiers/) [archive]
- (anglais) Liste des classes (http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/classes.html) [archive]
- (anglais) PyQt - Getting started - article by devshed (http://www.devshed.com/c/a/Python/PyQT-Getting-Started/) [archive]
- (anglais) GUI Programming with Python : QT Edition (http://www.commandprompt.com/community/pyqt) [archive]
- (anglais) Python, PyQt et Qt Designer : developpement rapide d’applications graphiques (http://dosimple.ch/articles/Python-PyQt/) [archive]
- (anglais) Tutorial : Creating GUI Applications in Python with QT (http://www.cs.usfca.edu/~afedosov/qttut/) [archive]

PyQt versus wxPython

Dans la partie qui suit nous allons tenter d'effectuer un petit comparatif objectif entre les librairies wxPython et PyQt. Ces deux librairies permettent de créer facilement des interfaces graphiques en Python, mais en se basant sur des principes différents. La première différence que l'on remarque entre les librairies Qt et wxWidgets (les libraires C++ de base) concerne la documentation de ces librairies. En effet, Qt semble plus professionnelle car il y a une société derrière, tandis que wxWidgets est une librairie communautaire.

Par conséquent, la documentation de Qt est bien plus fournie et complète que celle de wxWidgets. Néanmoins, ni PyQt ni wxPython ne disposent à l'heure actuelle de véritable documentation de référence, car il est extrêmement simple de passer de C++ à Python. Il faut toutefois noter que la documentation de wxWidgets comporte certaines informations sur wxPython, lorsque par exemple les méthodes de wxPython sont quelque peu différentes de celles en C++. De plus, il existe un livre dédié entièrement à wxPython.

Qt dispose d'un système de gestion des événements extrêmement puissant et pratique. En effet, le système de signaux et slots permet d'affecter à un événement (signal) de multiples actions (slot), et une action peut être activée par de multiples événements. WxPython ne permet malheureusement pas, pour un même événement, d'appeler plusieurs méthodes.

À l'inverse de wxWidgets qui n'est qu'une librairie d'interface, Qt propose un ensemble de classes autre que la gestion d'interfaces. Citons entre autre les bases de données, la gestion de XML, la possibilité de faire des connexions réseaux, etc. Qt est donc bien plus qu'une simple librairie d'interface. Cependant, elle garde sa simplicité d'utilisation.

wxWidgets met à disposition du programmeur un système très performant concernant le placement des éléments graphiques. Les sizers, qui sont un peu difficile à appréhender au départ, permettent de gérer facilement le positionnement et le redimensionnement des éléments, en définissant par exemple que les éléments doivent garder leurs proportions, que certains éléments peuvent s'agrandir ou se rétrécir jusqu'à une certaine limite, etc. De telles possibilités existent aussi en Qt, cependant avec l'utilisation du Qt Designer, il devient très simple de créer et de tester l'interface d'une application.

Nous avons aussi constaté que l'installation de wxPython semble plus simple que PyQt. Cela est certainement dû au fait que la librairie Qt est beaucoup plus bas niveau que wxWidgets. Qt dispose de ces propres routines d'affichage, tandis que wxWidgets se base sur ce qui est fourni par le système d'exploitation sur lequel il tourne.

Néanmoins, pour conclure ce petit comparatif, il ne semble pas qu'une librairie soit meilleure que l'autre. Tout dépend de ce que l'on désire faire. Pour une application avec uniquement une interface graphique la librairie wxPython ira très bien, cependant si on désire faire une application plus complète, avec par exemple la gestion du réseau, PyQt sera un meilleur choix. Notez cependant qu'il est tout à fait possible de créer une application interagissant avec le réseau en wxPython, Python disposant de classes permettant de gérer le réseau. Le seul avantage de Qt sur ce point et qu'il n'y a besoin que d'une seule librairie. Avec les inconvénients que cela génère, comme la lourdeur du chargement, même si Qt en version 4 résout ce problème en découpant la librairie en modules.

termes peuvent s’appliquer.
Voyez les termes d’utilisation pour plus de détails.
Développeurs