



This month, we'll explore yet another GUI designer, this time for Tkinter. Many people have an issue with Tkinter because it doesn't offer a built-in designer. While I've shown you how to easily design your applications without a designer, we will examine one now. It's called Page. Basically it's a version of Visual TCL with Python support on top. The current version is 3.2 and can be found at <http://sourceforge.net/projects/page/files/latest/download>.

Prerequisites

You need TCK/TK 8.5.4 or later, Python 2.6 or later, and pyttk - which you can get (if you don't already have it) from <http://pypi.python.org/pypi/pyttk>. You probably have all of these with the possible exception of pyttk.

Installation

You can't really ask for an easier installation routine. Simply unpack

the distribution file into a folder of your choice. Run the script called "configure" from the folder where you just unpacked everything. This will create your launch script called "page" which you use to get everything going. That's it.

Learning Page

When you start Page, you'll get three windows (forms). One is a "launch pad", one is a toolbox, and one shows the Attribute Editor.

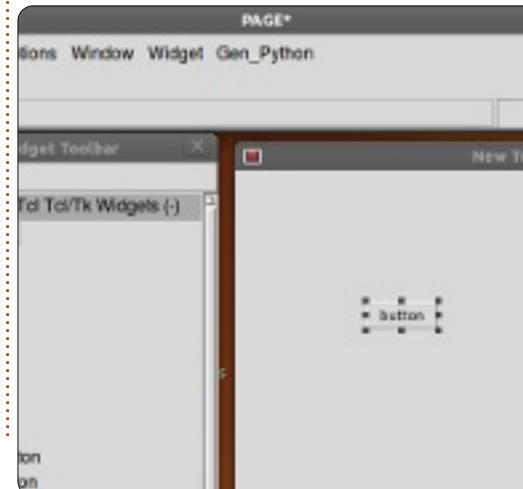
To start a new project, click on the Toplevel button in the toolbox.



This creates your main form. You can move it wherever you wish on your screen. Next, and from now on, click on a widget in the tool box and then click where you want it on the main form.



For now, let's do a button. Click on the Button button on the toolbox, and then click somewhere on the main form.



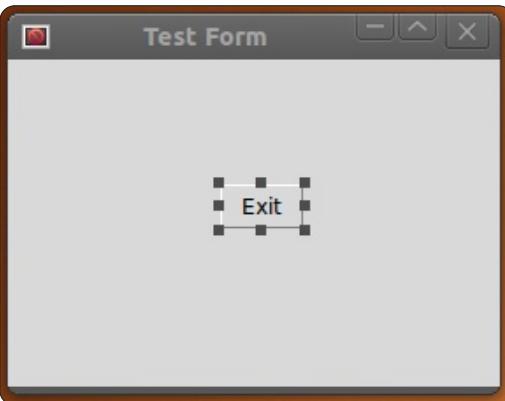
Next, in the launch pad form, click on Window and select Attribute Editor (if it's not already showing). Your single button should be highlighted already, so move it around the form and when you release the mouse button you should see the position change in the attribute editor form under 'x position' and 'y position'.

Here we can set other attributes such as the text on the button (or most any other widget), the alias for the widget (the name we will refer to in our code), color, the name we will call it and more. Near the bottom of the attribute editor is the text field. This is the text that appears to the user for, in this case, the button widget. Let's change this from "button" to "Exit". Notice that now the button says "Exit". Now resize the form to just show the button and recenter the button in the form.

Next click in the main form someplace where the button isn't. The attribute editor form now shows the attributes for the main form. Find the "title" field and

HOWTO - BEGINNING PYTHON 30

change this from "New Toplevel 1" to "Test Form".



Now, before we save our project, we need to create a folder to hold our project files. Create a folder somewhere on your drive called "PageProjects". Now, in the launch pad window, select File then Save As. Navigate to your PageProjects folder, and, in the dialog box, type TestForm.tcl and click the Save button. Notice this is saved as a TCL file, not a Python file. We'll create the python file next.

In the launch pad, find the Gen_Python menu item and click it. Select Generate Python and a new form appears.

Page has generated (as the name suggests) our python code for us and placed it in a window for

us to view. At the bottom of this form, are three buttons...Save, Run, and Close.

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

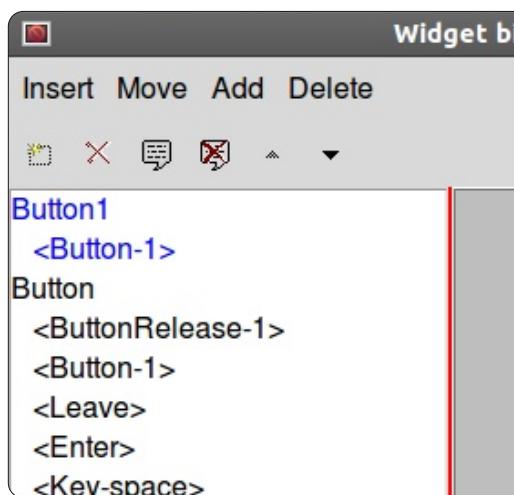
import sys

py2 = py30 = py31 = False
version = sys.hexversion
if version >= 0x020600FD and version <
    py2 = True # Python 2.6 or 2.7
    from Tkinter import *
    import ttk
elif version >= 0x03000000 and version <
    py30 = True
    from tkinter import *
    import ttk
elif version >= 0x03010000:
    py31 = True
    from tkinter import *
    import tkinter.ttk as ttk
else:
    print('***')
```

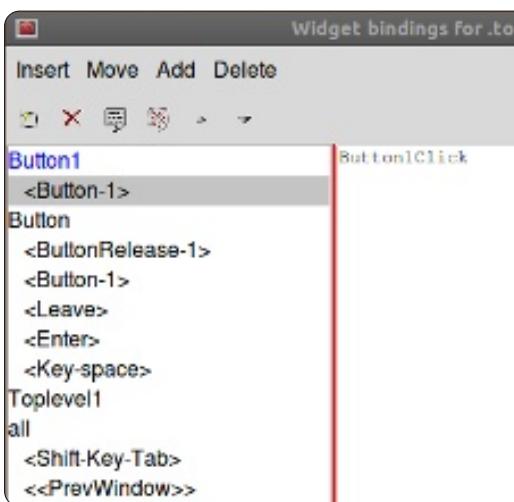
Click Save. If, at this point, you were to look in your PageProjects folder, you will see the python file (TestForm.py). Now click on the Run button. In a few seconds, you'll see the project start up. The button is not connected to anything yet, so it won't do anything if you click on it. Simply close the form with the "X" in the corner of the window. Now close the Python Console window with the close button at the bottom right.

Back at our main form, highlight

the Exit button and right click on it. Select "Bindings...". Under the menu is a set of buttons.

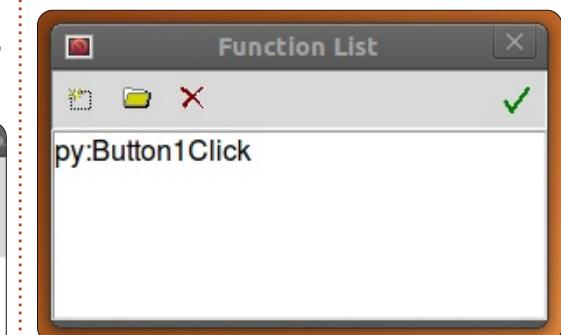


The first on the left allows you to create a new binding. Click on "Button-1". This allows us to enter the binding for the left mouse button. In the window on the right, type "Button1Click".



Save and generate the python code again. Scroll down in the Python Console to the bottom of the file. Above the "class Test_Form" code is the function we just asked to be created. Notice that at this point, it simply is passed. Look further down and you'll see the code that creates and controls our button. Everything is done for us already. However, we still have to tell the button what to do. Close the Python Console and we'll continue.

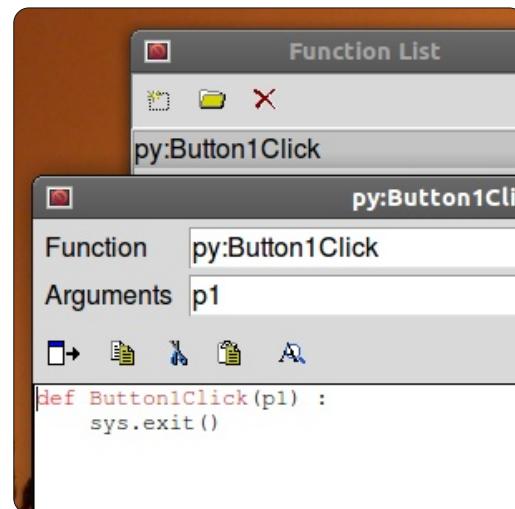
On the launch pad, click Window then select Function List. Here we will write our method to close the window.



The first button on the left is the Add button. Click it. In the Function box, type "py:Button1Click" and, in the Arguments box, type "p1", and

change the text in the lower box to...

```
def Button1Click(p1):  
    sys.exit()
```



Click on the checkmark and we are done with this.

Next we have to bind this routine to the button. Select the button in the form, right click it, and select "Bindings...". As before, click on the far left button on the toolbar and select Button-1. This is the event for the left mouse button click. In the right text box, enter "Button1Click". Make sure you use the same case that you did for the Function we just created. Click the checkmark on the right side.

Now save and generate your

python code.

You should see the following code near the bottom, but OUTSIDE of the Test_Form class...

```
def Button1Click(p1) :
```

```
    sys.exit()
```

And the last line of the class should be...

```
self.Button1.bind( '<Button-1>', Button1Click)
```

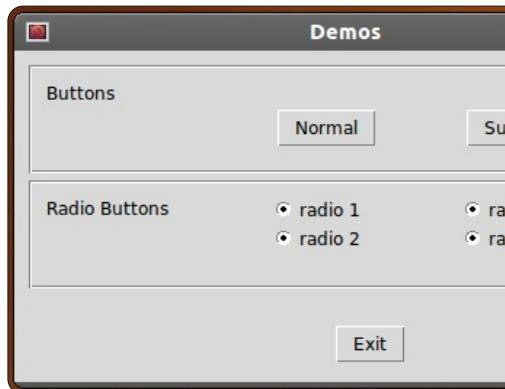
Now, if you run your code and click on the Exit button, the form should close properly.

Moving Forward

Now let's do something more complicated. We'll create a demo showing some of the widgets that are available. First close Page and restart it. Next, create a new Toplevel form. Add two frames, one above the other and expand them to pretty much take up the entire width of the form. In the top frame, place a label, and, using the attributes editor, change the text to "Buttons:". Next, add two buttons along the horizontal plane. Change the text of the left one to

"Normal", and the right one to "Sunken". While the sunken button is selected, change the relief to "sunken" and name it btnSunken. Name the "Normal" button "btnNormal". Save this project as "Demos.tcl".

Next, place in the lower frame a label saying "Radio Buttons" and four radio buttons like in the image below. Finally, place an Exit button below the bottom frame.



Before we work on the bindings, let's create our click functions. Open the Function List and create two functions. The first should be called btnNormalClicked and the other btnSunkenClicked. Make sure you set the arguments box to include p1. Here's the code you should have for them...

```
def btnNormalClicked(p1) :
```

```
print "Normal Button Clicked"  
  
def btnSunkenClicked(p1) :  
  
print "Sunken Button Clicked"
```

Let's add our button bindings. For each button, right click it, select "Bindings...", and add, as before, a binding to the functions we created. For the normal button, it would be "btnNormalClicked", and for the sunken button it would be btnSunkenClicked. Save and generate your code. Now, if you were to test the program under the "Run" option of the Python Console, and click any of the buttons, you won't see anything happen. However, when you close the application, you should see the print responses. This is normal for Page and if you simply run it from the command line as you normally do, things should work as expected.

Now for our radio buttons. We have grouped them in two "clusters". The first two (Radio 1 and Radio 2) will be cluster 1 and the other two will be cluster 2. Click on Radio1 and in the Attribute Editor, set the value to 0 and the variable to "rbc1". Set the variable for Radio 2 to "rbc1" and the value to 1. Do the same thing

for Radio 3 and Radio 4 but for both of these set the variable to "rbc2". If you want, you can deal with the click of the radiobuttons and print something to the terminal, but for now, the important thing is that the clusters work. Clicking Radio1 will deselect Radio2 and not influence Radio3 or Radio4, and the same for Radio2 and so on.

Finally, you should create a function for the Exit button, and bind it to the button like we did in the first example.

If you've been following along as we have done our other Tkinter applications, you should be able to understand the code shown above right. If not, please go back a few issues for a full discussion of this code.

You can see that using Page makes the basic design process much easier than doing it yourself. We've only scratched the surface of what Page can do, and we'll start doing something much more realistic next time.

The python code can be found on pastebin at
<http://pastebin.com/qq0YVgTb>.

```
def set_Tk_var():
    # These are Tk variables passed to Tkinter and must
    # be defined before the widgets using them are created.
    global rbc1
    rbc1 = StringVar()
    global rbc2
    rbc2 = StringVar()
    def btnExitClicked(p1) :
        sys.exit()
    def btnNormalClicked(p1) :
        print "Normal Button Clicked"
    def btnSunkenClicked(p1) :
        print "Sunken Button Clicked"
```

One note before we go for this month. You might have noticed that I've missed a couple of issues. This is due to my wife being diagnosed with cancer last year. As hard as I have tried to keep things from falling through the cracks, a number of things have. One of these things is my old domain/web site at

www.thedesignatedgeek.com. I blew it and missed the renewal. Due to this, the domain was sold out from under me. I have set up www.thedesignatedgeek.net with all the old stuff. I will be working hard the next month to bring it all up to date.

See you next time.



Greg Walters is owner of RainyDay Solutions, LLC, a consulting company in Colorado and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family. His website is www.thedesignatedgeek.net.



The Ubuntu Podcast covers all the latest news and issues facing Ubuntu Linux users and Free Software fans in general. The show appeals to the newest user and the oldest coder. Our discussions cover the development of Ubuntu but aren't overly technical. We are lucky enough to have some great guests on the show, telling us first hand about the latest exciting developments they are working on, in a way that we can all understand! We also talk about the Ubuntu community and what it gets up to.

The show is presented by members of the UK's Ubuntu Linux community. Because it is covered by the Ubuntu Code of Conduct it is suitable for all.

The show is broadcast live every fortnight on a Tuesday evening (British time) and is available for download the following day.

podcast.ubuntu-uk.org



HOW-TO

Written by Greg D. Walters

After our last meeting you should have a fairly good idea of how to use Page. If not, please read last month's article. We'll continue this time by creating a file list application with a GUI. The goal here is to create a GUI application that will recursively walk through a directory, looking for files with a defined set of extensions, and display the output in a treeview. For this example we will look for media files with the extensions of ".avi", ".mkv", ".mv4", ".mp3" and ".ogg".

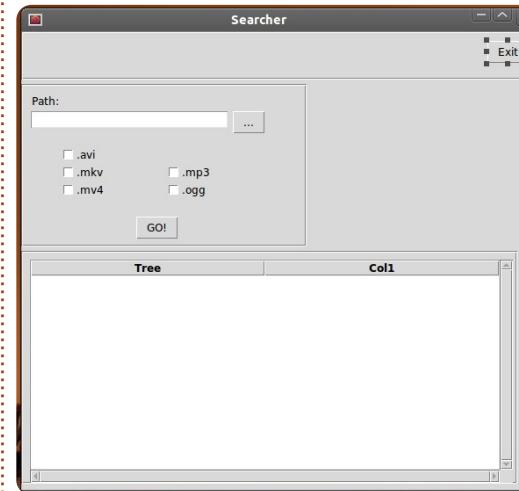
This time, the text might seem a bit terse in the design portion. All I'm going to do is give you directions for placement of widgets and the required attributes and values like this...

Widget

Attribute: Value

I will only quote text string when it is needed. For example for one of the buttons, the text should be set to "...".

Here's what the GUI of our application will look like...



As you can see, we have our main form, an exit button, a text entry box with a button that will call up an ask for directory dialog box, 5 check boxes for extension selecting extension types, a "GO!" button to actually start the processing and a treeview to display our output.

So, let's get started. Fire up Page and create a new top level widget. Using the Attribute Editor set the following attributes.

Alias: Searcher
Title: Searcher

Beginning Python - Part 31

Be sure to save often. When you save the file, save it as "Searcher". Remember, Page puts the .tcl extension for you and when you finally generate the python code, it will be saved in the same folder.

Next add a frame. It should go at the very top of the main frame. Set the attributes as follows.

Width: 595
Height: 55
x position: 0
y position: 0

In this frame, add a button. This will be our Exit button.

Alias: btnExit
Text: Exit

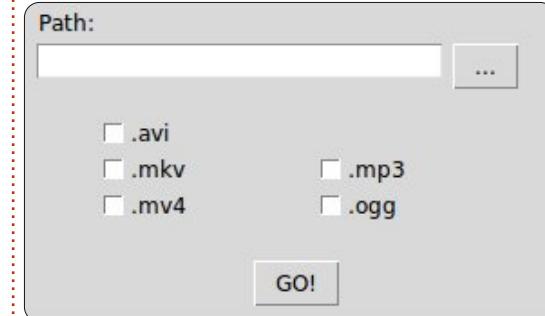
Move this close to the center of the frame or close to the frame's right side. I set mine to X 530 and Y 10.

Create another frame.

Width: 325
Height: 185
y position: 60

Here is what this frame will look

like, to give you a guide going forward through this section.



In this frame, add a label. Set the text attribute to "Path:". Move it close to the top left of the frame.

In the same frame, add an entry widget.

Alias: txtPath
Text: FilePath
Width: 266
Height: 21

Add a button to the right of the entry widget.

Alias: btnSearchPath
Text: "..." (no quotes)

Add five (5) check buttons. Put them in the following order...

x
x
x

The three check buttons on the left are for video files and the two on the right are for audio files. We will deal with the three on the left first, then the two on the right.

Alias: chkAVI
Text: ".avi" (no quotes)
Variable: VchkAVI

Alias: chkMKV
Text: ".mkv" (no quotes)
Variable: VchkMKV

Alias: chkMV4
Text: ".mv4" (no quotes)
Variable: VchkMV4

Alias: chkMP3
Text: ".mp3" (no quotes)
Variable: VchkMP3

Alias: chkOGG
Text: ".ogg" (no quotes)
Variable: VchkOGG

Finally, in this frame add a button somewhere below the five check boxes and somewhat centered within the frame.

Alias: btnGo
Text: GO!

Now add one more frame below our last frame.

Width: 565
Height: 265

I placed mine around X 0 Y 250. You might have to resize your main form to have the entire frame show. Within this frame, add a Scrolledtreeview widget.

Width: 550
Height: 254
X Position: 10
Y Position: 10

There. We've designed our GUI. Now all that is left to do is create our function list and bind the functions to our buttons.

In the Function list window, click the New button (the far left button). This brings up the new function editor. Change the text in the Function entry box from "py:xxx" to "py:btnExitClick()". In the arguments entry box type "p1". In the bottom multiline entry box, change the text to:

```
def btnExitClick(p1):  
    sys.exit()
```

Notice that this is not indented. Page will do that for us when it creates the python file.

Next create another function called btnGoClick. Remember to add a passed parameter of "p1". Leave the "pass" statement. We'll change that later.

Finally, add another function called "btnSearchPath". Again, leave the pass statement.

Lastly, we need to bind the buttons to the functions we just created.

Right-click on the exit button we created, select Bind. A large box will pop up. Click on the New binding button, Click on Button-1 and change the word "TODO" in the right text entry box to "btnExitClick". Do NOT include the parens () here.

Bind the GO button to btnGoClick and the "..." button to btnSearchPathClick.

Save your GUI and generate the python code.

Now all we have left is to create the code that "glues" the GUI together.

Open up the code we just

generated in your favorite editor. Let's start off by examining what Page created for us.

At the top of the file is our standard python header and a single import statement to import the sys library. Next is some rather confusing (at first glance) code. This basically looks at the version of python you are trying to run the application in and then to import the correct versions of the tkinter libraries. Unless you are using python 3.x, you can basically ignore the last two.

We'll be modifying the 2.x code portion to import other tkinter modules in a few moments.

Next is the "vp_start_gui()" routine. This is the program's main routine. This sets up our gui, sets the variables we need, and then calls the tkinter main loop. You might notice the line "w = None" below this. It is not indented and it isn't supposed to be.

Next are two routines (create_Searcher and destroy_Searcher) that are used to replace the main loop routine if we are calling this application as a

library. We don't need to worry about these.

Next is the "set_Tk_var" routine. We define the tkinter variables used that need to be set up before we create the widgets. You might recognize these as the text variable for the FilePath entry widget and the variables for our check boxes. The next three routines here are the functions we created using the function editor and an "init()" function.

Run the program now. Notice that the check buttons have grayed out checks in them. We don't want that in our "release" app, so we'll create some code to clear them before the form is displayed to the user. The only functioning thing other than the check boxes is the Exit button.

Go ahead and end the program.

Now, we'll take a look at the class that actually holds the GUI definition. That would be "class Searcher". Here is where all the widgets are defined and placed in our form. You should be familiar with this by now.

Two more classes are created for us that hold the code to support the scrolled tree view. We don't have to change any of this. It was all created by Page for us.

Now let's go back to the top of the code and start modifying.

We need to import a few more library modules, so under the "import sys" statement, add...

```
import os  
  
from os.path import join,  
    getsize, exists
```

Now find the section that has the line "py2 = True". As we said before, this is the section that deals with the tkinter imports for Python version 2.x. Below the "import ttk", we need to add the following to support the FileDialog library. We also need to import the tkFont module.

```
import tkFileDialog  
  
import tkFont
```

Next we need to add some variables to the "set_Tk_var()" routine. At the bottom of the routine, add the following lines...

```
global exts, fileList
```

```
exts = []
```

```
fileList=[]
```

Here we create two global variables (exts and fileList) that will be accessed later on in our code. Both are lists. "exts" is a list of the extensions that the user selects from the GUI. "FileList" holds a list of lists of the matching files found when we do our search. We'll use that to populate the treeview widget.

Since our "btnExitClick" is already done for us by Page, we'll deal with the "btnGoClick" routine. Comment out the pass statement and add the code so it looks like this...

```
def btnGoClick(p1) :  
  
    #pass  
  
    BuildExts()  
  
    fp = FilePath.get()  
  
    e1 = tuple(exts)  
  
    Walkit(fp,e1)
```

LoadDataGrid()

This is the routine that will be called when the user clicks the "GO!" button. We call a routine called "BuildExts" which creates the list of the extensions that the user has selected. Then we get the path that the user has selected from the AskDirectory dialog and assign that to the fp variable. We then create a tuple from the extension list, which is needed when we check for files. We then call a routine called "Walkit", passing the target directory and the extension tuple.

Finally we call a routine called "LoadDataGrid".

Next we need to flesh out the "btnSearchPathClick" routine. Comment out the pass statement and change the code to look like this...

```
def btnSearchPathClick(p1) :  
  
    #pass  
  
    path =  
        tkFileDialog.askdirectory()  
    #**self.file_opt)  
  
    FilePath.set(path)
```

HOWTO - BEGINNING PYTHON 31

The init routine is next. Again, make the code look like this...

```
def init():

    #pass

    # Fires AFTER Widgets
    # and Window are created...

    global treeview

    BlankChecks()

    treeview =
    w.Scrolledtreeview1

    SetupTreeview()
```

Here we create a global called "treeview". We then call a routine that will clear the gray checks from the check boxes, assign the "treeview" variable to point to the Scrolled treeview in our form and call "SetupTreeview" to set the headers for the columns.

Here's the code for the BlankChecks routine which needs to be next.

```
def BlankChecks():

    VchkAVI.set('0')

    VchkMKV.set('0')

    VchkMP3.set('0')
```

```
VchkMV4.set('0')

VchkOGG.set('0')
```

Here, all we are doing is setting the variables (which automatically sets the check state in our check boxes) to "0". If you remember, whenever the check box is clicked, this variable is automatically updated. If the variable is changed by our code, the check box responds as well. Now (above right) we'll deal with the routine that builds the list of extensions from what the user has clicked.

Cast your memory back to my ninth article in FCM#35. We wrote some code to create a catalog of MP3 files. We'll use a shortened version of that routine (middle right). Refer back to FCM#35 if you have questions about this routine.

Next (bottom right) we call the SetupTreeview routine. It's fairly straightforward. We define a variable "ColHeads" with the headings we want in each column of the treeview. We

do this as a list. We then set the heading attribute for each column. We also set the column width to the size of this header.

Finally we have to create the "LoadDataGrid" routine (next page, top right) which is where we load our data into the treeview. Each row of the treeview is one entry in the FileList list variable. We also adjust the width of each column (again) to match the size of the column data.

That's it for the first blush of

```
def Walkit(musicpath,extensions):
    rcntr = 0
    fl = []
    for root, dirs, files in os.walk(musicpath):
        rcntr += 1 # This is the number of folders we have walked
        for file in [f for f in files if f.endswith(extensions)]:
            fl.append(file)
            fl.append(root)
            fileList.append(fl)
    fl=[]
```

```
def SetupTreeview():
    global ColHeads
    ColHeads = ['Filename','Path']
    treeview.configure(columns=ColHeads,show="headings")
    for col in ColHeads:
        treeview.heading(col, text = col.title(),
                        command = lambda c = col: sortby(treeview, c, 0))
    ## adjust the column's width to the header string
    treeview.column(col, width =
tkFont.Font().measure(col.title()))
```

```
def BuildExts():
    if VchkAVI.get() == '1':
        exts.append(".avi")
    if VchkMKV.get() == '1':
        exts.append(".mkv")
    if VchkMP3.get() == '1':
        exts.append(".mp3")
    if VchkMV4.get() == '1':
        exts.append(".mv4")
    if VchkOGG.get() == '1':
        exts.append(".ogg")
```

the application. Give it a run and see how we did. Notice that if you have a large number of files to go through, the program looks like it's not responding. This is something

that needs to be fixed. We'll create routines to change our cursor from the default to a "watch" style cursor and back so when we do something that takes a long time, the user will notice.

In the "set_Tk_var" routine, add the following code at the bottom.

```
global  
busyCursor, preBusyCursors, bus  
yWidgets  
  
busyCursor = 'watch'  
  
preBusyCursors = None  
  
busyWidgets = (root, )
```

What we do here is set up global variables, assign them and then we set the widget(s) (in busyWidgets) we wish to respond to the cursor change. In this case we set it to root which is our full window. Notice that this is a tuple.

Next we create two routines to set and unset the cursor. First the set routine, which we will call "busyStart". After our "LoadDataGrid" routine, insert the code shown middle right.

We first check to see if a value was passed to "newcursor". If not, we default to the busyCursor. Then

we walk through the busyWidgets tuple and set the cursor to whatever we want.

Now put the code shown bottom right below it.

In this routine, we basically reset the cursor for the widgets in our busyWidget tuple back to our default cursor.

Save and run your program. You should find that the cursor changes whenever you have a long list of files to go through.

While this application doesn't really do much but show you how to use Page to create really fast code development. From today's article, you can see how having a good design of your GUI ahead of time can make the development process easy and

```
def LoadDataGrid():  
    global ColHeads  
    for c in FileList:  
        treeview.insert(' ', 'end', values=c)  
        # adjust column's width if necessary to fit each value  
        for ix, val in enumerate(c):  
            col_w = tkFont.Font().measure(val)  
            if treeview.column(ColHeads[ix], width=None)<col_w:  
                treeview.column(ColHeads[ix], width=col_w)
```

```
def busyStart(newcursor=None):  
    global preBusyCursors  
    if not newcursor:  
        newcursor = busyCursor  
    newPreBusyCursors = {}  
    for component in busyWidgets:  
        newPreBusyCursors[component] = component['cursor']  
        component.configure(cursor=newcursor)  
        component.update_idletasks()  
    preBusyCursors = (newPreBusyCursors, preBusyCursors)
```

```
def busyEnd():  
    global preBusyCursors  
    if not preBusyCursors:  
        return  
    oldPreBusyCursors = preBusyCursors[0]  
    preBusyCursors = preBusyCursors[1]  
    for component in busyWidgets:  
        try:  
            component.configure(cursor=oldPreBusyCursors[component])  
        except KeyError:  
            pass  
        component.update_idletasks()
```

fairly painless.

The tcl file is saved in pastebin at <http://pastebin.com/AA1kE4Dy> and the python code is saved at

<http://pastebin.com/VZm5un3e>.

See you next time.

