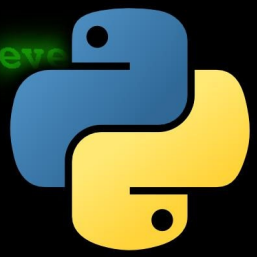
Full Circle

LA RIVISTA INDIPENDENTE PER LA COMUNITÀ LINUX UBUNTU

EDIZIONE SPECIALE SERIE PROGRAMMAZIONE

EDIZIONE SPECIALE
SERIE PROGRAMMAZIONE



python™

PROGRAMMARE IN PYTHON VOLUME 4

Cos'è Full Circle

Full Circle è una rivista gratuita e indipendente, dedicata alla famiglia Ubuntu dei sistemi operativi Linux. Ogni mese pubblica utili articoli tecnici e articoli inviati dai lettori.

Full Circle ha anche un podcast di supporto, il Full Circle Podcast, con gli stessi argomenti della rivista e altre interessanti notizie.

Si prega di notare che questa edizione speciale viene fornita senza alcuna garanzia: né chi ha contribuito né la rivista Full Circle hanno alcuna responsabilità circa perdite di dati o danni che possano derivare ai computer o alle apparecchiature dei lettori dall'applicazione di quanto pubblicato.

Full Circle

LA RIVISTA INDIPENDENTE PER LA COMUNITÀ LINUX UBUNTU

Ecco a voi un altro Speciale monotematico!

Come richiesto dai nostri lettori, stiamo assemblando in edizioni dedicate alcuni degli articoli pubblicati in serie.

Quella che avete davanti è la ristampa della serie **Programmare in Python, parti 22-26**, pubblicata nei numeri 48-52: niente di speciale, giusto quello che abbiamo già pubblicato.

Vi chiediamo, però, di badare alla data di pubblicazione: le versioni attuali di hardware e software potrebbero essere diverse rispetto ad allora. Controllate il vostro hardware e il vostro software prima di provare quanto descritto nelle guide di queste edizioni speciali. Potreste avere versioni più recenti del software installato o disponibile nei repository delle vostre distribuzioni.

Buon divertimento!

Come contattarci

Sito web:
http://www.fullcirclemagazine.org/

Forum:
http://ubuntuforums.org/forumdisplay.php?f=270

IRC:#fullcirclemagazine su chat.freenode.net

Gruppo editoriale

Capo redattore: Ronnie Tucker
(aka: RonnieTucker)
ronnie@fullcirclemagazine.org

Webmaster: Rob Kerfia
(aka: admin / linuxgeekery-
admin@fullcirclemagazine.org

Podcaster: Robin Catling
(aka RobinCatling)
podcast@fullcirclemagazine.org

Manager delle comunicazioni:
Robert Clipsham
(aka: mrmonday) -
mrmonday@fullcirclemagazine.org

Programmare in Python - Parte 22

Errata Corrige

Il mese scorso, nella parte 21, vi era stato detto di salvare il lavoro come "PlaylistMaker.glade" ma, nel codice, vi si faceva riferimento come "playlistmaker.glade". Sono sicuro che avrete notato che una versione ha le maiuscole e l'altra no. Il codice funzionerà solo usando la stessa versione, con o senza le maiuscole.

Per iniziare col piede giusto dovrete avere playlistmaker.glade e playlistmaker.py del mese scorso. In caso contrario, provvedete di conseguenza. Prima di iniziare, diamo un'occhiata a cos'è un file playlist. Esistono molte versioni di liste d'esecuzione e tutte hanno estensioni differenti. Quella che creeremo noi sarà del tipo *.m3u. Nella sua forma più semplice, è solo un file di testo che inizia con "#EXTM3U" e presenta una riga per ciascuna canzone da ascoltare, incluso l'intero percorso. È possibile aggiungere informazioni ulteriori prima di ciascuna voce come la durata della canzone, il titolo dell'album, il numero di traccia e il titolo. Per il momento ci concentreremo sulla versione base.

Ecco un esempio di playlist M3U...

```
#EXTM3U
Adult Contemporary/Chris Rea/Collection/02 - On The Beach.mp3
Adult Contemporary/Chris Rea/Collection/07 - Fool (If You Think It's Over).mp3
Adult Contemporary/Chris Rea/Collection/11 - Looking For The Summer.mp3
```

Tutti i percorsi sono relativi alla posizione del file playlist.

Ok... iniziamo a scrivere codice. A destra trovate l'inizio del codice sorgente del mese scorso.

Ora dobbiamo creare una funzione per la gestione di ciascun evento già configurato. Notate che on_MainWindow_destroy e on_tbtnQuit_clicked sono già stati realizzati, così ne dobbiamo realizzare solamente altri 10

```
#!/usr/bin/env python
import sys
from mutagen.mp3 import MP3
try:
    import pygtk
    pygtk.require("2.0")
except:
    pass
try:
    import gtk
    import gtk.glade
except:
    sys.exit(1)
```

quindi la definizione della classe

```
class PlayListCreator:
    def __init__(self):
        self.gladefile = "playlistmaker.glade"
        self.wTree = gtk.glade.XML(self.gladefile,"MainWindow")
```

e la routine main

```
if __name__ == "__main__":
      plc = PlayListCreator()
      gtk.main()
```

Quindi abbiamo il dizionario che dovrebbe essere inserito dopo la funzione __init__ .

```
def SetEventDictionary(self):
  dict = {"on_MainWindow_destroy": gtk.main_quit,
    "on_tbtnQuit_clicked": gtk.main_quit,
    "on_tbtnAdd_clicked": self.on_tbtnAdd_clicked,
    "on_tbtnDelete_clicked": self.on_tbtnDelete_clicked,
    "on_tbtnClearAll_clicked": self.on_tbtnClearAll_clicked,
    "on_tbtnMoveToTop_clicked": self.on_tbtnMoveToTop_clicked,
    "on_tbtnMoveUp_clicked": self.on_tbtnMoveUp_clicked,
    "on_tbtnMoveDown_clicked": self.on_tbtnMoveDown_clicked,
    "on_tbtnMoveToBottom_clicked": self.on_tbtnMoveToBottom_clicked,
    "on_tbtnAbout_clicked": self.on_tbtnAbout_clicked,
    "on_btnGetFolder_clicked": self.on_btnGetFolder_clicked,
    "on_btnSavePlaylist_clicked": self.on_btnSavePlaylist_clicked}
  self.wTree.signal_autoconnect(dict)
```

(mostrati in alto a destra). Per ora create solo lo scheletro.

Le modificheremo tra pochi minuti. Per il momento questo dovrebbe essere sufficiente a eseguire l'applicazione e potremo fare delle prove nel corso dello sviluppo. Ma prima di poter eseguire l'applicazione dobbiamo inserire un'ulteriore riga alla funzione __init__. Dopo la riga self.wTree, aggiungete...

```
self.SetEventDictionary()
```

Ora potete eseguire l'applicazione, osservare la finestra e fare clic sul Pulsante Esci della barra strumenti per uscire correttamente dall'applicazione. Salvate il codice come "playlistmaker-1a.py" e provate. Ricordate di salvarlo nella stessa cartella insieme al file glade dell'ultima volta o copiate il file glade nella stessa cartella in cui avete salvato questo codice.

Dobbiamo anche definire alcune variabili per un uso successivo. Inserite le seguenti dopo la chiamata SetEventDictionary nella funzione __init__.

```
self.CurrentPath = ""
self.CurrentRow = 0
self.RowCount = 0
```

Ora creeremo una funzione che ci permette di mostrare una finestra a comparsa ogniqualvolta si debbano dare informazioni all'utente. Ci sono alcune funzioni predefinite che useremo, ma creeremo una routine nostra per semplificarci il lavoro. È la funzione gtk.MessageDialog e la sintassi è la seguente...

```
gtk.MessageDialog(parent,flags,MessageType,Buttons,message)
```

Sono necessari alcuni chiarimenti prima di procedere. Il tipo di messaggio può essere uno dei seguenti...

```
GTK_MESSAGE_INFO - Messaggio informativo
GTK_MESSAGE_WARNING - Messaggio di avviso non fatale
GTK_MESSAGE_QUESTION - Domanda richiedente una scelta
GTK_MESSAGE_ERROR - Messaggio di errore fatale
```

E i tipi di pulsante sono...

```
GTK_BUTTONS_NONE - nessun pulsante
GTK_BUTTONS_OK - un pulsante OK
GTK_BUTTONS_CLOSE - un pulsante Chiudi
GTK_BUTTONS_CANCEL - Un pulsante Annulla
GTK_BUTTONS_YES_NO - pulsanti Sì e No
GTK_BUTTONS_OK_CANCEL - pulsanti Ok e Annulla
```

Normalmente userete il seguente, o simile, codice per creare una finestra di dialogo, mostrarla, attendere la risposta e quindi eliminarla dalla memoria.

```
dlg = gtk.MessageDialog(None,0,gtk.MESSAGE_INFO,gtk.BUTTONS_OK,"This is a test message...")
response = dlg.run()
dlg.destroy()
```

```
    def on_tbtnAdd_clicked(self,widget):
        pass
    def on_tbtnDelete_clicked(self,widget):
        pass
    def on_tbtnClearAll_clicked(self,widget):
        pass
    def on_tbtnMoveToTop_clicked(self,widget):
        pass
    def on_tbtnMoveUp_clicked(self,widget):
        pass
    def on_tbtnMoveDown_clicked(self,widget):
        pass
    def on_tbtnMoveToBottom_clicked(self,widget):
        pass
    def on_tbtnAbout_clicked(self,widget):
        pass
    def on_btnGetFolder_clicked(self,widget):
        pass
    def on_btnSavePlaylist_clicked(self,widget):
        pass
```

Comunque, se volete mostrare un messaggio all'utente più di una o due volte, si tratta di scrivere parecchio. La regola generale è che se scrivete una serie di righe più di una o due volte è certamente meglio creare una funzione e quindi chiamarla. Ragionate in questo modo: se vogliamo mostrare un messaggio all'utente, diciamo dieci volte, equivale a scrivere 10x3 (o 30) righe di codice. Creando una funzione (usando l'esempio appena presentato) avremo 10+3 righe di codice da scrivere. Più volte richiamiamo il messaggio, meno

codice ci tocca scrivere e più leggibile risulterà il nostro codice. La nostra funzione (in alto a destra) ci permetterà di chiamare ciascuno dei quattro tipi di messaggio con solo una funzione e parametri differenti.

Si tratta di una funzione molto semplice che possiamo chiamare in questo modo:

```
self.MessageBox("info","The button QUIT was clicked")
```

Notate che se scegliamo di usare il tipo MESSAGE_QUESTION, la finestra potrà ritornare due possibili risposte , "Si" o "No". Qualunque pulsante l'utente scelga riceveremo l'informazione nel nostro codice. Per usare la finestra di dialogo domanda, la chiamata sarà simile a...

```
response = self.MessageBox("question","Are you sure you want to do this now?")
if response == gtk.RESPONSE_YES:
    print "Yes was clicked"
elif response == gtk.RESPONSE_NO:
    print "NO was clicked"
```

Potete vedere come controllare il valore del pulsante scelto. Quindi

```
def MessageBox(self,level,text):
    if level == "info":
        dlg = gtk.MessageDialog(None,0,gtk.MESSAGE_INFO,gtk.BUTTONS_OK,text)
    elif level == "warning":
        dlg = gtk.MessageDialog(None,0,gtk.MESSAGE_WARNING,gtk.BUTTONS_OK,text)
    elif level == "error":
        dlg = gtk.MessageDialog(None,0,gtk.MESSAGE_ERROR,gtk.BUTTONS_OK,text)
    elif level == "question":
        dlg = gtk.MessageDialog(None,0,gtk.MESSAGE_QUESTION,gtk.BUTTONS_YES_NO,text)
    if level == "question":
        resp = dlg.run()
        dlg.destroy()
        return resp
    else:
        resp = dlg.run()
        dlg.destroy()
```

ora sostituiamo la chiamata "pass" in ciascuna delle funzioni di gestione degli eventi con qualcosa simile a quello mostrato in basso a destra.

```
def on_tbtnAdd_clicked(self,widget):
    self.MessageBox("info","Button Add was clicked...")
def on_tbtnDelete_clicked(self,widget):
    self.MessageBox("info","Button Delete was clicked...")
def on_tbtnClearAll_clicked(self,widget):
    self.MessageBox("info","Button ClearAll was clicked...")
def on_tbtnMoveToTop_clicked(self,widget):
    self.MessageBox("info","Button MoveToTop was clicked...")
def on_tbtnMoveUp_clicked(self,widget):
    self.MessageBox("info","Button MoveUp was clicked...")
def on_tbtnMoveDown_clicked(self,widget):
    self.MessageBox("info","Button MoveDown was clicked...")
def on_tbtnMoveToBottom_clicked(self,widget):
    self.MessageBox("info","Button MoveToBottom was clicked...")
def on_tbtnAbout_clicked(self,widget):
    self.MessageBox("info","Button About was clicked...")
def on_btnGetFolder_clicked(self,widget):
    self.MessageBox("info","Button GetFolder was clicked...")
def on_btnSavePlaylist_clicked(self,widget):
    self.MessageBox("info","Button SavePlaylist was clicked...")
```

Non è la versione definitiva, ma vi darà un'indicazione visiva che i pulsanti funzionano come desiderato. Salvate il codice come "playlistmaker-1b.py" e testate il programma. Ora creeremo una funzione per impostare i riferimenti per i nostri widget. Questa funzione sarà chiamata una sola volta, ma renderà il codice più gestibile e leggibile. In pratica, vogliamo creare variabili locali che referenziano i widget nella finestra di glade, così da poterli richiamare, se necessario, a piacimento.

Inserite questa funzione (pag. precedente, in alto a destra) sotto la funzione SetEventDictionary.

Si prega di notare che c'è una cosa non referenziata nella nostra routine. Si tratta del widget vista albero. Creeremo il riferimento quando imposteremo la vista stessa. Notate anche l'ultima riga della funzione. Per usare la barra di stato dovremo riferirci ad essa attraverso il suo id contestuale. Lo useremo più avanti.

Proseguiamo impostando la funzione che mostra la finestra "informazioni" quando si fa clic sul pulsante "Informazioni" nella barra strumenti. Di nuovo, a tal scopo c'è una funzione predefinita fornita dalla libreria GTK. Inserite il codice in basso a destra dopo la funzione MessageBox.

Salvate e fate una prova. Dovreste vedere una finestra a comparsa, centrata rispetto all'applicazione, che mostra qualunque cosa abbiamo inserito. La finestra Informazioni possiede altri attributi (documentati all'indirizzo http://www.pygtk.org/docs/pygtk/class-gtkaboutdialog.html), ma questi sono quelli che considero di base.

```
    def SetWidgetReferences(self):
        self.txtFilename = self.wTree.get_widget("txtFilename")
        self.txtPath = self.wTree.get_widget("txtPath")
        self.tbtnAdd = self.wTree.get_widget("tbtnAdd")
        self.tbtnDelete = self.wTree.get_widget("tbtnDelete")
        self.tbtnClearAll = self.wTree.get_widget("tbtnClearAll")
        self.tbtnQuit = self.wTree.get_widget("tbtnQuit")
        self.tbtnAbout = self.wTree.get_widget("tbtnAbout")
        self.tbtnMoveToTop = self.wTree.get_widget("tbtnMoveToTop")
        self.tbtnMoveUp = self.wTree.get_widget("tbtnMoveUp")
        self.tbtnMoveDown = self.wTree.get_widget("tbtnMoveDown")
        self.tbtnMoveToBottom = self.wTree.get_widget("tbtnMoveToBottom")
        self.btnGetFolder = self.wTree.get_widget("btnGetFolder")
        self.btnSavePlaylist = self.wTree.get_widget("btnSavePlaylist")
        self.sbar = self.wTree.get_widget("statusbar1")
        self.context_id = self.sbar.get_context_id("Statusbar")
```

e quindi aggiungete una chiamata ad essa giusto dopo la chiamata self.SetEventDictionary() nella funzione __init__.

```
        self.SetWidgetReferences()
```

Prima di proseguire dobbiamo discutere di quello che avverrà esattamente da questo momento. L'idea generale è che l'utente faccia clic sul pulsante "Aggiungi" nella barra strumenti, si apra una finestra per aggiungere file alla playlist e quindi vengano mostrate le informazioni dei file nella vista ad albero. Da qui è possibile aggiungere altri file, cancellare singole voci, cancellare tutto, muovere un elemento in alto, in basso,

```
    def ShowAbout(self):
        about = gtk.AboutDialog()
        about.set_program_name("Playlist Maker")
        about.set_version("1.0")
        about.set_copyright("(c) 2011 by Greg Walters")
        about.set_comments("Written for Full Circle Magazine")
        about.set_website("http://thedesignatedgeek.com")
        about.run()
        about.destroy()
```

Ora, commentate (o rimuovete semplicemente) la chiamata messagebox nella funzione on_tbtnAbout_clicked e sostituitela con una chiamata alla funzione ShowAbout. Fate in modo che somigli a questa.

```
  def on_tbtnAbout_clicked(self,widget):
        #self.MessageBox("info","Button About was clicked...")
        self.ShowAbout()
```

all'inizio o alla fine della vista ad albero. Eventualmente, si potrà definire il percorso dove salvare il file, fornire un nome con estensione "m3u" e fare clic sul pulsante salva file. Anche se questo sembra tutto abbastanza semplice, molto deve avvenire dietro le quinte. Poiché molta della magia riguarda il widget vista ad albero, incominciamo a parlarne. Ci addentreremo abbastanza, quindi leggete con attenzione dato che la corretta comprensione vi eviterà errori futuri.

Una vista ad albero può essere semplice come la rappresentazione di una colonna di dati di un foglio di lavoro o di un database, o può essere più complessa come una lista di file e cartelle con elementi genitori e figli, dove una cartella potrebbe essere il genitore e i file in quella cartella potrebbero essere i figli, o qualcosa di ancora più complesso. Per questo progetto useremo il primo esempio, una lista colonnare. Questa lista conterrà tre colonne. Una per il nome del file musicale, una per l'estensione del file stesso (mp3, ogg, wav, etc) e la colonna finale per il percorso. Combinando il tutto in una stringa (percorso, nome e estensione) otterremo la voce da inserire nella playlist. Potete, se vi dovesse occorrere, aggiungere altre colonne ma per il momento ci fermeremo a queste tre.

Una vista ad albero è un contenitore che conserva e mostra un modello. Il modello è il "dispositivo" attuale che contiene e manipola i nostri dati. Ci sono due modelli predefiniti che possono essere usati con la vista albero, ma ovviamente potete crearne di vostri. Detto questo, nel circa il 98% dei casi uno dei due modelli predefiniti sarà sufficiente ai nostri bisogni. I due tipi sono GTKListStore e GTKTreeStore. Come suggerisce il loro nome il modello ListStore si usa di solito per le liste, TreeStore per gli alberi. Nella nostra applicazione useremo GTKListStore. I passi base sono:

• Creare un riferimento al widget TreeView.
• Aggiungere le colonne.
• Impostare il tipo di visualizzazione da usare.
• Creare una ListStore
• Impostare come modello della vista albero il nostro modello
• Inserire i dati

```
def SetupTreeview(self):
    self.cFName = 0
    self.cFType = 1
    self.cFPath = 2
    self.sFName = "Filename"
    self.sFType = "Type"
    self.sFPath = "Folder"
    self.treeview = self.wTree.get_widget("treeview1")
    self.AddPlaylistColumn(self.sFName,self.cFName)
    self.AddPlaylistColumn(self.sFType,self.cFType)
    self.AddPlaylistColumn(self.sFPath,self.cFPath)
    self.playList = gtk.ListStore(str,str,str)
    self.treeview.set_model(self.playList)
    self.treeview.set_grid_lines(gtk.TREE_VIEW_GRID_LINES_BOTH)
```

Il terzo passo riguarda la scelta del tipo di visualizzazione usato dalla colonna per mostrare i dati. Si tratta di una semplice funzione per disegnare i dati nel modello ad albero. GTK fornisce molti tipi di visualizzazioni differenti ma quelli che userete maggiormente sono GtkCellRenderText e GtkCellRendererToggle.

Allora, creiamo una funzione (mostrata in alto) per configurare il widget vista albero. La chiameremo SetupupTreeview. Prima definiremo alcune variabili per le colonne, quindi la variabile che fa riferimento alla stessa vista albero, aggiungiamo le colonne, impostiamo ListStore e il modello. Inseritela dopo la funzione SetWidgetReferences.

Le variabili cFName, cFType e cFPath definiscono i numeri di colonna. Le variabili sFName, sFType e sFPath conterranno i nomi delle colonne in fase di visualizzazione. La settima riga configura la variabile che fa riferimento al widget vista albero utilizzando lo stesso nome usato nel nostro file glade.

Procediamo chiamando una funzione (pagina seguente, in alto a destra), che creeremo a breve, per ciascuna colonna. Quindi definiamo GTKListStore con tre campi di testo e per finire impostiamo l'attributo modello del widget vista albero con

il nostro GTKListStore. Proseguiamo creando la funzione AddPlaylistColumn. Inseritela dopo SetupTreeview.

Ciascuna colonna viene creata utilizzando questa funzione. Le passiamo il titolo della colonna (quello che viene mostrato in cima a ciascuna colonna) e un identificativo. In questo caso passeremo le variabili create in precedenza (sFName and cFname). Quindi nel widget vista albero creiamo una colonna assegnandole un titolo, il tipo di visualizzazione che userà e, per finire, un identificativo. Quindi rendiamo la colonna ridimensionabile, ordinabile in base all'identificativo e finiamo aggiungendo la colonna alla vista ad albero.

Aggiungete queste due funzioni. Ho deciso di inserirle subito dopo la funzione SetWidgetReferences, ma potete inserirle ovunque nella classe PlayListCreator. Aggiungete la seguente riga dopo la chiamata a SetWidgetReferences() nella funzione __init__ per richiamare la funzione.

```
self.SetupTreeview()
```

Salvate ed eseguite il programma e vedrete che ora il nostro widget vista albero contiene tre colonne con intestazioni.

```
def AddPlaylistColumn(self,title,columnId):
    column = gtk.TreeViewColumn(title,gtk.CellRendererText(),text=columnId)
    column.set_resizable(True)
    column.set_sort_column_id(columnId)
    self.treeview.append_column(column)
```

Ci sono molte cose ancora da fare. Dobbiamo trovare un modo per ricevere dall'utente i nomi dei file audio e inserirli nell'albero come righe di dati. Dobbiamo creare le funzioni di eliminazione, pulizia e spostamento, quella di salvataggio e quella di gestione del percorso dei file, più altre cose "carine" che daranno un aspetto più professionale all'applicazione. Iniziamo con la funzione Aggiungi. Dopo tutto è il primo pulsante della nostra barra strumenti. Quando l'utente fa clic sul pulsante Aggiungi, vogliamo che compaia una finestra standard per i file che ci permetta una selezione multipla. Quando l'utente ha ultimato la selezione, dobbiamo recuperare i dati e inserirli nella lista, come detto sopra. Quindi la prima cosa logica da fare è lavorare sulla finestra di scelta dei file. Ancora, GTK fornisce un sistema per richiamare una finestra "standard". Potremmo inserire il codice direttamente nel gestore evento on_tbtnAdd_clicked ma creiamo una classe a sé per lo scopo. Visto che ci siamo, la realizzeremo in maniera tale che non solo gestisca la finestra APRI ma anche quella SELEZIONA. Come visto prima con la funzione MessageBox, potete raccogliere tutti questi frammenti di codice in un file di funzioni riutilizzabili. Iniziamo definendo una nuova classe chiamata FileDialog che conterrà solo una funzione chiamata ShowDialog. La funzione accetterà due parametri, uno chiamato 'which' ('0' o '1'), che indicherà se creare una finestra per aprire un file o selezionare una cartella, e l'altro,chiamato CurrentPath, è il percorso sui cui la vista si aprirà. Create questa classe subito prima la funzione main in fondo al file sorgente.

```
class FileDialog:
    def
ShowDialog(self,which,Current
Path):
```

La prima parte del codice dovrebbe essere un'istruzione IF

```
if which == 0: # file chooser
    ...
else: # folder chooser
    ...
```

Prima di procedere ulteriormente, esploriamo come la finestra per file/cartelle è chiamata e utilizzata. La sintassi è la seguente

```
gtk.FileChooserDialog(title,p
arent,action,buttons,backend)
```

e ritorna un oggetto di tipo finestra di dialogo. La prima riga (sotto if which == 0) sarà la riga mostrata sotto.

Come potete vedere, il titolo è

```
dialog = gtk.FileChooserDialog("Select files to add...",None,
                    gtk.FILE_CHOOSER_ACTION_OPEN,
                    (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
                    gtk.STOCK_OPEN, gtk.RESPONSE_OK))
```

"Select files to add...", il genitore è impostato su None. In questo caso l'azione da compiere sarà Apri File e vogliamo i pulsanti Annulla e Apri, entrambi che usano le icone predefinite. Stiamo anche impostando i codici di ritorno per gtk.RESPONSE_CANCEL e gtk.RESPONSE_OK quando l'utente fa la sua scelta. La chiamata per Scelta Cartella sotto la clausola Else è simile.

In pratica, l'unica differenza tra le due definizioni è il titolo (mostrato in alto a destra) e il tipo di azione. Quindi il codice finale per la classe ora dovrebbe essere quello mostrato al centro a destra.

Abbiamo impostato come risposta predefinita il pulsante OK, e quindi attivato la seleziona multipla così che l'utente può selezionare (avete indovinato) più file da aggiungere. Se non lo avessimo fatto, la finestra avrebbe permesso la selezione di un file alla volta dato che set_select_multiple è impostato, di default, a falso. Le righe successive configurano il percorso corrente e quindi viene mostrata la finestra stessa. Prima di inserire il codice, fatemi spiegare perché è importante configurare un percorso. Ogni volta che si apre una finestra di dialogo per i file e NON impostate un percorso, viene scelto quello in cui risiede il programma. Quindi diciamo che i file musicali che cerchiamo sono in /media/music_files/ e sono quindi suddivisi per genere, quindi per artista e quindi per album. Ipotizziamo ancora che l'utente abbia installato l'applicazione in /home/utente2/playlistmaker. Ogni volta che richiamiamo la finestra la cartella iniziale sarebbe /home/utente2/playlistmaker. Ben presto questo stuferebbe l'utente che vorrebbe come cartella iniziale quella aperta per ultima. Ha senso? Ok. Allora, in basso a destra ci sono le prossime righe di codice.

Qui controlliamo le risposte ritornate. Se l'utente ha fatto clic sul pulsante 'Apri' che restituisce gtk.RESPONSE_OK, otteniamo il nome o i nomi dei file selezionati

```
dialog = gtk.FileChooserDialog("Select Save Folder..",None,
                               gtk.FILE_CHOOSER_ACTION_SELECT_FOLDER,
                              (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
                               gtk.STOCK_OPEN, gtk.RESPONSE_OK))
```

```
class FileDialog:
    def ShowDialog(self,which,CurrentPath):
        if which == 0: #file chooser
            #gtk.FileChooserDialog(title,parent,action,buttons,backend)
            dialog = gtk.FileChooserDialog("Select files to add...",None,

                                   gtk.FILE_CHOOSER_ACTION_OPEN,
                                   (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
                                    gtk.STOCK_OPEN, gtk.RESPONSE_OK))
        else:           #folder chooser
            dialog = gtk.FileChooserDialog("Select Save Folder..",None,
                                    gtk.FILE_CHOOSER_ACTION_SELECT_FOLDER,
                                   (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
                                    gtk.STOCK_OPEN, gtk.RESPONSE_OK))
```

Le due righe seguenti saranno (fuori dall'istruzione IF/ELSE)...

```
        dialog.set_default_response(gtk.RESPONSE_OK)
        dialog.set_select_multiple(True)
```

```
        if CurrentPath != "":
            dialog.set_current_folder(CurrentPath)
        response = dialog.run()
```

Quindi dobbiamo gestire la risposta dalla finestra di dialogo.

```
        if response == gtk.RESPONSE_OK:
            fileselection = dialog.get_filenames()
            CurrentPath = dialog.get_current_folder()
            dialog.destroy()
            return (fileselection,CurrentPath)
        elif response == gtk.RESPONSE_CANCEL:
            print 'Closed, no files selected'
            dialog.destroy()
```

dall'utente, si imposta il percorso corrente alla cartella in cui siamo, rimuove la finestra di dialogo dalla memoria e ritorna i dati alla funzione chiamante. Se, d'altra parte, l'utente ha scelto il pulsante 'Annulla', si rimuove semplicemente la finestra dalla memoria. Ho inserito un'istruzione print solo per mostrarvi che la pressione del pulsante ha funzionato. Potete lasciarla o rimuoverla. Notate che selezionando il pulsante Apri saranno restituiti due insiemi di valori:
'fileselection', una lista di file selezionati dall'utente, e CurrentPath.

Affinché la funzione faccia qualcosa aggiungete la seguente riga sotto la routine on_tbtnAdd_click...

```
fd = FileDialog()
selectedfiles,self.CurrentPath =
fd.ShowDialog(0,self.CurrentPath)
```

Qui recuperiamo i due valori di ritorno. Per il momento, aggiungete il codice seguente per vedere l'aspetto dell'informazione ritornata.

```
for f in selectedfiles:
    print "User selected %s" %f
print "Current path is %s" %self.CurrentPath
```

Quando eseguite il programma, fate clic sul pulsante 'Aggiungi'. Vedrete la finestra di dialogo. Ora spostatevi dove tenete i file e selezionateli. Potete tenere premuto il tasto [ctrl] e fare clic su più file per selezionarli individualmente, o il tasto [shift] per selezionare più file contigui. Fate clic sul pulsante 'Apri' e controllate la risposta nel terminale. Si noti che se fate clic sul pulsante 'Annulla' proprio ora, otterrete un messaggio di errore. Questo perché il codice sopra presume che nessun file sia selezionato. Per il momento non preoccupatevene, ce ne occuperemo a breve. Quello che mi preme evidenziare è che vediate cosa otteniamo quando si sceglie il pulsante 'Apri'. Una cosa che dovremmo fare è aggiungere un filtro alla finestra per i file. Dato che ci aspettiamo che l'utente selezioni file musicali, noi dovremmo (1) dare l'opzione per visualizzare solo questi e (2) dare la possibilità per mostrare tutti i file, se necessario. Lo facciamo configurando l'attributo FileFilter della finestra di dialogo. Ecco il codice da inserire nella sezione which == 0 subito dopo la riga che imposta la finestra.

```
filter = gtk.FileFilter()
filter.set_name("Music Files")
filter.add_pattern("*.mp3")
filter.add_pattern("*.ogg")
filter.add_pattern("*.wav")
dialog.add_filter(filter)
filter = gtk.FileFilter()
filter.set_name("All files")
filter.add_pattern("*")
dialog.add_filter(filter)
```

Stiamo configurando due "gruppi", uno per i file audio (filter.set_name("Music Files")) e l'altro per tutti i file. Usiamo uno schema per definire i tipi di file che desideriamo. Ho definito tre schemi, ma potete aggiungerli o rimuoverli a piacere. Ho inserito prima il filtro per la musica, dato che dovrebbe essere quello che più interessa all'utente. Quindi i passi sono...

- Definire la variabile filter.
- Impostare un nome.
- Aggiungere uno schema.
- Aggiungere il filtro alla finestra di dialogo.

Potete avere quanti filtri volete. Notate anche che una volta aggiunto il filtro è possibile riutilizzarlo.

Ritorniamo alla funzione on_tbtnAdd_clicked, commentate le ultime righe inserite e sostituitele con questa

```
line.self.AddFilesToTreeview(selec tedfiles)
```

così ora la nostra funzione appare come il codice mostrato nella pagina seguente.

Così, quando riceviamo la risposta della finestra di dialogo, invieremo la lista contenente i file selezionati a questa funzione. Una volta qui, definiamo una variabile contatore (quanti file stiamo aggiungendo), quindi scorriamo la lista. Ricordate che ciascuna voce contiene il nome completo di percorso ed estensione. Divideremo questa stringa in percorso, nome ed estensione. Prima troviamo l'ultimo 'punto' e assumiamo che questo sia l'inizio dell'estensione e assegniamo la sua posizione alla stringa extStart. Quindi cerchiamo l'ultimo '/' per determinare l'inizio del nome del file. Quindi suddividiamo la stringa in estensione, nome del file e percorso. Inseriamo questi valori in una lista chiamata 'data' e l'aggiungiamo alla playlist ListStore. Poiché abbiamo

```
    def on_tbtnAdd_clicked(self,widget):
        fd = FileDialog()
        selectedfiles,self.CurrentPath =
fd.ShowDialog(0,self.CurrentPath)
        self.AddFilesToTreeview(selectedfiles)
```

Ora dobbiamo creare la funzione che abbiamo appena chiamato. Inserite questa funzione dopo on_btnSavePlaylist_clicked.

```
    def AddFilesToTreeview(self,FileList):
        counter = 0
        for f in FileList:
            extStart = f.rfind(".")
            fnameStart = f.rfind("/")
            extension = f[extStart+1:]
            fname = f[fnameStart+1:extStart]
            fpath = f[:fnameStart]
            data = [fname,extension,fpath]
            self.playList.append(data)
            counter += 1
        self.RowCount += counter
        self.sbar.push(self.context_id,"%s files added
for a total of %d" % (counter,self.RowCount))
```

terminato il lavoro, incrementiamo il contatore. Per finire incrementiamo la variabile RowCount che contiene il numero totale delle righe in ListStore e quindi mostriamo un messaggio nella barra di stato.
Ora potete avviare il programma e vedere i dati nella vista ad albero.

Come sempre, l'intero codice può essere trovato all'indirizzo http://pastebin.com/JtrhuE71.

La prossima volta finalizzeremo l'applicazione, aggiungendo le funzioni mancanti, ecc.

Greg Walters è il proprietario della RainyDay Solutions, LLC, una società di consulenza in Aurora, Colorado e programma dal 1972. Ama cucinare, fare escursioni, ascoltare musica e passare il tempo con la sua famiglia.

Programmare in Python - Parte 23

Con questa lezione termineremo il programma playlistmaker. L'ultima volta ne abbiamo realizzato buona parte, ma lasciando alcune cose incomplete. Non possiamo salvare la playlist, le funzioni di spostamento non sono disponibili, non è possibile scegliere il percorso di salvataggio del file, e così via. Ci sono, però, ancora alcune cose da fare prima di iniziare col codice di oggi. Primo, dobbiamo trovare un'immagine logo da inserire nella finestra Informazioni e quando il programma è minimizzato. Potreste cercare in /usr/share/icons un'icona di vostro gradimento, o andare su internet e recuperarne una o crearne una voi stessi. Qualunque sia la vostra scelta, inseritela nella cartella con il file glade e il codice sorgente del mese scorso. Chiamatela logo.png. Quindi, dobbiamo aprire il file glade e apportare alcune modifiche.

Prima di tutto, usando MainWindow andate nella scheda Generale, scorrete in basso fino a trovare Icona. Usando lo strumento sfoglia, cercate la vostra icona e selezionatela. Ora il campo di testo dovrebbe contenere "logo.png". Proseguite selezionando treeview1 nell'ispettore, attivate la scheda Segnali e in corrispondenza di GtkTreeView|cursorchanged selezionate on_treeview1_cursor_changed. Ricordate, come vi ho detto il mese scorso, di fare clic all'esterno per applicare la modifica. Per finire, sempre nell'Ispettore selezionate txtFilename e aprite la scheda Segnali. Scorrete fino a trovare 'GtkWidget', quindi spostatevi in basso un altro po' fino a 'key_press_event'. Selezionate 'on_txtFilename_key_press_event'. Salvate il vostro progetto e chiudete Glade.

Ora è arrivato il momento di completare il progetto. Riprenderemo da dove abbiamo interrotto, usando il codice dell'ultimo mese.

```
elif response == gtk.RESPONSE_CANCEL:
    print 'Closed, no files selected'
    dialog.destroy()
```

Notate come non venga restituito nulla. Questo causa l'errore. Quindi, per correggerlo, dobbiamo inserire la riga seguente dopo dialog.destroy().

```
Return ([],"")
```

Questa eviterà che si verifichi l'errore. Quindi, aggiungiamo il gestore evento del campo di testo creato con glade. Al nostro dizionario, inseriamo la seguente riga.

```
"on_txtFilename_key_press_event": self.txtFilenameKeyPress,
```

Come ricorderete, così si crea la funzione per gestire l'evento keypress. Creiamo a seguire la funzione.

```
def txtFilenameKeyPress(self,widget,data):
   if data.keyval == 65293: # The value of the return key
      self.SavePlaylist()
```

La prima cosa che voglio fare è modificare il codice della classe FileDialog. Se ricordate dall'ultima volta, se l'utente faceva clic sul pulsante 'Annulla' compariva un errore. Prima sistemeremo questo. Alla fine della funzione avrete il codice mostrato in alto.

Come potete immaginare, non fa altro che controllare il valore di ciascun tasto premuto, quando l'utente si trova nel campo di testo TxtFilename e lo confronta con il valore 65293, che sarebbe il codice del tasto INVIO. Se corrisponde allora chiama la funzione SavePlaylist. L'utente non deve fare clic su nessun pulsante.

Ora il nuovo codice. Occupiamoci del pulsante Pulisci della barra degli

strumenti. Quando è premuto dall'utente, vogliamo che treeview e ListScore vengano puliti. È una semplice riga che possiamo inserire nella funzione on_tbtnClearAll_clicked.

```
def on_tbtnClearAll_clicked(self,widget):
    self.playList.clear()
```

Stiamo semplicemente chiedendo a ListStore di pulirsi da solo. È stato facile. Passiamo al pulsante Elimina. Più difficile, ma una volta all'interno, capirete.

Prima di tutto dobbiamo discutere su come si ricava una selezione dal widget vista albero e da ListScore. È complicato, quindi procediamo con calma. Per ottenere i dati da ListScore dobbiamo prima ricavare gtk.TreeSelection che non è altro che un oggetto di supporto per maneggiare la selezione in una vista ad albero. Quindi usiamo questo oggetto di supporto per recuperare il tipo di modello e un iteratore che contiene le righe selezionate.

So che state pensando "Cosa diamine è un iteratore?" Beh, li avete già usati senza saperlo. Considerate il codice seguente (in alto a destra) dalla funzione AddFilesToTreeview, del mese scorso.

```
def AddFilesToTreeview(self,FileList):
    counter = 0
    for f in FileList:
        extStart = f.rfind(".")
        fnameStart = f.rfind("/")
        extension = f[extStart+1:]
        fname = f[fnameStart+1:extStart]
        fpath = f[:fnameStart]
        data = [fname,extension,fpath]
        self.playList.append(data)
        counter += 1
```

Osservate l'istruzione 'for'. Usiamo un iteratore per spostarci nella lista chiamata FileList. In pratica, in questo caso, l'iteratore si sposta semplicemente attraverso ciascun elemento della lista restituendoceli uno alla volta. Quello che andremo a fare è creare un iteratore, riempirlo con le righe selezionate nella vista albero e usarlo come una lista. Quindi il codice (a destra, al centro) di on_tbtnDelete_clicked sarà.

```
def on_tbtnDelete_clicked(self,widget):
    sel = self.treeview.get_selection()
    (model,rows) = sel.get_selected_rows()
    iters=[]
    for row in rows:
        iters.append(self.playList.get_iter(row))
    for i in iters:
        if i is not None:
            self.playList.remove(i)
            self.RowCount -= 1
    self.sbar.push(self.context_id,"%d files in list." %
(self.RowCount))
```

La prima riga crea l'oggetto TreeSelection. Lo usiamo per ricavare le righe selezionate (che saranno solo una perché non abbiamo previsto che il modello supporti la selezione multipla) che inseriamo nella lista chiamata iters e quindi la scorriamo rimuovendo (come il metodo .clear). Di pari passo decrementiamo la variabile RowCount e quindi mostriamo il numero di file nella barra di stato.

Ora, prima di passare alle funzioni di spostamento, concentriamoci su quella di salvataggio del file. Useremo la classe FileDialog come prima. Inseriremo l'intero codice (in basso a destra) nella funzione on_btnGetFolder_clicked.

```
def on_btnGetFolder_clicked(self,widget):
    fd = FileDialog()
    filepath,self.CurrentPath = fd.ShowDialog(1,self.CurrentPath)
    self.txtPath.set_text(filepath[0])
```

L'unica differenza sta nell'ultima riga. Inseriremo il percorso ottenuto da FileDialog nel campo di testo impostato precedentemente usando il metodo set_text. Ricordate che i dati ritornati sono sotto forma di lista, anche nel caso di una sola voce. Per questo usiamo 'filepath[0]'.

Passiamo alla funzione per salvare il file. Possiamo

tranquillamente farlo prima di occuparci delle funzioni di spostamento. Creeremo una funzione chiamata savePlaylist. La prima cosa da fare (in alto a destra) è controllare se txtPath contiene qualcosa. Quindi se il campo di testo txtFilename contiene il nome del file. In entrambi i casi usiamo il metodo .get_text().

Ora che sappiamo di avere un percorso (fp) ed un nome (fn) possiamo aprire il file, inserirvi l'intestazione M3U e processare la playlist. Il percorso è contenuto (se ben ricordate) nella colonna 2, il nome del file nella colonna 0 e l'estensione nella colonna 1. Semplicemente creiamo (a destra) una stringa e la scriviamo nel file e quindi lo chiudiamo.

Possiamo ora iniziare a lavorare sulle funzioni di spostamento. Iniziamo dalla routine Sposta all'inizio. Proprio come abbiamo fatto nel caso della funzione Elimina, recuperiamo la selezione e quindi la riga selezionata. Poi dobbiamo muoverci tra le righe per recuperare due variabili. Le chiameremo path1 e path2. path2, in questo caso, sarà impostata a 0, che è la riga "bersaglio". path1 è il percorso selezionato dall'utente. Finiamo usando il metodo model.move_before() per muovere la riga selezionata fino a quella 0, praticamente spostando tutto giù. Inseriremo il codice (in basso a destra) direttamente nella funzione

```
def SavePlaylist(self):
    fp = self.txtPath.get_text()      # Get the filepath from the text box
    fn = self.txtFilename.get_text() # Get the filename from the filename text box
```

Ora controlliamo i valori...

```
    if fp == "":                      # IF the path is blank...
        self.MessageBox("error","Please provide a filepath for the playlist.")
    elif fn == "":                    # IF the filename is blank...
        self.MessageBox("error","Please provide a filename for the playlist file.")
    else:                             # Otherwise we are good to go.
```

```
plfile = open(fp + "/" + fn,"w")  # Open the file
plfile.writelines('#EXTM3U\n')    # Print the M3U Header
for row in self.playList:
    plfile.writelines("%s/%s.%s\n" % (row[2],row[0],row[1])) #Write the line data
plfile.close   # Finally close the file
```

Per finire, facciamo comparire la finestra che informa l'utente che il file è stato salvato.

```
self.MessageBox("info","Playlist file saved!")
```

Dobbiamo ora inserire una chiamata a questa funzione nella routine di gestione dell'evento on_btnSavePlaylist_clicked.

```
def on_btnSavePlaylist_clicked(self,widget):
    self.SavePlaylist()
```

Salvate il codice e testatelo. La playlist dovrebbe essere salvata correttamente e somigliare all'esempio fornito il mese scorso.

```
def on_tbtnMoveToTop_clicked(self,widget):
    sel = self.treeview.get_selection()
    (model,rows) = sel.get_selected_rows()
    for path1 in rows:
        path2 = 0
    iter1=model.get_iter(path1)
    iter2 = model.get_iter(path2)
    model.move_before(iter1,iter2)
```

on_tbtnMoveToTop_clicked.

Per la funzione MoveToBottom, useremo quasi lo stesso codice di MoveToTop ma, invece del metodo model.move_before() useremo model.move_after() e invece di impostare path2 a 0 sarà impostato a self.RowCount-1. Ora è evidente il senso della variabile RowCount. Ricordate che il conteggio incomincia da 0 così dobbiamo usare RowCount-1 (in alto a destra).

Ora diamo un'occhiata a cosa serve per creare la funzione MoveUp. Ancora una volta è molto simile alle ultime due. Questa volta useremo il numero della riga selezionata, assegnata a path1, ed assegnandolo, diminuito di uno, a path2. Quindi se path2 (la riga bersaglio) è maggiore o uguale a 0, useremo il metodo model.swap() (la seconda in basso a destra).

Stessa cosa per MoveDown. Questa volta però controlleremo se path2 è MINORE o uguale al valore di self.RowCount-1 (la terza in basso a destra).

Ora facciamo alcuni cambiamenti alle funzioni della nostra playlist. Nell'articolo del mese scorso vi mostrai il formato base del file playlist (in basso).

Ma vi dissi anche che c'era un formato esteso nel quale prima di ciascuna voce può essere aggiunta una riga contenente informazioni extra sulla canzone. La nuova riga ha il seguente formato...

```
#EXTINF:[Length of song in seconds],[Artist Name] – [Song Title]
```

Potreste esservi chiesti perché abbiamo incluso la libreria mutagen fin dall'inizio senza averla mai usata. Bene, lo faremo ora. Per rinfrescarvi la memoria, la libreria mutagen serve per accedere alle informazioni ID3 contenute all'interno dell'mp3. Per una discussione completa fate riferimento al numero 35 di Full Circle che contiene la parte 9 di questa serie. Creeremo una funzione che si occuperà di leggere il file mp3 e che restituisca il nome dell'artista, il titolo della canzone, la sua durata in secondi, che sono i tre dati di cui abbiamo bisogno per la riga di informazioni extra. Inserite la

```
def on_tbtnMoveToBottom_clicked(self,widget):
        sel = self.treeview.get_selection()
        (model,rows) = sel.get_selected_rows()
        for path1 in rows:
            path2 = self.RowCount-1
        iter1=model.get_iter(path1)
        iter2 = model.get_iter(path2)
        model.move_after(iter1,iter2)
```

```
def on_tbtnMoveUp_clicked(self,widget):
        sel = self.treeview.get_selection()
        (model,rows) = sel.get_selected_rows()
        for path1 in rows:
            path2 = (path1[0]-1,)
        if path2[0] >= 0:
            iter1=model.get_iter(path1)
            iter2 = model.get_iter(path2)
            model.swap(iter1,iter2)
```

```
def on_tbtnMoveDown_clicked(self,widget):
        sel = self.treeview.get_selection()
        (model,rows) = sel.get_selected_rows()
        for path1 in rows:
            path2 = (path1[0]+1,)
        iter1=model.get_iter(path1)
        if path2[0] <= self.RowCount-1:
            iter2 = model.get_iter(path2)
            model.swap(iter1,iter2)
```

funzione dopo ShowAbout all'interno della classe PlaylistCreator (pagina seguente, in alto a destra).

Sempre per rinfrescarvi la memoria, analizzerò il codice. Prima ripuliremo le tre variabili di ritorno così se non accade nulla resteranno vuote una volta restituite. Quindi passiamo il nome del file mp3 da

```
#EXTM3U
Adult Contemporary/Chris Rea/Collection/02 - On The Beach.mp3
Adult Contemporary/Chris Rea/Collection/07 - Fool (If You Think It's Over).mp3
Adult Contemporary/Chris Rea/Collection/11 - Looking For The Summer.mp3
```

controllare. Poi inseriamo le chiavi (sì, avete indovinato) nell'iteratore e lo controlleremo in cerca di due etichette specifiche. Sono 'TPE1' che contiene il nome dell'artista, e 'TIT2' che è il titolo della canzone. Ora, se la chiave non esiste, otterremo un errore così inseriamo in ciascuna chiamata l'istruzione 'try|except'. Quindi estraiamo la durata della canzone dall'attributo audio.info.length e restituiamo il tutto.

Ora dobbiamo modificare la funzione SavePlaylist affinché supporti le informazioni extra. Controlliamo se il nome del file esiste e, in caso affermativo, segnalarlo all'utente e uscire dalla funzione. Inoltre, per facilitare l'utente, dato che non supportiamo altri tipi di file, aggiungiamo automaticamente, se non esiste, l'estensione '.m3u' al percorso e nome del file. Prima inserite all'inizio del codice, tra import sys e import mutagen, una riga che importi os.path (in basso a destra).

Come nella funzione AddFilesToTreeview, useremo il metodo 'rfind' per trovare la posizione dell'ultimo punto ('.') nel nome del file fn. Se non c'è, il valore restituito è -1. Quindi controlliamo se il valore è -1 e, in questa eventualità, aggiungiamo l'estensione e subito dietro il nome del file, nel campo di testo, rendendolo più gradevole.

```
if os.path.exists(fp + "/" + fn):

self.MessageBox("error","The file already exists. Please select another.")
```

Quindi, nel resto della

```
def GetMP3Info(self,filename):
        artist = ''
        title = ''
        songlength = 0
        audio = MP3(filename)
        keys = audio.keys()
        for key in keys:
            try:
                if key == "TPE1":          # Artist
                    artist = audio.get(key)
            except:
                artist = ''
            try:
                if key == "TIT2":          # Song Title
                    title = audio.get(key)
            except:
                title = ''
            songlength = audio.info.length    # Audio Length
        return (artist,title,songlength)
```

```
import os.path
```

Quindi proseguite e commentate la funzione SavePlaylist preesistente e sostituitela.

```
   def SavePlaylist(self):
        fp = self.txtPath.get_text()      # Get the file path from the text box
        fn = self.txtFilename.get_text() # Get the filename from the text box
        if fp == "":  # IF filepath is blank...
            self.MessageBox("error","Please provide a filepath for the playlist.")
        elif fn == "": # IF filename is blank...
            self.MessageBox("error","Please provide a filename for the playlist file.")
        else:  # Otherwise
```

Fino a questo punto, la funzione è la stessa. Ecco dove iniziano le modifiche.

```
            extStart = fn.rfind(".") # Find the extension start position
            if extStart == -1:
                fn += '.m3u' #append the extension if there isn't one.
                self.txtFilename.set_text(fn) #replace the filename in the text box
```

funzione, inseriamo una clausola IF|ELSE (in alto a destra) così se il file esiste già, usciremo semplicemente dalla funzione. Useremo os.path.exists(filename) per eseguire il controllo.

Il resto del codice è praticamente simile al precedente, ma controlliamolo comunque.

La riga 2 apre il file su cui andiamo a scrivere. La riga 3 inserisce l'intestazione M3U. La riga 4 inizia la fase di processamento di ListStore. La riga 5 crea il nome del file dalle tre colonne di ListStore. La riga 6 chiama GetMP3Info e salva i valori restituiti nelle variabili. La riga 7 controlla se le tre variabili contengono un valore. In caso affermativo, inseriamo le informazioni estese con la riga 8, altrimenti no. La riga 9 scrive il nome del file come prima. La riga 10 chiude correttamente il file e la riga 11 fa comparire il messaggio che informa l'utente che il processo è terminato.

```
else:
        plfile = open(fp + "/" + fn,"w")  # Open the file
        plfile.writelines('#EXTM3U\n')    #Print the M3U header
        for row in self.playList:
           fname = "%s/%s.%s" % (row[2],row[0],row[1])
           artist,title,songlength = self.GetMP3Info(fname)
           if songlength > 0 and (artist != '' and title != ''):
              plfile.writelines("#EXTINF:%d,%s - %s\n" % (songlength,artist,title))
           plfile.writelines("%s\n" % fname)
        plfile.close  # Finally Close the file
        self.MessageBox("info","Playlist file saved!")
```

Proseguite salvando il codice e testatelo.

A questo punto manca solo di aggiungere qualche suggerimento per i nostri controlli quando l'utente si ferma sopra col puntatore del mouse. Aggiunge un tocco di professionalità (in basso). Creiamo ora la funzione.

Usiamo i riferimenti ai widget creati precedentemente e impostiamo il testo del suggerimento attraverso (avete indovinato) l'attributo set_tooltip_text. Quindi dobbiamo aggiungere la chiamata alla funzione. Ritorniamo alla routine __init__ e aggiungiamo, dopo self.SetWidgetReferences,

```
self.SetupToolTops()
```

Infine, ma certamente non per importanza, vogliamo inserire il nostro logo nella finestra Informazioni. Come per tutto il resto, c'è un attributo apposito. Aggiungete la seguente riga alla

```
def SetupToolTips(self):
        self.tbtnAdd.set_tooltip_text("Add a file or files to the playlist.")
        self.tbtnAbout.set_tooltip_text("Display the About Information.")
        self.tbtnDelete.set_tooltip_text("Delete selected entry from the list.")
        self.tbtnClearAll.set_tooltip_text("Remove all entries from the list.")
        self.tbtnQuit.set_tooltip_text("Quit this program.")
        self.tbtnMoveToTop.set_tooltip_text("Move the selected entry to the top of the list.")
        self.tbtnMoveUp.set_tooltip_text("Move the selected entry up in the list.")
        self.tbtnMoveDown.set_tooltip_text("Move the selected entry down in the list.")
        self.tbtnMoveToBottom.set_tooltip_text("Move the selected entry to the bottom of the list.")
        self.btnGetFolder.set_tooltip_text("Select the folder that the playlist will be saved to.")
        self.btnSavePlaylist.set_tooltip_text("Save the playlist.")
        self.txtFilename.set_tooltip_text("Enter the filename to be saved here.  The extension '.m3u' will be added for
you if you don't include it.")
```

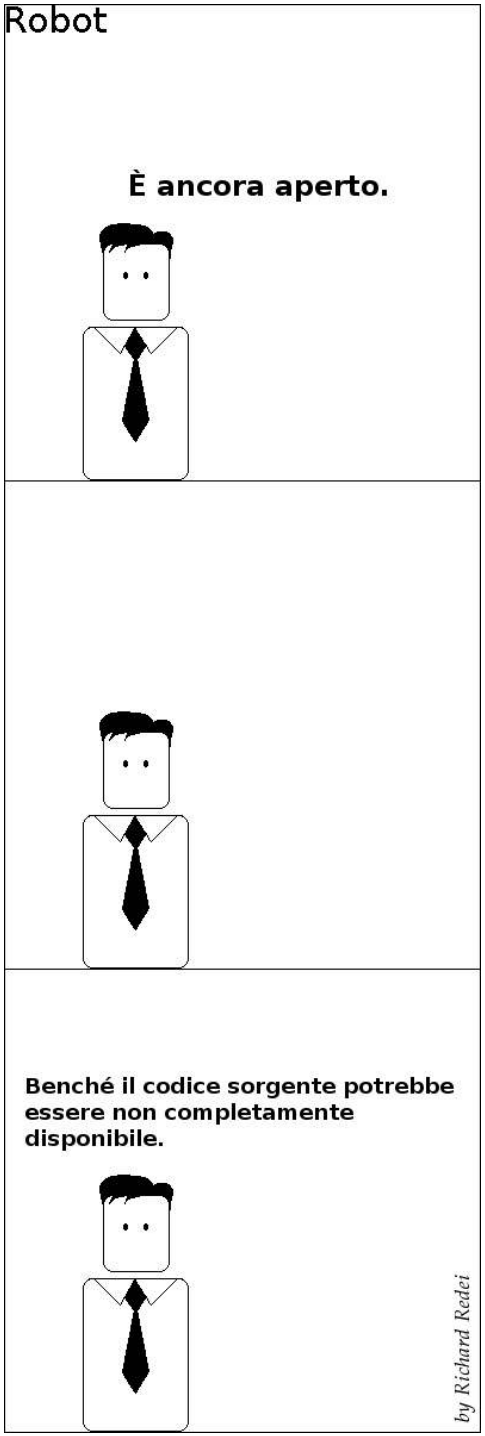
funzione ShowAbout.

```
about.set_logo(gtk.gdk.pixbuf_new_from_file("logo.png"))
```

Questo è tutto. Ora avete un programma totalmente funzionante, bello e che che fa un ottimo lavoro nella creazione delle playlist per i vostri file musicali.

L'intero codice sorgente, incluso il file glade creato il mese scorso, lo potete trovare su pastebin: http://pastebin.com/tQJizcwT

Fino alla prossima volta, gioite delle vostre nuove abilità.



Greg Walters è il proprietario della RainyDay Solutions, LLC, una società di consulenza in Aurora, Colorado e programma dal 1972. Ama cucinare, fare escursioni, ascoltare musica e passare il tempo con la sua famiglia. Il suo sito web: www.thedesignatedgeek.com.

Programmare in Python - Parte 24

Caspita! È difficile credere che questa è la parte 24. Sono due anni che stiamo imparando Python! Avete fatto una lunga strada.

Questa volta ci occuperemo di due argomenti. Il primo è stampare con una stampante, il secondo è la creazione di file RTF (Rich Text Format).

Stampare su Linux

Allora iniziamo con la stampa tramite stampante. L'idea per questo argomento è venuta da una mail inviata da Gord Campbell. Oggi è facile stampare su Linux, più facile che su altri sistemi operativi che iniziano per "Win", di cui non mi occupo.

Finché non si vuole che stampare del semplice testo, senza grassetto, corsivo, caratteri multipli, ecc è tutto facile. Ecco un'applicazione semplice che stamperà direttamente con la vostra stampante...

```
import os

pr = os.popen('lpr','w')

pr.write('print test from
linux via python\n')

pr.write('Print finished\n')

pr.close()
```

È facile da comprendere se espandete un po' la vostra mente. Nel codice sopra, 'lpr' è lo spooler di stampa. L'unico requisito è aver già configurato e avviato 'lpd'. Ci sono buone possibilità che usando Ubuntu sia già tutto pronto. In genere ci si riferisce a 'lpd' come ad un "filtro magico" capace di convertire automaticamente diversi tipi di documenti in qualcosa che può essere compreso dalla stampante. Stamperemo sul dispositivo/oggetto 'lpr'. Pensate ad esso come fosse un semplice file. Lo apriamo. Dobbiamo importare 'os'. Quindi nella riga 2, apriamo 'lpr' con permessi di scrittura, assegnandolo alla variabile 'pr'. Quindi usiamo 'pr.write' passandogli tutto quello che vogliamo stampare. Per finire, riga 5, chiudiamo il file e i dati saranno inviati alla stampante.

Possiamo anche creare un file di testo e inviarlo alla stampante in questo modo...

```
import os

filename = 'dummy.file'

os.system('lpr %s'
%filename)
```

In questo caso stiamo ancora usando l'oggetto lpr ma tramite l'istruzione 'os.system' che in pratica crea un comando simile a quello che avremmo usato da terminale.

Ora vi lascio giocarci.

PyRTF

Ora occupiamoci dei file RTF. Il formato RTF (che è come dire numero PIN, dato che PIN sta per Numero di Identificazione Personale, cosicché si traduce in Numero Numero-di-Identificazione-Personale. Forse dal dipartimento del dipartimento di ridondanza, eh?) fu creato in origine da Microsoft Corporation nel 1987 e la sua sintassi venne influenzata dal linguaggio typesetting TeX. PyRTF è una libreria meravigliosa che rende semplice scrivere file RTF. Dovete pianificare in anticipo l'aspetto dei vostri file, ma il risultato ne vale la pena.

> **Caspita! È difficile credere che questa è la parte 24. Sono due anni che stiamo imparando Python!**

Prima di tutto dovete scaricare e installare il pacchetto PyRTF. Andate su http://pyrtf.sourceforge.net e recuperate il pacchetto PyRTF-0.45.tar.gz. Salvatelo e usate il gestore degli archivi per scompattarlo. Quindi dal terminale posizionatevi dove lo avete estratto. Prima bisogna installarlo, quindi digitate "sudo python setup.py install" e verrà installato per voi. Notate che c'è una cartella chiamata examples. Contiene informazioni utili su come eseguire operazioni avanzate.

Eccoci qua. Come nostra abitudine creiamo lo scheletro del programma che è mostrato nella pagina seguente, in alto a destra.

Prima di procedere, discuteremo di cosa abbiamo fatto. La riga 2 importa la libreria PyRTF. Osservate l'uso di un formato di import diverso dal solito. Serve per importare tutto, dalla libreria.

La nostra routine principale di lavoro è MakeExample. Per ora solo lo scheletro. La funzione OpenFile crea il file con il nome passatogli, aggiunge l'estensione "rtf", attiva la modalità "scrittura" e restituisce l'oggetto.

Abbiamo già discusso in precedenza la funzione __name__ ma, per rinfrescarvi la memoria, se eseguiamo il programma da solo, la variabile interna __name__ è impostata a "__main__". Se invece lo usiamo importandolo in un altro programma allora questa porzione di codice sarà ignorata.

Qui creiamo una istanza dell'oggetto Render, chiamiamo la routine MakeExample e recuperiamo l'oggetto restituito doc. Quindi scriviamo il file (in doc) usando la funzione OpenFile.

Ora la parte principale della routine MakeExample. Sostituite l'istruzione pass con il codice mostrato in basso.

Diamo un'occhiata a quello che abbiamo fatto. Nella prima riga abbiamo creato un'istanza di Document. Quindi abbiamo creato una istanza del foglio di stile. Proseguiamo creando un'istanza dell'oggetto section e lo aggiungiamo al documento. Pensate a una sezione come a un capitolo di un libro. Quindi creiamo un paragrafo usando lo stile Normale. L'autore di PyRTF lo ha definito con carattere Arial, 11 punti. Possiamo inserire qualunque testo nel paragrafo, aggiungentelo alla sezione e restituendo il nostro documento doc.

È molto facile. Ancora, dovete pianificare l'aspetto finale con molta attenzione, ma non in maniera onerosa.

Salvate il programma come "rtftesta.py" ed eseguitelo. Quando ha finito, usate OpenOffice (o LibreOffice) per aprire e controllare il file.

Ora facciamo qualcosa di più sfizioso. Primo, aggiungeremo un'intestazione. Ancora una volta, l'autore di PyPDF ci ha fornito uno stile predefinito chiamato Header1. Lo useremo per la nostra intestazione.

Tra le righe doc.Sections.append e p = Paragraph, aggiungete quanto segue.

```
#!/usr/bin/env python
from PyRTF import *

def MakeExample():
    pass

def OpenFile(name) :
    return file('%s.rtf' % name, 'w')

if __name__ == '__main__' :
    DR = Renderer()
    doc = MakeExample()
    DR.Write(doc, OpenFile('rtftesta'))
    print "Finished"
```

```
p = Paragraph(ss.ParagraphStyles.Heading1)

p.append('Example Heading 1')

section.append(p)
```

```
doc = Document()
ss  = doc.StyleSheet
section = Section()
doc.Sections.append(section)

p = Paragraph(ss.ParagraphStyles.Normal)
p.append('This is our first test writing to a RTF file. '
         'This first paragraph is in the preset style called normal '
         'and any following paragraphs will use this style until we change it.')
section.append(p)

return doc
```

Cambiate il nome del file rtf in "rtftestb". Dovrebbe assomigliare a questo:

```
DR.Write(doc,
OpenFile('rtftestb'))
```

Salvatelo come rtftestb.py ed eseguitelo. Ora abbiamo un'intestazione. Sono sicuro che starete già pensando alle numerose operazioni che potremmo fare. Ecco un elenco degli stili predefiniti forniti dall'autore.

Normale, Normale Breve, Intestazione 1, Intestazione 2, Normale Numerato, Normale Numerato 2. C'è anche lo stile Elenco, che lascerò scoprire a voi stessi. Se volete saperne di più, su questo o altre cose, gli stili sono definiti nel file Elements.py, nel percorso di installazione.

Anche se questi stili vanno generalmente bene, vorreste poter usare qualcosa di nuovo. Vediamo come cambiare al volo tipo, dimensione e attributi (grassetto, corsivo, ecc) dei caratteri. Dopo il nostro paragrafo ma prima di restituire l'oggetto document inseriamo il codice mostrato in alto a destra, e cambiamo il nome del file risultante in rtftestc. Salvate il file come rtftestc.py. Ed eseguitelo. La nuova porzione di codice dovrebbe somigliare a questo...

> **Vediamo come cambiare al volo tipo, dimensione e attributi (grassetto, corsivo, ecc) dei caratteri.**

```
p = Paragraph(ss.ParagraphStyles.Normal)
p.append( 'It is also possible to provide overrides for elements of a style. ',
        'For example you can change just the font ',
        TEXT(' size to 24 point', size=48),
        ' or',
        TEXT(' typeface to Impact', font=ss.Fonts.Impact),
        ' or even more Attributes like',
        TEXT(' BOLD',bold=True),
        TEXT(' or Italic',italic=True),
        TEXT(' or BOTH',bold=True,italic=True),
        '.' )
section.append(p)
```

È anche possibile fornire nuove versioni per gli elementi di uno stile. Per esempio, potete cambiare solo la dimensione del carattere a 24 punti o il tipo a Impact o anche più attributi come grassetto o corsivo o entrambi.

Cosa abbiamo fatto adesso? La riga 1 crea un nuovo paragrafo. Quindi iniziamo, come fatto prima, a inserire il nostro testo. Osservate la quarta riga (TEXT(' size to 24 point', size = 48),). Usando il qualificatore TEXT, stiamo dicendo a PyRTF di fare qualcosa di diverso nel mezzo della frase, che in questo caso è cambiare la dimensione del carattere (Arial in questo punto) a 24 punti, seguito dal comando 'size = '. Ma, aspettate un attimo. 'size =' dice 48 e quello che stiamo stampando dice 24 punti, e l'output sarà a 24 punti. Cosa è accaduto? Bé, il comando size è in mezzi punti. Così se vogliamo un carattere a 8 punti dobbiamo usare size = 16. Ha senso?

Quindi, proseguiamo e cambiamo il carattere con il comando 'font ='. Ancora, riguarderà tutto quello inserito tra gli apici singoli del qualificatore TEXT.

Ok, Se tutto questo ha senso, cos'altro possiamo fare?

Possiamo impostare il colore del testo del comando in linea TEXT. Come questo.

```
p = Paragraph()

p.append('This is a new
paragraph with the word',

TEXT('RED',colour=ss.Colours
.Red),

' in Red text.')

section.append(p)
```

Notate che non abbiamo bisogno di impostare lo stile del paragrafo su Normale, poiché verrà usato fino al prossimo cambiamento. Notate anche che se vivete negli U.S.A. dovrete usare il nome di colore "appropriato".

Ecco i colori (di nuovo) predefiniti: Black, Blue, Turquoise, Green, Pink, Red, Yellow, White, BlueDark, Teal, GreenDark, Violet, RedDark, YellowDark, GreyDark and Grey.

Ed ecco un elenco dei caratteri predefiniti (nella notazione per usarli):

Arial, ArialBlack, ArialNarrow, BitstreamVeraSans, BitstreamVeraSerif, BookAntiqua, BookmanOldStyle, BookmanOldStyle, Castellar, CenturyGothic, ComicSansMS, CourierNew, FranklinGothicMedium, Garamond, Georgia, Haettenschweiler, Impact, LucidaConsole, LucidaSansUnicode, MicrosoftSansSerif, PalatinoLinotype, MonotypeCorsiva, Papyrus, Sylfaen, Symbol, Tahoma, TimesNewRoman, TrebuchetMS e Verdana.

Ora starete pensando che tutto questo è bello e buono, ma come creare degli stili propri? È molto facile. Spostiamoci all'inizio del nostro file e, prima della riga di intestazione, aggiungiamo il codice seguente.

```
result = doc.StyleSheet

NormalText = TextStyle(TextPropertySet(result.Fonts.CourierNew,16))
```

```
    p = Paragraph(ss.ParagraphStyles.Courier)
    p.append('Now we are using the Courier style at 8 points. '
             'All subsequent paragraphs will use this style automatically. '
             'This saves typing and is the default behaviour for RTF documents.',LINE)
    section.append(p)
    p = Paragraph()
    p.append('Also notice that there is a blank line between the previous paragraph ',
             'and this one.  That is because of the "LINE" inline command.')

    section.append(p)
```

```
ps2 = ParagraphStyle('Courier',NormalText.Copy())

result.ParagraphStyles.append(ps2)
```

Prima di scrivere il codice per usarlo, osserviamo cosa abbiamo fatto. Abbiamo creato una istanza di un nuovo foglio di stile chiamato result. Nella seconda riga, abbiamo impostato il carattere a Courier New 8 punti e quindi registrato lo stile come Courier. Ricordate, dobbiamo usare 16 come dimensione dato che la dimensione del carattere è in mezzi punti.

Ora, prima della riga return in fondo alla funzione, inseriamo un nuovo paragrafo usando lo stile Courier.

Così ora possedete un nuovo stile che potrete usare a piacimento.

Potete usare qualunque carattere dall'elenco precedente e creare uno stile personalizzato. Basta copiare il codice e sostituire carattere e dimensione come desiderato. Possiamo fare anche questo...

```
NormalText = TextStyle(TextPropertySet(result.Fonts.Arial,22,bold=True,colour=ss.Colours.Red))

ps2 = ParagraphStyle('ArialBoldRed',NormalText.Copy())

result.ParagraphStyles.append(ps2)
```

E aggiungere il codice in basso...

```
p = Paragraph(ss.ParagraphStyles.ArialBoldRed)

p.append(LINE,'And now we are using the ArialBoldRed style.',LINE)

section.append(p)
```

per stampare lo stile ArialBoldRed.

Tabelle

Molte volte, le tabelle sono l'unico modo per rappresentare propriamente dati in un documento. Creare tabelle in modalità testo è difficile e, in alcuni casi, è più semplice con PyRTF. Vi spiegherò questa affermazione più avanti in questo articolo.

Osserviamo una tabella standard (mostrata in basso) in OpenOffice/LibreOffice. Sembra un foglio di calcolo, dove tutto è incolonnato.

Le righe vanno da sinistra a destra,

Column Header 1	Column Header 2	Column Header 3
Row 1 data 1	Row 1 data 2	Row 1 data 3
Row 2 data 1	Row 2 data 2	Row 2 data 3

le colonne verso il basso. Un concetto semplice.

Iniziamo una nuova applicazione e chiamiamola rtfTable-a.py. Partiamo dal nostro codice standard (mostrato nella pagina seguente) e proseguiamo da lì.

Non c'è motivo di discuterne dato che è in pratica lo stesso codice usato prima. Ora, arricchiremo la funzione TableExample. Sto, in pratica, usando parte del codice d'esempio fornito dall'autore di PyRTF. Sostituite l'istruzione pass nella funzione con il codice seguente...

```
doc = Document()

ss = doc.StyleSheet

section = Section()

doc.Sections.append(section)
```

Questa parte è la stessa della precedente, quindi glissiamo.

```
table =
Table(TabPS.DEFAULT_WIDTH *
7,


TabPS.DEFAULT_WIDTH * 3,


TabPS.DEFAULT_WIDTH * 3)
```

Questa riga (sì, è davvero una riga, ma spezzettata per renderla più leggibile) crea la nostra tabella base. Stiamo creando una tabella con 3 colonne, la prima è larga 7 tabulazioni, la seconda e la terza 3 tabulazioni. Oltre alle tabulazioni è possibile usare anche il twip. Di più a riguardo tra un attimo.

```
c1 = Cell(Paragraph('Row
One, Cell One'))

c2 = Cell(Paragraph('Row
One, Cell Two'))

c3 = Cell(Paragraph('Row
One, Cell Three'))

table.AddRow(c1,c2,c3)
```

Qui stiamo impostando i dati che andranno in ciascuna cella della prima riga.

```
c1 =
Cell(Paragraph(ss.ParagraphS
tyles.Heading2,'Heading2
Style'))

c2 =
Cell(Paragraph(ss.ParagraphS
tyles.Normal,'Back to Normal
Style'))

c3 = Cell(Paragraph('More
Normal Style'))
```

```
#!/usr/bin/env python

from PyRTF import *

def TableExample():
    pass

def OpenFile(name):
    return file('%s.rtf' % name, 'w')

if __name__ == '__main__':
    DR = Renderer()
    doc = TableExample()
    DR.Write(doc, OpenFile('rtftable-a'))
    print "Finished"
```

```
table.AddRow(c1,c2,c3)
```

Questo gruppo di codice imposta i dati per la seconda riga. Notate come sia possibile impostare uno stile differente per una o più celle.

```
c1 =
Cell(Paragraph(ss.ParagraphS
tyles.Heading2,'Heading2
Style'))

c2 =
Cell(Paragraph(ss.ParagraphS
tyles.Normal,'Back to Normal
Style'))

c3 = Cell(Paragraph('More
Normal Style'))

table.AddRow(c1,c2,c3)
```

Questo imposta la riga finale.

```
section.append(table)

return doc
```

Questo aggiunge la tabella alla sezione e restituisce il documento per la stampa.

Salvate ed eseguite l'applicazione. Notate che tutto è come previsto tranne che per l'assenza di bordi nella tabella. Questo ci complica il lavoro. Sistemiamolo. Ancora, userò principalmente il codice dal file d'esempio fornito dall'autore di PyRTF.

Salvate il file come rtftable-b.py. Ora, cancellate tutto quello tra 'doc.Sections.append(section)' e 'return doc' nella funzione TableExample e sostituetolo con il

seguente...

```
thin_edge =
BorderPS(width=20,
style=BorderPS.SINGLE )

thick_edge =
BorderPS(width=80,
style=BorderPS.SINGLE )

thin_frame =
FramePS(thin_edge,
thin_edge, thin_edge,
thin_edge )

thick_frame =
FramePS(thick_edge,
thick_edge, thick_edge,
thick_edge )

mixed_frame =
FramePS(thin_edge,
thick_edge, thin_edge,
thick_edge )
```

Qui stiamo impostando le definizioni dei margini e delle cornici che useremo.

```
table = Table(
    TabPS.DEFAULT_WIDTH * 3,
    TabPS.DEFAULT_WIDTH * 3,
    TabPS.DEFAULT_WIDTH * 3
)

c1 = Cell( Paragraph( 'R1C1'
), thin_frame )

c2 = Cell( Paragraph( 'R1C2'
) )

c3 = Cell( Paragraph( 'R1C3'
), thick_frame )

table.AddRow( c1, c2, c3 )
```

Nella prima riga, la cella nella prima colonna (cornice sottile) e quella nella terza colonna (cornice spessa) avranno un bordo tutto in torno.

```
c1 = Cell( Paragraph( 'R2C1'
) )

c2 = Cell( Paragraph( 'R2C2'
) )

c3 = Cell( Paragraph( 'R2C3'
) )

table.AddRow( c1, c2, c3 )
```

Nessuna cella della seconda riga avrà il bordo.

```
c1 = Cell( Paragraph( 'R3C1'
), mixed_frame )

c2 = Cell( Paragraph( 'R3C2'
) )

c3 = Cell( Paragraph( 'R3C3'
), mixed_frame )

table.AddRow( c1, c2, c3 )
```

Ancora una volta, le celle della prima e terza colonna della terza riga presentano una cornice mista.

```
section.append( table )
```

Finito. Avete quasi tutto quello necessario per creare, col codice, documenti RTF.

Ci vediamo la prossima volta!

Il codice può essere recuperato, come al solito, su pastebin. La prima parte la trovate all'indirizzo http://pastebin.com/3Rs7T3D7 che è la somma di rtftest.py (a-e), mentre la seconda, rtftable.py (a-b), la trovate all'indirizzo http://pastebin.com/XbaE2uP7.

Greg Walters è il proprietario della RainyDay Solutions, LLC, una società di consulenza in Aurora, Colorado e programma dal 1972. Ama cucinare, fare escursioni, ascoltare musica e passare il tempo con la sua famiglia. Il suo sitoweb è www.thedesignatedgeek.com.

Programmare in Python - Parte 25

Alcuni di voi hanno lasciato commenti sugli articoli dedicati alla programmazione di GUI e su quanto vi siano piaciuti. Come risposta inizieremo a dare un'occhiata ad un diverso gruppo di strumenti per creare interfacce grafiche, chiamato Tkinter. Si tratta degli strumenti "ufficiali" per creare GUI in python. Tkinter è in circolazione da parecchio e si è fatto una brutta fama per il suo stile un po' retró. Recentemente le cose sono cambiate così penso possiamo cercare di sfatare questo pregiudizio.

NOTATE - Tutto il codice qui presentato riguarda unicamente Python 2.x. In uno dei prossimi articoli parleremo di Tkinter con Python 3.x. Se siete "costretti" a usare la versione 3.x sostituite l'istruzione import con "from tkinter import *".

Una breve storia e qualche antefatto

TKinter sta per "**Tk inter**face", interfaccia Tk. Tk è un linguaggio di programmazione unico e il modulo Tkinter permette di usarne i componenti grafici. Ci sono diversi widget nativi. Il contenitore detto Toplevel (la finestra), pulsanti, etichette, cornici, campi di testo, pulsanti di selezione singola o multipla, canvas, campi di testo multi-riga ed altri. Ci sono molti moduli che aggiungono ulteriori funzionalità. Questo mese ci concentreremo su 4 widget. Il Toplevel (da questo momento semplicemente finestra radice), cornici, etichette e pulsanti. Nel prossimo articolo ci occuperemo più approfonditamente di altri widget.

In pratica il contenitore di alto livello contiene gli altri widget. Si tratta della finestra radice o master. All'interno di questa posizioniamo i controlli che vogliamo usare nel nostro programma. Ciascuno di questi, ad eccezione della finestra principale, ha un genitore. Non necessariamente il genitore deve essere la finestra. Può essere un altro widget. Di più a riguardo il mese prossimo. Questo mese tutti i widget avranno come genitore la finestra principale.

Per posizionare e visualizzare i widget figli, dobbiamo usare la cosiddetta "gestione geometrica". Si tratta di come i vari componenti sono inseriti nella finestra. La maggior parte dei programmatori usa uno dei tre tipi di organizzazione, pack, grid o place. In base alla mia umile opinione il metodo pack è molto sciatto. Lascio a voi constatarlo. Il metodo di gestione place permette un accurato posizionamento dei widget, ma può risultare complicato. Ne parleremo in uno dei prossimi articoli. Per questa volta ci concentreremo sul metodo grid, a griglia.

Pensate a un foglio di calcolo. Ci sono righe e colonne. Le colonne sono verticali, le righe orizzontali. Ecco una semplice rappresentazione testuale degli indirizzi delle celle in una tabella di 5 colonne e 4 righe (in alto a destra).

```
    COLONNE - >
RIGHE| 0,0 | 1,0 | 2,0 | 3,0 | 4,0 |
  |  | 0,1 | 1,1 | 2,1 | 3,1 | 4,1 |
     | 0,2 | 1,2 | 2,2 | 3,2 | 4,2 |
     | 0,3 | 1,3 | 2,3 | 3,3 | 4,3 |
```

Allora, il genitore contiene la griglia, i widget vanno nelle celle. A prima vista questo può sembrare molto limitante. Però i widget possono estendersi e occupare più celle di una colonna, di una riga o di entrambe.

Primo esempio

Il nostro primo esempio è MOLTO facile (solo quattro righe), ma è un buon inizio.

```
from Tkinter import *

root = Tk()

button = Button(root, text =
"Hello FullCircle").grid()

root.mainloop()
```

Allora, cosa abbiamo fatto? La riga uno importa la libreria Tkinter. Nella seguente, istanziamo

l'oggetto Tk usando root (Tk è parte di Tkinter). Ecco la riga tre:

```
button = Button(root, text = "Hello FullCircle").grid()
```

Creiamo un pulsante chiamato button, impostiamo come suo genitore la finestra root, impostiamo il suo testo a "Hello FullCircle" e lo sistemiamo nella griglia. Per finire, chiamiamo il mainloop della finestra. Molto semplice dalla nostra prospettiva, ma molto avviene dietro la scena. Per fortuna, non dobbiamo capirlo in questo momento.

Eseguite il programma e osservate cosa accade. Sul mio computer la finestra principale è mostrata nell'angolo in basso a sinistra dello schermo. Nel vostro caso, potrebbe essere da tutt'altra parte. Sistemiamo la cosa con il prossimo esempio.

Secondo esempio

Questa volta creeremo una classe chiamata App. Conterrà la nostra finestra. Iniziamo.

```
from Tkinter import *
```

È l'istruzione per la libreria Tkinter.

```
class App:
    def __init__(self, master):
        frame = Frame(master)
        self.lblText = Label(frame, text = "This is a label widget")
        self.btnQuit = Button(frame, text="Quit", fg="red", command=frame.quit)
        self.btnHello = Button(frame, text="Hello", command=self.SaySomething)
        frame.grid(column = 0, row = 0)
        self.lblText.grid(column = 0, row = 0, columnspan = 2)
        self.btnHello.grid(column = 0, row = 1)
        self.btnQuit.grid(column = 1, row = 1)
```

Definiamo la nostra classe e, nella funzione __init__, configuriamo i widget e li posizioniamo nella griglia.

La prima riga nella routine __init__ crea una cornice (frame) che sarà il genitore di tutti gli altri widget. Il genitore della cornice è la finestra (widget Toplevel). Quindi definiamo una etichetta e due pulsanti. Osserviamo la riga che crea l'etichetta.

```
self.lblText = Label(frame, text = "This is a label widget")
```

Abbiamo creato l'etichetta chiamandola self.lblText. È eridata dall'oggetto Label. Impostiamo il suo genitore (frame) e il testo che vogliamo visualizzare (text = "this is a label widget"). Tutto quì.

Potremmo fare di più, ma per il momento è sufficiente. A seguire configuriamo i due pulsanti di cui abbiamo bisogno:

```
self.btnQuit = Button(frame, text="Quit", fg="red", command=frame.quit)

self.btnHello = Button(frame, text="Hello", command=self.SaySomething)
```

Assegnamo un nome, il loro genitore (frame) e il testo da mostrare. btnQuit ha l'attributo fg impostato su "red". Avrete intuito che serve a colorare di rosso il colore in primo piano o il colore del testo. L'ultimo attributo serve a configurare il comando di supporto (callback) che vogliamo utilizzare quando l'utente fa clic sul pulsante. In questo caso si tratta di frame.quit, che termina il programma. È una funzione predefinita, quindi non dobbiamo crearla. Nel caso di btnHello si tratta della funzione chiamata self.SaySomething. Questa dobbiamo crearla, ma prima abbiamo altro da fare.

Dobbiamo inserire i widget nella griglia. Ecco ancora le righe:

```
frame.grid(column = 0, row = 0)

self.lblText.grid(column = 0, row = 0, columnspan = 2)

self.btnHello.grid(column = 0, row = 1)

self.btnQuit.grid(column = 1, row = 1)
```

Prima assegnamo una griglia alla cornice. Quindi impostiamo l'attributo grid di ciascun widget con la posizione nella griglia. Notate la riga columnspan per l'etichetta (self.lblText). Serve per far occupare all'etichetta due

colonne. Poiché abbiamo solo due colonne, allora si tratta dell'intera larghezza dell'applicazione. Ora possiamo creare la funzione di supporto:

```
def SaySomething(self):

   print "Hello to FullCircle
Magazine Readers!!"
```

Questo, semplicemente, stampa nel terminale il messaggio "Hello to FullCircle Magazine Readers!!"

Per finire instanziamo la classe Tk, la nostra classe App, ed eseguiamo il ciclo principale.

```
root = Tk()

app = App(root)

root.mainloop()
```

Proviamola. Ora le cose effettivamente funzionano. Ma di nuovo la posizione della finestra è inappropriata. Sistemiamola nel prossimo esempio.

Terzo esempio

Salvate l'ultimo esempio come example3.py. È tutto esattamente uguale tranne che per una riga. È in fondo alla nostra funzione principale. Ecco le righe precedenti con quella nuova:

```
root = Tk()

root.geometry('150x75+550+150
')

app = App(root)

root.mainloop()
```

```
class Calculator():
    def __init__(self,root):
        master = Frame(root)
        self.CurrentValue = 0
        self.HolderValue = 0
        self.CurrentFunction = ''
        self.CurrentDisplay = StringVar()
        self.CurrentDisplay.set('0')
        self.DecimalNext = False
        self.DecimalCount = 0
        self.DefineWidgets(master)
        self.PlaceWidgets(master)
```

Il suo scopo è forzare la finestra iniziale ad una larghezza di 150 pixel e ad una altezza di 75 pixel. Inoltre si sceglie di posizionare l'angolo in alto a sinistra a 550 pixel dal margine sinistro dello schermo e a 150 dal margine superiore. Come ho ottenuto questi numeri? Ho iniziato con dei valori approssimati e li ho perfezionati un po' alla volta. È un metodo un po' complicato, ma il risultato è meglio di niente.

Quarto esempio - Un semplice calcolatore

Ora proviamo a realizzare qualcosa di più complicato. Questa volta creeremo una calcolatrice "4 funzioni", ovvero con le quattro operazioni base: addizione, sottrazione, moltiplicazione e divisione.

```
----------------
|            0 |
----------------
| 1 | 2 | 3 | + |
----------------
| 4 | 5 | 6 | - |
----------------
| 7 | 8 | 9 | * |
----------------
| - | 0 | . | / |
----------------
|       =      |
----------------
|     CLEAR    |
----------------
```

```
from Tkinter import *

def StartUp():
    global val, w, root
    root = Tk()
    root.title('Easy Calc')
    root.geometry('247x330+469+199')
    w = Calculator(root)
    root.mainloop()
```

Ci dedicheremo ad essa e spiegherò man mano il codice (al centro a destra).

Ad eccezione dell'istruzione geometry, questo (a sinistra) dovrebbe ora essere facile da comprendere. Ricordate, prendete dei valori approssimati, aggiustateli e proseguite.

Iniziamo con la definizione della classe e impostiamo la funzione __init__. Configuriamo tre variabili come segue:

- CurrentValue: conterrà il valore attuale inserito nella calcolatrice.
- HolderValue: conterrà il valore inserito prima di scegliere un'operazione.
- CurrentFunction: è un semplice promemoria dell'operazione attualmente usata.

A seguire, definiamo la variabile

CurrentDisplay e l'assegnamo all'oggetto StringVar. Si tratta di un oggetto speciale di Tkinter. Qualunque widget gli venga assegnato il valore di quest'ultimo è aggiornato automaticamente. In questo caso, lo useremo per conservare qualunque cosa vogliamo che il widget etichetta, bé..., mostri. Dobbiamo instanziarlo prima di assegnarlo al widget. Quindi usiamo la funzione nativa 'set'. Poi definiamo una variabile booleana chiamata DecimalNext, la variabile DecimalCount e chiamiamo prima la funzione DefineWidgets, che crea tutti i widget, e poi PlaceWidget, che li posiziona nella finestra.

```
def
DefineWidgets(self,master):

self.lblDisplay =
Label(master,anchor=E,relief
=
SUNKEN,bg="white",height=2,te
xtvariable=self.CurrentDispla
y)
```

Precedentemente abbiamo definito una etichetta. Però questa volta aggiungeremo altri attributi. Notate che non usiamo l'attributo 'text'. Invece assegneremo l'etichetta al genitore (master), quindi imposteremo l'ancoraggio (o, per i nostri scopi, l'allineamento) del testo, quando viene scritto. In questo caso, stiamo dicendo all'etichetta di allineare tutto il testo a Est o sul lato destro del widget. Esiste l'attributo justify, ma è utilizzato per i campi di testo multi-riga. L'attributo anchor ha le seguenti opzioni: N, NE, E, SE, S, SW, W, NW e CENTER. Quello predefinito è CENTER. Dovete immaginare una bussola. In condizioni normali, gli unici valori davvero utilizzabili sono E (destra), W(sinistra) e Center.

Quindi configuriamo il margine o stile visivo dell'etichetta. Le opzioni "legali" sono FLAT (piatto), SUNKEN (incavato), RAISED (rialzato), GROOVE (scanalato) e RIDGE (rialzato). Il valore predefinito, se non viene specificato nulla, è FLAT. Siete liberi di provare voi stessi le altre combinazioni. A seguire impostiamo uno sfondo (bg) bianco per differenziarlo un po' dal resto della finestra. Impostiamo l'altezza a 2 (cioè due righe di testo, non pixel) e, per finire, assegnamo la variabile definita un attimo fa (self.CurrentDisplay) all'attributo textvariable. Ogniqualvolta il valore di self.CurrentDisplay cambia, l'etichettà rifletterà tale cambiamento.

```
self.btn1 = Button(master, text = '1',width = 4,height=3)
self.btn1.bind('<ButtonRelease-1>', lambda e: self.funcNumButton(1))
self.btn2 = Button(master, text = '2',width = 4,height=3)
self.btn2.bind('<ButtonRelease-1>', lambda e: self.funcNumButton(2))
self.btn3 = Button(master, text = '3',width = 4,height=3)
self.btn3.bind('<ButtonRelease-1>', lambda e: self.funcNumButton(3))
self.btn4 = Button(master, text = '4',width = 4,height=3)
self.btn4.bind('<ButtonRelease-1>', lambda e: self.funcNumButton(4))
```

Come mostrato in basso, creeremo alcuni pulsanti.

Vi ho mostrato solo quattro pulsanti. Questo perché, come potete vedere, il codice è quasi esattamente lo stesso. Di nuovo, abbiamo creato i pulsanti precedentemente in questa lezione, ma osserviamo più attentamente cosa abbiamo fatto.

Iniziamo definendo il genitore (master), il testo che vogliamo sul pulsante, larghezza e altezza. Notate come la larghezza sia espressa in caratteri e l'altezza in righe di testo. Se avete intenzione di inserire un'immagine nel pulsante allora dovrete usare i pixel. Questa differenza potrebbe confondere un po' prima di prenderci la mano. A seguire, impostiamo l'attributo bind. Quando, negli esempi precedenti, abbiamo creato i pulsanti abbiamo usato l'attributo 'command=' per indicare la funzione da chiamare in caso di clic dell'utente. Questa volta useremo l'attributo '.bind'. È praticamente la stessa cosa, ma più semplice e con la possibilità di passare argomenti alla funzione, che è statica. Notate che stiamo usando '<ButtonRelease-1>' come innesco dell'associazione. In questo caso, vogliamo assicurarci che solo dopo il rilascio del pulsante sinistro del mouse venga chiamata la funzione di callback. Infine, definiamo la funzione di supporto da chiamare e cosa le andremo a passare. Ora i più astuti, cioe tutti voi, avranno notato una novità. La chiamata 'lambda e:'.

In Python, si usa Lambda per definire funzioni anonime che

appaiono all'interprete come istruzioni valide. Questo ci permette di inserire segmenti multipli in una singola riga di codice. Immaginatela come una mini-funzione. In questo caso stiamo impostando il nome della funzione di supporto e il valore che si vuole inviarle, oltre al tag event (e:). Parleremo più approfonditamente di Lambda in uno dei prossimi articoli. Per ora limitatevi a seguire l'esempio.

Vi ho fornito i primi quattro pulsanti. Copiate e incollate il codice sopra per i pulsanti da 5 a 9 e per lo 0. Sono uguali ad eccezione del nome e del valore inviato alla funzione callback. I passi seguenti sono mostrati a destra.

Le uniche cose di cui non ci siamo ancora occupati sono columnspan e l'attributo sticky. Come detto prima, un widget può estendersi a più colonne o righe. In questo caso, si "stira" il widget etichetta per quattro colonne. Questo è quello che determina l'attributo "columnspan". Esiste un equivalente "rowspan". L'attributo "sticky" dice al widget dove allineare i suoi margini. Pensate ad esso a come il widget riempie la

```
        self.btnDash = Button(master, text = '-',width = 4,height=3)
        self.btnDash.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('ABS'))
        self.btnDot = Button(master, text = '.',width = 4,height=3)
        self.btnDot.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Dec'))
```

btnDash rende assoluto il valore inserito. 523 resta 523 e -523 diventa 523. Il pulsante btnDot inserisce il punto decimale. Questi esempi, e quelli sotto, usano la funzione di callback funcFuncButton.

```
        self.btnPlus = Button(master,text = '+', width = 4, height=3)
        self.btnPlus.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Add'))
        self.btnMinus = Button(master,text = '-', width = 4, height=3)
        self.btnMinus.bind('<ButtonRelease-1>', lambda e:
self.funcFuncButton('Subtract'))
        self.btnStar = Button(master,text = '*', width = 4, height=3)
        self.btnStar.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Multiply'))
        self.btnDiv = Button(master,text = '/', width = 4, height=3)
        self.btnDiv.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Divide'))
        self.btnEqual = Button(master, text = '=')
        self.btnEqual.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Eq'))
```

Ecco i quattro pulsanti per le funzioni matematiche.

```
        self.btnClear = Button(master, text = 'CLEAR')
        self.btnClear.bind('<ButtonRelease-1>', lambda e: self.funcClear())
```

Per finire, ecco il pulsante clear. Ovviamente, ripulisce le variabili contenitore e lo schermo. Ora posizioniamo i widget nella funzione PlaceWidget. Prima inizializziamo la griglia, quindi mettiamo i widget in essa. Ecco la prima parte della routine.

```
    def PlaceWidgets(self,master):
        master.grid(column=0,row=0)
        self.lblDisplay.grid(column=0,row=0,columnspan = 4,sticky=EW)
        self.btn1.grid(column = 0, row = 1)
        self.btn2.grid(column = 1, row = 1)
        self.btn3.grid(column = 2, row = 1)
        self.btn4.grid(column = 0, row = 2)
        self.btn5.grid(column = 1, row = 2)
        self.btn6.grid(column = 2, row = 2)
        self.btn7.grid(column = 0, row = 3)
        self.btn8.grid(column = 1, row = 3)
        self.btn9.grid(column = 2, row = 3)
        self.btn0.grid(column = 1, row = 4)
```

griglia. In alto a sinistra c'è il resto dei pulsanti.

Prima di procedere ulteriormente diamo un'occhiata a come funziona il tutto quando l'utente preme i pulsanti.

Diciamo che l'utente vuole inserire 563 + 127 ed ottenere la risposta. Lui premerà o farà clic (logicamente) su 5, 6, 3, quindi "+" e 1, 2, 7, e infine il pulsante "=". Come lo traduciamo in codice? Abbiamo già impostato la funzione di supporto per i pulsanti dei numeri a funcNumButton. Ci sono due modi di procedere. Possiamo mantenere l'informazione inserita come stringa e quando ne abbiamo bisogno convertirla in un numero, o conservarla tutto il tempo sottoforma numerica. Useremo la seconda strada. Per farlo, conserveremo il valore inserito (0 quando si inizia) in una variabile chiamata "self.CurrentValue". Quando viene inserito un numero, si prende la variabile, la si moltiplica per 10 e si aggiunge il nuovo valore. Così, quando l'utente inserisce 5, 6 e 3, facciamo quanto segue...

```
L'utente fa clic su 5 — 0 * 10 + 5 (5)
```

```
self.btnDash.grid(column = 0, row = 4)
self.btnDot.grid(column = 2, row = 4)
self.btnPlus.grid(column = 3,row = 1)
self.btnMinus.grid(column = 3, row = 2)
self.btnStar.grid(column = 3, row = 3)
self.btnDiv.grid(column=3, row = 4)
self.btnEqual.grid(column=0,row=5,columnspan = 4,sticky=NSEW)
self.btnClear.grid(column=0,row=6,columnspan = 4, sticky = NSEW)
```

```
def funcNumButton(self,val):
    if self.DecimalNext == True:
        self.DecimalCount += 1
        self.CurrentValue = self.CurrentValue + (val * (10**-self.DecimalCount))
    else:
        self.CurrentValue = (self.CurrentValue * 10) + val
    self.DisplayIt()
```

```
L'utente fa clic su 6 — 5 * 10 + 6 (56)

L'utente fa clic su 3 — 56 * 10 + 3 (563)
```

Ovviamente visualizziamo il contenuto di "self.CurrentValue" nell'etichetta.

Quindi l'utente fa clic su "+". Prendiamo il valore in "self.CurrentValue" e lo mettiamo in "self.HolderValue" e azzeriamo "self.CurrentValue". Quindi ripetiamo il procedimento per 1, 2 e 7. Quando l'utente fa clic su "=" allora sommiamo i valori di "self.CurrentValue" e "self.HolderValue", lo mostriamo e poi ripuliamo entrambe le variabili per continuare.

In alto c'è il codice per iniziare a definire le funzioni di supporto.

La routine "funcNumButton" riceve il valore che le abbiamo passato alla pressione del pulsante. L'unica differenza dall'esempio sopra è quello che accade se l'utente preme il pulsante col punto decimale ("."). In basso, vedrete che usiamo una variabile booleana per ricordare questa occorrenza e, al clic successivo, lo gestiamo. Questo è il significato della riga "if self.DecimalNext == True:". Analizziamola.

L'utente fa clic su 3, poi 2, quindi il punto decimale, poi 4 per creare "32.4". Gestiamo i clic su 3 e 2 con la funzione "funcNumButton". Controlliamo self.DecimalNext per vedere se il valore è vero (che in questo caso non lo è finche non si fa clic sul pulsante "."). In caso contrario, moltiplichiamo il valore conservato (self.CurrentValue) per 10 e aggiungiamo il valore in arrivo. Quando l'utente fa clic su ".", la funzione di callback "funcFuncButton" è chiamata con il valore "Dec". Tutto quello che facciamo è impostare la variabile booleana "self.DecimalNext" a vero. Quando l'utente fa clic su 4, testeremo il valore di "self.DecimalNext" e, poiché è vero, faremo una magia. Prima

incrementiamo la variabile self.DecimalCount. Questa ci dice con quanti posti decimali stiamo lavorando. Quindi prendiamo il valore in arrivo, lo moltiplichiamo per (10**-self.DecimalCount). Questo operatore magico ci fornisce la funzione “eleva alla potenza di”. Per esempio 10**2 restituisce 100. 10**-2 restituisce 0.01. Potrebbe accadere che usando questa funzione si incappi in problemi di arrotondamento, ma, nel caso della nostra semplice calcolatrice, funzionerà per la maggior parte dei numeri decimali. Lascio a voi il compito di trovare una funzione migliore. Prendetelo come il compito a casa per questo mese.

La funzione “funcClear” azzera semplicemente le due variabili contenitore, quindi imposta la visualizzazione.

```
def funcClear(self):

self.CurrentValue = 0

self.HolderValue = 0

self.DisplayIt()
```

Ora le funzioni. Abbiamo già parlato di cosa accade con la funzione “Dec”. La impostiamo

```
    def funcFuncButton(self,function):
      if function =='Dec':
          self.DecimalNext = True
      else:
            self.DecimalNext = False
            self.DecimalCount = 0
            if function == 'ABS':
                self.CurrentValue *= -1
                self.DisplayIt()
```

La funzione ABS prende semplicemente il valore corrente e lo moltiplica per -1.

```
            elif function == 'Add':
                self.HolderValue = self.CurrentValue
                self.CurrentValue = 0
                self.CurrentFunction = 'Add'
```

La funzione Add copia “self.CurrentValue” in “self.HolderValue”, pulisce “self.CurrentValue” e imposta “self.CurrentFunction” su “Add”. Le funzioni sottrazione, moltiplicazione e divisione fanno la stessa cosa inserendo il termine appropriato in “self.CurrentFunction”.

```
            elif function == 'Subtract':
                self.HolderValue = self.CurrentValue
                self.CurrentValue = 0
                self.CurrentFunction = 'Subtract'
            elif function == 'Multiply':
                self.HolderValue = self.CurrentValue
                self.CurrentValue = 0
                self.CurrentFunction = 'Multiply'
            elif function == 'Divide':
                self.HolderValue = self.CurrentValue
                self.CurrentValue = 0
                self.CurrentFunction = 'Divide'
```

La funzione “Eq” (uguale) è dove avviene la magia. Sara semplice per voi capire ora il codice seguente.

```
            elif function == 'Eq':
                if self.CurrentFunction == 'Add':
                    self.CurrentValue += self.HolderValue
                 elif self.CurrentFunction == 'Subtract':
                    self.CurrentValue = self.HolderValue - self.CurrentValue
                 elif self.CurrentFunction == 'Multiply':
                    self.CurrentValue *= self.HolderValue
                 elif self.CurrentFunction == 'Divide':
                    self.CurrentValue = self.HolderValue / self.CurrentValue
                 self.DisplayIt()
                 self.CurrentValue = 0
                 self.HolderValue = 0
```

prima con l'istruzione "if". Ci spostiamo all'"else" e, se la funzione è qualcos'altro, ripuliamo le variabili "self.DecimalNext" e "self.DecimalCount".

Il seguente insieme di passi sono mostrati nella pagina seguente (il box a destra).

La funzione DisplayIt semplicemente imposta il valore nell'etichetta display. Ricordate quando dicemmo alla etichetta di "monitorare" la variabile "self.CurrentDisplay". Quando cambia, l'etichetta automaticamente cambia la visualizzazione per corrisponderle. Usiamo il metodo ".set" per cambiare il valore.

```
def DisplayIt(self):

print('CurrentValue = {0} -
HolderValue =
{1}'.format(self.CurrentValue
,self.HolderValue))

self.CurrentDisplay.set(self.
CurrentValue)
```

Finalmente abbiamo le righe di avvio.

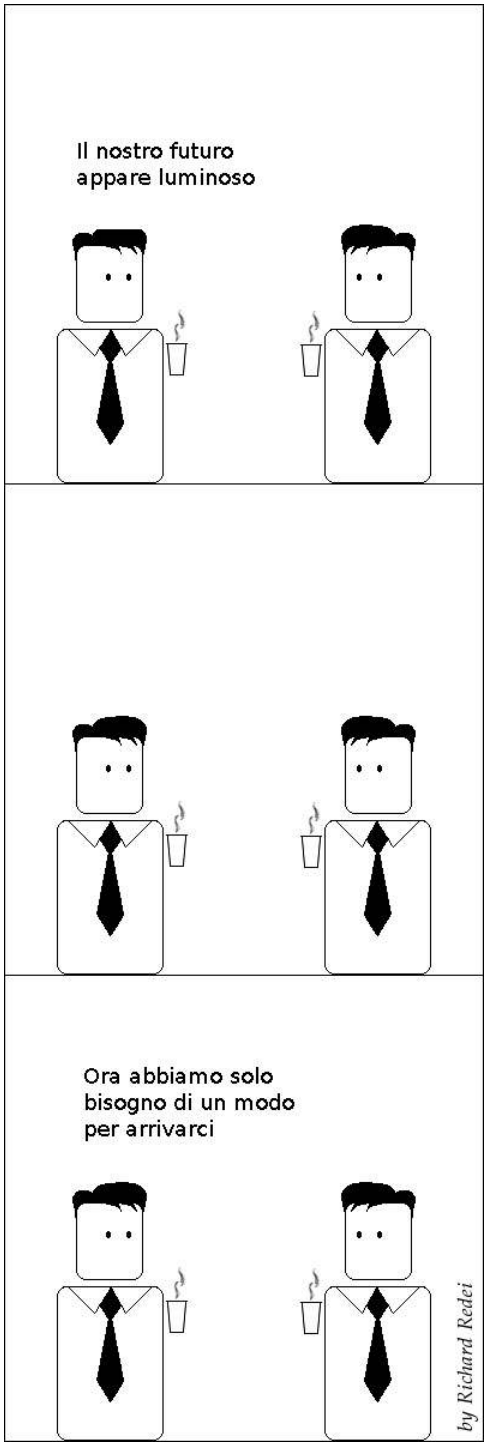
```
if __name__ == '__main__':

StartUp()
```

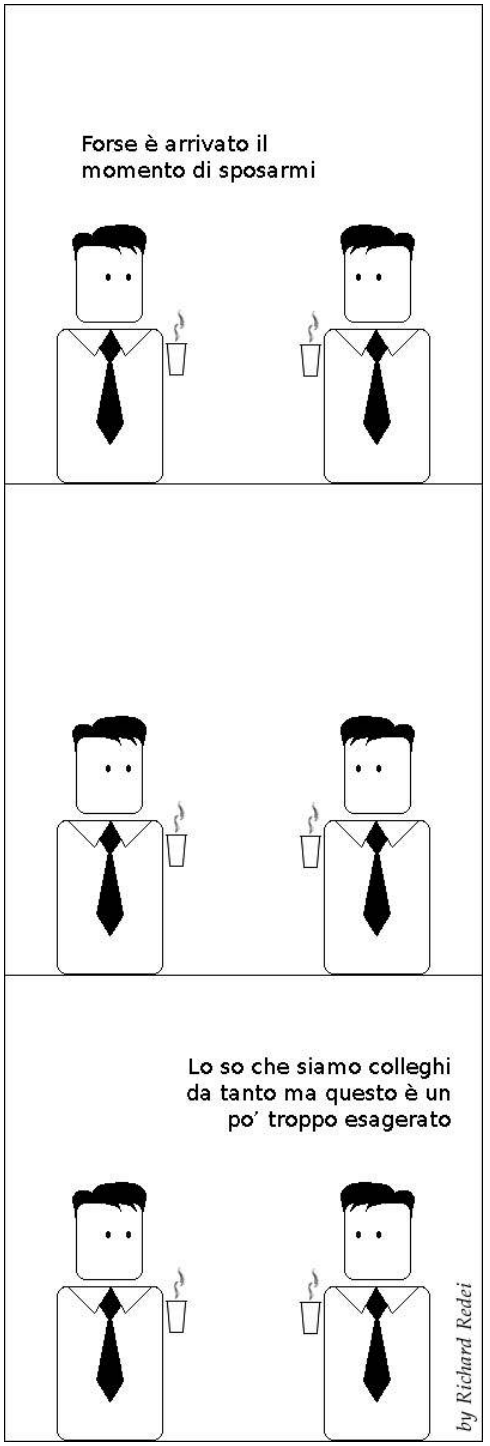
Ora possiamo eseguire il programma e provarlo.

Come sempre, il codice dell'articolo può essere trovato su PasteBin. Gli esempi 1, 2 e 3 sono all'indirizzo http://pastebin.com/mBAS1Umm e l'esempio Calc.py si trova all'indirizzo http://pastebin.com/LbMibF0u

Il mese prossimo continueremo con Tkinter e i suoi tanti widget. In un articolo futuro parleremo del disegnatore di GUI per Tkinter chiamato PAGE. Nel frattempo, divertitevi. Penso che gradirete Tkinter.

Greg Walters è il proprietario della RainyDay Solutions, LLC, una società di consulenza in Aurora, Colorado e programma dal 1972. Ama cucinare, fare escursioni, ascoltare musica e passare il tempo con la sua famiglia. Il suo sito web è: www.thedesignatedgeek.com.





Programmare in Python - Parte 26

Scritto da Greg Walters

Il mese scorso abbiamo parlato di tkInter e di quattro dei suoi widget: TopLevel, cornici, pulsanti ed etichette. Sempre il mese scorso vi dissi che avremmo discusso di come usare altri widget, oltre che TopLevel, come genitore.

Così, questo mese, oltre che approfondire cornici, pulsanti ed etichette introdurremo pulsanti a selezione multipla (checkbox) e singola (radiobutton), campi di testo (textbox, widget di inserimento), caselle di riepilogo (listbox) con barra di scorrimento verticale e messaggi (messagebox). Prima di iniziare, esaminiamo qualcuno di questi widget.

I pulsanti a selezione multipla hanno due opzioni, selezionato e non selezionato, anche identificabili con gli stati on e off. Sono solitamente usati per fornire una serie di opzioni dove è possibile selezionarne qualcuna, molte o tutte. È possibile impostare un evento che informi quando un controllo a selezione multipla è selezionato o che valuti il valore del widget in qualunque momento.

I pulsanti a selezione singola hanno anch'essi due possibilità, on e off. Però sono raggruppati per fornire una serie di opzioni tra cui sceglierne logicamente una sola. È possibile avere più gruppi di pulsanti a selezione singola che, se propriamente programmati, non interagiranno tra loro.

La casella di riepilogo fornisce all'utente una lista di elementi tra cui scegliere. Il più delle volte si vuole che l'utente possa selezionare un singolo elemento, ma ci sono occasioni in cui sono consentite selezioni multiple. Può anche essere presente una barra di scorrimento orizzontale o verticale per permettere all'utente una esplorazione più semplice.

Il nostro progetto consisterà in una finestra principale e sette cornici che raggrupperanno visivamente i nostri widget:

- La prima cornice sarà molto semplice. Consisterà di varie etichette mostranti le differenti opzioni.
- La seconda, ancora semplice, conterrà i pulsanti relativi alle differenti opzioni.
- In questa cornice avremo 2 pulsanti a selezione multipla ed un pulsante per cambiare il loro stato e ogni variazione sarà trasmessa alla finestra terminale.
- A seguire avremo due gruppi di tre pulsanti a selezione singola, ciascuno dei quali invierà un messaggio alla finestra terminale quando selezionato. I due gruppi sono separati.
- Questo ha qualche campo di testo o di inserimento, che non vi sono nuovi, ma c'è anche un pulsante per abilitare o disabilitare uno dei due. Quando disabilitato nel campo di testo non è possibile inserire nulla.
- Questa è una casella di riepilogo con una barra di scorrimento verticale che invia un messaggio al terminale ogni volta si seleziona un elemento, e contiene due pulsanti. Uno per ripulire la casella e l'altro per inserire valori fittizzi.
- La cornice finale conterrà una serie di pulsanti che richiameranno vari messaggi.

```
# widgetdemo1.py
# Labels
from Tkinter import *

class Demo:
    def __init__(self,master):
        self.DefineVars()
        f = self.BuildWidgets(master)
        self.PlaceWidgets(f)
```

Quindi ora diamo inizio al progetto. Chiamiamolo "widgetdemo1.py". Assicuratevi di salvarlo perché lo scriveremo a piccoli blocchi fino all'applicazione finale. Ciascun blocco riguarda una cornice. Noterete che man mano inserirò dei commenti a cui potrete far riferimento per capire cosa abbiamo fatto. In alto ci sono le righe iniziali.

Le prime due (commenti) contengono il nome dell'applicazione e l'oggetto del blocco. La riga tre contiene l'istruzione import. Quindi definiamo la nostra classe. La riga successiva dà inizio alla routine __init__ che ormai dovreste ben conoscere ma, se vi siete appena uniti a noi, è il codice eseguito quando si istanzia la funzione nel main del programma. Le passiamo la finestra

Toplevel o radice, che sarà quella principale. Le ultime, per il momento, tre righe chiamano tre funzioni differenti. La prima, DefineVars, configurerà varie variabili che ci serviranno via via. La seguente, BuildWidgets, sarà quella che conterrà le definizioni dei nostri widget, e l'ultima, PlaceWidgets, sarà usata per posizionare i controlli nella finestra principale. Come abbiamo fatto l'ultima volta, useremo la gestione geometrica a griglia. Notate che BuildWidgets restituirà l'oggetto "f" (la nostra finestra principale) e lo passeremo alla funzione PlaceWidgets.

```
def BuildWidgets(self,master):
    # Define our widgets
    frame = Frame(master)
    # Labels
    self.lblframe = Frame(frame,relief = SUNKEN,padx = 3, pady = 3,
                          borderwidth = 2, width = 500)
    self.lbl1 = Label(self.lblframe,text="Flat Label",relief = FLAT,
                      width = 13,borderwidth = 2)
    self.lbl2 = Label(self.lblframe,text="Sunken Label", relief = SUNKEN,
                      width = 13, borderwidth = 2)
    self.lbl3 = Label(self.lblframe,text="Ridge Label", relief = RIDGE, width = 13,
                      borderwidth = 2)
    self.lbl4 = Label(self.lblframe,text="Raised Label", relief = RAISED,
                      width = 13, borderwidth = 2)
    self.lbl5 = Label(self.lblframe,text="Groove Label", relief = GROOVE,
                      width = 13, borderwidth = 2)
    return frame
```

In alto a destra c'è la funzione BuildWidgets. Le righe che iniziano con "self." sono state divise per due motivi. Il primo, è buona abitudine contenere la lunghezza delle righe negli 80 caratteri o meno. Il secondo, è più semplice nel nostro meraviglioso editor. Potete fare due cose. La prima, creare righe lunghe o tenerle come sono. Python consente la suddivisione delle righe fintantoché sono racchiuse tra parentesi tonde o graffe. Come detto prima, definiamo i widget prima di inserirli nella griglia. Noterete con una delle prossime funzioni che è possibile definire un widget contemporaneamente al suo posizionamento nella griglia, ma la definizione prima del posizionamento rende più semplice il controllo, dato che stiamo facendo la maggior parte delle definizioni in questa routine.

Allora, prima definiamo la cornice principale. Qui inseriremo i rimanenti widget. Quindi definiamo una cornice, figlia di quella principale, che conterrà cinque etichette e la chiamiamo lblframe. Qui settiamo i vari attributi del frame. Impostiamo i margini a 'SUNKEN', una distanza di 3 pixel a sinistra e destra (padx) e 3 pixel in alto e in basso (pady). Impostiamo anche borderwidth a 2 pixel così che l'effetto incavato sia più visibile, rispetto al valore predefinito 0. Per finire impostiamo la larghezza totale del frame a 500 pixel.

Quindi definiamo ciascuna etichetta che andremo ad usare. Come genitore useremo self.lblframe e non frame. In questo modo tutte le etichette saranno figlie di lblframe che sarà a sua volta figlia di frame. Notate come ogni definizione sia praticamente la stessa per tutte le cinque etichette ad eccezione del nome del widget (lbl1, lbl2, etc), del testo, del bordo o effetto visivo. Per finire, restituiamo il frame alla funzione chiamante (__init__).

La pagina seguente (in alto a destra) mostra la funzione PlaceWidgets.

Come parametro usiamo l'oggetto cornice chiamato master. Lo assegnamo a frame per adeguarci a quanto già fatto nella funzione BuildWidgets. A seguire impostiamo la griglia principale (frame.grid(column = 0, row = 0)). Se non lo facessimo, nulla funzionerebbe correttamente. Quindi iniziamo a inserire i widget nelle celle della griglia. Prima inseriamo la cornice lblframe che contiene le etichette e configuriamo i suoi attributi. La mettiamo nella colonna 0, riga 1, impostiamo una distanza di 5 pixel su tutti i lati e le facciamo ricoprire 5 colonne (da sinistra a destra) e, per

finire, usiamo l'attributo "sticky" per far espandere il frame a destra e sinistra ("WE", dalle iniziali degli equivalenti in inglese dei nostri ovest ed est). Ora arriva la parte in cui violeremo la regola di cui vi dicevo prima. Stiamo per posizionare una etichetta come primo widget della cornice, ma non lo abbiamo ancora definito. Lo facciamo ora. Impostiamo come genitore lblframe, come per le altre etichette. Il testo a "Labels |", la larghezza a 15 e l'ancoraggio a destra ('e'). Se ricordate, la volta passata usando l'attributo anchor abbiamo indicato dove deve essere posizionato il testo nel widget. In questo caso è lungo il lato destro. Ora la parte divertente. Definiamo la posizione nella griglia, e ogni altro attributo necessario, semplicemente aggiungendo ".grid" in coda della definizione dell'etichetta.

Continuiamo allineando tutte le etichette nella griglia, iniziando dalla colonna 1, riga 0.

Ecco la funzione DefineVars. Notate che per il momento stiamo semplicemente usando l'istruzione pass. La sostituiremo dopo, visto che ora non ci serve:

```
def DefineVars(self):
  # Define our resources
  pass
```

E per finire il codice della funzione main:

```
root = Tk()
root.geometry('750x40+150+150')
root.title("Widget Demo 1")
demo = Demo(root)
root.mainloop()
```

Prima instanziamo Tk. Quindi impostiamo la dimensione della finestra principale a 750 x 40 pixel e la posizioniamo a 150 pixel dal margine superiore e sinistro dello schermo. Quindi il titolo e instanziamo l'oggetto Demo e, per finire, chiamiamo mainloop di Tk.

Provatela. Dovreste vedere le 5 etichette più l'etichetta aggiunta alla fine, in tutta la loro gloria.

Pulsanti

Ora salvate il tutto come widgetdemo1a.py e aggiungiamo qualche pulsante. Poiché il programma base a cui li aggiungeremo è già pronto, aggiungeremo solo le parti mancanti. Iniziamo con la funzione BuildWidgets. Dopo la definizione delle etichette, e prima della riga "return frame", aggiungiamo quanto mostrato nella pagina seguente, in alto a destra.

```
def PlaceWidgets(self, master):
    frame = master
    # Place the widgets
    frame.grid(column = 0, row = 0)
    # Place the labels
    self.lblframe.grid(column = 0, row = 1, padx = 5, pady = 5,
                       columnspan = 5,sticky='WE')
    l = Label(self.lblframe,text='Labels |',width=15,
              anchor='e').grid(column=0,row=0)
    self.lbl1.grid(column = 1, row = 0, padx = 3, pady = 5)
    self.lbl2.grid(column = 2, row = 0, padx = 3, pady = 5)
    self.lbl3.grid(column = 3, row = 0, padx = 3, pady = 5)
    self.lbl4.grid(column = 4, row = 0, padx = 3, pady = 5)
    self.lbl5.grid(column = 5, row = 0, padx = 3, pady = 5)
```

Nulla di nuovo fin qui. Abbiamo definito i pulsanti, con i loro attributi, e settato le funzioni di supporto (callback) attraverso .bind. Notate l'uso di lambda per l'invio dei valori da 1 a 5 in base al pulsante premuto. Questo permetterà di sapere, nella funzione di callback, quale pulsante stiamo trattando. Passiamo alla funzione PlaceWidgets. Inserite il codice sotto dopo aver posizionato l'ultima etichetta.

Ancora una volta, nessuna novità, quindi continuiamo. In basso a destra c'è la funzione callback. Inseritela dopo la routine DefineVars.

Ancora niente di davvero

```
# Place the buttons
self.btnframe.grid(column=0, row = 2, padx = 5,
                   pady = 5, columnspan = 5,sticky = 'WE')
l = Label(self.btnframe,text='Buttons |',width=15,
          anchor='e').grid(column=0,row=0)
self.btn1.grid(column = 1, row = 0, padx = 3, pady = 3)
self.btn2.grid(column = 2, row = 0, padx = 3, pady = 3)
self.btn3.grid(column = 3, row = 0, padx = 3, pady = 3)
self.btn4.grid(column = 4, row = 0, padx = 3, pady = 3)
self.btn5.grid(column = 5, row = 0, padx = 3, pady = 3)
```

divertente. Usiamo solo una serie di istruzioni IF/ELIF per stampare quale pulsante è stato premuto. La cosa principale da osservare, eseguendo il programma, è che il pulsante con l'aspetto incavato non si muove quando facciamo clic. Generalmente non si dovrebbe usare questo effetto a meno che non si crei un pulsante che resti "schiacciato" quando premuto. Per finire, dobbiamo aggiustare la geometria per supportare il widget extra inserito:

```
root.geometry('750x110+150+150')
```

OK. Questo è pronto. Salvate ed eseguitelo.

Salvatelo come widgetdemo1b.py e passiamo ai pulsanti a scelta multipla.

Pulsanti a scelta multipla (checkbox)

Come detto prima, questa parte del programma contiene due pulsanti normali e due pulsanti a scelta multipla. Il primo checkbox ha l'aspetto che ci aspetteremmo. Il secondo somiglia più a un pulsante "appiccicoso": quando non è selezionato, sembra un pulsante normale, quando lo si seleziona, ha l'aspetto di un pulsante costantemente premuto. Questa funzione si abilita impostando l'attributo indicatoron su falso. Ogni pressione del pulsante "normale" cambierà lo stato dei checkbox da selezionato a non selezionato e vice versa. Otteniamo questo chiamando il metodo .toggle del checkbox. Colleghiamo il rilascio del pulsante sinistro del mouse alla funzione che invia, in questo caso, un messaggio al terminale. In aggiunta a tutto questo, configuriamo due variabili (una per ciascun checkbox) che possiamo interrogare in qualunque momento. In questo caso, ogniqualvolta si fa clic su un checkbox si controlla questo valore e lo si stampa. Si faccia attenzione alla porzione di codice relativo alla variabile. Sarà usata per più widget.

Nella funzione BuildWidget, dopo il codice del pulsante appena inserito e prima dell'istruzione return, inserite il codice mostrato nella pagina seguente, in alto a destra.

```
# Buttons
self.btnframe = Frame(frame,relief = SUNKEN,padx = 3, pady = 3,
                      borderwidth = 2, width = 500)
self.btn1 = Button(self.btnframe,text="Flat Button",
                   relief = FLAT, borderwidth = 2)
self.btn2 = Button(self.btnframe,text="Sunken Button",
                   relief = SUNKEN, borderwidth = 2)
self.btn3 = Button(self.btnframe,text="Ridge Button",
                   relief = RIDGE, borderwidth = 2)
self.btn4 = Button(self.btnframe,text="Raised Button",
                   relief = RAISED, borderwidth = 2)
self.btn5 = Button(self.btnframe,text="Groove Button",
                   relief = GROOVE, borderwidth = 2)
self.btn1.bind('<ButtonRelease-1>',lambda e: self.BtnCallback(1))
self.btn2.bind('<ButtonRelease-1>',lambda e: self.BtnCallback(2))
self.btn3.bind('<ButtonRelease-1>',lambda e: self.BtnCallback(3))
self.btn4.bind('<ButtonRelease-1>',lambda e: self.BtnCallback(4))
self.btn5.bind('<ButtonRelease-1>',lambda e: self.BtnCallback(5))
```

```
    def BtnCallback(self,val):
        if val == 1:
            print("Flat Button Clicked...")
        elif val == 2:
            print("Sunken Button Clicked...")
        elif val == 3:
            print("Ridge Button Clicked...")
        elif val == 4:
            print("Raised Button Clicked...")
        elif val == 5:
            print("Groove Button Clicked...")
```

Ancora, tutto questo lo avete già visto. Abbiamo creato la cornice per contenere i controlli. Abbiamo impostato un pulsante e due checkbox. Posizioniamoli ora con il codice della prossima pagina, al centro.

Ora definiamo le due variabili per monitorare il valore di ciascun checkbox. In DefineVars, commentate l'istruzione pass, e aggiungete quanto segue...

```
self.Chk1Val = IntVar()
self.Chk2Val = IntVar()
```

Dopo la funzione di callback del pulsante, inserite il testo mostrato in basso a destra.

E, per finire, sostituite l'istruzione geometry con questa:

```
root.geometry('750x170+150+150')
```

Salvate ed eseguite. Salvate come widgetdemo1c.py e passiamo ai pulsanti a scelta singola.

Pulsanti a scelta singola (radiobutton)

Se siete vecchi abbastanza da ricordare le autoradio con i pulsanti per selezionare le stazioni memorizzate, allora potete capire perché sono chiamati Radiobutton. Quando si usano i pulsanti a scelta singola l'attributo variable è molto importante. È ciò che tiene insieme questi pulsanti. Nel programma, il primo gruppo di pulsanti è raggruppato per mezzo della variabile chiamata self.RBVal. Il secondo è raggruppato attraverso la variabile RBVal2. Dobbiamo anche impostare il valore dell'attributo in fase di definizione. Questo assicura che i pulsanti restituiranno un valore valido quando premuti.

Tornando a BuildWidgets e, giusto prima l'istruzione return, aggiungete il codice mostrato nella pagina seguente.

Una osservazione. Notate la definizione delle etichette nella funzione PlaceWidget. Queste lunghe righe sono spezzettate per mostrare come l'uso delle parentesi ci permetta di ben formattare righe lunghe, con il codice che ancora funziona.

```
# Check Boxes
self.cbframe = Frame(frame, relief = SUNKEN, padx = 3, pady = 3,
                     borderwidth = 2, width = 500)
self.chk1 = Checkbutton(self.cbframe, text = "Normal Checkbox",
                        variable=self.Chk1Val)
self.chk2 = Checkbutton(self.cbframe, text = "Checkbox",
                        variable=self.Chk2Val,indicatoron = False)
self.chk1.bind('<ButtonRelease-1>',lambda e: self.ChkBoxClick(1))
self.chk2.bind('<ButtonRelease-1>',lambda e: self.ChkBoxClick(2))
self.btnToggleCB = Button(self.cbframe,text="Toggle Cbs")
self.btnToggleCB.bind('<ButtonRelease-1>',self.btnToggle)
```

```
# Place the Checkboxes and toggle button
self.cbframe.grid(column = 0, row = 3, padx = 5, pady = 5,
                  columnspan = 5,sticky = 'WE')
l = Label(self.cbframe,text='Check Boxes |',width=15,
          anchor='e').grid(column=0,row=0)
self.btnToggleCB.grid(column = 1, row = 0, padx = 3, pady = 3)
self.chk1.grid(column = 2, row = 0, padx = 3, pady = 3)
self.chk2.grid(column = 3, row = 0, padx = 3, pady = 3)
```

```
def btnToggle(self,p1):
    self.chk1.toggle()
    self.chk2.toggle()
    print("Check box 1 value is {0}".format(self.Chk1Val.get()))
    print("Check box 2 value is {0}".format(self.Chk2Val.get()))

def ChkBoxClick(self,val):
    if val == 1:
        print("Check box 1 value is {0}".format(self.Chk1Val.get()))
    elif val == 2:
        print("Check box 2 value is {0}".format(self.Chk2Val.get()))
```

In DefineVars aggiungete:

```
self.RBVal = IntVar()
```

Aggiungete le funzioni per il clic:

```
def RBClick(self):

    print("Radio Button clicked
- Value is
{0}".format(self.RBVal.get()))

def RBClick2(self):
    print("Radio Button clicked
- Value is
{0}".format(self.RBVal2.get()))
```

e, per finire, rielaborate l'istruzione geometry come segue.

```
root.geometry('750x220+150+150
')
```

Salvate il progetto come widgetdemo1d.py ed eseguitelo. Ora

```
# Radio Buttons
self.rbframe = Frame(frame, relief = SUNKEN, padx = 3, pady = 3, borderwidth = 2, width = 500)
self.rb1 = Radiobutton(self.rbframe, text = "Radio 1", variable = self.RBVal, value = 1)
self.rb2 = Radiobutton(self.rbframe, text = "Radio 2", variable = self.RBVal, value = 2)
self.rb3 = Radiobutton(self.rbframe, text = "Radio 3", variable = self.RBVal, value = 3)
self.rb1.bind('<ButtonRelease-1>',lambda e: self.RBClick())
self.rb2.bind('<ButtonRelease-1>',lambda e: self.RBClick())
self.rb3.bind('<ButtonRelease-1>',lambda e: self.RBClick())
self.rb4 = Radiobutton(self.rbframe, text = "Radio 4", variable = self.RBVal2, value = "1-1")
self.rb5 = Radiobutton(self.rbframe, text = "Radio 5", variable = self.RBVal2, value = "1-2")
self.rb6 = Radiobutton(self.rbframe, text = "Radio 6", variable = self.RBVal2, value = "1-3")
self.rb4.bind('<ButtonRelease-1>',lambda e: self.RBClick2())
self.rb5.bind('<ButtonRelease-1>',lambda e: self.RBClick2())
self.rb6.bind('<ButtonRelease-1>',lambda e: self.RBClick2())
```

In PlaceWidgets aggiungete questo

```
# Place the Radio Buttons and select the first one
self.rbframe.grid(column = 0, row = 4, padx = 5, pady = 5, columnspan = 5,sticky = 'WE')
l = Label(self.rbframe,
    text='Radio Buttons |',
    width=15,anchor='e').grid(column=0,row=0)
self.rb1.grid(column = 2, row = 0, padx = 3, pady = 3, sticky = 'EW')
self.rb2.grid(column = 3, row = 0, padx = 3, pady = 3, sticky = 'WE')
self.rb3.grid(column = 4, row = 0, padx = 3, pady = 3, sticky = 'WE')
self.RBVal.set("1")
l = Label(self.rbframe,text='| Another Set |',
    width = 15,
    anchor = 'e').grid(column = 5, row = 0)
self.rb4.grid(column = 6, row = 0)
self.rb5.grid(column = 7, row = 0)
self.rb6.grid(column = 8, row = 0)
self.RBVal2.set("1-1")
```

inizieremo a lavorare sui campi di testo standard (o widget d'inserimento).

Campi di testo (textbox)

Ancora, li abbiamo usati in varie GUI precedentemente. Comunque questa volta, come detto prima, vi mostrerò come prevenire, disabilitandole, la loro modifica da parte dell'utente. È utile quando si voglia solo mostrare qualcosa e permetterne cambiamenti solo nella modalità "modifica". A questo punto dovreste essere abbastanza sicuri che la prima cosa da fare è aggiungere il codice a destra alla funzione BuildWidget.

Casella di riepilogo (listbox)

Ora dobbiamo occuparci della casella di riepilogo. In BuildWidgets, aggiungete il codice dalla prossima pagina, a destra.

Come al solito, creiamo una cornice. Quindi la barra di scorrimento verticale. Lo facciamo prima di creare la listbox perché dobbiamo referenziare il metodo '.set' della barra di scorrimento. Notate l'attributo 'height = 5'. Questo forza la casella a mostrare 5 elementi alla volta. Nell'istruzione .bind usiamo '<<ListboxSelect>>' come evento. È chiamato evento virtuale, poiché non è un evento "ufficiale".

Ora ci occuperemo del codice aggiuntivo per la funzione PlaceWidgets, mostrato nella pagina seguente, a sinistra.

Messaggi

Questa sezione è semplicemente una serie di normalissimi pulsanti che chiameranno vari tipi di messaggi. Li abbiamo visti precedentemente usando altri strumenti per creare GUI. Ne considereremo solo 5 tipi, ma ce ne sono di più. In questa sezione ci

```
        # Textboxes
        self.tbframe = Frame(frame, relief = SUNKEN, padx = 3, pady =
3, borderwidth = 2, width = 500)
        self.txt1 = Entry(self.tbframe, width = 10)
        self.txt2 = Entry(self.tbframe, disabledbackground="#cccccc",
width = 10)
        self.btnDisable = Button(self.tbframe, text =
"Enable/Disable")
        self.btnDisable.bind('<ButtonRelease-1>',
self.btnDisableClick)
```

A seguire, aggiungete questo codice nella funzione PlaceWidget:

```
        # Place the Textboxes
        self.tbframe.grid(column = 0, row = 5, padx = 5, pady = 5,
columnspan = 5,sticky = 'WE')
        l = Label(self.tbframe,text='Textboxes |',width=15,
anchor='e').grid(column=0,row=0)
        self.txt1.grid(column = 2, row = 0, padx = 3, pady = 3)
        self.txt2.grid(column = 3, row = 0, padx = 3, pady = 3)
        self.btnDisable.grid(column = 1, row = 0, padx = 3, pady = 3)
```

Aggiungete questa riga alla fine della funzione DefineVars:

```
        self.Disabled = False
```

Ora aggiungete la funzione che risponde all'evento clic del pulsante:

```
    def btnDisableClick(self,p1):
        if self.Disabled == False:
            self.Disabled = True
            self.txt2.configure(state='disabled')
        else:
            self.Disabled = False
            self.txt2.configure(state='normal')
```

E per finire, rielaborate l'istruzione geometry:

```
root.geometry('750x270+150+150')
```

Salvate come widgetdemo1d.py ed eseguitelo.

```
        # Place the Listbox and support buttons
        self.lstframe.grid(column = 0, row = 6, padx = 5,
pady = 5, columnspan = 5,sticky = 'WE')
        l = Label(self.lstframe,text='List Box |',width=15,
anchor='e').grid(column=0,row=0,rowspan=2)
        self.lbox.grid(column = 2, row = 0,rowspan=2)
        self.VScroll.grid(column = 3, row = 0,rowspan = 2,
sticky = 'NSW')
        self.btnClearLBox.grid(column = 1, row = 0, padx =
5)
        self.btnFillLBox.grid(column = 1, row = 1, padx =
5)
```

In DefineVars aggiungete questo...

```
        # List for List box items
        self.examples = ['Item One','Item Two','Item
Three','Item Four']
```

E aggiungete le seguenti funzioni di supporto:

```
    def ClearList(self):
        self.lbox.delete(0,END)

    def FillList(self):
        # Note, clear the listbox first...no check is done
        for ex in self.examples:
            self.lbox.insert(END,ex)
        # insert([0,ACTIVE,END],item)

    def LBoxSelect(self,p1):
        print("Listbox Item clicked")
        items = self.lbox.curselection()
        selitem = items[0]
        print("Index of selected item =
{0}".format(selitem))
        print("Text of selected item =
{0}".format(self.lbox.get(selitem)))
```

Per finire, aggiornate la riga geometry.

```
root.geometry('750x370+150+150')
```

Salvate come widgetdemo1e.py ed eseguitelo. Ora faremo l'ultima modifica alla nostra applicazione.

```
      # List Box Stuff
      self.lstframe = Frame(frame,
            relief = SUNKEN,
            padx = 3,
            pady = 3,
            borderwidth = 2,
            width = 500
        )
        # Scrollbar for list box
        self.VScroll = Scrollbar(self.lstframe)
        self.lbox = Listbox(self.lstframe,
            height = 5,
            yscrollcommand = self.VScroll.set)
      # default height is 10

self.lbox.bind('<<ListboxSelect>>',self.LBoxSelect)
        self.VScroll.config(command =
self.lbox.yview)
        self.btnClearLBox = Button(
            self.lstframe,
            text = "Clear List",
            command = self.ClearList,
            width = 11
        )
        self.btnFillLBox = Button(
            self.lstframe,
            text = "Fill List",
            command = self.FillList,
            width = 11
        )
        # <<ListboxSelect>> is virtual event
        # Fill the list box
        self.FillList()
```

occuperemo dei messaggi Informazione, Avviso, Errore, Domanda e Si/No. Sono molto utili quando si voglia trasmettere un'informazione all'utente in maniera palese. Nella funzione BuildWidgets aggiungete il codice mostrato sotto.

Ecco la routine di supporto. Per i primi tre (Informazione, Avviso ed Errore) si chiama semplicemente 'tkMessageBox.showinfo', o qualunque sia necessario, con due parametri. Il primo è il titolo della finestra del messaggio e il secondo è il messaggio stesso da mostrare. L'icona è gestita direttamente da tkinter. Per i messaggi che richiedono una risposta (domanda, si/no), bisogna fornire una variabile che riceve il valore corrispondente al pulsante selezionato. Nel caso del messaggio con domanda, la risposta sarà "si" o "no", nel caso del messaggio si/no la risposta sarà "vero" o "falso".

Per finire, modifichiamo la riga con geometry:

```
root.geometry('750x490+550+150')
```

Salvate come widgetdemo1f.py e giocateci.

Ho inserito il codice di widgetdemo1f.py su pastebin all'indirizzo http://pastebin.com/ZqrgHcdG.

```
def ShowMessageBox(self,which):
    if which == 1:
        tkMessageBox.showinfo('Demo','This is an INFO messagebox')
    elif which == 2:
        tkMessageBox.showwarning('Demo','This is a WARNING messagebox')
    elif which == 3:
        tkMessageBox.showerror('Demo','This is an ERROR messagebox')
    elif which == 4:
        resp = tkMessageBox.askquestion('Demo','This is a QUESTION messagebox?')
        print('{0} was pressed...'.format(resp))
    elif which == 5:
        resp = tkMessageBox.askyesno('Demo','This is a YES/NO messagebox')
        print('{0} was pressed...'.format(resp))
```

```
# Buttons to show message boxes and dialogs
self.mbframe = Frame(frame,relief = SUNKEN,padx = 3, pady = 3, borderwidth = 2)
self.btnMBInfo = Button(self.mbframe,text = "Info")
self.btnMBWarning = Button(self.mbframe,text = "Warning")
self.btnMBError = Button(self.mbframe,text = "Error")
self.btnMBQuestion = Button(self.mbframe,text = "Question")
self.btnMBYesNo = Button(self.mbframe,text = "Yes/No")
self.btnMBInfo.bind('<ButtonRelease-1>', lambda e: self.ShowMessageBox(1))
self.btnMBWarning.bind('<ButtonRelease-1>', lambda e: self.ShowMessageBox(2))
self.btnMBError.bind('<ButtonRelease-1>', lambda e: self.ShowMessageBox(3))
self.btnMBQuestion.bind('<ButtonRelease-1>', lambda e: self.ShowMessageBox(4))
self.btnMBYesNo.bind('<ButtonRelease-1>', lambda e: self.ShowMessageBox(5))
```

Ora aggiungete il codice per la funzione PlaceWidgets:

```
# Messagebox buttons and frame
self.mbframe.grid(column = 0,row = 7, columnspan = 5, padx = 5, sticky = 'WE')
l = Label(self.mbframe,text='Message Boxes |',width=15, anchor='e').grid(column=0,row=0)
self.btnMBInfo.grid(column = 1, row = 0, padx= 3)
self.btnMBWarning.grid(column = 2, row = 0, padx= 3)
self.btnMBError.grid(column = 3, row = 0, padx= 3)
self.btnMBQuestion.grid(column = 4, row = 0, padx= 3)
self.btnMBYesNo.grid(column = 5, row = 0, padx= 3)
```