Inventaire des widgets PyQt

1. Widget QWidget

a. Propriétés et méthodes de classes

Le meilleur moyen d'aborder un widget PyQt est de s'intéresser d'emblée à la documentation en ligne relative à l'objet Qt lui-même, dont est issu ce widget PyQt.

Dans le cas de `QWidget`, la documentation en ligne se trouve à cette adresse : https://doc.qt.io/qt-5/qwidget.html

Après consultation, on peut rapidement partager les remarques suivantes :

- La classe `QWidget` hérite de plusieurs dizaines de classes qui participent à sa définition.
- La classe `QWidget` possède plusieurs dizaines de propriétés.
- La classe `QWidget` possède plusieurs dizaines de méthodes.

Certaines propriétés ne sont pas définies avec des types élémentaires (les types élémentaires sont en particulier le type booléen `-bool` et le type entier `-int`).

En effet, on peut voir des types comme `QString` qui correspond à l'encapsulation d'un type chaîne de caractères (`string`) en Qt. On cite également le type QSize, qui encapsule les informations relatives à une dimension, c'est-à-dire par exemple le nombre d'éléments d'un tableau.

On rencontre fréquemment en Qt, et donc en PyQt, des encapsulations de types. Ainsi, Qt apparaît comme une couche parfaitement étanche et autonome. On a donc la couche Qt sur laquelle se greffe la couche PyQt, la seule à être utilisée par le code Python lui-même. Ainsi, jamais nous n'utiliserons directement Qt.

Voici ci-dessous la liste exhaustive des propriétés de la classe `QWidget`. On s'aperçoit qu'elles sont très nombreuses : cette liste est donc proposée ici de manière exceptionnelle. L'usage est en effet de consulter méthodiquement la documentation Qt en ligne pour connaître les différentes possibilités offertes par une classe, y compris dans un

contexte de programmation PyQt.

```
- QWidget(QWidget *parent = nullptr, Qt::WindowFlags f = Qt::WindowFlags())
- virtual    ~QWidget()
- bool   acceptDrops() const
- QString      accessibleDescription() const
- QString      accessibleName() const
- QList<QAction *>     actions() const
- void   activateWindow()
- void   addAction(QAction *action)
- void   addActions(QList<QAction *> actions)
- void   adjustSize()
- bool   autoFillBackground() const
- Qpalette::ColorRole backgroundRole() const
- QBackingStore *     backingStore() const
- QSize  baseSize() const
- QWidget *    childAt(int x, int y) const
- QWidget *    childAt(const QPoint &p) const
- QRect  childrenRect() const
- QRegion      childrenRegion() const
- void   clearFocus()
- void   clearMask()
- QMargins     contentsMargins() const
- QRect  contentsRect() const
- Qt::ContextMenuPolicy      contextMenuPolicy() const
- QCursor      cursor() const
- WId    effectiveWinId() const
- void   ensurePolished() const
- Qt::FocusPolicy     focusPolicy() const
- QWidget *    focusProxy() const
- QWidget *    focusWidget() const
- const QFont & font() const
- QFontInfo    fontInfo() const
- QFontMetrics  fontMetrics() const
- QPalette::ColorRole foregroundRole() const
- QRect  frameGeometry() const
- QSize  frameSize() const
- const QRect & geometry() const
- QPixmap   grab(const QRect &rectangle = QRect(QPoint(0, 0),
  QSize(-1, -1)))
```

```
- void   grabGesture(Qt::GestureType gesture, Qt::GestureFlags
  flags = Qt::GestureFlags())
- void   grabKeyboard()
- void   grabMouse()
- void   grabMouse(const QCursor &cursor)
- int    grabShortcut(const QKeySequence &key, Qt::ShortcutContext
  context = Qt::WindowShortcut)
- QGraphicsEffect *   graphicsEffect() const
- QGraphicsProxyWidget *     graphicsProxyWidget() const
- bool   hasEditFocus() const
- bool   hasFocus() const
- virtual bool  hasHeightForWidth() const
- bool   hasMouseTracking() const
- bool   hasTabletTracking() const
- int    height() const
- virtual int   heightForWidth(int w) const
- Qt::InputMethodHints       inputMethodHints() const
- virtual QVariant    inputMethodQuery(Qt::InputMethodQuery    query) const
- void   insertAction(QAction *before, QAction *action)
- void   insertActions(QAction *before, QList<QAction *> actions)
- bool   isActiveWindow() const
- bool   isAncestorOf(const QWidget *child) const
- bool   isEnabled() const
- bool   isEnabledTo(const QWidget *ancestor) const
- bool   isFullScreen() const
- bool   isHidden() const
- bool   isMaximized() const
- bool   isMinimized() const
- bool   isModal() const
- bool   isVisible() const
- bool   isVisibleTo(const QWidget *ancestor) const
- bool   isWindow() const
- bool   isWindowModified() const
- QLayout *     layout() const
- Qt::LayoutDirection layoutDirection() const
- QLocale       locale() const
- QPoint mapFrom(const QWidget *parent, const QPoint &pos) const
- QPoint mapFromGlobal(const QPoint &pos) const
- QPoint mapFromParent(const QPoint &pos) const
- QPoint mapTo(const QWidget *parent, const QPoint &pos) const
```

```
- QPoint mapToGlobal(const QPoint &pos) const
- QPoint mapToParent(const QPoint &pos) const
- QRegion     mask() const
-int    maximumHeight() const
- QSize  maximumSize() const
-int    maximumWidth() const
-int    minimumHeight() const
- QSize  minimumSize() const
- virtual QSize minimumSizeHint() const
-int    minimumWidth() const
-void   move(const QPoint &)
-void   move(int x, int y)
- QWidget *     nativeParentWidget() const
- QWidget *     nextInFocusChain() const
- QRect  normalGeometry() const
-void   overrideWindowFlags(Qt::WindowFlags flags)
-const QPalette &    palette() const
- QWidget *    parentWidget() const
- QPoint pos() const
- QWidget *    previousInFocusChain() const
- QRect  rect() const
-void   releaseKeyboard()
-void   releaseMouse()
-void   releaseShortcut(int id)
-void   removeAction(QAction *action)
-void   render(QPaintDevice *target, const QPoint &targetOffset = QPoint(),
const QRegion &sourceRegion = QRegion(),  QWidget::RenderFlags renderFlags =
RenderFlags(DrawWindowBackground | DrawChildren))
-void   render(QPainter *painter, const QPoint &targetOffset = QPoint(),
const QRegion &sourceRegion = QRegion(), QWidget::RenderFlags renderFlags =
RenderFlags(DrawWindowBackground | DrawChildren))
-void   repaint(int x, int y, int w, int h)
-void   repaint(const QRect &rect)
-void   repaint(const QRegion &rgn)
-void   resize(const QSize &)
-void   resize(int w, int h)
- bool   restoreGeometry(const QByteArray &geometry)
- QByteArray   saveGeometry() const
- QScreen *    screen() const
-void   scroll(int dx, int dy)
```

- void scroll(int dx, int dy, const QRect &r)
- void setAcceptDrops(bool on)
- void setAccessibleDescription(const QString &description)
- void setAccessibleName(const QString &name)
- void setAttribute(Qt::WidgetAttribute attribute, bool on = true)
- void setAutoFillBackground(bool enabled)
- void setBackgroundRole(QPalette::ColorRole role)
- void setBaseSize(const QSize &)
- void setBaseSize(int basew, int baseh)
- void setContentsMargins(int left, int top, int right, int bottom)
- void setContentsMargins(const QMargins &margins)
- void setContextMenuPolicy(Qt::ContextMenuPolicy policy)
- void setCursor(const QCursor &)
- void setEditFocus(bool enable)
- void setFixedHeight(int h)
- void setFixedSize(const QSize &s)
- void setFixedSize(int w, int h)
- void setFixedWidth(int w)
- void setFocus(Qt::FocusReason reason)
- void setFocusPolicy(Qt::FocusPolicy policy)
- void setFocusProxy(QWidget *w)
- void setFont(const QFont &)
- void setForegroundRole(QPalette::ColorRole role)
- void setGeometry(const QRect &)
- void setGeometry(int x, int y, int w, int h)
- void setGraphicsEffect(QGraphicsEffect *effect)
- void setInputMethodHints(Qt::InputMethodHints hints)
- void setLayout(QLayout *layout)
- void setLayoutDirection(Qt::LayoutDirection direction)
- void setLocale(const QLocale &locale)
- void setMask(const QBitmap &bitmap)
- void setMask(const QRegion &region)
- void setMaximumHeight(int maxh)
- void setMaximumSize(const QSize &)
- void setMaximumSize(int maxw, int maxh)
- void setMaximumWidth(int maxw)
- void setMinimumHeight(int minh)
- void setMinimumSize(const QSize &)
- void setMinimumSize(int minw, int minh)
- void setMinimumWidth(int minw)

- void setMouseTracking(bool enable)
- void setPalette(const QPalette &)
- void setParent(QWidget *parent)
- void setParent(QWidget *parent, Qt::WindowFlags f)
- void setShortcutAutoRepeat(int id, bool enable = true)
- void setShortcutEnabled(int id, bool enable = true)
- void setSizeIncrement(const QSize &)
- void setSizeIncrement(int w, int h)
- void setSizePolicy(QSizePolicy)
- void setSizePolicy(QSizePolicy::Policy horizontal, QSizePolicy::Policy vertical)
- void setStatusTip(const QString &)
- void setStyle(QStyle *style)
- void setTabletTracking(bool enable)
- void setToolTip(const QString &)
- void setToolTipDuration(int msec)
- void setUpdatesEnabled(bool enable)
- void setWhatsThis(const QString &)
- void setWindowFilePath(const QString &filePath)
- void setWindowFlag(Qt::WindowType flag, bool on = true)
- void setWindowFlags(Qt::WindowFlags type)
- void setWindowIcon(const QIcon &icon)
- void setWindowModality(Qt::WindowModality windowModality)
- void setWindowOpacity(qreal level)
- void setWindowRole(const QString &role)
- void setWindowState(Qt::WindowStates windowState)
- void setupUi(QWidget *widget)
- QSize size() const
- virtual QSize sizeHint() const
- QSize sizeIncrement() const
- QSizePolicy sizePolicy() const
- void stackUnder(QWidget *w)
- QString statusTip() const
- QStyle * style() const
- Qstring styleSheet() const
- bool testAttribute(Qt::WidgetAttribute attribute) const
- QString toolTip() const
- int toolTipDuration() const
- bool underMouse() const
- void ungrabGesture(Qt::GestureType gesture)

- void unsetCursor()
- void unsetLayoutDirection()
- void unsetLocale()
- void update(int x, int y, int w, int h)
- void update(const QRect &rect)
- void update(const QRegion &rgn)
- void updateGeometry()
- bool updatesEnabled() const
- QRegion visibleRegion() const
- Qstring whatsThis() const
- int width() const
- WId winId() const
- QWidget * window() const
- QString windowFilePath() const
- Qt::WindowFlags windowFlags() const
- QWindow * windowHandle() const
- QIcon windowIcon() const
- Qt::WindowModality windowModality() const
- qreal windowOpacity() const
- QString windowRole() const
- Qt::WindowStates windowState() const
- QString windowTitle() const
- Qt::WindowType windowType() const
- int x() const
- int y() const
- Reimplemented Public Functions
- virtual QPaintEngine * paintEngine() const override
- Public Slots
- bool close()
- void hide()
- void lower()
- void raise()
- void repaint()
- void setDisabled(bool disable)
- void setEnabled(bool)
- void setFocus()
- void setHidden(bool hidden)
- void setStyleSheet(const QString &styleSheet)
- virtual void setVisible(bool visible)
- void setWindowModified(bool)

```
- void   setWindowTitle(const QString &)
- void   show()
- void   showFullScreen()
- void   showMaximized()
- void   showMinimized()
- void   showNormal()
- void   update()
```

b. Les fenêtres QWidget de niveau supérieur

Commençons par consulter le constructeur C++ de la classe `QWidget` dont le prototype correspond à la ligne suivante :

```
QWidget(QWidget *parent = nullptr, Qt::WindowFlags f = Qt::WindowFlags())
```

Dans ce code, le premier paramètre du constructeur permet de définir un parent à la fenêtre `QWidget` instanciée. Exprimé autrement, on peut avoir une première fenêtre [instance 1 de `QWidget]` qui a comme enfant une seconde fenêtre [instance 2 de `QWidget]`. Quand ce paramètre n'est pas renseigné on crée alors une fenêtre `QWidget` de niveau supérieur. Elle peut avoir un certain nombre de fenêtres enfants, mais elle n'a aucun parent.

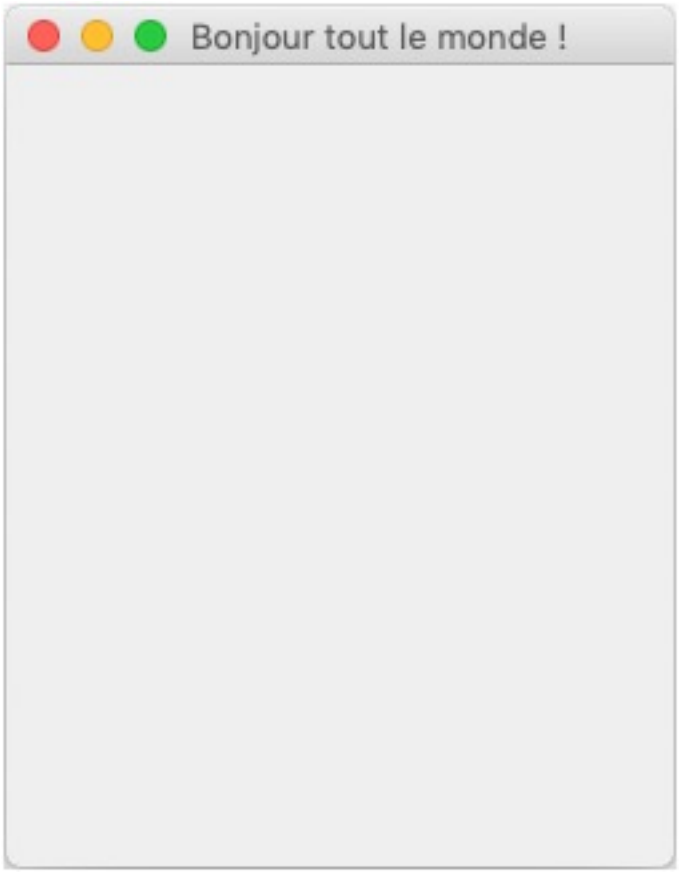
c. Les principales propriétés de QWidget

Reprenons le second code d'exemple vu dans le chapitre précédent. En effet, cet exemple utilisait QWidget, et nous allons à présent détailler son fonctionnement et son utilisation.

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget

class FenetreSimple(QWidget):

    def __init__(self):

        super().__init__()
```

```
        self.execute()

    def execute(self):

        self.resize(250, 300)
        self.move(50, 500)
        self.setWindowTitle("Bonjour tout le monde !")
        self.show()

application = QApplication(sys.argv)

fenetre = FenetreSimple()

sys.exit(application.exec_())
```

Pour rappel, ce code affiche une fenêtre toute simple, ayant pour titre de fenêtre « Bonjour tout le monde ! » comme ci-dessous.



Fenêtre simple

Ajouter une icône de fenêtre

Après avoir consulté la documentation de `QWidget` telle que partiellement reproduite précédemment, nous pouvons essayer d'agrémenter cette petite fenêtre. Pour rappel, la documentation en ligne de QWidget est à cette adresse web : https://doc.qt.io/qt-5/qwidget.html

On peut commencer par ajouter une icône de fenêtre en haut à gauche. La fonction de classes qui nous permettra de faire cela a, selon la documentation, le prototype suivant en C++ :

```
- void    setWindowIcon(const QIcon &icon)
```

On ne va donc pas pouvoir déclarer directement une image. Dans le cas de la petite icône située en général en haut à gauche d'une fenêtre, il s'agit en PyQt de le déclarer grâce à la classe QIcon. Une fois cette classe instanciée, cette dernière sera passée en paramètre de la fonction en charge de son affichage dans la fenêtre.

On commence par importer l'objet `QIcon`.

```
from PyQt5.QtGui import QIcon
```

Puis on instancie l'objet `QIcon` en passant en paramètre l'image souhaitée comme icône. Ensuite, on passe l'instance en paramètre de la fonction `setWindowIcon`.

```
icone = QIcon('icone.png')
self.setWindowIcon(icone)
```

On utilise la ligne "`if __name__ == __main__ :"`, classique en développement Python.

Le code amélioré devient donc celui-ci montrant en gras les lignes importantes quant à l'affichage de l'icône.

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget
from PyQt5.QtGui import QIcon

class FenetreSimple(QWidget):

    def __init__(self):

        super().__init__()
        self.execute()

    def execute(self):

        self.resize(250, 300)
        self.move(50, 500)
        self.setWindowTitle("Bonjour tout le monde ! ")

        icone = QIcon('icone.png')
        self.setWindowIcon(icone)

        self.show()

if __name__ == '__main__':
    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    sys.exit(application.exec_())
```

Fenêtre modale

Une fenêtre modale est une fenêtre qui prend le contrôle total du clavier et de l'écran. C'est-à-dire qu'elle est en attente d'une action utilisateur et interdit toute action sur une autre fenêtre de l'applicatif.

Grâce à la fonction `setWindowModality`, il possible de définir notre fenêtre comme modale. Pour cela, nous avons besoin de définir le comportement de de la fenêtre à une valeur `Qt.ApplicationModal` qui est stockée dans `PyQt5.QtCore`.

On modifie le programme précédent en conséquence et on teste que la fenêtre est bien modale grâce à la fonction `isModal`.

Voici le code modifié avec en gras les lignes concernant la définition de la fenêtre modale :

```
import sys
from PyQt5.QtWidgets import * #QApplication, QWidget
from PyQt5.QtCore import *

class FenetreSimple(QWidget):

    def __init__(self):

        super().__init__()
        self.execute()

    def execute(self):

        self.resize(250, 300)
        self.move(50, 500)
        self.setWindowTitle("Bonjour tout le monde ! ")

        self.setWindowModality(Qt.ApplicationModal)

        self.show()

if __name__ == '__main__':
    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    print("Fenêtre modale ? : " + str(fenetre.isModal()))
    sys.exit(application.exec_())
```

Choix du curseur de la souris

Modifions à nouveau le programme utilisé jusqu'à présent pour cette fois modifier le curseur de la souris tel qu'il est affiché lors du survol de la fenêtre. On utilise pour cela la fonction `setOverrideCursor` sur l'application elle-même.

Voici l'extrait du code avec la ligne concernée en gras :

```
if __name__ == '__main__':
    application = Qapplication(sys.argv)
```

```
application.setOverrideCursor(Qt.WaitCursor)
fenetre = FenetreSimple()
sys.exit(application.exec_())
```

`Qt.WaitCursor` correspond au curseur d'attente. Les différents curseurs utilisables sont listés sur la page de documentation en ligne relative aux curseurs de souris dans Qt : https://doc.qt.io/qt-5/qcursor.html

Notion de flags et différents affichages possibles

Les différentes valeurs de l'énumération sont explicitées sur la page suivante de la documentation en ligne : https://doc.qt.io/qt-5.9/qt.html#WindowType-enum. En fait, on définit grâce à cela la fonction de la fenêtre créée : pop-up, dialogue, écran au chargement d'un logiciel.

On utilise pour cela la fonction `setWindowFlags` comme dans le code ci-dessous où l'on définit la fenêtre en pop-up (`Qt.Popup`) :

```
class FenetreSimple(QWidget)

    def __init__(self):

        super().__init__()
        self.execute()

    def execute(self):

        self.resize(250, 300)
        self.move(50, 500)
        self.setWindowTitle("Bonjour tout le monde ! ")
        self.setWindowFlags(Qt.Popup)
        self.show()
```

Le tableau suivant reprend les principales valeurs possibles de l'énumération :

`Qt.Widget`	Il s'agit de la valeur par défaut. Elle correspond à une utilisation de `QWidget` de manière indépendante, sans parent déclaré.
`Qt.Window`	Cette fenêtre est d'un type assez proche du type par défaut `Qt.Widget`. La nuance est qu'on ne se préoccupe pas ici de savoir si notre fenêtre a un parent ou non.
`Qt.Dialog`	Cette fenêtre de dialogue n'a habituellement pas les boutons « classiques » : fermeture, maximisation et minimisation.
`Qt.Popup`	Cette fenêtre est une fenêtre de pop-up, c'est-à-dire sans barre de statut (*status bar*).
`Qt.ToolTip`	Cette fenêtre permet de définir un *tooltip*, c'est-à-dire l'affichage au survol d'une information complémentaire.
`Qt.SplashScreen`	Cette fenêtre *SplashScreen* correspond à ce qui s'affiche au chargement d'un logiciel pendant quelques secondes en général.

Nous allons maintenant étudier les différents widgets qui peuvent être utilisés sur un `QWidget`. D'autres aspects sur l'utilisation de `QWidget` lui-même seront développés par la suite.

2. Widget QLabel

a. Introduction

Nous allons en effet commencer à étudier les widgets qui participent justement à composer une fenêtre `QWidget` : en premier lieu, le `QLabel`. Ce composant permet d'afficher un texte simple ou une image simple.

Reprenons notre fenêtre simple et affichons un texte ainsi qu'une image grâce à ce widget `QLabel`. Un rapide passage par la documentation Qt peut nous apprendre notamment les fonctions disponibles pour préciser l'affichage d'un texte ou d'une image : https://doc.qt.io/qt-5/qlabel.html

b. Exemple d'utilisation

Commençons par créer un `QLabel` et d'emblée associons-le à la fenêtre courante.

```
label1 = QLabel("PyQt", self)
```

Nous pouvons à présent choisir de modifier le texte affiché par le `QLabel` en accédant à la valeur affichée, en lui concaténant le chiffre 5 puis en l'affectant à ce qui doit être affiché.

```
label1.setText(label1.text() + "5")
```

On peut définir une marge intérieure au widget `QLabel`.

```
label1.setMargin(5)
```

On peut également indenter le widget lui-même, par rapport au côté gauche de la fenêtre.

```
label1.setIndent(15)
```

Pour l'affichage d'une image avec `QLabel`, il s'agit à peu près de la même technique. Toutefois, nous manipulons pour cela un objet `Qpixmap`. Il faut donc commencer par référencer le bon module pour pouvoir y accéder.

```
from PyQt5.QtGui import *
```

Puis on déclare l'image à afficher.

```
pic = Qpixmap("logo.png")
```

Ensuite, on crée un `QLabel` que l'on associe à la fenêtre courante.

```
label2 = QLabel(self)
```

On définit l'image préalablement créée comme contenu à afficher (ce n'est pas du texte cette fois-ci).

```
label2.setPixmap(pic)
```

Puis on positionne l'affichage en définissant les coordonnées du coin supérieur gauche de l'image dans le repère orthonormé associé à la fenêtre elle-même.

```
label2.move(60, 100)
```

Le code donne ceci à présent (code concernant `QLabel` en gras) :

```
import sys
from PyQt5.QtWidgets import * #QApplication, QWidget
from PyQt5.QtCore import *
from PyQt5.QtGui import * #QPixmap

class FenetreSimple(QWidget):

    def __init__(self):

        super().__init__()
        self.execute()
```

```
    def execute(self):

        self.resize(250, 300)
        self.move(50, 500)
        self.setWindowTitle("Bonjour tout le monde ! ")

        label1 = QLabel("PyQt", self)
        label1.setText(label1.text() + "5")
        label1.setMargin(5)
        label1.setIndent(15)

        pic = QPixmap("logo.png")
        label2 = QLabel(self)
        label2.setPixmap(pic)
        label2.move(60, 100)

        self.show()

if __name__ == '__main__':
    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    sys.exit(application.exec_())
```

Ce code permet l'affichage de cette petite fenêtre d'exemple :

Affichage d'un texte et d'une image avec QLabel

Les chapitres suivants nous permettront de proposer d'autres exemples d'utilisation de `QLabel`.
Poursuivons l'inventaire avec le widget `QLineEdit` que l'on nomme parfois *text box* ou *text input*, c'est-à-dire en français « zone de texte ».

3. Widget QLineEdit

a. Introduction

On a fréquemment besoin de laisser à l'utilisateur(trice) d'une interface graphique la possibilité d'entrer des données, dans un formulaire par exemple. C'est exactement le propos du widget `QLineEdit` qui autorise l'entrée d'une chaîne de caractères. La documentation en ligne est disponible à l'adresse suivante : https://doc.qt.io/qt-5/qlineedit.html

b. Exemple d'utilisation

Pour ajouter un widget `QLineEdit` à une fenêtre, on procède selon les étapes suivantes. On commence par l'instancier.

```
line_edit = QLineEdit(self)
```

Puis on le positionne à l'endroit désiré.

```
line_edit.move(5, 30)
```

On change éventuellement les dimensions du widget (largeur, hauteur).

```
line_edit.resize(150, 20)
```

On peut enfin définir une valeur par défaut.

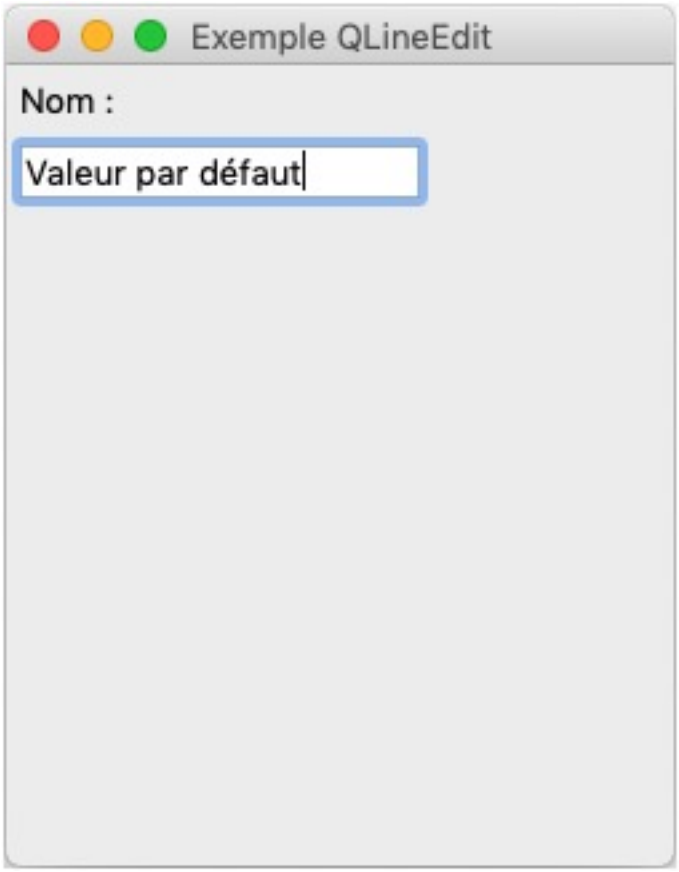
```
line_edit.setText("Valeur par défaut")
```

Cela nous donne le code suivant (lignes concernant `QLineEdit` en gras) :

```
import sys
from PyQt5.QtWidgets import * #QApplication, QWidget
from PyQt5.QtCore import *
from PyQt5.QtGui import * #QPixmap

class FenetreSimple(QWidget):

    def __init__(self):
        super().__init__()
        self.execute()

    def execute(self):

        self.resize(250, 300)
```

```
        self.move(50, 500)
        self.setWindowTitle("Exemple QLineEdit")

        label = QLabel("Nom : ", self)
        label.move(5, 5)

        line_edit = QLineEdit(self)
        line_edit.move(5, 30)
        line_edit.resize(150, 20)
        line_edit.setText("Valeur par défaut")

        self.show()

if __name__ == '__main__':
    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    sys.exit(application.exec_())
```

Ce code permet l'affichage suivant :



Affichage d'une zone de texte avec QLineEdit

4. Widget QPushButton

a. Introduction

Nous avons vu comment définir un label de type texte ou image et comment définir une zone de texte éditable. Voyons à présent comment afficher un bouton. Nous aurons alors l'essentiel pour créer quelque chose de similaire à un formulaire web. Nous en profiterons également pour étudier quelques autres widgets qui héritent de `QPushButton`. Pour le moment, on ne se préoccupe pas de la gestion des évènements ni des signaux, c'est-à-dire de ce qui permet de gérer l'action induite par l'appui sur le bouton.

b. Utilisation

Tout d'abord, la documentation Qt de ce widget se trouve à cette adresse : https://doc.qt.io/qt-5/qpushbutton.html

De façon assez similaire à ce que nous faisons depuis le début de ce chapitre, on commence par instancier un `QPushButton` en l'associant à la fenêtre courante.

```
button = QPushButton(self)
```

Puis on le positionne à un emplacement judicieux.

```
button.move(5, 60)
```

Enfin, on donne une valeur au texte affiché dans le bouton lui-même.

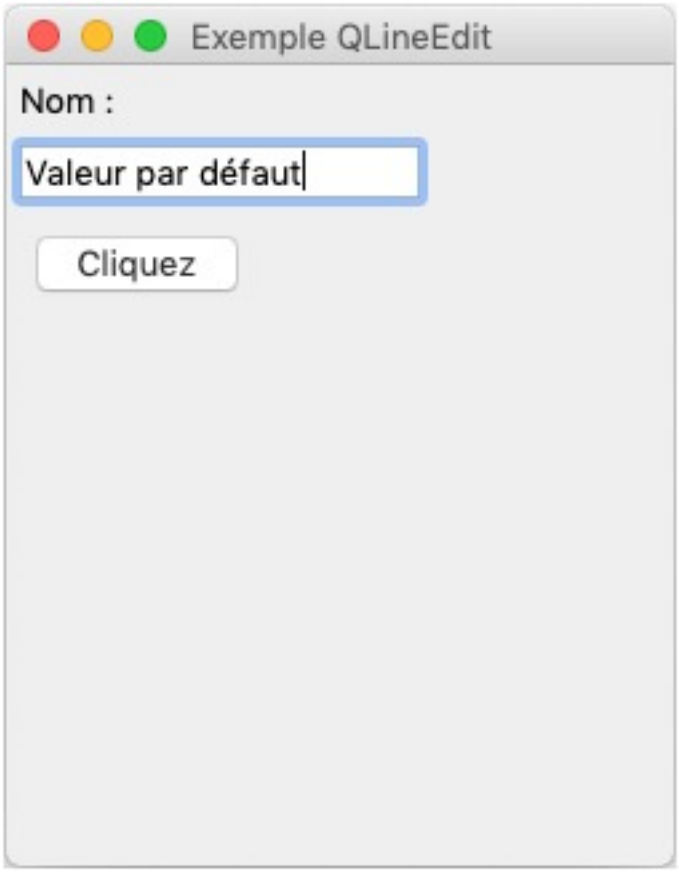
```
button.setText("Cliquez")
```

Voici le code global de l'exemple courant (lignes concernant `QPushButton` en gras) :

```
import sys
from PyQt5.QtWidgets import * #QApplication, QWidget
from PyQt5.QtCore import *
from PyQt5.QtGui import * #QPixmap

class FenetreSimple(QWidget):

    def __init__(self):

        super().__init__()
        self.execute()

    def execute(self):

        self.resize(250, 300)
        self.move(50, 500)
        self.setWindowTitle("Exemple QLineEdit")
```

```
        label = QLabel("Nom : ", self)
        label.move(5, 5)

        line_edit = QLineEdit(self)
        line_edit.move(5, 30)
        line_edit.resize(150, 20)
        line_edit.setText("Valeur par défaut")

        button = QPushButton(self)
        button.move(5, 60)
        button.setText("Cliquez")

        self.show()

if __name__ == '__main__':
    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    sys.exit(application.exec_())
```

Ce code permet d'afficher la fenêtre suivante :



Intégration d'un QPushButton dans la fenêtre

Plusieurs classes du framework dérivent de la classe `QPushButton`. Il s'agit notamment de :

- `QCheckBox` : permet de définir une case à cocher.
- `QRadioButton` : permet de définir un bouton radio.

De façon assez similaire à ce que l'on a fait précédemment, on ajoute une case à cocher ainsi qu'un bouton radio à notre fenêtre.

Commençons par le `QCheckBox`.

```
cb = QCheckBox(self)
cb.setText("Case à cocher")
cb.setChecked(True)
cb.move(5, 90)
```

Puis insérons un bouton radio (`QRadioButton`).

```
rb = QRadioButton(self)
rb.setText("Bouton radio")
rb.setChecked(True)
rb.move(5, 110)
```

On peut également utiliser la fonction `isChecked` permettant de savoir si ce type de widget est coché ou non.

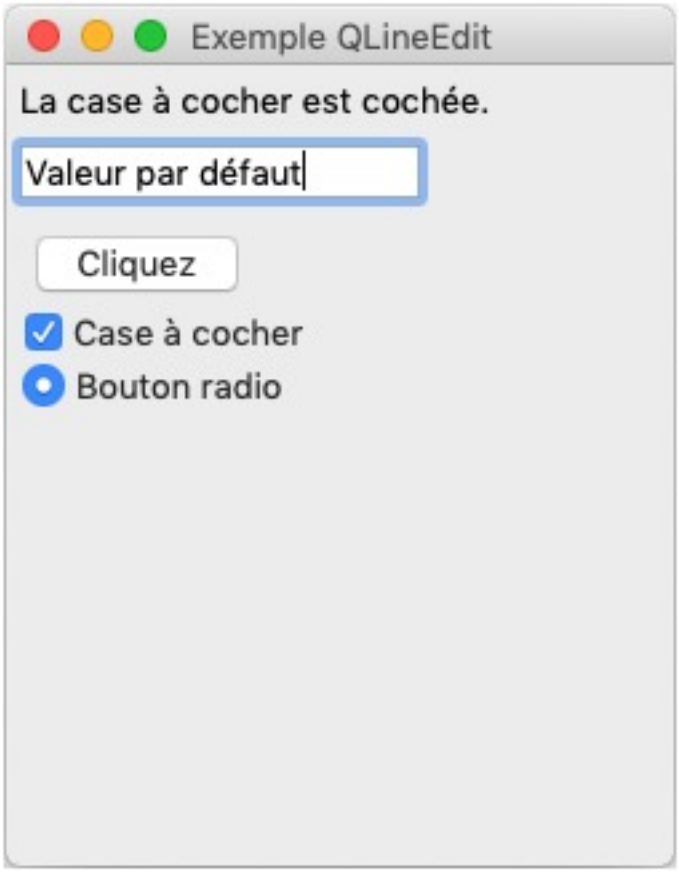
```
if cb.isChecked() == True:
    label.setText("La case à cocher est cochée.")
else:
    label.setText("La case à cocher n'est pas cochée.")
```

Le code complet est à présent celui-ci :

```
import sys
from PyQt5.QtWidgets import * #QApplication, QWidget
from PyQt5.QtCore import *
from PyQt5.QtGui import * #QPixmap

class FenetreSimple(QWidget):


   def __init__(self):

      super().__init__()
      self.execute()

   def execute(self):

      self.resize(250, 300)
      self.move(50, 500)
      self.setWindowTitle("Exemple QLineEdit")

      label = QLabel("Nom : ", self)
      label.move(5, 5)
```

```
        line_edit = QLineEdit(self)
        line_edit.move(5, 30)
        line_edit.resize(150, 20)
        line_edit.setText("Valeur par défaut")

        button = QPushButton(self)
        button.move(5, 60)
        button.setText("Cliquez")

        cb = QCheckBox(self)
        cb.setText("Case à cocher")
        cb.setChecked(True)
        cb.move(5, 90)

        rb = QRadioButton(self)
        rb.setText("Bouton radio")
        rb.setChecked(True)
        rb.move(5, 110)

        if cb.isChecked() == True:
            label.setText("La case à cocher est cochée.")
        else:
            label.setText("La case à cocher n'est pas cochée.")

        self.show()

if __name__ == '__main__':
    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    sys.exit(application.exec_())
```

On obtient avec ce code le résultat suivant :



Intégration d'une case à cocher et d'un bouton radio

5. Widget QComboBox

a. Introduction

Voyons à présent le widget `QComboBox` qui correspond comme son nom l'indique à une *combobox* ou en français à une « boîte combinée ». Ce composant est effectivement l'un de ceux les plus utilisés en conception d'interfaces graphiques.

b. Utilisation

On commence, comme à l'habitude, par déclarer une instance de `QComboBox`, mais que l'on définit comme propriété de notre classe afin d'y accéder par la suite.

```
self.cb = QcomboBox(self)
```

On peut ensuite ajouter des valeurs à notre boîte combinée, grâce à la fonction `addItem`.

```
self.cb.addItem("France")
self.cb.addItem("Italie")
```

On peut également ajouter plusieurs valeurs en une fois grâce à la fonction `addItems`.

```
self.cb.addItems(["Espagne", "Allemagne", "Belgique"])
```

Associons une fonction à l'évènement de changement de valeur dans la *combobox*. Pour cela, on connecte avec la fonction `connect` l'évènement `currentIndexChanged` à une fonction que l'on nomme ici `selectionchange`.

```
self.cb.currentIndexChanged.connect(self.selectionchange)
```

On affiche au passage toutes les valeurs de la *combobox* en itérant sur la collection des pays.

```
for ii in range(self.cb.count()):
    print(self.cb.itemText(ii)
```

Cela nous donne, lors de l'exécution, l'affichage suivant dans la console :

```
> France
> Italie
> Espagne
> Allemagne
> Belgique
```

Définissons enfin notre fonction de rappel (*callback*) liée à l'évènement de changement de valeur dans la *combobox*. Nous avons appelé précédemment cette fonction

`selectionchange`. On récupère en paramètre l'index courant correspondant à la sélection, mais il est plus simple d'utiliser la fonction `currentText` pour afficher dans notre label la valeur courante de la *combobox*.

```
def selectionchange(self, i):

    self.label.setText(self.cb.currentText())
```

Le code global de notre exemple est le suivant (lignes concernées en gras) :

```
import sys
from PyQt5.QtWidgets import * #QApplication, QWidget
from PyQt5.QtCore import *
from PyQt5.QtGui import * #QPixmap

class FenetreSimple(QWidget):

  def __init__(self):

    super().__init__()
    self.execute()

  def execute(self):

    self.resize(250, 300)
    self.move(50, 500)
    self.setWindowTitle("Exemple QComboBox")

    self.label = QLabel("Utilisation de QComboBox", self)
    self.label.move(5, 5)

    self.cb = QComboBox(self)
    self.cb.addItem("France")
    self.cb.addItem("Italie")
    self.cb.addItems(["Espagne", "Allemagne", "Belgique"])
    self.cb.move(5, 30)
    self.cb.currentIndexChanged.connect(self.selectionchange)

    self.show()
```

```
        for ii in range(self.cb.count()):
            print(self.cb.itemText(ii))

    def selectionchange(self, i):

        self.label.setText(self.cb.currentText())

if __name__ == '__main__':
    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    sys.exit(application.exec_())
```

Ce code permet d'afficher la fenêtre suivante. Quand on sélectionne un pays, la valeur est affichée dans le label.

Utilisation d'un QComboBox

6. Widget QSpinBox

a. Introduction

Un `QSpinBox` permet d'afficher une zone de sélection numérique, parfois appelée en français un « bouton fléché » et plus couramment, en anglais, un *spinner*. Il consiste en un petit widget avec des flèches permettant de sélectionner une valeur entière (1, 2, 3, etc.). On peut définir la valeur minimale (fonction `setMinimum`) et la valeur maximale (fonction `setMaximum`). La valeur courante peut se définir avec la fonction `setValue`.

b. Exemple d'utilisation

Comme d'habitude, on définit notre instance de widget.

```
self.spinner = QSpinBox(self)
```

Puis on la positionne à un emplacement de la fenêtre.

```
self.spinner.move(5, 35)
```

Enfin, on associe à l'évènement de changement de valeur une fonction `valuechange`.

```
self.spinner.valueChanged.connect(self.valuechange)
```

Le code de la fonction `valuechange` est le suivant. On affiche la valeur courante dans un label de la fenêtre.

```
def valuechange(self):

    self.label.setText("Valeur courante : " +
str(self.spinner.value()))
```

Le code global est le suivant :

```
import sys
from PyQt5.QtWidgets import * #QApplication, QWidget
from PyQt5.QtCore import *
from PyQt5.QtGui import * #QPixmap

class FenetreSimple(QWidget):

    def __init__(self):

        super().__init__()
        self.execute()

    def execute(self):

        self.resize(250, 300)
        self.move(50, 500)
        self.setWindowTitle("Exemple QSpinBox")

        self.label = QLabel("", self)
        self.label.move(5, 5)
        self.label.resize(150, 20)

        self.spinner = QSpinBox(self)
        self.spinner.move(5, 35)
        self.spinner.valueChanged.connect(self.valuechange)

        self.show()

    def valuechange(self):

        self.label.setText("Valeur courante : " +
str(self.spinner.value()))

if __name__ == '__main__':

    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    sys.exit(application.exec_())
```

Ce code donne l'affichage suivant :

Utilisation d'un QSpinBox

7. Widget QDateTimeEdit

a. Introduction

Le widget `QDateTimeEdit` ressemble graphiquement au widget précédent `QSpinBox`. Comme son nom le laisse penser, il sert à choisir ou à définir une date et une heure.

b. Utilisation

La documentation en ligne de ce widget se trouve à cette adresse : https://doc.qt.io/qt-5/qdatetimeedit.html

Tout d'abord, on instancie le widget.

```
self.dt = QDateTimeEdit(self)
```

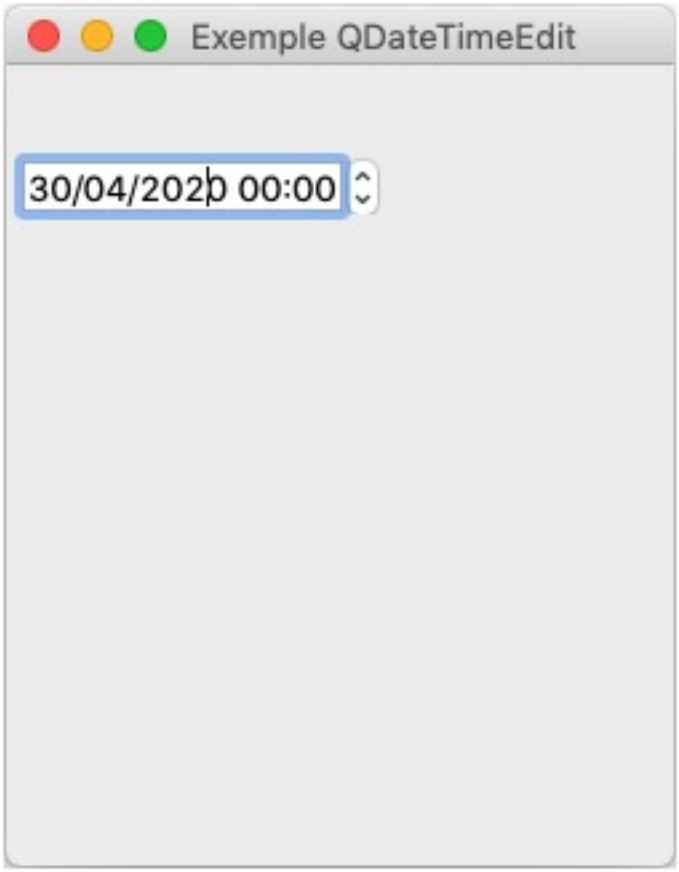
On peut lui passer une valeur par défaut, comme par exemple la date du jour courant.

```
self.dt.setDate(QDate.currentDate())
```

Le code global de l'exemple est le suivant :

```
import sys
from PyQt5.QtWidgets import * #QApplication, QWidget
from PyQt5.QtCore import *

class FenetreSimple(QWidget):

    def __init__(self):

        super().__init__()
        self.execute()

    def execute(self):

        self.resize(250, 300)
        self.move(50, 500)
        self.setWindowTitle("Exemple QDateTimeEdit")

        self.dt = QDateTimeEdit(self)
        self.dt.move(5, 35)
        self.dt.setDate(QDate.currentDate())

        self.show()


if __name__ == '__main__':

    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    sys.exit(application.exec_())
```

Ce code donne l'affichage suivant :



Exemple d'utilisation de QDateTimeEdit avec par défaut la date du 30 avril 2020

c. Un mot au sujet de QDate et de QTime

Dans le dernier exemple, nous avons utilisé la classe `QDate` pour obtenir la date du jour. La documentation de cette classe est accessible à l'adresse suivante : https://doc.qt.io/qt-5/qdate.html

Il existe également une classe spécialisée pour la gestion de l'heure. Il s'agit de la classe `QTime`, dont la documentation est disponible à cette adresse : https://doc.qt.io/qt-5/qtime.html

Ces deux classes fournissent toutes les fonctions qui permettent d'obtenir la date courante, l'heure courante, mais également de construire une date et une heure à partir d'autres données. Il est enfin possible d'exprimer une date et une heure en divers formats. Sont également accessibles plusieurs autres calendriers (chinois, hébreu, islamique)

permettant de passer facilement d'un système à l'autre.

L'exemple suivant, qui utilise uniquement la ligne de commande, propose un aperçu des possibilités :

```
from PyQt5.QtCore import *

maintenant = QDate.currentDate()

print(maintenant.toString(Qt.ISODate))
print(maintenant.toString(Qt.DefaultLocaleLongDate))

maintenant_datetime = QDateTime.currentDateTime()

print(maintenant_datetime.toString())

heure = QTime.currentTime()

print(heure.toString(Qt.DefaultLocaleLongDate))
```

Ce programme donne ce résultat en sortie :

```
2020-05-13
13 May 2020
Wed May 13 17:25:24 2020
17:25:24 CEST
```

8. Widget QProgressBar

a. Introduction

Une barre de progression permet de renseigner l'utilisateur(trice) sur l'avancée d'un traitement (un calcul ou un téléchargement par exemple). Ce type de composant graphique s'implémente avec le widget `QProgressBar`.

b. Exemple d'utilisation

On déclare une barre de progression en spécifiant ses dimensions et en imposant la valeur initiale.

```
        self.progress = QProgressBar(self)
        self.progress.setGeometry(5, 20, 150, 30)
        self.progress.setValue(0)
```

Dans cet exemple, on déclare ensuite un chronomètre qui va faire défiler la barre de progression.

```
self.timer.timeout.connect(self.handleTimer)
self.timer.start(1000)
```

La fonction suivante permet de faire évoluer la valeur courante de la barre de progression.

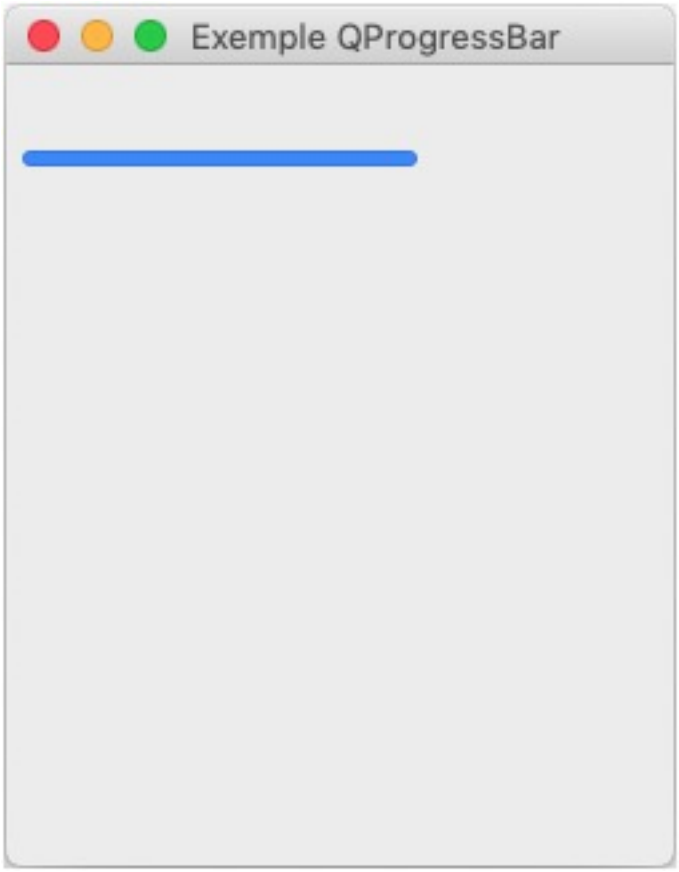
```
def handleTimer(self):
  valeur = self.progress.value()
  if valeur < 100:
    valeur = valeur + 10
    self.progress.setValue(valeur)
  else:
    self.timer.stop()
```

Le code global est le suivant :

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

class FenetreSimple(QWidget):
   def __init__(self):
      super().__init__()
      self.execute()

   def execute(self):
```

```
        self.resize(250, 300)
        self.move(50, 500)
        self.setWindowTitle("Exemple QProgressBar")

        self.progress = QProgressBar(self)
        self.progress.setGeometry(5, 20, 150, 30)
        self.progress.setValue(0)

        self.show()

        self.timer = QTimer()
        self.timer.timeout.connect(self.handleTimer)
        self.timer.start(1000)

    def handleTimer(self):
        valeur = self.progress.value()
        if valeur < 100:
            valeur = valeur + 10
            self.progress.setValue(valeur)
        else:
            self.timer.stop()

if __name__ == '__main__':
    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    sys.exit(application.exec_())
```

Voici-ci-dessous la fenêtre relative au code précédent, implémentant une barre de progression en PyQt :



Utilisation de QProgressBar

9. Widget QTextEdit

a. Introduction

Pour compléter cet inventaire des principaux widgets PyQt, abordons rapidement le widget `QTextEdit`. Nous avons vu précédemment `QLineEdit`, qui est une zone de texte simple, sur une ligne. `QTextEdit` permet non seulement d'avoir plusieurs lignes dans le widget, mais également d'utiliser un formatage WYSIWYG.

WYSIWYG, en anglais *What you see is what you get* (littéralement : « ce que vous voyez est ce que vous obtenez ») est relatif au formatage à l'écran qui donne une idée précise du résultat final. C'est le cas par exemple dans un traitement de texte.

La documentation en ligne est disponible à cette adresse : https://doc.qt.io/qt-5/qtextedit.html

b. Exemple d'utilisation

Prenons un premier exemple avec une mise en forme allant plus loin que du simple texte. Pour cela, nous allons utiliser du HTML, le langage à balisage utilisé par le Web et compris par les navigateurs. Entre autres formatages, un widget `QTextEdit` « comprend » en effet nativement le HTML.

Dans l'exemple, nous allons afficher une phrase en gras de couleur rouge, passer une ligne, puis une ligne en bleu et en italique.

Le code HTML correspondant est le suivant :

```
<font color='red'><b>Ligne en rouge et en gras</b></font>
<br />
<font color='blue'><i>Ligne en bleu et en italique</i></font><br />
```

Voici ce que donne le code de l'exemple :

```
import sys
from PyQt5.QtWidgets import * #QApplication, QWidget

class FenetreSimple(QWidget):
    def __init__(self):
        super().__init__()
        self.execute()
```

```
    def execute(self):
        self.resize(250, 300)
        self.move(50, 500)
        self.setWindowTitle("Exemple QTextEdit")

        label = QLabel("Nom : ", self)
        label.move(5, 5)

        text_edit = QTextEdit(self)
        text_edit.move(5, 30)
        text_edit.resize(150, 60)
        text_edit.append(u"<font color='red'><b>Ligne en rouge
et en gras</b></font><br /><font color='blue'><i>Ligne en bleu
et en italique</i></font><br />")
        self.show()

if __name__ == '__main__':
    application = QApplication(sys.argv)
    fenetre = FenetreSimple()
    sys.exit(application.exec_())
```

Ce petit code donne l'affichage ci-dessous. On voit qu'une barre de défilement (*scroll bar*) apparaît automatiquement sur la droite du widget. En effet, le contenu est trop grand pour le contenant et les barres de défilement sont un aspect du fonctionnement de `QTextEdit` dans ce cas.

Exemple d'utilisation de QTextEdit

Ce widget `QTextEdit` est donc extrêmement puissant, car il permet non seulement d'afficher du texte brut, c'est-à-dire du texte simple sans aucun formatage (*plain text* en anglais), mais également du texte formaté, stylisé, comme nous venons de le démontrer. Dans certains contextes, il peut être utile et intéressant de n'utiliser que du texte brut (exemple de l'éditeur de texte de type Notepad). Dans ces cas, l'usage de `QTextEdit` peut paraître disproportionné. Il existe un widget très similaire en PyQt, mais qui ne prend en charge que le texte brut. Il s'agit de `QPlainTextEdit`.

c. Un mot sur le widget QPlainTextEdit

Comme exposé précédemment, la grande nuance entre `QTextEdit` et `QPlainTextEdit` est que ce dernier ne prend en charge que du texte brut. Si le logiciel à développer doit manipuler du texte sans formatage particulier, alors le widget `QPlainTextEdit` est le plus indiqué. C'est d'ailleurs celui que nous utiliserons dans un des développements (celui d'un petit Notepad) du chapitre Modèle-Vue-Contrôleur et

premiers exemples d'applications en PyQt.

Le prochain chapitre est consacré spécifiquement aux contrôles de disposition dont la vocation est d'aider à l'organisation et l'agencement au sein d'une fenêtre des différents widgets étudiés dans le présent chapitre.