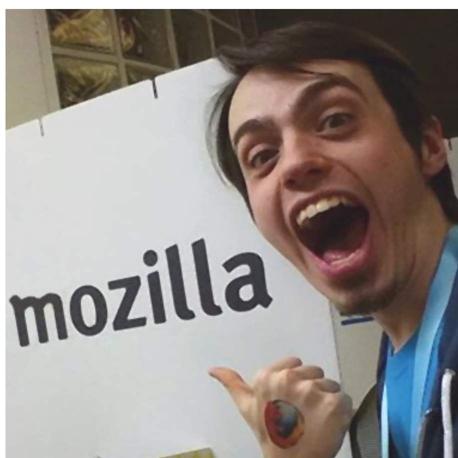
Introduzione a PyQT 2 Ed.

Premessa

Questa guida/tutorial (molto informale) è stata scritta perchè di questo ambiente Python-QT esiste poco materiale, incompleto e con pochi esempi.
Premetto che Python non mi piace, I suoi trackback sono inutili la maggior parte delle volte e l'errore riportato è troppo spesso generico.
Come punto di riferimento per il codice verranno utilizzati miei programmi LearnHotkeys (il predefinito) e Plessc oltre che al boilerplate.
La prima edizione di questa guida è stata pubblicata il 21/04/2013.
Nella guida non è prevista una sezione per l'installazione di PyQT quindi usate Google :-)

Su di me

Utente Debian Gnu/Linux dal 2009 e contributore OpenSource.
Volontario Mozilla, membro di KDE Italia, LibreItalia e Dartlang Italia, Blogger per il Network AndMore dove scrivo articoli dedicati a HTML5, jQuery e Wordpress. Sulla fiducia è diventato da poco redattore su Chimera Revo con carta bianca su qualunque argomento. Inoltre sono stato uno speaker più volte riguardo questi argomenti.
Possiede un canale Youtube dove parla di Open Source, informatica e di programmazione. Co Founder di Codeat una web agency romana, codeat.it.



Twitter: @Mte90.Net
G+: https://plus.google.com/u/0/+DanieleScasciafratteMte90Net
Linkedin: http://www.linkedin.com/profile/view?id=173136999
Sito: www.mte90.net

Guida rilasciata sotto GPLv3 - http://gplv3.fsf.org/ -
Unica richiesta per la diffusione di questo libro è lasciare questa pagina.
Per segnalazioni, errata e commenti contattatemi sui miei profili social. Grazie!!

Indice

Introduzione
– Cos'è PyQT 3
– Differenze tra Python 2 e 3 3
Programmiamo 4
– Vai con la GUI 4
– Pattern per le nomenclature 4
– Includere il form 5
GUI o non GUI 6
– QtDesigner 6
– Cosa sono I layout 7
– Daje con I Widget 7
I segnali e gli slot 9
– Cosa sono 9
– Proviamo un segnale 9
Soluzioni e codice già pronto 10

È richiesta una conoscenza basilare di programmazione ad oggetti per comprendere la guida.

Introduzione

Cos'è PyQT

PyQT è il primo bindings (collegamento tra due entità differenti) QT (si pronuncia cute) per il linguaggio Python creato da un privato (Riverbank) molti anni fà e mantenuto ancora oggi.
Il secondo bindings è **PySide** creato dalla Nokia perchè **PyQT** ha una licenza commerciale che Nokia non voleva. L'API tra i due è compatibile ma il supporto per i tools di programmazione **PySide** ancora non ha raggiunto il livello di quelli di PyQT. Per un maggior approfondimento delle differenze vi rimando al link dedicato su qt-project.org.

Differenze tra Python 2 e 3

Il supporto per Python 2.x e 3.x per PyQT è differente. Non sono riuscito a trovare un'elenco completo e chiaro ma l'unica differenza che ho trovato sta nell'utilissima funzione .toString().
Chi conosce il .NET Framework l'avrà già sentita nominare, converte l'oggetto o il suo contenuto in stringa ed è molto utile per verificare se esiste o contiene qualcosa.
Per chi lavora in JavaScript può essere utilizzato come il lenght per verificare se l'oggetto esiste.
L'unico esempio (ripetuto più volte nel programma in modi diversi) potete vederlo nel file defdialog.py alla riga 29.
Se si stampa il contenuto di un oggetto l'output è una **QVariant**, che è una classe unitiva di tutti i tipi di dati in ambito Qt e tramite questa classe si ottiene qualunque tipo di oggetto.
Se si usa Python 2.x l'output è un oggetto di tipo **QVariant** mentre in Python 3.x è una stringa generica.
Le Classi **QVariant** e **QString** sono state rimosse in PyQt per Python 3 per supportare le funzionalità native di Python.
Se si usa la funzione toString() in Python 3.x si ottiene un errore, quindi problemi di compatibilità, l'unica soluzione è quella usata in LearnHotkeys ovvero verificare la versione di Python ed eseguire la funzione supportata.
Python supporta Unicode, in Python 2 tramite le **PyQT** bisognava convertire il testo ma in Python 3.x il testo è in questo formato mentre in Python 2.x è di tipo Qstring.

In poche parole il concetto <è usiamo lo standard Python e le sue cose a livello nativo per standardizzare e semplificare il codice invece di supportare cose non necessarie>.

Tutto questo e altri dettagli nelle differenze si possono trovare nella pagina dedicata al libro "Rapid GUI Programming With Python and QT" (in mio possesso :-D).

Programmiamo!

Vai con la GUI

Qt Designer è il programma per la creazione di interfacce QT che vengono salvate in formato XML in file .ui.
QtCreator e **QtDesigner** ancora non supportano Python quindi useremo solamente **QtDesigner** per la realizzazione di interfacce, per poterlo fare al suo avvio scegliete Main Window così verrà creato un nuovo form o finestra.
Per chi ha usato **Visual Studio** si troverà a casa, per chi ha usato **Glade** si troverà davanti un mondo e gli altri diranno dafuq!?
La spiegazione di questo strumento la approfondiremo nella seconda parte perchè prima di divertirsi con le finestre abbiamo altro da vedere.
Tutti gli esempi d'uso per **PyQT** che ho trovato convertono i file .ui in codice Python tramite pyuic. Per semplificare questo lavoro noioso lo scrittore del libro, già citato, ha scritto un tool grafico che data la cartella effettua la conversione di tutti i file .ui in (codice Python) .py, potete scaricarlo tramite questo link.
Vista la comodità utilizzo anch'io questo script quando devo lavorare con le gui in Python.
Questa conversione viene effettuata per avere un miglior debug e supporto degli editor che non riconoscono questo formato mentre il codice Python lo elaborano senza problemi.

Pattern per le nomenclature

Ogni sviluppatore ha il suo metodo per le nomenclature. Visto che si tratta di widget conviene utilizzare il camelCase (utilizzato dalle QT per praticamente tutto). Il metodo utilizzato è molto semplice, il primo termine è un abbreviazione del tipo di widget e poi la definizione dello stesso. Per esempio se si tratta del pulsante per il download lo chiameremo pushDownload (il pulsante in QT si chiama **PushButton**) mentre se si tratta di textbox io vi consiglio di inserire direttamente la definizione perchè sono i widget più usati nel codice quindi è preferibile semplificare il nome.

Codice Basilare

Le prime righe sono le seguenti:

```
#!/usr/bin/env python
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

Per chi conosce Bash la prima riga gli è familiare, serve ad indicare con quale interprete eseguire lo script, nel nostro caso Python. Le righe successive servono a Python per caricare i moduli necessari a QT. Il primo è il cuore quindi tutto ciò che non riguarda la gui

(l'essenziale ovviamente) e il secondo proprio la parte relativa alla gui.
Per chi volesse usare Python3 è deve cambiare la prima riga, invece di */usr/bin/env python* si scriverà */usr/bin/env python3*. Se nel sistema è presente la versione 3 verrà eseguita altrimenti lo script non verrà eseguito.

Includere il form

L'inclusione del form realizzato con QtDesigner e poi convertito in Python è molto semplice!
La riga da aggiungere è

```
from ui_mainwindow import Ui_MainWindow
```

Naturalmente sostituite il nome del form (MainWindow) con il vostro. La prima parte è il nome del file e il secondo è il nome della classe che corrisponde al nome del form inserito in QtDesigner.
Aggiungete la riga che crea la classe nel file python e che inizializza la gui:

```
class MainWindow ( QMainWindow , Ui_MainWindow):
```

Sostituite il nome del form con il vostro e cambiate anche il nome della classe se volete.
All'interno della classe create una funzione di nome **init** in questo modo:

```
def __init__ ( self, parent = None ):
    QMainWindow.__init__( self, parent )
    self.ui = Ui_MainWindow()
    self.ui.setupUi( self )
    self.show()
```

Con questo codice inizializzate il form e lo mostrate ma non è finita qui!
Con il codice fuori dalla classe verrà creata una **QApplication** contenente la classe che abbiamo appena creato:

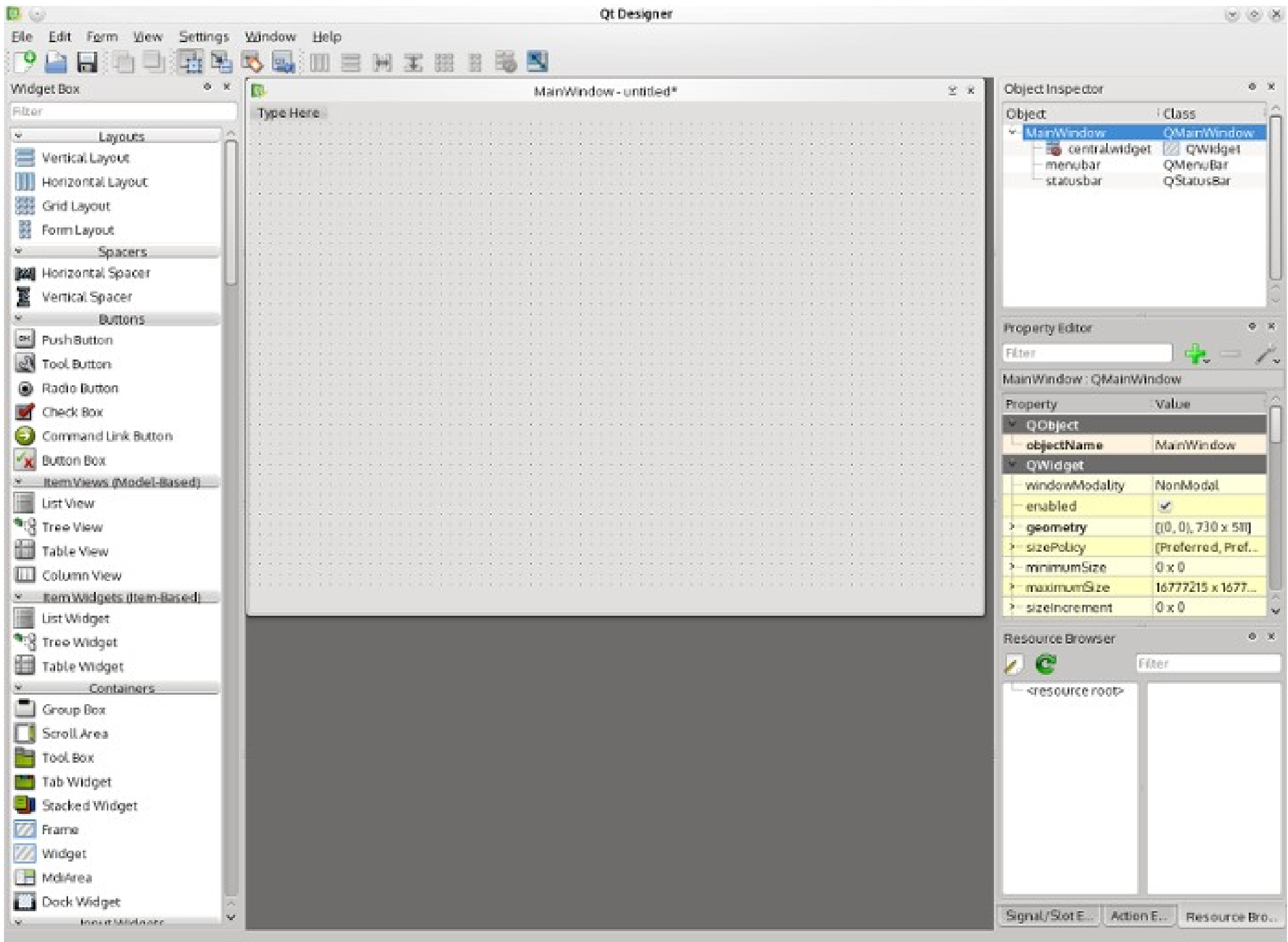
```
def main():
    app = QApplication(sys.argv)
    MainWindow_ = QMainWindow()
    ui = MainWindow()
    ui.setupUi(MainWindow_)
    sys.exit(app.exec_())

main()
```

Come al solito rimpiazzate i nomi della classe con il vostro.

Gui o non Gui

QtDesigner

Installate sul vostro computer QtDesigner, apritelo e scegliete Main Window nella finestra che si aprirà in questo modo verrà creato un nuovo form generico come da foto.



Come da screenshot potete vedere che il programma è diviso in tre colonne.
La prima colonna contiene i widget che in italiano potrebbero essere tradotti come elementi grafici differenti tra loro (per funzioni e caratteristiche) che permettono una interazione da parte dell'utente.
Questi widget sono l'elemento fondamentale di ogni toolkit grafico (GTK, Qt e wxWidgets per esempio) perchè semplificano allo sviluppatore la necessità di dover sviluppare le caratteristiche per ogni widget essendo già presenti nel framework.
Per esempio prendiamo il campo di testo, in QT ne esistono diversi tipi a seconda delle esigenze:

- QLineEdit è un editor di testi da una sola riga (supporto copia, incolla, taglia, indietro, avanti e drag and drop)

- QTextEdit supporta testo puro e Rich Text (formattazione del testo tramite css)

- QPlainTextEdit è un editor di testi avanzato per testo puro

Nella colonna centrale è presente il form su cui trascinare i widget di cui abbiamo bisogno.
Nella terza colonna abbiamo diverse schede, i più importanti per lo sviluppo in Python sono: Object inspector che contiene un elenco ad albero dei widget inseriti nel form e Property Editor che permette di gestire le varie proprietà di un widget.
Gli altri pannelli come Signal and Slot sono utili per chi sviluppa in C e vuole integrare direttamente i segnali da file ui (approfondiremo i segnali più avanti).
I segnali secondo il mio parere è preferibile farli da codice in modo che la ui sia indipendente dal codice.

Cosa sono i layout

Le Qt hanno il concetto dei layout per i widget, questi hanno lo scopo di disporre i widget a seconda del loro contenuto e della dimensione della finestra.
LearnHotkeys all'inizio non li sfruttava e il ridimensionamento della finestra sballava tutto e i widget non erano allineati tra di loro.
Riorganizzando l'interfaccia usando i layout è diverso, è possibile notare cosa succede quando si ridimensiona la finestra o cambia il contenuto delle label che portano al riallineamento dei widget.
Per prenderci la mano è necessaria molto pratica quindi ecco qualche dritta per semplificarvi l'uso!
Per poter sfruttare al meglio i layout conviene combinarli insieme a seconda della disposizione che si vuole avere. Su QtDesigner i layout si riconoscono dai bordi rossi.
Un esempio di combinazione non inclusiva la potete vedere aprendo il file mainwindow.ui su QtDesigner. Il form è diviso in due layout diversi con a loro volta altri layout.
Per un layout inclusivo potete vedere il file *editor.ui* dove c'è un solo layout che ne contiene altri.
La peculiarità fra i due tipi è se è presente un solo layout inclusivo può essere impostato come layout principale del form quindi ne assumerà le dimensioni totali mentre l'altro tipo (ovvero senza un genitore) assumerà come sue dimensioni quelle impostate.
Come creare un layout principale è molto semplice!
Seleziona un layout e trascinatelo sul form (valutate quale usare solitamente uso il vertical layout con un grid o un horyzontal). Fate un click col tastro destro sul form, poi andate in layout e impostate il tipo di layout.

Daje con i Widget

Fatta la premessa sui widget trasciniamone uno sul form!
Come potrete vedere prenderà subito le dimensioni datele dal layout, invece tramite la funzione vista poco fà impostando Break Layer le dimensioni andranno impostate

manualmente.
Se avrete il focus su un widget potrete vedere in Property Editor le sue proprietà.
La principale che fà parte della classe QObject è quella che definisce il nome del widget (per un pattern di nomenclatura fate riferimento alla puntata precedente).
In QWidget invece ci sono le proprietà generiche di ogni widget tra cui le dimensioni o il font.
In QAbstract* ci sono le proprietà generiche della famiglia del widget se si parla di pulsanti avremo l'icona o il toggle (nelle Qt si chiama checkable), se si tratta di liste avremo il drag and drop per l'ordinamento e così via.
Inoltre a seconda del widget avremo le sue proprietà specifiche o altre proprietà dovute all'insieme di widget di cui è composto.
Questa modularità (o ereditarietà) dei widget è interessante perchè semplifica l'API rendendola uniforme e non comporta una grande difficoltà per ricordarsi tutte queste proprietà nella scrittura del codice.

Il mio consiglio è di avere sempre sottomano la documentazione delle Qt per trovare subito la proprietà o metodo del widget di cui si ha bisogno. Potete farlo tramite il sito ufficiale o con QtAssistant.

I segnali e gli slot

Cosa sono

I segnali e gli slot sono una delle peculiarità e fondamento del framework Qt.
Gli altri framework e linguaggi sfruttano il sistema di callback che consiste in una funzione scatenata all'avvenimento dell'evento. Il segnale non è altro che il tipo di evento che attende l'esecuzione dello stesso e scatena lo slot che potremmo definire callback.
La peculiarità è che il segnale può scatenarne altri a sua volta o possono essere sviluppati ad hoc mentre lo slot è una classica funzione.
N.B una callback può essere anche una funzione mentre un segnale accetta solo funzioni.

Proviamo un segnale

Un esempio di segnale possiamo vederlo in *learnhotkeys.pyw* alla riga riga 33.

```
self.ui.newQuestionButton.clicked.connect(self.new_question)
```

Il segnale in questione si tratta di clicked su un **pushButton** a cui è passata una funzione.
Ogni widget ha i propri segnali che possono essere letti nella pagina dedicata del widget sulla documentazione che contiene inoltre anche tutte i metodi.

La guida si chiude perché è tempo di vedere degli esempi, anche avanzati, con le Qt che nel mio sviluppo ho scritto o cercato. Questi snippet mostrano le potenzialità del framework Qt in Python.

La maggior parte del codice degli esempi trovato sulla rete in C++ e poi lo ho convertito in Python perché il framework è lo stesso quindi se non si trova codice Python basta cambiare il linguaggio.

Soluzioni e Codice già pronto

Chiudere la GUI da shell

Per permettere la chiusura del programma dalla console è molto facile!
Se provate con il classico Ctrl + C in console l'applicazione non viene uccisa quindi è richiesta qualche riga di codice in più.
Aggiungete tra gli import:

```
import signal
```

Poi dentro la funzione **init**

```
signal.signal(signal.SIGINT, signal.SIG_DFL)
```

Salviamo le impostazioni con QSettings

QSettings è quella classe che semplifica il salvataggio e la gestione delle configurazioni di un programma o script. Come si può vedere alla riga 10:

```
settings = QSettings('Mte90','LearnHotkeys')
settings.setFallbacksEnabled(False)
```

In questo modo inizializziamo la classe in questione, creando in ambiente unix, una cartella in /home/utente/.config/Mte90/ e impostando solo l'uso di file per il salvataggio di dati.
Il primo valore da inserire è il nome del produttore e poi il nome del programma, (ovviamente ognuno è libero di fare quel che vuole) nella cartella verrà creato un file a seconda della piattaforma (con una sintassi propria ma genericamente si tratta del formato .INI). In ambiente Windows e OSX il percorso del file sarà diverso ma non c'è bisogno di preoccuparsi di questo per usare questa classe, visto che non sarà mai necessario editare manualmente il file.

Per inserire un valore è molto semplice basta guardare la riga 47:

```
self.settings.setValue("file_name_default",
self.ui.comboDef.currentText())
```

Prima si inserisce il nome del valore/proprietà/opzione/ecc e poi il suo contenuto che verrà convertito in stringa, quindi se si salvano numeri bisognerà ricordarsi questo dettaglio.
Per leggere il contenuto di un valore:

```
self.settings.value('file_name_default')
```

Via con le MessageBox

Le messagebox sono i classici alert che contengono informazioni veloci.
Un esempio a riga 52:

```
QMessageBox.information(self.window(), "LearnHotkeys","The file
list has been updated.")
```

Con il metodo information avremo l'icona classica della i, question avremo il triangolo giallo con il ?, con warning avremmo una X su sfondo rosso e con about potremmo inserire l'icona di nostro piacimento.
L'uso è molto semplice, nel primo parametro impostiamo la finestra madre, poi il titolo e alla fine il contenuto, con gli altri parametri è possibile scegliere quali pulsanti mostrare.
Questa classe è molto ampia e permette di creare degli alert o dialog semplicemente gestendo i parametri ma non fà parte di questa guida quindi sperimentate.

Aprire una seconda finestra

Come si può vedere dalla riga 102 il codice per aprire una seconda finestra è uguale a quello di apertura della principale solo che per aprirla si usa la funzione *exec_*.
In questa funzione serviva un sistema per riconoscere quanto la finestra veniva chiusa e tramite la riga 107 è possibile farlo.
Per il passaggio di variabili tra una finestra e l'altra non sono riuscito a trovare un sistema e l'unica soluzione è stata quella di usare QSettings.

Come aprire una Dialog

Per aprire una dialog basta una sola riga di codice!
Dialog di apertura file:

```
self.input_file = QFileDialog.getOpenFileName(self, 'Choose
file',self.ui.inputFile.text(),'file (*.ext)')
```

Dialog di salvataggio file:

```
self.output_file = QFileDialog.getSaveFileName(self, 'Choose
file',self.ui.inputFile.text(),'file (*.ext)')
```

Nella variabile verrà inserito il percorso del file, semplice no?
Il primo parametro è il contenitore, il secondo è il titolo della finestra, il terzo il percorso di default e il quarto è il contenuto della lista per i vari formati che si può ampliare sfruttando il carattere | aggiungendo altre estensioni.

Come aprire una Dialog di input

Sto parlando delle finestre di prompt che in alcuni soluzioni sono più veloci rispetto a creare un nuovo fomr con un solo campo di testo.

```
key, ok = QInputDialog.getText(self, 'Authorization Key', 'Insert the key:', QLineEdit.Normal, self.settings.value('Key'))
```

Come possiamo vedere ci sono due variabili che aspettano l'output perché **QinputDialog** ritorna due valori. In key in questo caso abbiamo il testo inserito e con ok possiamo verificare se è stato inserito un valore (in poche parole se la finestra è stata chiusa con il pulsante del prompt invece della X).

Il prompt però permette di inserire anche un testo vuoto quindi se necessario è preferibile verificare il valore ottenuto.

Come cambiare cursore

In ambiente desktop per far capire all'utente lo status del programma la soluzione più semplice è quella di cambiare il cursore. Con questo comando impostiamo un cursore di attesa nella classe **QCursor.**

```
Qapplication.setOverrideCursor(QCursor(Qt.WaitCursor))
```

Come fare una lista di checkbox

Per poter ottenere questo tipo di widget bisogna combinare diverse cose una **QlistView**, un **QstandardItemModel** e del **codice** per generare questa lista.

Prendendo il mio progetto Qasana a riga 110:

```
qsubtasks = QstandardItemModel()

for i in proj_tasks:

    item = QStandardItem(i['name'])

    item.setCheckState(Qt.Unchecked)

    item.setCheckable(True)

    qsubtasks.appendRow(item)

self.ui.listTasks.setModel(qsubtasks)
```

```
qsubtasks.itemChanged.connect(self.checkTasks)
```

Facciamo una variabile con inizializzata la classe **QstandardItemModel**, facciamo un ciclo per generare questo modello con **QstandardItemModel** ed impostiamo I suoi stati.

Alla fine impostiamo il modello e gli associamo uno slot quando viene cambiato lo stato dell'elemento.

Come avviare un processo

QProcess è la classe per poter avviare processi ed avere l'output (stdout e famiglia) degli stessi.

Quindi facendo un:

```
selfproc = Qprocess()

self.proc.finished.connect(self.checkLog)
```

Associamo uno slot alla fine dell'esecuzione del comando!
All'interno della funzione *checkLog* potremo elaborare l'output, ho realizzato una funzione che pulisca l'output togliendo I codici bash per l'accapo.

Con un codice del genere prima chiudiamo il canale di **QProcess** per poter associare un'altro comando.

```
self.proc.closeWriteChannel()

self.proc.start(command)

self.proc.waitForFinished()

self.proc.closeWriteChannel()
```

Con start inseriamo il comando e con I comandi successivi aspettiamo che finisca l'elaborazione.

Come realizzare l'autocompletamento nelle aree di testo

Con la classe **QCompleter** possiamo semplificare questo lavoro:

```
completer_input =
```

```
Qcompleter(self.history_field['input'],self.ui.inputFile)

self.ui.inputFile.setCompleter(completer_input)
```

Inizializziamo la classe passandogli una lista con i termini e la textbox e poi lo associamo.

Se aggiorniamo la lista in altri punti del codice verrà aggiornato anche l'autocompletamento.

Come fare un watcher

Con watcher ci si riferisce ad una funzione che verifica se il file ha subito delle modifiche.

Conviene innanzitutto verificare se il watcher è aperto in modo da chiuderlo per poterlo usare:

```
try:

        self.watcher.fileChanged.disconnect()

except (RuntimeError, TypeError, NameError):

        pass
```

Dopodichè aggiungiamo i percorsi e lo slot per eseguire il codice se il file è cambiato:

```
self.watcher.addPath(self.settings.value('input_file'))

self.watcher.fileChanged.connect(self.compileIt)
```

Come aprire una seconda finestra (dialog)

Carichiamo la finestra così poi potremo avviarla:

```
from defdialog import DefWindow
```

Il codice è molto semplice:

```
window = QDialog()

ui = DefWindow()
```

```
ui.setupUi(window)

if ui.exec_() == 1:

      self.loadHotkeys()
```

L'ultima riga serve per capire quando la finestra è chiusa ed eseguire una funzione.

In caso non servisse basterà sostituire le ultime due righe con:

```
ui.exec_()
```