PYQT4

0. Introduzione

PyQt4 è il porting delle librerie grafiche Qt per il linguaggio Python.
PyQt4 è basato su moltissime classi e moduli. Alcuni esempi di moduli sono:

- QtCore: classi di base, non riguardanti la GUI
- QtGui: tutto ciò che riguarda la GUI (finestre, dialog, pulsanti, edit boxes, widgets vari)
- QtNetwork: classi per la programmazione di rete.
- QtOpenGL: supporto OpenGL
- QtSql: interazione con database SQL

I primi due saranno usati molto spesso nel corso di questo tutorial.

1. Installazione

Per iniziare ad usare queste librerie bisogna installare PyQt4 e i relativi file di sviluppo. Con APT il pacchetto è python-qt4-dev.

2. Creazione di una semplice finestra

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore # importiamo i moduli necessari

class MainWindow(QtGui.QMainWindow):
        def __init__(self):
                QtGui.QMainWindow.__init__(self) # da porre sempre all'inizio
                                                 # inizializza alcuni metodi importanti come resize

                self.resize(350, 250) # ridimensiona la finestra
                self.setWindowTitle('MainWindow')

                self.statusBar().showMessage('Messaggio') # crea una veloce barra di stato

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

Andiamo ad analizzare le parti salienti del codice:

```
app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

La prima istruzione inizializza PyQt4, e deve essere sempre posta all'inizio di ogni codice.
Alle due righe successive, main viene inizializzato come oggetto MainWindow, quindi “contenente” tutti i widgets che abbiamo inizializzato nel costruttore della classe. A questo punto il metodo show() serve a mostrare a schermo la finestra e tutto il resto. Ogni QWidget ha una serie di metodi predefiniti, come ad esempio show(), per le operazioni più comuni, e vedremo i più importanti strada facendo. L'ultima istruzione, infine, serve a “congelare” la finestra. Se manca, appena finito di visualizzare tutti i widget il programma si chiuderà, e voi non avrete tempo di visualizzare nulla.

2. **Primo widget**

Chi non ha mai usato librerie grafiche si chiederà cosa sono i widgets. I widgets sono tutti gli “oggetti” che si possono inserire in una finestra: pulsanti, checkboxes, edit boxes, orologi, barre di stato e moltissimi altri. Dato l'elevato numero di widget esistenti, non saranno trattati tutti in questo tutorial. Mi limiterò ad utilizzarne alcuni, i più comuni od utili; per conoscerne altri è opportuno consultare la reference (http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/classes.html)

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
    def __init__(self):
                    QtGui.QMainWindow.__init__(self)

                    self.setWindowTitle('Pulsanti')

                    button = QtGui.QPushButton('Quit');
                    button.setFont(QtGui.QFont("Times", 10, QtGui.QFont.Bold));
                    self.connect(button, QtCore.SIGNAL('clicked()'), QtCore.SLOT('close()'));

                    self.setCentralWidget(button);

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

Le quattro righe “nuove” sono:

```
                    button = QtGui.QPushButton('Quit');
```

```
        button.setFont(QtGui.QFont("Times", 10, QtGui.QFont.Bold));
        self.connect(button, QtCore.SIGNAL('clicked()'), QtCore.SLOT('close()'));

        self.setCentralWidget(button);
```

Con la prima, creiamo un nuovo pulsante con il nome Quit. La definizione completa di questo costruttore è:

__init__ (self, QIcon icon, QString text, QWidget parent = None)

Come vedete possiamo definire un'icona da associare al bottone, la stringa di testo, e il widget "genitore". Nel nostro caso possiamo lasciare il valore di default, ma vedremo che in seguito tale parametro ci sarà utile. Il metodo setFont è abbastanza esplicativo.
setCentralWidget: ogni volta che creiamo una finestra in PyQt4, essa richiede che sia settato uno e un solo widget come centrale. Se ciò non viene fatto correttamente, non potrete visualizzare finestre più complesse, con un numero elevato di widget, o con dei menu/toolbar.
Per adesso abbiamo creato un solo pulsante, e quindi non c'è nessun problema: cosa faremo, invece, quando ce ne saranno più di uno? Lo vedremo in seguito.

3. Segnali e slot

La funzione connect infine è di particolare importanza. Questa infatti permette di associare ad un segnale (ovvero ad un evento), un particolare "slot", o funzione definita dall'utente. Il primo parametro definisce il widget da cui deve provenire tale evento (button, nell'esempio precedente), successivamente si definisce il tipo di signale, e infine il nome della funzione (nel codice, uno slot predefinito di QtCore che provoca la chiusura dell'applicazione).
I segnali rappresentano tutte le azioni che l'utente può compiere con un determinato widget (nel caso di un pulsante, l'utente può cliccarlo): nella reference troverete una lista di segnali accettati da ogni widget. Nel nostro caso, vogliamo che quando il pulsante viene cliccato (segnale clicked()), il programma venga chiuso.
Altri segnali saranno visti nel corso di questo tutorial, basta ricordare che per associare un segnale di un particolare widget ad uno slot (o funzione personale) bisogna usare connect.

È possibile anche emettere un particolare segnale utilizzando emit.

4. Layout

Passiamo all'inserimento di più widgets.
Il posizionamento di più widgets all'interno della finestra si può realizzare in due modi: tramite il cosiddetto posizionamento assoluto, o definendo dei layout.

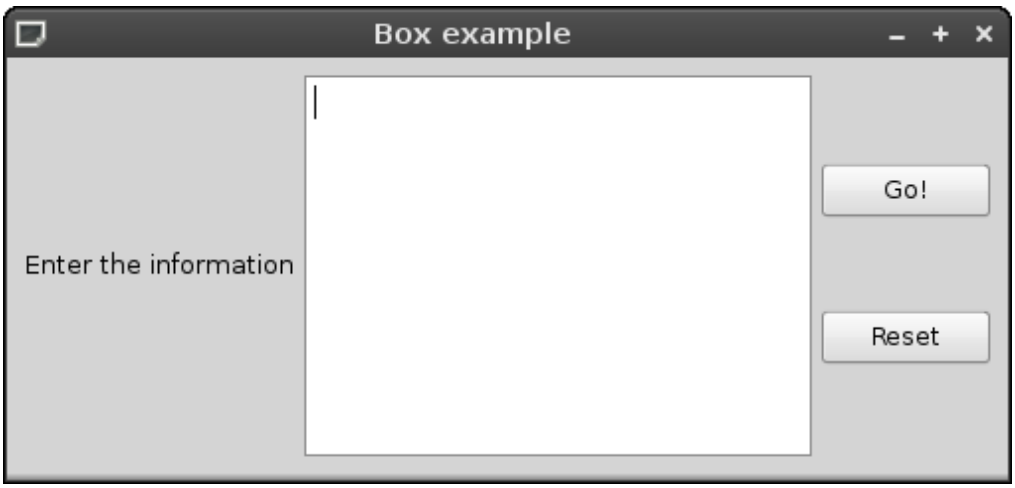
Il primo metodo consiste nello specificare la posizione in pixel di ogni layout: è decisamente **sconsigliato**. Infatti l'applicazione potrebbe risultare diversa su diverse piattaforme, se apportate una piccola modifica (ad esempio cambiate font) dovrete rivedere tutto, e la posizione non cambia se l'utente ridimensiona la finestra.
Tuttavia, se volete usarlo, ecco un piccolo esempio (non un codice completo):

```
label = QtGui.QLabel('Testo');
label.move(34, 40); # definiamo la posizione assoluta in pixel con move()
```

Passiamo all'utilizzo di layout. Ci sono due tipi di layout: le "boxes" (verticali od orizzontali) e la grid (griglia, o comunemente tabella). Nelle box è possibile "impilare" i vari widgets uno dietro l'altro, verticalmente od orizzontalmente a seconda del tipo.
All'interno di un programma si possono usare quanti layout si desiderano, anche annidati (ad esempio, una box verticale dentro una cella di una griglia), ma bisogna sempre ricordarsi di

definirne uno “genitore” che contenga tutti gli altri, e che sarà impostato come il layout della finestra corrente.
Andiamo a fare un esempio dell'utilizzo dei due tipi di box.

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):

	def __init__(self):
		QtGui.QMainWindow.__init__(self)
		self.setWindowTitle('Box example')
		cWidget = QtGui.QWidget(self)

		hBox = QtGui.QHBoxLayout()
		hBox.setSpacing(5)

		randomLabel = QtGui.QLabel('Enter the information', cWidget)
		hBox.addWidget(randomLabel)

		textEdit = QtGui.QTextEdit(cWidget)
		if (textEdit.isReadOnly() == True):
			textEdit.setReadOnly(False)
		hBox.addWidget(textEdit)

		vBox = QtGui.QVBoxLayout()
		vBox.setSpacing(2)
		hBox.addLayout(vBox)

		button1 = QtGui.QPushButton('Go!', cWidget)
		vBox.addWidget(button1)
		button2 = QtGui.QPushButton('Reset', cWidget)
		vBox.addWidget(button2)

		cWidget.setLayout(hBox)
		self.setCentralWidget(cWidget)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```



Intanto notiamo che sono stati introdotti due nuovi widget: la label e la textEdit. La label, come potete vedere, è una porzione di testo libero, mentre la textEdit permette all'utente di inserire un testo particolarmente lungo, o al programma di mostrare dell'output.
Il metodo setReadOnly() permette o vieta all'utente di scrivere nella textEdit: in realtà il comportamento predefinito è di accettare input dall'utente, ma in questo caso ce ne "accertiamo" con isReadOnly().

```
cWidget = QtGui.QWidget(self)
```

Con questa riga creiamo un widget "astratto", che non ha alcuna funzione grafica se non quella di contenere tutti gli altri. Infatti è indicato in ogni widget successivo come il parametro "parent". A cosa serve tutto ciò? Per prima cosa, quando in un programma più complesso andremo ad inserire menu o toolbar, il programma ha bisogno di un solo widget da riconoscere come centrale: se ne mettiamo uno qualsiasi (ad esempio la textEdit), tutti gli altri non verranno visualizzati. Se omettiamo di specificare il widget centrale, non verrà visualizzato nulla al di là del menu. Il workaround per ciò è appunto creare un widget astratto che contenga tutti quelli desiderati.
C'è anche un'altra comodità: se abbiamo bisogno, nel corso del programma, di distruggere/nascondere il nostro lavoro per sostituirlo ad esempio con dell'altro output, basterà un semplice

```
cWidget.hide()
```

per rendere il tutto invisibile. Lo stesso vale al contrario:

```
cWidget.show()
```

mostrerà tutto di nuovo.
Un utente si potrebbe chiedere perchè i costruttori delle due boxes non riportano cWidget come parametro genitore. Non ne abbiamo bisogno, poiché alla fine indichiamo il layout di cWidget con il metodo setLayout(). Al contrario, avremmo potuto dichiarare la hbox come:

```
hBox = QtGui.QHBoxLayout(cWidget)
```

e poi omettere la chiamata a setLayout().
Ma passiamo alla parte importante, le boxes.

```
hBox = QtGui.QHBoxLayout()
hBox.setSpacing(5)
```

Come facilmente intuibile, ciò serve a creare una horizontal box (box orizzontale). Il metodo setSpacing() è comune a tutti i layout (è presente anche nelle griglie) per impostare i pixel di spazio fra i widget appartenenti al layout stesso.

```
randomLabel = QtGui.QLabel('Enter the information', cWidget)
hBox.addWidget(randomLabel) # aggiunge la label alla hbox
```

Se vogliamo che randomLabel sia inserita nella hbox, bisogna ovviamente aggiungerla, con il metodo addWidget().

```
hBox.addLayout(vBox)
```

Se invece vogliamo inserire un altro layout, al suo posto si usa addLayout().

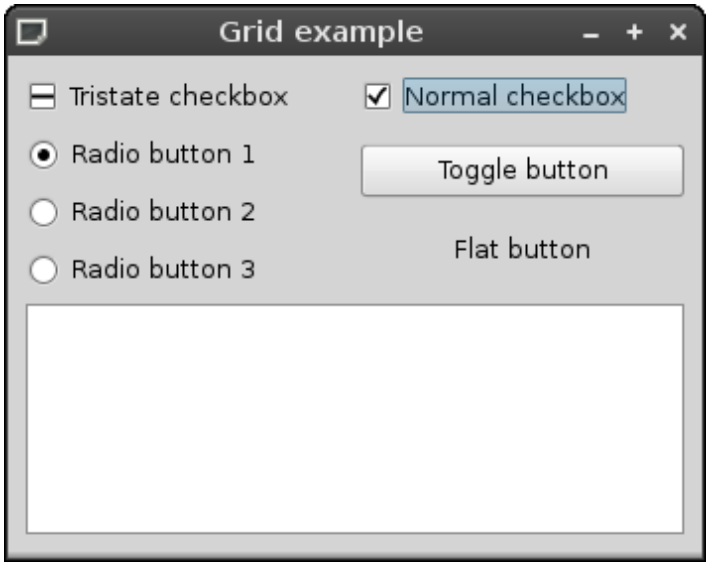
```
cWidget.setLayout(hBox)
```

La hbox è il layout principale (quello che prima ho definito "genitore") e quindi va impostato come

centrale con questa chiamata.
Le box sono comode quando si tratta di inserire in fila pochi widget, ma per grafiche complesse è consigliabile usare le grid, o tabelle. Vediamo un esempio di utilizzo:

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
        def __init__(self):
                QtGui.QMainWindow.__init__(self)

                self.resize(350, 250)
                self.setWindowTitle('Grid example')

                cWidget = QtGui.QWidget(self)

                grid = QtGui.QGridLayout(cWidget) # dichiarazione della griglia

                checkBox = QtGui.QCheckBox("Normal checkbox", cWidget)
                checkBox.setChecked(True)

                triCheck = QtGui.QCheckBox("Tristate checkbox", cWidget)
                triCheck.setTristate(True)
                triCheck.setCheckState(QtCore.Qt.PartiallyChecked)

                vBox = QtGui.QVBoxLayout()
                radio1 = QtGui.QRadioButton("Radio button 1", cWidget)
                radio2 = QtGui.QRadioButton("Radio button 2", cWidget)
                radio3 = QtGui.QRadioButton("Radio button 3", cWidget)
                radio1.setChecked(True)
                vBox.addWidget(radio1)
                vBox.addWidget(radio2)
                vBox.addWidget(radio3)

                vBox2 = QtGui.QVBoxLayout()
                toggle = QtGui.QPushButton("Toggle button")
                toggle.setCheckable(True)
                flat = QtGui.QPushButton("Flat button")
                flat.setFlat(True)
                vBox2.addWidget(toggle)
                vBox2.addWidget(flat)

                textEdit = QtGui.QTextEdit(cWidget)

                grid.addWidget(triCheck, 0, 0)
                grid.addWidget(checkBox, 0, 1)
                grid.addLayout(vBox, 1, 0)
                grid.addLayout(vBox2, 1, 1)
                grid.addWidget(textEdit, 2, 0, 1, 2)

                cWidget.setLayout(grid)
                self.setCentralWidget(cWidget)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
```

```
main.show()
sys.exit(app.exec_())
```



Per prima cosa introduciamo brevemente i widgets nuovi:

```
checkBox = QtGui.QCheckBox("Normal checkbox", cWidget)
checkBox.setChecked(True)
```

Una checkbox può avere due stati: checked o unchecked. Il cambio di stato è “catturato” dal segnale stateChanged().
Le Qt mettono a disposizione un secondo tipo di checkbox, detto “tristate” perchè appunto possiede tre stati diversi: la si può vedere nel secondo esempio.

```
radio1 = QtGui.QRadioButton("Radio button 1", cWidget)
```

A differenza delle checkbox, quando ci sono diversi radio button, solo uno può essere selezionato.
Un toggle button è definito dal metodo setCheckable(), mentre l'ultimo è un flat button.

Torniamo però a parlare delle griglie.
Se addWidget (o il corrispondente addLayout) è usato per delle griglie, oltre all'oggetto da aggiungere, bisogna anche specificare la posizione. La definizione completa è:

QGridLayout.addWidget (self, QWidget, int row, int column, int rowSpan, int columnSpan, Qt.Alignment=0)

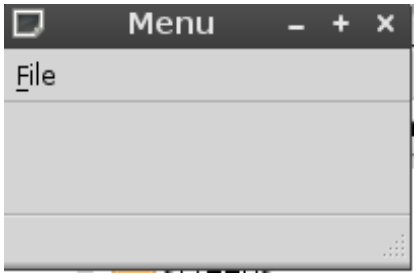
Dove row e column indicano la riga e la colonna in cui inserire il widget, rowSpan e columnSpan nell'esempio sono usati solo a proposito della textEdit: se vogliamo che essa occupi uno spazio maggiore di quello assegnato ad una singola celletta, bisogna indicare esattamente quante righe o colonne si vuole far occupare al widget. La nostra textEdit occupa quindi una sola riga ma due colonne.
I metodi per impostare le proprietà delle griglie sono tanti, eccone alcuni:
setHorizontalSpacing: imposta la distanza fra due widget orizzontali.
setVerticalSpacing: imposta la distanza fra due widget verticali.
setSpacing: le due precedenti, combinate.
setColumnMinimumWidth: larghezza minima della colonna specificata come primo parametro.
setRowMinimumWidth: stessa cosa per una riga.
int columnCount: ritorna il numero di colonne della griglia.

int rowCount: ritorna il numero di righe.

5. Menu

Andiamo a definire un semplice menu.

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):

        def mySlot(self):
                print "Menu voice has been hovered"

        def __init__(self):
                QtGui.QMainWindow.__init__(self)
                self.setWindowTitle('Menu')

                quit = QtGui.QAction(QtGui.QIcon("icons/cancel.png"), "Quit", self)
                quit.setShortcut("Ctrl+Q")
                quit.setStatusTip("Quit application")
                self.connect(quit, QtCore.SIGNAL('triggered()'), QtCore.SLOT('close()'))

                sep = QtGui.QAction(self)
                sep.setSeparator(True)

                info = QtGui.QAction(QtGui.QIcon("icons/information.png"), "Information", self)
                info.setShortcut("Ctrl+I")
                info.setStatusTip("Show information")
                self.connect(info, QtCore.SIGNAL('hovered()'), self.mySlot)

                self.statusBar().show()

                menuB = self.menuBar()
                file = menuB.addMenu('&File')
                file.addAction(quit)
                file.addAction(sep)
                file.addAction(info)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```



In ordine di “importanza”, i passi sono:

- creare una menuBar (la nostra menuB)

- aggiungere tutti i menu che vogliamo creare con il metodo addMenu(). Questo metodo prende come argomento una stringa. Il carattere preceduto dal simbolo & funge da scorciatoia: premendo Alt+carattere attiveremo il menu corrispondente.
- ad ogni menu vanno poi aggiunge le singole QAction, ovvero le voci di menu.

```
quit = QtGui.QAction(QtGui.QIcon("icons/cancel.png"), "Quit", self)
quit.setShortcut("Ctrl+Q")
quit.setStatusTip("Quit application")
self.connect(quit, QtCore.SIGNAL('triggered()'), QtCore.SLOT('close()'))
```

Il costruttore di QAction permette di specificare subito il path dell'icona e il testo della voce di menu, oppure di ometterlo e di indicarlo dopo con i metodi appositi:

```
quit = QtGui.QAction(self)
quit.setText("Quit")
quit.setIcon(QtGui.QIcon("icons/quit.png")
```

Questi ultimi due codici danno lo stesso identico risultato.
Il metodo setText() merita particolare attenzione: infatti esso si può trovare in diversi widget che comprendono un testo (pulsanti, etichette, etc...) per modificare il testo che accompagna il widget stesso. Il corrispondente che invece ritorna il testo corrente sotto forma di QString è semplicemente text().
Eccezione: per le textEdit, invece di text() si ha toPlainText().
Per ritornare la shortcut o lo statustip correnti potete usare i metodi statusTip() e shortcut().
Il secondo oggetto non rappresenta una voce di menu, ma semplicemente un separatore: per questo nel costruttore non passiamo nessun argomento.

Nel codice precedente sono mostrati due segnali tipici dei menu: triggered() viene emesso quando l'utente clicca su una voce di menu. ATTENZIONE: non viene dunque emesso il segnale clicked() come si potrebbe pensare! Invece hovered() è emesso quando il mouse passa sulla voce di menu.
Ultima nota: per le voci di menu è presente anche il metodo setCheckable() (e setChecked(), per specificare l'azione iniziale), identico a quello usato per i toggle button in precedenza: se viene passato il parametro True, l'icona della voce si comporterà proprio come un toggle button.

6. Toolbar

Una toolbar è un insieme di pulsanti che assolvono ad alcune funzioni di base. In alcuni casi può essere più comoda di un menu perché permette all'utente di scegliere l'azione più velocemente.

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):

    def __init__(self):
        QtGui.QMainWindow.__init__(self)

        self.setWindowTitle('Toolbar')

        quit = QtGui.QAction(QtGui.QIcon("icons/cancel.png"), "Quit", self)
        quit.setShortcut("Ctrl+Q")
```

```
                quit.setStatusTip("Quit application")
                self.connect(quit, QtCore.SIGNAL('triggered()'), QtCore.SLOT('close()'))

                sep = QtGui.QAction(self)
                sep.setSeparator(True)

                info = QtGui.QAction(QtGui.QIcon("icons/information.png"), "Info", self)
                info.setShortcut("Ctrl+I")
                info.setStatusTip("Show information")

                self.statusBar().show()

                toolbar = self.addToolBar('My tool')
                toolbar.addAction(quit)
                toolbar.addAction(info)
                toolbar.setToolButtonStyle(QtCore.Qt.ToolButtonTextUnderIcon)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

Come è possibile notare, il codice non è troppo diverso da quello usato per i menu. Le varie "action" si dichiarano sempre allo stesso modo, ma alla fine invece di dichiarare barre e menu, ci limitiamo ad una toolbar.
Il metodo setToolButtonStyle() permette di specificare la presenza del testo e nel caso, la sua posizione. Gli argomenti che possiamo passargli (oltre a quello indicato, che posiziona il testo sotto l'icona) sono: Qt.ToolButtonTextBesideIcon (testo accanto all'icona), Qt.ToolButtonTextOnly (solo testo), Qt.ToolButtonIconOnly (solo l'icona, l'azione predefinita).
Altro metodo è setOrientation(): accetta come argomenti Qt.Horizontal o Qt.Vertical.

7. Dialogs

Le dialog window sono "popup", piccole finestrelle che possono essere dipendenti o meno da quella principale. Vengono utilizzate per visualizzare brevi messaggi o per prendere input.
PyQt4 mette a disposizione diversi dialogs standard per le operazioni più comuni: vediamo in un unico codice come si dichiarano, e come vengono ritornate le informazioni.

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
        def __init__(self):
                QtGui.QMainWindow.__init__(self)
                buttons = [0] * 13
                self.titles = ["Get integer", "Get double", "Get item", "Get text", "Set color", "Set font", "Set
```

```
directory", "Open file", "Save File", "Critical message", "Info message", "Question", "Warning"]
                slots = [self.getInt, self.getDouble, self.getItem, self.getText, self.getColor, self.getFont,
self.getDirectory, self.openFile, self.saveFile, self.Critical, self.Info, self.Question, self.Warning]

                self.resize(350, 250)
                self.setWindowTitle('Dialogs')

                widget = QtGui.QWidget(self)

                grid = QtGui.QGridLayout(widget)
                grid.setVerticalSpacing(10)
                grid.setHorizontalSpacing(8)

                row = 0
                col = 0

                for i in range(13):
                        buttons[i] = QtGui.QPushButton(self.titles[i], widget)
                        self.connect(buttons[i], QtCore.SIGNAL('clicked()'), slots[i])

                        grid.addWidget(buttons[i], row, col)

                        if col == 2:
                                col = 0
                                row += 1
                        else:
                                col += 1

                self.textEdit = QtGui.QTextEdit(widget)
                self.textEdit.setReadOnly(True)
                grid.addWidget(self.textEdit, 5, 0, 1, 3)

                self.setLayout(grid)
                self.setCentralWidget(widget)

        def getInt(self):
                integer, ok = QtGui.QInputDialog.getInteger(self, self.titles[0], "Integer: ", 9, 0, 1000, 1)

                if ok:
                        self.textEdit.append(self.tr("%1").arg(integer))

        def getDouble(self):
                double, ok = QtGui.QInputDialog.getDouble(self, self.titles[1], self.tr("Amount:"), 37.56,
-10000, 10000, 1)

                if ok:
                        self.textEdit.append(self.tr("%1").arg(double))

        def getItem(self):
                items = QtCore.QStringList()
                items << "GNU/Linux" << "Windows" << "Macintosh" << "QNX"

                item, ok = QtGui.QInputDialog.getItem(self, self.titles[2], "OS", items, 0, False)

                if ok and item.isEmpty() == False:
                        self.textEdit.append(item)

        def getText(self):
```

```
                text, ok = QtGui.QInputDialog.getText(self, self.titles[3], "Enter your name:")

                if ok and text.isEmpty() == False:
                        self.textEdit.append(text)

        def getColor(self):
                color = QtGui.QColorDialog.getColor(QtCore.Qt.blue, self)

                if color.isValid():
                        self.textEdit.setTextColor(color)
                        self.textEdit.append(color.name())

        def getFont(self):
                font, ok = QtGui.QFontDialog.getFont()

                if ok:
                        self.textEdit.setFont(font)
                        self.textEdit.append(font.key())

        def getDirectory(self):
                directory = QtGui.QFileDialog.getExistingDirectory(self, self.titles[6], "Select directory:",
QtGui.QFileDialog.ShowDirsOnly)

                if directory.isEmpty() == False:
                        self.textEdit.append(directory)

        def openFile(self):
                fName = QtGui.QFileDialog.getOpenFileName(self, self.titles[7], "Open new file", self.tr("All
Files (*);;Text Files (*txt)"))

                if fName.isEmpty() == False:
                        self.textEdit.append(fName)

        def saveFile(self):
                fName = QtGui.QFileDialog.getSaveFileName(self, self.titles[8], "Save a new file", self.tr("All
Files(*)"))

                if fName.isEmpty() == False:
                        self.textEdit.append(fName)

        def Critical(self):
                reply = QtGui.QMessageBox.critical(self, self.titles[9], "Critical message!",
QtGui.QMessageBox.Abort, QtGui.QMessageBox.Ignore, QtGui.QMessageBox.Retry)

                if reply == QtGui.QMessageBox.Abort:
                        self.textEdit.append("Abort")
                elif reply == QtGui.QMessageBox.Retry:
                        self.textEdit.append("Retry")
                else:
                        self.textEdit.append("Ignore")

        def Info(self):
                QtGui.QMessageBox.information(self, self.titles[10], "Information message")

        def Question(self):
                reply = QtGui.QMessageBox.question(self, self.titles[11], "Are you sure?",
QtGui.QMessageBox.Yes, QtGui.QMessageBox.No)
```

```
                if reply == QtGui.QMessageBox.Yes:
                        self.textEdit.append("Yes")
                else:
                        self.textEdit.append("No")

        def Warning(self):
                reply = QtGui.QMessageBox.warning(self, self.titles[12], "Warning!", "Try again", "Continue")

                if reply == 0:
                        self.textEdit.append("Try again")
                else:
                        self.textEdit.append("Continue")

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

Per guadagnare in chiarezza, abbiamo posto i pulsanti, i testi degli stessi e gli slots in degli array. Il codice all'interno del ciclo for si occupa di inizializzarli e sistemarli all'interno della griglia.

```
                integer, ok = QtGui.QInputDialog.getInteger(self, self.titles[0], "Integer: ", 9, 0, 1000, 1)
```

Il metodo `getInteger()` richiede i seguenti parametri: self, una stringa da usare come titolo, un'altra da inserire all'interno del dialog, e quattro valori numerici. Il primo indica il valore di default, il secondo il valore minimo selezionabile, il terzo il valore massimo, e infine l'ultimo valore indica di quanti numeri si deve andare avanti/indietro quando si premono le freccette a lato. Ad esempio se mettete 2, e poi cliccate sulla freccetta in alto, si passerà da 9 direttamente ad 11.

```
                double, ok = QtGui.QInputDialog.getDouble(self, self.titles[1], self.tr("Amount:"), 37.56,
-10000, 10000, 1)
```

Cosa simile vale per l'input di valori double, ovvero con virgola mobile.

```
                items << "GNU/Linux" << "Windows" << "Macintosh" << "QNX"
```

```
        item, ok = QtGui.QInputDialog.getItem(self, self.titles[2], "OS", items, 0, False)
```

In questo dialog, ci viene proposta una scelta fra degli elementi predefiniti. La variabile items li contiene. Gli ultimi due parametri di getItem() indicano rispettivamente l'indice della stringa da visualizzare di default e un'indicazione rispetto al layout. Provare a sostituire False con True per vedere la differenza.

```
    def getColor(self):
        color = QtGui.QColorDialog.getColor(QtCore.Qt.blue, self)
```

Qui l'unico parametro interessante è il primo: il colore di default. È un parametro opzionale, e se viene omesso è settato a QtCore.Qt.white.

8. Altri widgets

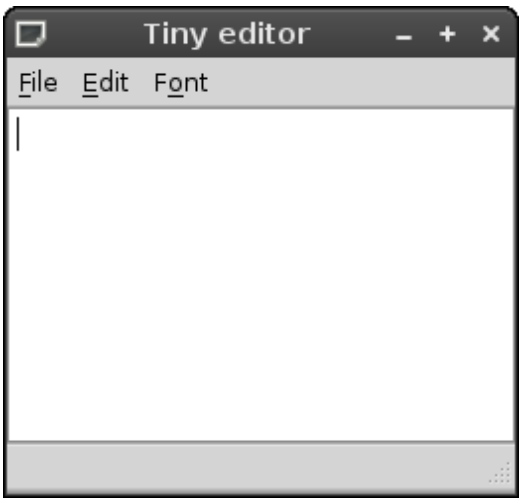
- **textEdit**

Anche se abbiamo già incontrato questo widget in precedenza, in realtà esistono altri metodi di cui non abbiamo parlato. Il prossimo codice ne usa una buona parte per creare un rudimentale editor, di nome Tiny Editor.

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
    def createMenuVoice(self, iconPath, name, shortcut, tip, slot):
        voice = QtGui.QAction(QtGui.QIcon(iconPath), name, self)
        voice.setShortcut(shortcut)
        voice.setStatusTip(tip)
        self.connect(voice, QtCore.SIGNAL('triggered()'), slot)

        return voice

    def createSeparator(self):
        sVoice = QtGui.QAction(self)
        sVoice.setSeparator(True)

        return sVoice

    def createMenu(self):
        tinyMenu = self.menuBar()

        file = tinyMenu.addMenu("&File")
        edit = tinyMenu.addMenu("&Edit")
        font = tinyMenu.addMenu("F&ont")

        new = self.createMenuVoice("icons/new.png", "New", "Ctrl+N", "New file", self.textEdit.clear)
        open = self.createMenuVoice("icons/open.png", "Open file", "Ctrl+O", "Open a new file",
self.openNewFile)
        save = self.createMenuVoice("icons/save.png", "Save file", "Ctrl+S", "Save file",
self.saveNewFile)
        sep = self.createSeparator()
        quit = self.createMenuVoice("icons/quit.png", "Quit", "Ctrl+Q", "Quit TinyEditor",
```

```
QtCore.SLOT('close()'))
                    file.addAction(new)
                    file.addAction(open)
                    file.addAction(save)
                    file.addAction(sep)
                    file.addAction(quit)

                    undo = self.createMenuVoice("icons/undo.png", "Undo", "Ctrl+U", "Undo operation",
self.textEdit.undo)
                    redo = self.createMenuVoice("icons/redo.png", "Redo", "Ctrl+R", "Redo operation",
self.textEdit.redo)
                    sep1 = self.createSeparator()
                    cut = self.createMenuVoice("icons/cut.png", "Cut", "Ctrl+X", "Cut selected text",
self.textEdit.cut)
                    copy = self.createMenuVoice("icons/copy.png", "Copy", "Ctrl+C", "Copy selected text",
self.textEdit.copy)
                    paste = self.createMenuVoice("icons/paste.png", "Paste", "Ctrl+V", "Paste text", self.paste)
                    sep2 = self.createSeparator()
                    selectAll = self.createMenuVoice("icons/selectAll.png", "Select all", "Ctrl+A", "Select all text",
self.textEdit.selectAll)
                    edit.addAction(undo)
                    edit.addAction(redo)
                    edit.addAction(sep1)
                    edit.addAction(cut)
                    edit.addAction(copy)
                    edit.addAction(paste)
                    edit.addAction(sep2)
                    edit.addAction(selectAll)

                    setFont = self.createMenuVoice("icons/setfont.png", "Set font", "Ctrl+F", "Set font",
self.setFont)
                    underline = self.createMenuVoice("icons/underline.png", "Underline", "Ctrl+L", "Underline
text", self.underline)
                    italic = self.createMenuVoice("icons/italic.png", "Italic", "Ctrl+I", "Set text to italic", self.italic)
                    tColor = self.createMenuVoice("icons/color.png", "Set text color", "Ctrl+R", "Set text color",
self.setColor)
                    delete = self.createMenuVoice("icons/delete.png", "Delete format", "Ctrl+D", "Delete format",
self.deleteFormat)
                    font.addAction(setFont)
                    font.addAction(underline)
                    font.addAction(italic)
                    font.addAction(tColor)
                    font.addAction(delete)

          def __init__(self):
                    QtGui.QMainWindow.__init__(self)

                    self.setWindowTitle('Tiny editor')

                    self.textEdit = QtGui.QTextEdit()
                    self.textEdit.setReadOnly(False)

                    if self.textEdit.isUndoRedoEnabled() == False:
                              self.textEdit.setUndoRedoEnabled(True)

                    self.createMenu()
                    self.statusBar()
                    self.setCentralWidget(self.textEdit)
```

```
        def openNewFile(self):
                fName = QtGui.QFileDialog.getOpenFileName(self, "Open text file", "Open new file",
self.tr("Text Files (*.txt)"))

                if fName.isEmpty() == False:
                        fptr = open(fName, 'r')

                        content = fptr.read()
                        self.textEdit.append(content)
                        fptr.close()

        def saveNewFile(self):
                fName = QtGui.QFileDialog.getSaveFileName(self, "Save text file", "Save a new file",
self.tr("Text Files (*.txt)"))

                if fName.isEmpty() == False:
                        fptr = open(fName, 'w')

                        fptr.write(self.textEdit.toPlainText())
                        fptr.close()

        def paste(self):
                if self.textEdit.canPaste():
                        self.textEdit.paste()
                else:
                        QtGui.QMessageBox.critical(self, "Error", "Impossible to paste text",
QtGui.QMessageBox.Ok)


        def setFont(self):
                font, ok = QtGui.QFontDialog.getFont()

                if ok:
                        self.textEdit.setFont(font)

        def underline(self):
                self.textEdit.setFontUnderline(True)

        def italic(self):
                self.textEdit.setFontItalic(True)

        def setColor(self):
                color = QtGui.QColorDialog.getColor(QtCore.Qt.blue, self)

                if color.isValid():
                        self.textEdit.setTextColor(color)

        def deleteFormat(self):
                self.textEdit.setFontUnderline(False)
                self.textEdit.setFontItalic(False)
                self.textEdit.setTextColor(QtCore.Qt.black)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

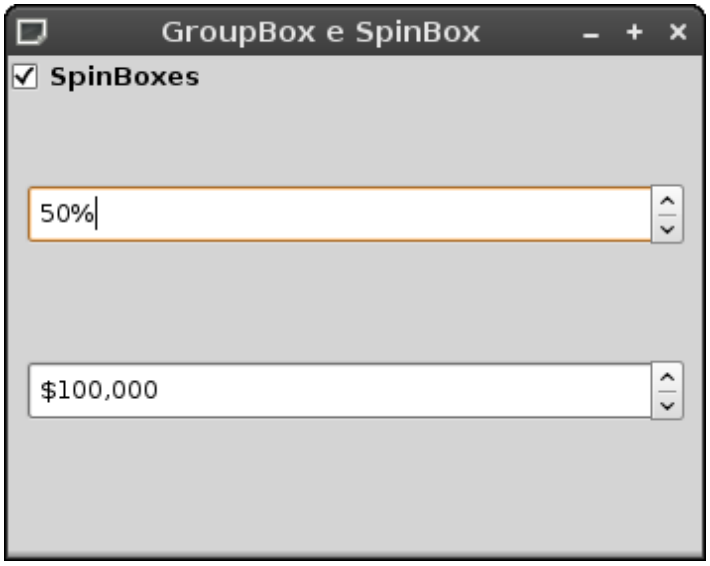


Il tutto è piuttosto intuitivo.
Come notate, la maggior parte delle azioni tipiche di un editor (taglia, copia, incolla, seleziona tutto, annulla operazione...) non hanno bisogno di un'implementazione a parte del programmatore, ma ci si può avvalere dei metodi predefiniti del widget.
Come noterete, sono riapparsi anche precedenti dialog predefiniti.
Come esercizio potreste tentare di migliorare questo editor!

- GroupBox e SpinBox
Vediamo altri due widgets utili: GroupBox e SpinBox. Il primo serve a dividere diverse sezioni di programma tramite dei titoletti in “grassetto”, mentre la seconda è utile per l'input di dati numerici.

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
        def __init__(self):
                QtGui.QMainWindow.__init__(self)

                self.resize(350, 250)
                self.setWindowTitle('GroupBox e SpinBox')

                widget = QtGui.QWidget(self)

                self.gBox = QtGui.QGroupBox("SpinBoxes", widget)
                self.gBox.setCheckable(True)
                self.gBox.setChecked(True)
                self.connect(self.gBox, QtCore.SIGNAL('clicked()'), self.showSpin)

                self.iSpin = QtGui.QSpinBox(widget)
                self.iSpin.setRange(0, 100)
                self.iSpin.setSuffix("%")
                self.iSpin.setValue(50)
                self.iSpin.setSingleStep(5)

                self.dSpin = QtGui.QDoubleSpinBox(widget)
                self.dSpin.setDecimals(3)
                self.dSpin.setPrefix("$")
                self.dSpin.setRange(0, 1000.0)
                self.dSpin.setSingleStep(0.5)
```

```
                self.dSpin.setValue(100)

                vBox = QtGui.QVBoxLayout(widget)
                vBox.setSpacing(3)

                vBox.addWidget(self.iSpin)
                vBox.addWidget(self.dSpin)

                widget.setLayout(vBox)
                self.setCentralWidget(widget)

        def showSpin(self):
                if self.gBox.isChecked():
                        self.dSpin.show()
                        self.iSpin.show()
                else:
                        self.dSpin.hide()
                        self.iSpin.hide()
```



In questo caso, associamo alla GroupBox una checkbox che ci permette di decidere fra due comportamenti. In realtà una GroupBox di default appare solo come un titoletto in grassetto, senza alcuna checkbox. Quando si cambia lo stato della checkbox viene emesso un segnale clicked().

```
                if self.gBox.isChecked():
```

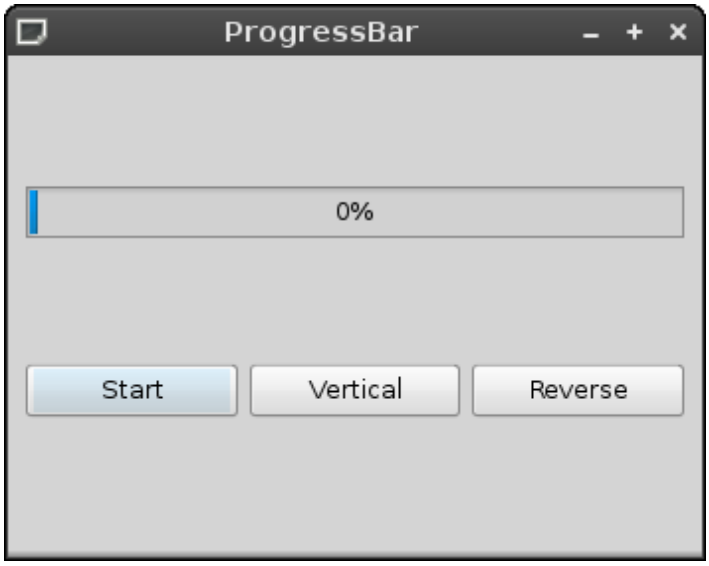
Se questo metodo ritorna True, la checkbox è stata attivata, altrimenti è stata disattivata.
Come vediamo, esistono due principali tipi di spinbox: la classica QSpinBox, che contiene interi, e la QDoubleSpinBox, che contiene valori double.
È possibile specificare un range di valori accettabili, un valore di default, il “salto” fra due valori contigui, e stringhe da usare come suffissi o prefissi dei valori numerici.

- ProgressBar

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore
```

```
class MainWindow(QtGui.QMainWindow):
        def __init__(self):
                QtGui.QMainWindow.__init__(self)

                self.resize(350, 250)
                self.setWindowTitle('ProgressBar')
                widget = QtGui.QWidget()

                grid = QtGui.QGridLayout(widget)
                self.progressBar = QtGui.QProgressBar(widget)
                self.progressBar.setRange(0, 100)
                self.progressBar.setValue(0)
                self.progressBar.setTextVisible(True)

                self.button = QtGui.QPushButton('Start', widget)
                self.connect(self.button, QtCore.SIGNAL('clicked()'), self.StartProgress)

                self.horiz = QtGui.QPushButton('Vertical', widget)
                self.horiz.setCheckable(True)
                self.connect(self.horiz, QtCore.SIGNAL('clicked()'), self.changeOrientation)

                self.direction = QtGui.QPushButton('Reverse', widget)
                self.direction.setCheckable(True)
                self.connect(self.direction, QtCore.SIGNAL('clicked()'), self.Reverse)

                grid.addWidget(self.progressBar, 0, 0, 1, 3)
                grid.addWidget(self.button, 1, 0)
                grid.addWidget(self.horiz, 1, 1)
                grid.addWidget(self.direction, 1, 2)

                self.timer = QtCore.QBasicTimer()
                self.step = 0

                widget.setLayout(grid)
                self.setCentralWidget(widget)

        def Reverse(self):
                if self.direction.isChecked():
                        self.progressBar.setInvertedAppearance(True)
                else:
                        self.progressBar.setInvertedAppearance(False)

        def changeOrientation(self):
                if self.horiz.isChecked():
                        self.progressBar.setOrientation(QtCore.Qt.Vertical)
                else:
                        self.progressBar.setOrientation(QtCore.Qt.Horizontal)

        def timerEvent(self, event):
                if self.step >= 100:
                        self.timer.stop()
                        return
                self.step = self.step+1
                self.progressBar.setValue(self.step)

        def StartProgress(self):
                if self.timer.isActive():
```

```
                        self.timer.stop()
                        self.button.setText('Start')
                else:
                        self.timer.start(100, self)
                        self.button.setText('Stop')

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```



La dichiarazione della progress bar è questa:

```
                self.progressBar = QtGui.QProgressBar(widget)
                self.progressBar.setRange(0, 100)
                self.progressBar.setValue(0)
                self.progressBar.setTextVisible(True)
```

Nota: i tre metodi che chiamo dopo il costruttore in realtà potrebbero essere omessi, in quanto quelli specificati sono già i comportamenti di default. Li ho inseriti solamente a scopo dimostrativo.

```
                self.timer = QtCore.QBasicTimer()
                self.step = 0
```

Qui dichiariamo un timer, un oggetto di basso livello molto semplice e adatto alle nostre esigenze. Ogni tot millisecondi (il valore precisato nel metodo start()), il timer genera un evento che noi intercettiamo con la nostra funzione. Ogni 100 millisecondi dunque avanziamo di 1 nella progress bar.
Se vogliamo che essa sia più veloce, dobbiamo ovviamente diminuire il valore numerico passato a start().
Le due funzioni importanti sono Reverse e changeOrientation: essi, a seconda del valore del toggle button corrispondente, “rovesciano” il testo della progress bar oppure il suo orientamento.

- Tooltip, password e combo box
Finiamo con tre widget.

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
        def __init__(self):
                QtGui.QMainWindow.__init__(self)

                self.resize(350, 250)
                self.setWindowTitle('Tooltip e password')
                widget = QtGui.QWidget()
                hbox = QtGui.QHBoxLayout(widget)
                hbox.setSpacing(10)

                self.label = QtGui.QLabel("Enter your username", widget)
                self.line = QtGui.QLineEdit(widget)
                comboBox = QtGui.QComboBox()
                comboBox.addItem("Username")
                comboBox.addItem("Password")
                self.connect(comboBox, QtCore.SIGNAL("activated(int)"), self.changeEcho)

                hbox.addWidget(self.label)
                hbox.addWidget(self.line)
                hbox.addWidget(comboBox)
                self.setToolTip('This is a tooltip')
                self.setCentralWidget(widget)

        def changeEcho(self, index):
                if index == 0:
                        self.line.setEchoMode(QtGui.QLineEdit.Normal)
                        self.label.setText("Enter your username")
                else:
                        self.line.setEchoMode(QtGui.QLineEdit.Password)
                        self.label.setText("Enter your password")

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

Il widget QLineEdit è comodo per inserire piccole porzioni di testo. Se vogliamo una maggiore sicurezza possiamo anche impostare il tipo di “echo” a password, come viene fatto nello slot.
La combo box permette all'utente di scegliere diverse opzioni: la selezione è catturata dal segnale activated(), che manda al proprio slot un parametro intero, ovvero l'indice della voce momentaneamente attivata.
Il tooltip può essere visto soffermando il cursore su un punto qualsiasi della finestra.

Conclusioni.
Se avete qualsiasi suggerimento, correzione ai codici, miglioria che volete vedere in questo tutorial, scrivetemi pure a syn.shainer@gmail.com

Lisa