Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

PROGRAMMING SERIES SPECIAL EDITION

PROGRAMMING SERIES SPECIAL EDITION

PROGRAM IN PYTHON

Volume Five

Parts 27-31

Full Circle Magazine Specials

Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

About Full Circle

Full Circle is a free, independent, magazine dedicated to the Ubuntu family of Linux operating systems. Each month, it contains helpful how-to articles and reader-submitted stories.

Full Circle also features a companion podcast, the Full Circle Podcast which covers the magazine, along with other news of interest.

Welcome to another 'single-topic special'

In response to reader requests, we are assembling the content of some of our serialised articles into dedicated editions.

For now, this is a straight reprint of the series **'Programming in Python', Parts 27-31** from issues #53 through #59; and yes, peerless Python professor Gregg Walters took some time off during this run!

Please bear in mind the original publication date; current versions of hardware and software may differ from those illustrated, so check your hardware and software versions before attempting to emulate the tutorials in these special editions. You may have later versions of software installed or available in your distributions' repositories.

Enjoy!

Find Us

Website:
http://www.fullcirclemagazine.org/

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

IRC: #fullcirclemagazine on chat.freenode.net

Editorial Team

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

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

Editing & Proofreading
Mike Kennedy, Lucas Westermann, Gord Campbell, Robert Orsino, Josh Hertel, Bert Jerred

Our thanks go to Canonical and the many translation teams around the world.

Written by Greg Walters

Program In Python - Part 27

If you've ever waited in line to buy a movie ticket, you've been in a queue. If you've ever had to wait in traffic at rush hour, you've been in a queue. If you've ever waited in a government office with one of those little tickets that says you’re number 98, and the sign says "Now serving number 42," you've been in a queue.

In the world of computers, queues are common. As a user, most times, you don't have to think about them. They are invisible to the user. But if you ever have to deal with realtime events, you will eventually have to deal with them. It's just data of one type or another, waiting in line for its turn to be processed. Once it's in the queue, it's there until it gets accessed, and then it's gone. You can't get the value of the next data item unless you pull it out of the queue. You can't, for example, get the value of the 15th item in the queue. You have to access the other 14 items first. Once it's accessed, it's out of the queue. It's gone, and unless you save it to a long-term variable, there's no way to get the data back.

There are multiple types of queues. The most common ones are FIFO (First In, First Out), LIFO (Last In, First Out), Priority, and Ring. We'll talk about ring queues another time.

FIFO queues are what we see in everyday life. All of the examples I listed above are FIFO queues. The first person in the line gets handled first, moves on, then everyone moves up one spot in the line. In a FIFO buffer, there is (within reason) no limit to the number of items it can hold. They just stack up in order. As an item is handled, it is pulled out (or dequeued) of the queue, and everything moves closer to the front of the queue by one position.

LIFO Queues are less common in life, but there are still real-world examples. The one that comes to mind most quickly is a stack of dishes in your kitchen cabinet. When the dishes are washed and dryed, they get stacked in the cabinet. The last one in on the stack is the first one that comes out to be used. All the rest have to wait, maybe for days, to be used. It's a good thing that the movie ticket queue is FIFO, isn't it? Like the FIFO queue, within reason, there is no limit to the size of a LIFO queue. The first item in the queue has to wait as newer items are pulled out of the buffer (plates pulled off the stack) until it's the only one left.

Priority queues are a bit harder for many people to imagine right off the bat. Think of a company that has one printer. Everyone uses that one printer. The print jobs are handled by department priority. Payroll has a higher priority (and thankfully so) than say, you, a programmer. You have a higher priority (and thankfully so) than the receptionist. So in short, the data that has a higher priority gets handled, and gets out of the queue, before data that has a lower priority.

FIFO

There are multiple types of queues. The most common ones are FIFO (First In, First Out), LIFO (Last In, First Out), Priority, and Ring.

FIFO queues are easy to visualize in terms of data. A python list is an easy mental representation. Consider this list...

```
[1,2,3,4,5,6,7,8,9,10]
```

There are 10 items in the list. As a list, you access them by index. However, in a queue, you can't access the items by index. You have to deal with the next one in the line and the list isn't static. It's VERY dynamic. As we request the next item in the queue, it gets removed. So using the example above, you request one item from the queue. It returns the first item (1) and the queue then looks like this.

```
[2,3,4,5,6,7,8,9,10]
```

contents ^

```
import Queue
fifo = Queue.Queue()
for i in range(5):
    fifo.put(i)

while not fifo.empty():
    print fifo.get()
```

Request two more and you get 2, then 3, returned, and then the queue looks like this.

```
[4,5,6,7,8,9,10]
```

I'm sure you get the idea. Python provides a simple library, surprisingly enough, called Queue, that works well for small-to-medium sized queues, up to about 500 items. Above is a simple example to show it.

In this example, we initialize the queue (fifo = Queue.Queue()) then put the numbers 0 through 4 into our queue (fifo.put(i)). We then use the internal method .get() to pull items off the queue until the queue is empty, .empty(). What is returned is 0,1,2,3,4. You can also set the maximum number of items that the queue can handle by initializing it with the size of the queue like this.

```
import Queue

fifo = Queue.Queue(12)
for i in range(13):
    if not fifo.full():
        fifo.put(i)

while not fifo.empty():
    print fifo.get()
```

```
fifo = Queue.Queue(300)
```

Once the maximum number of items have been loaded, the Queue blocks any additional entries going into the queue. This has a side effect of making the program look like it's "locked" up, though. The easiest way to get around this is to use the Queue.full() check (above right).

In this case, the queue is set for a maximum of 12 items. As we put items into the queue, we start with '0' and get up to '11'. When we hit number 12, though, the buffer is already full. Since we check to see if the buffer is full before we try to put the item in, the last item is simply discarded.

There are other options, but they can cause other side-effects, and we will address this in a future article. So, for the majority of the time, the bottom line is either use a queue with no limit or make sure you have more space in your queue than you will need.

LIFO

The Queue library also supports LIFO queues. We'll use the above list as a visual example. Setting up our queue, it looks like this:

```
[1,2,3,4,5,6,7,8,9,10]
```

```
import Queue
lifo = Queue.LifoQueue()
for i in range(5):
    lifo.put(i)
while not lifo.empty():
    print lifo.get()
```

Pulling three items from the queue, it then looks like this:

```
[1,2,3,4,5,6,7]
```

Remember that in a LIFO queue, items are removed in a LAST-in FIRST-out order. Here's the simple example modified for a LIFO queue:

When we run it, we get "4,3,2,1,0".

As with the FIFO queue, you have the ability to set the size of the queue, and you can use the .full() check.

```
pq = Queue.PriorityQueue()
pq.put((3,'Medium 1'))
pq.put((4,'Medium 2'))
pq.put((10,'Low'))
pq.put((1,'high'))

while not pq.empty():
    nex = pq.get()
    print nex
    print nex[1]
```

PRIORITY

While it's not often used, a Priority queue can sometimes be helpful. It's pretty much the same as the other queue structures, but we need to pass a tuple that holds both the priority and the data. Here's an example using the Queue library:

```
(1, 'high')
high
(3, 'Medium')
Medium
(4, 'Medium')
Medium
(10, 'Low')
Low
```

First, we initialize the queue. Then we put four items into the queue. Notice we use the format (priority, data) to put our data. The library sorts our data in a ascending order based on the priority value. When we pull the data, it comes back as a tuple, just like we put it in. You can address by index the data. What we get back is...

In our first two examples, we simply printed the data that comes out of our queue. That's fine for these examples, but in real-world programming, you probably need to do something with that information as soon as it comes out of the queue, otherwise it's lost. When we use the 'print fifo.get', we send the data to the terminal and then it's destroyed. Just something to keep in mind.

Now let's use some of what we've already learned about tkinter to create a queue demo program. This demo will have two frames. The first will contain (to the user) three buttons. One for a FIFO queue, one for a LIFO queue, and one for a PRIORITY queue. The second frame will contain an entry widget, two buttons, one for adding to the queue, and one for pulling from the queue, and three labels, one showing when the queue is empty, one showing when the queue is full, and one to display what has been pulled from the queue. We'll also be writing some code to automatically center the window within the screen. Above left is the beginning of the code.

```
import sys
from Tkinter import *
import ttk
import tkMessageBox
import Queue

class QueueTest:
    def __init__(self,master = None):
        self.DefineVars()
        f = self.BuildWidgets(master)
        self.PlaceWidgets(f)
        self.ShowStatus()
```

Here we have our imports and the beginning of our class. As before, we create the __init__ routine with the DefineVars, BuildWidgets, and PlaceWidgets routines. We also have a routine called ShowStatus (above right) which will... well, show the status of our queue.

We now create our DefineVars routine. We have four StringVar() objects, an empty variable called

```
    def DefineVars(self):
        self.QueueType = ''
        self.FullStatus = StringVar()
        self.EmptyStatus = StringVar()
        self.Item = StringVar()
        self.Output = StringVar()
        # Define the queues
        self.fifo = Queue.Queue(10)
        self.lifo = Queue.LifoQueue(10)
        self.pq = Queue.PriorityQueue(10)
        self.obj = self.fifo
```

```
    def BuildWidgets(self,master):
        # Define our widgets
        frame = Frame(master)
        self.f1 = Frame(frame,
            relief = SUNKEN,
            borderwidth=2,
            width = 300,
            padx = 3,
            pady = 3
        )
        self.btnFifo = Button(self.f1,
            text = "FIFO"
        )
        self.btnFifo.bind('<ButtonRelease-1>',
            lambda e: self.btnMain(1)
        )
        self.btnLifo = Button(self.f1,
            text = "LIFO"
        )
        self.btnLifo.bind('<ButtonRelease-1>',
            lambda e: self.btnMain(2)
        )
        self.btnPriority = Button(self.f1,
            text = "PRIORITY"
        )
        self.btnPriority.bind('<ButtonRelease-1>',
            lambda e: self.btnMain(3)
        )
```

QueueType, and three queue objects - one for each of the types of queues that we are going to play with. We have set the maximum size of the queues at 10 for the purposes of the demo. We also have created an object called obj, and assigned it to the FIFO queue. When we select a queue type from the buttons, we will set this object to the queue that we want. This way, the queue is maintained when we switch to another queue type (code is on previous page, bottom right).

Here we start the widget definitions. We create our first frame, the three buttons, and their bindings. Notice we are using the same routine to handle the binding callback. Each button sends a value to the callback routine to denote which button was clicked. We could just as easily have created a dedicated routine for each button. However, since all three buttons are dealing with a common task, I thought it would be good to work them as a group (code shown right).

Next (below right), we set up the second frame, the entry widget, and the two buttons. The only thing here that is out of the ordinary is the binding for the entry widget. Here we bind the self.AddToQueue routine to the <Return> key. This way, the user doesn't have to use the mouse to add the data. They can just enter the data into the entry widget, and press <Return> if they want to.

Here (next page, bottom) is the last three widget definitions. All three are labels. We set the textvariable attribute to the variables we defined earlier. If you remember, when that variable changes, so does the text in the label. We also do something a bit different on the lblData label. We will use a different font to make it stand out when we display the data pulled from the queue. Remember that we have to return the frame object so it can be used in the PlaceWidget routine.

This (next page, middle) is the beginning of the PlaceWidgets routine. Notice here that we put five empty labels at the very top of the root window. I'm doing this to set spacing. This is an easy way to "cheat" and make your window placement much easier. We then set the first frame, then another

```
self.f2 = Frame(frame,
    relief = SUNKEN,
    borderwidth=2,
    width = 300,
    padx = 3,
    pady = 3
)
self.txtAdd = Entry(self.f2,
    width=5,
    textvar=self.Item
)
self.txtAdd.bind('<Return>',self.AddToQueue)
self.btnAdd = Button(self.f2,
    text='Add to Queue',
    padx = 3,
    pady = 3
)
self.btnAdd.bind('<ButtonRelease-1>',self.AddToQueue)
self.btnGet = Button(self.f2,
    text='Get Next Item',
    padx = 3,
    pady = 3
)
self.btnGet.bind('<ButtonRelease-1>',self.GetFromQueue)
```

```
        self.lblEmpty = Label(self.f2,
            textvariable=self.EmptyStatus,
            relief=FLAT
        )
        self.lblFull = Label(self.f2,
            textvariable=self.FullStatus,
            relief=FLAT
        )
        self.lblData = Label(self.f2,
            textvariable=self.Output,
            relief = FLAT,
            font=("Helvetica", 16),
            padx = 5
        )

        return frame
```

"cheater" label, then the three buttons.

Here we place the second frame, another "cheater" label, and the rest of our widgets.

```
def Quit(self):
    sys.exit()
```

Next we have our "standard" quit routine which simply calls sys.exit() (above right).

Now our main button callback routine, btnMain. Remember we are sending in (through the p1 parameter) which button was clicked. We use the self.QueueType variable as a reference to which queue type we are dealing with, then we assign self.obj to the proper queue, and finally change the title of our root window to display the queue type we are using. After that, we print the queue type to the terminal window (you don't really have to do that), and call the ShowStatus routine. Next (following page, top right) we'll make the ShowStatus routine.

As you can see, it's pretty simple. We set the label variables to their proper state so they

```
    def btnMain(self,p1):
        if p1 == 1:
            self.QueueType = 'FIFO'
            self.obj = self.fifo
            root.title('Queue Tests - FIFO')
        elif p1 == 2:
            self.QueueType = 'LIFO'
            self.obj = self.lifo
            root.title('Queue Tests - LIFO')
        elif p1 == 3:
            self.QueueType = 'PRIORITY'
            self.obj = self.pq
            root.title('Queue Tests - Priority')

        print self.QueueType
        self.ShowStatus()
```

```
        self.f2.grid(column = 0,row = 2,sticky='nsew',columnspan=5,padx = 5, pady = 5)
        l = Label(self.f2,text='',width = 15,anchor = 'e').grid(column = 0, row = 0)
        self.txtAdd.grid(column=1,row=0)
        self.btnAdd.grid(column=2,row=0)
        self.btnGet.grid(column=3,row=0)
        self.lblEmpty.grid(column=2,row=1)
        self.lblFull.grid(column=3,row = 1)
        self.lblData.grid(column = 4,row = 0)
```

```
    def PlaceWidgets(self, master):
        frame = master
        # Place the widgets
        frame.grid(column = 0, row = 0)
        l = Label(frame,text='',relief=FLAT,width = 15, anchor = 'e').grid(column = 0, row = 0)
        l = Label(frame,text='',relief=FLAT,width = 15, anchor = 'e').grid(column = 1, row = 0)
        l = Label(frame,text='',relief=FLAT,width = 15, anchor = 'e').grid(column = 2, row = 0)
        l = Label(frame,text='',relief=FLAT,width = 15, anchor = 'e').grid(column = 3, row = 0)
        l = Label(frame,text='',relief=FLAT,width = 15, anchor = 'e').grid(column = 4, row = 0)

        self.f1.grid(column = 0,row = 1,sticky='nsew',columnspan=5,padx = 5,pady = 5)
        l = Label(self.f1,text='',width = 25,anchor = 'e').grid(column = 0, row = 0)
        self.btnFifo.grid(column = 1,row = 0,padx = 4)
        self.btnLifo.grid(column = 2,row = 0,padx = 4)
        self.btnPriority.grid(column = 3, row = 0, padx = 4)
```

display if the queue we are using is either full, empty, or somewhere in between.

The AddToQueue routine (next page, bottom right) is also fairly straight-forward. We get the data from the entry box using the .get() function. We then check to see if the current queue type is a priority queue. If so, we need to make sure it's in the correct format. We do that by checking for the presence of a comma. If it isn't, we complain to the user via an error message box. If everything seems correct, we then check to see if the queue that we are currently using is full. Remember, if the queue is full, the put routine is blocked and the program will hang. If everything is fine, we add the item to the queue and update the status.

The GetFromQueue routine (middle right) is even easier. We check to see if the queue is empty so as not to run into a blocking issue, and, if not, we pull the data from the queue, show the data, and update the status.

```
if __name__ == '__main__':
    def Center(window):
        # Get the width and height of the screen
        sw = window.winfo_screenwidth()
        sh = window.winfo_screenheight()
        # Get the width and height of the window
        rw = window.winfo_reqwidth()
        rh = window.winfo_reqheight()
        xc = (sw-rw)/2
        yc = (sh-rh)/2
        window.geometry("%dx%d+%d+%d"%(rw,rh,xc,yc))
        window.deiconify()
```

We are getting to the end of our application. Here is the center window routine (above left). We first get the screen width and screen height of the screen we are on. We then get the width and height of the root window by using the winfo_reqwidth() and winfo_reqheight() routines built into tkinter. These routines, when called at the right time, will return the width and height of the root window based on the widget placement. If you call it too early, you'll get data, but it won't be what you really need. We then subtract the required window

```
    def ShowStatus(self):
        # Check for Empty
        if self.obj.empty() == True:
            self.EmptyStatus.set('Empty')
        else:
            self.EmptyStatus.set('')
        # Check for Full
        if self.obj.full() == True:
            self.FullStatus.set('FULL')
        else:
            self.FullStatus.set('')
```

```
    def GetFromQueue(self,p1):
        self.Output.set('')
        if not self.obj.empty():
            temp = self.obj.get()
            self.Output.set("Pulled
{0}".format(temp))
        self.ShowStatus()
```

```
    def AddToQueue(self,p1):
        temp = self.Item.get()
        if self.QueueType == 'PRIORITY':
            commapos = temp.find(',')
            if commapos == -1:
                print "ERROR"
                tkMessageBox.showerror('Queue Demo',
                    'Priority entry must be in format\r(priority,data)')
            else:
                self.obj.put(self.Item.get())
        elif not self.obj.full():
            self.obj.put(self.Item.get())
        self.Item.set('')
        self.ShowStatus()
```

width from the screen width, and divide it by two, and do the same thing for the height information. We then use that information to set the geometry call. In MOST instances, this works wonderfully. However, there might be times that you need to set the required width and height by hand.

Finally, we instantiate the root window, set the base title, instantiate the QueueTest class. We then call root.after, which waits x number of milliseconds (in this case 3) after the root window is instantiated, and then calls the Center routine. This way, the root window has been completely set up and is ready to go, so we can get the root window width and height. You might have to tweak the delay time a bit. Some machines are much faster than others. 3 works fine on my machine, your mileage may vary. Last but not least, we call the root window mainloop to get the application to run.

```
root = Tk()
root.title('Queue Tests - FIFO')
demo = QueueTest(root)
root.after(3,Center,root)
root.mainloop()
```

As you play with the queues, notice that if you put some data in one queue (let's say the FIFO queue) then switch to another queue (let's say the LIFO queue), the data that was put into the FIFO queue is still there and waiting for you. You can completely or partially fill all three queues, then start playing with them.

Well, that's it for this time. Have fun with your queues. The QueueTest code can be found at http://pastebin.com/5BBUiDce.

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.com.

Below Zero

Zero Downtime

Below Zero is a Co-located Server Hosting specialist in the UK.

Uniquely we only provide rack space and bandwidth. This makes our service more reliable, more flexible, more focused and more competitively priced. We concentrate solely on the hosting of Co-located Servers and their associated systems, within Scotland's Data Centres.

At the heart of our networking infrastructure is state-of-the-art BGP4 routing that offers optimal data delivery and automatic multihomed failover between our outstanding providers. Customers may rest assured that we only use the highest quality of bandwidth; our policy is to pay more for the best of breed providers and because we buy in bulk this doesn't impact our extremely competitive pricing.

At Below Zero we help you to achieve Zero Downtime.

www.zerodowntime.co.uk

Written by Greg Walters

Program In Python - Part 28

We are going to explore even more widgets provided by tkinter. This time we will look at menus, combo boxes, spin boxes, separator bar, progress bars and notebooks. Let's talk about them one at a time.

You've seen menus in almost every application that you have ever used. Tkinter makes it VERY easy for us to make menus. Combo Boxes are similar to the list box that we explored in the last widget demo article, except the list “pops down” instead of being visible at all times. Spin box controls are great for giving a fixed range of values that can “scroll” up or down. For example, if we want the user to be able to choose from integers between 1 and 100, we can easily use a spin box. Progress bars are a wonderful way to show that your application hasn't locked up when something takes a lot of time, like reading records from a database. It can show the percentage of completion of a task. There are two types of progress bars, Determinate and Indeterminate.

You use a determinate progress bar when you know just how many items you are dealing with. If you don't know the number of items or the percentage of how done your task is at any point, you would use the Indeterminate version. We will work with both. Finally a notebook widget (or tabbed widget) is used many times for things like configuration screens. You can logically group a series of widgets on each tab.

So, let's get started. As usual, we will create a base application and build on to it with each extra widget we add. Shown right is the first part of our application. You've seen most of this before.

Save all of this as widgetdemo2a.py. Remember we will use this as the base to build the full demo. Now we will start the process of creating the menu. Here are the steps we need to do. First, we define a variable to hold the menu instance. Like most any widget we use, the format is...

```
OurVariable = Widget(parent,
options).
```

```
import sys
from Tkinter import *
import ttk
# Shows how to create a menu
class WidgetDemo2:

    def __init__(self,master = None):
        self.DefineVars()
        f = self.BuildWidgets(master)
        self.PlaceWidgets(f)

    def DefineVars(self):
        pass
```

And here is the bottom of our program. Again, you have seen this before. Nothing new here.

```
if __name__ == '__main__':
    def Center(window):
        # Get the width and height of the screen
        sw = window.winfo_screenwidth()
        sh = window.winfo_screenheight()
        # Get the width and height of the window
        rw = window.winfo_reqwidth()
        rh = window.winfo_reqheight()
        xc = (sw-rw)/2
        yc = (sh-rh)/2
        print "{0}x{1}".format(rw,rh)
        window.geometry("%dx%d+%d+%d"%(rw,rh,xc,yc))
        window.deiconify()

    root = Tk()
    root.title('More Widgets Demo')
    demo = WidgetDemo2(root)
    root.after(13,Center,root)
    root.mainloop()
```

HOWTO - PROGRAM IN PYTHON - PART 28

In this case, we are using the Menu widget and we will assign it to master as the parent. We do this under the BuildWidgets routine. Next we create another menu item, this time calling it filemenu. We add commands and separators as needed. Finally we add it to the menu bar and do it all over again until we are done. In our example, we'll have the menubar, a File pulldown, an Edit pulldown and a Help pulldown (top right). Let's get started.

Next (middle right) we concentrate on the File Menu. There will be five elements. New, Open, Save, a separator and Exit. We'll use the .add_command method to add the command. All we really need to do is call the method with the text (label =) and then provide a callback function to handle when the user clicks the item. Finally we use the menubar.add_cascade function to attach the menu to the bar.

Notice that the Exit command uses “root.quit” to end the program. No call back needed for that. Next we'll do the same thing for the Edit and Help menus.

Notice the part in each of the menu group definitions that says “tearoff=0”. If you were to change the “=0” to “=1”, the menu would start with what looks like a dashed line and if you drag it, it “tears off” and creates its own window. While this might be helpful sometime in the future, we don't want that here.

Last but not least, we need to place the menu. We don't do a normal placement with the .grid() function. We simply add it by using the parent.config function (bottom right).

All of this has gone in the BuildWidgets routine. Now (next page, top right) we need to add a generic frame and set the return statement before we move on to the PlaceWidgets routine.

Finally (next page, bottom right) we need to create all the callbacks we defined earlier. For the demo, all we'll do is print something in the terminal used to launch the program.

That's it. Save and run the

```
def BuildWidgets(self,master):
    frame = Frame(master)
    #==============================
    #       MENU STUFF
    #==============================
    # Create the menu bar
    self.menubar = Menu(master)
```

```
# Create the File Pull Down, and add it to the menu bar
   filemenu = Menu(self.menubar, tearoff = 0)
   filemenu.add_command(label = "New", command = self.FileNew)
   filemenu.add_command(label = "Open", command = self.FileOpen)
   filemenu.add_command(label = "Save", command = self.FileSave)
   filemenu.add_separator()
   filemenu.add_command(label = "Exit", command = root.quit)
   self.menubar.add_cascade(label = "File", menu = filemenu)
```

```
# Create the Edit Pull Down
editmenu = Menu(self.menubar, tearoff = 0)
editmenu.add_command(label = "Cut", command = self.EditCut)
editmenu.add_command(label = "Copy", command = self.EditCopy)
editmenu.add_command(label = "Paste", command = self.EditPaste)
self.menubar.add_cascade(label = "Edit", menu = editmenu)
# Create the Help Pull Down
helpmenu = Menu(self.menubar, tearoff=0)
helpmenu.add_command(label = "About", command = self.HelpAbout)
self.menubar.add_cascade(label = "Help", menu = helpmenu)
```

```
  # Now, display the menu
master.config(menu = self.menubar)
  #=========================================
  #             End of Menu Stuff
  #=========================================
```

program. Click on each of the menu options (saving File|Exit for last).

```
    def FileNew(self):
        print "Menu - File New"

    def FileOpen(self):
        print "Menu - File Open"

    def FileSave(self):
        print "Menu - File Save"

    def EditCut(self):
        print "Menu - Edit Cut"

    def EditCopy(self):
        print "Menu - Edit Copy"

    def EditPaste(self):
        print "Menu - Edit Paste"

    def HelpAbout(self):
        print "Menu - Help About"
```

Now (below) we'll deal with the combo box. Save your file as widgetdemo2b.py and we'll get started. The imports, class definition and the def __init__ routines are all the same, as is the bottom part of the program. We'll add two lines to the DefineVars routine. Either comment out the "pass" statement or erase it and put in the following code. (I included the definition line just for clarity.)

```
    def DefineVars(self):
        self.cmbo1Val = StringVar()
        self.c1Vals = ['None','Option 1','Option 2','Option 3']
```

After our the self.f1 definition in BuildWidgets and before the "return frame" line insert the following code.

```
        # Combo Box
        self.lblcb = Label(self.f1, text = "Combo Box: ")
        self.cmbo1 = ttk.Combobox(self.f1,
                                  height = "19",
                                  width = 20,
                                  textvariable = self.cmbo1Val
                                 )
        self.cmbo1['values'] = self.c1Vals
        # Bind the virtual event to the callback
        self.cmbo1.bind("<<ComboboxSelected>>",self.cmbotest)
```

First we define a label, which we've done before. Next we define the combo box. We use "ttk.Combobox", define the parent and set the height to 19, the width to 20 and the textvariable to "self.cmbo1Val". Remember that we set textvariables in the last widget demo, but just in case you forgot...this is changed anytime the value in the combo box is changed. We defined it in DefineVars as a StringVar object. Next we load the values that we want the user to choose from, again we defined that in DefineVars. Finally we bind the virtual event

```
        self.f1 = Frame(frame,
                        relief = SUNKEN,
                        borderwidth = 2,
                        width = 500,
                        height = 100
                        )

        return frame
```

Next we (as we have done multiple times) deal with placing our other widgets.

```
    def PlaceWidgets(self,master):
        frame = master
        frame.grid(column = 0, row = 0)

        self.f1.grid(column = 0,
                     row = 0,
                     sticky = 'nsew'
                    )
```

<<ComboboxSelected>> to the cmbotest routine that we will flesh out in a minute.

Next let's place the combo box and the label into our form (top right).

```
self.lblcb.grid(column = 0,row = 2)
self.cmbo1.grid(column = 1,
                row = 2,
                columnspan = 4,
                pady = 2
                )
```

And finally we put in the callback which simply prints what the user selected into the terminal window.

```
def cmbotest(self,p1):
    print self.cmbo1Val.get()
```

Save everything and test it out.

Now save as widgetdemo2c.py and we'll start with the separator bar. This is SO super easy. While the updated tkinter provides a separator bar widget, I've never been able to get it to work. Here's an easy work around. We use a frame with a height of 2. The only changes to our program will be the definition of the frame in BuildWidgets after the combo box bind statement and placing the frame in the Place Widgets routine. So, in BuildWidgets put in the following lines (shown middle right)...

```
self.fsep = Frame(self.f1,
                  width = 140,
                  height = 2,
                  relief = RIDGE,
                  borderwidth = 2
                  )
```

And in PlaceWidgets put in this …

```
self.fsep.grid(column = 0,
               row = 3,
               columnspan = 8,
               sticky = 'we',
               padx = 3,
               pady = 3
               )
```

Once again, you've seen all this before. Save and test it. You'll probably have to expand the topmost window to see the separator, but it will become much more evident in the next demo. Save as widgetdemo2d.py and we'll add the spin control.

Under DefineVars, add the following line...

```
self.spinval = StringVar()
```

By now, you know that this is so we can get the value at any time we want. Next, we'll add some code to the BuildWidgets routine...just before the "return frame" line (bottom right).

```
self.lblsc = Label(self.f1, text = "Spin Control:")
self.spin1 = Spinbox(self.f1,
                     from_ = 1.0,
                     to = 10.0,
                     width = 3,
                     textvariable = self.spinval,
                     wrap=True
                     )
```

Here we define a label and the spin control. The spin control definition is as follows:

```
ourwidget =
Spinbox(parent,low value,
high value, width,
textvariable, wrap)
```

The low value must be called as "from_" since the word "from" is a keyword and using that would simply confuse everyting. The values "from_" and "to" must be defined as float values. In this case we want it to have a low value of 1 and a high value of 10. Finally the wrap option says that if the value is (in our case) 10 and the user clicks on the up arrow, we want it to wrap around to the low value and keep going. The same works for the low value. If the user clicks the down arrow of the control and the value is 1, it wraps to 10 and keeps going. If you set “wrap=False”, the control simply stops at whichever direction the user is going.

Now we'll place the widgets in PlaceWidgets (below).

Again, that's it. Save and play. You'll really notice the separator now.

Save as widgetdemo2e.py and we'll do the progress bars.

Again, we need to define some variables, so in the DefineVars routine add the following code...

```
self.spinval2 = StringVar()
self.btnStatus = False
self.pbar2val = StringVar()
```

It should be pretty obvious what the two StringVar variables are. We'll discuss the “self.btnStatus” in a moment. For now, let's go and define the widgets for this portion in BuildWidgets (right).

Again this goes before the “return frame” line. What we are doing is setting up a frame for us to put the widgets into. Then we set up two labels as guides. Next we define the first progress bar. Here the only things that might be strange are length, mode and maximum. Length is the size in pixels of our bar. Maximum is the highest value that will be seen. In this case it's 100 since we are looking at percentage. Mode in this case is 'indeterminate'. Remember, we use this mode when we don't know how far we've gotten in a task so we just want to let the user know that something is happening.

Next we add a button (you've done this before), another label another progress bar and another spin control. The mode for the second progress bar is “determinate”. We will use the spin control to set the “percentage” of completion. Next add the following lines (next page, top left) into the PlaceWidgets routine.

```
        self.lblsc.grid(column = 0, row = 4)
        self.spin1.grid(column = 1,
                        row = 4,
                        pady = 2
                        )
```

```
        #========================================
        # Progress Bar Stuff
        #========================================
        self.frmPBar = Frame(self.f1,
                        relief = SUNKEN,
                        borderwidth = 2
                        )

        self.lbl0 = Label(self.frmPBar,
                        text = "Progress Bars"
                        )
        self.lbl1 = Label(self.frmPBar,
                        text = "Indeterminate",
                        anchor = 'e'
                        )
        self.pbar = ttk.Progressbar(self.frmPBar,
                        orient = HORIZONTAL,
                        length = 100,
                        mode = 'indeterminate',
                        maximum = 100
                        )
        self.btnptest = Button(self.frmPBar,
                        text = "Start",
                        command = self.TestPBar
                        )
        self.lbl2 = Label(self.frmPBar,
                        text = "Determinate"
                        )
        self.pbar2 = ttk.Progressbar(self.frmPBar,
                        orient = HORIZONTAL,
                        length = 100,
                        mode = 'determinate',
                        variable = self.pbar2val
                        )
        self.spin2 = Spinbox(self.frmPBar,
                        from_ = 1.0,
                        to = 100.0,
                        textvariable = self.spinval2,
                        wrap = True,
                        width = 5,
                        command = self.Spin2Do
                        )
```

Lastly, we add two routines to control our progress bars (botom right).

The TestPBar routine controls the indeterminate progress bar. Basically, we are starting and stopping an internal timer that is built into the progress bar. The line “self.pbar.start(10)” sets the timer to 10 milliseconds. This makes the bar move fairly quickly. Feel free to play with this value up and down on your own. The Spin2Do routine simply sets the progress bar to whatever value the spin control has. We print it as well to the terminal.

That's all the changes for this. Save and play.

Now save as widgetdemo2f.py and we'll deal with the tabbed notebook widgets. In BuildWidgets put the following code (below) before the “return frame” line...

Let's look at what we did. First, we define a frame for our

```
        #========================================
        #              NOTEBOOK
        #========================================
        self.nframe = Frame(self.f1,
                            relief = SUNKEN,
                            borderwidth = 2,
                            width = 500,
                            height = 300
                            )
        self.notebook = ttk.Notebook(self.nframe,
                                     width = 490,
                                     height = 290
                                     )
        self.p1 = Frame(self.notebook)
        self.p2 = Frame(self.notebook)
        self.notebook.add(self.p1,text = 'Page One')
        self.notebook.add(self.p2,text = 'Page Two')
        self.lsp1 = Label(self.p1,
                          text = "This is a label on
page number 1",
                          padx = 3,
                          pady = 3
                          )
```

```
        # Progress Bar
        self.frmPBar.grid(column = 0,
                          row = 5,
                          columnspan = 8,
                          sticky = 'nsew',
                          padx = 3,
                          pady = 3
                          )
        self.lbl0.grid(column = 0, row = 0)
        self.lbl1.grid(column = 0,
                       row = 1,
                       pady = 3
                       )
        self.pbar.grid(column = 1, row = 1)
        self.btnptest.grid(column = 3, row = 1)
        self.lbl2.grid(column = 0,
                       row = 2,
                       pady = 3
                       )
        self.pbar2.grid(column = 1, row = 2)
        self.spin2.grid(column = 3, row = 2)
```

```
    def TestPBar(self):
        if self.btnStatus == False:
            self.btnptest.config(text="Stop")
            self.btnStatus = True
            self.pbar.start(10)
        else:
            self.btnptest.config(text="Start")
            self.btnStatus = False
            self.pbar.stop()

    def Spin2Do(self):
        v = self.spinval2.get()
        print v
        self.pbar2val.set(v)
```

notebook widget. Now we define the widget. All the options are ones we've seen before. Next we define two frames named self.p1 and self.p2. These act as our pages. The next two lines (self.notebook.add) attach the frames to the notebook widget and they get a tab attached to them. We also set the text for the tabs. Finally, we put a label on page number one. We'll put one on page number two when we place the controls just for fun.

In the PlaceWidgets routine put the following code (below).

The only thing that might possibly be strange is the label on page two. We combine the definition and placement in the grid with the same command. We did that when we did our first widget demo app.

That's it. Save and play.

As always the full code for the full application is up on pastebin at http://pastebin.com/qSPkSNU1.

Enjoy. Next time we'll deal with some more database stuff.

```
        self.nframe.grid(column = 0,
                         row = 6,
                         columnspan = 8,
                         rowspan = 7,
                         sticky = 'nsew'
                         )
        self.notebook.grid(column = 0,
                           row = 0,
                           columnspan = 11,
                           sticky = 'nsew'
                           )
        self.lsp1.grid(column = 0,row = 0)
        self.lsp2 = Label(self.p2,
                          text = 'This is a label on PAGE 2',
                          padx = 3,
                          pady = 3
                          ).grid(
                                column = 0,
                                row = 1
                                )
```

Below Zero

Zero Downtime

Below Zero is a Co-located Server Hosting specialist in the UK.

Uniquely we only provide rack space and bandwidth. This makes our service more reliable, more flexible, more focused and more competitively priced. We concentrate solely on the hosting of Co-located Servers and their associated systems, within Scotland's Data Centres.

At the heart of our networking infrastructure is state-of-the-art BGP4 routing that offers optimal data delivery and automatic multihomed failover between our outstanding providers. Customers may rest assured that we only use the highest quality of bandwidth; our policy is to pay more for the best of breed providers and because we buy in bulk this doesn't impact our extremely competitive pricing.

At Below Zero we help you to achieve Zero Downtime.

www.zerodowntime.co.uk

contents ^

Written by Greg Walters

Program In Python - Part 29

A little while ago, I was asked to convert a MySQL database to SQLite. Looking around the web for a quick and easy (and free) solution, I found nothing that worked with the current version of MySQL for me. So I decided to go ahead and "roll my own".

The MySQL Administrator program allows you to backup a database into a flat text file. Many SQLite browsers allow you to read a flat sql definition file and create the database from there. However, there are many things that MySQL supports that SQLite doesn't. So this month, we'll write a conversion program that reads a MySQL dump file and creates a SQLite version.

Let's start by looking at the MySQL dump file. It consists of a section that creates the database, and then sections that create each table within the database followed by the data for that table, if it's included in the dump file. (There's an option to export the table schema(s) only). Shown above right is an example of one of the create table sections.

```
DROP TABLE IF EXISTS `categoriesmain`;
CREATE TABLE `categoriesmain` (
  `idCategoriesMain` int(10) unsigned NOT NULL
auto_increment,
  `CatText` char(100) NOT NULL default '',
  PRIMARY KEY  (`idCategoriesMain`)
) ENGINE=InnoDB AUTO_INCREMENT=40 DEFAULT
CHARSET=latin1;
```

The first thing that we would need to get rid of is in the last line. Everything after the ending parenthesis needs to go away. (SQLite does not support an InnoDB database). In addition to that, SQLite doesn't support the "PRIMARY KEY" line. In SQLite, we set a primary key by using "INTEGER PRIMARY KEY AUTOINCREMENT" when we define the field. The other thing that SQLite doesn't support is the "unsigned" keyword.

When it comes to the data, the "INSERT INTO" statements are also non-compatible. The problem here is that SQLite doesn't allow multiple inserts within the same statement. Here's a short example from the MySql dump file. Notice (right) that the end-of-line marker is a semicolon.

```
INSERT INTO `categoriesmain`
(`idCategoriesMain`,`CatText`) VALUES
 (1,'Appetizer'),
 (2,'Snack'),
 (3,'Barbecue'),
 (4,'Cake'),
 (5,'Candy'),
 (6,'Beverages');
```

To make this compatible, we need to change this from a single statement format to a series of single statements like this:

```
INSERT INTO `categoriesmain`
(`idCategoriesMain`,`CatText`) VALUES (1,'Appetizer');
INSERT INTO `categoriesmain`
(`idCategoriesMain`,`CatText`) VALUES (2,'Snack');
INSERT INTO `categoriesmain`
(`idCategoriesMain`,`CatText`) VALUES (3,'Barbecue');
INSERT INTO `categoriesmain`
(`idCategoriesMain`,`CatText`) VALUES (4,'Cake');
INSERT INTO `categoriesmain`
(`idCategoriesMain`,`CatText`) VALUES (5,'Candy');
INSERT INTO `categoriesmain`
(`idCategoriesMain`,`CatText`) VALUES (6,'Beverages');
```

We will also ignore any comment lines, and the CREATE DATABASE and USE statements. Once we have the converted SQL file, we'll use a program similar to the public domain program SQLite Database Browser to actually deal with the process of creating the database, tables, and data.

Let's get started. Start a new project folder and a new python file. Name it MySQL2SQLite.py.

Shown above right is the import statement, the class definition, and the __init__ routine.

This will be a commandline driven program, so we'll need to create the “if __name__” statement, a command line argument handler, and a usage routine (if the user doesn't know how to use the program). This goes at the very end of the program. All other code we create will go above this:

```
def error(message):

    print >> sys.stderr,
str(message)
```

Below is the handler that does the printing of the usage statement.

The DoIt() routine is called if our program is being run stand-alone from the command line, which is the design. However, if we want to keep this as a library to be included in another program at another time, we can just use the class. Here we set up a number of variables to make sure that everything works correctly. The code shown bottom right then parses the command line arguments passed to our program, and gets things ready for the main routines.

```
#!/usr/bin/env python
#=====================================
#  MySQL2SQLite.py
#=====================================
#            IMPORTS
import sys
#=====================================

#=====================================
#     BEGIN CLASS MySQL2SQLite
#=====================================
class MySQL2SQLite:
    def __init__(self):
        self.InputFile = ""
        self.OutputFile = ""
        self.WriteFile = 0
        self.DebugMode = 0
        self.SchemaOnly = 0
        self.DirectMode = False
```

```
def DoIt():
    #========================================
    #            Setup Variables
    #========================================
    SourceFile = ''
    OutputFile = ''
    Debug = False
    Help = False
    SchemaOnly = False
    #========================================
```

```
if len(sys.argv) == 1:
    usage()
else:
    for a in sys.argv:
        print a
        if a.startswith("Infile="):
            pos = a.find("=")
            SourceFile = a[pos+1:]
        elif a.startswith("Outfile="):
            pos = a.find("=")
            OutputFile = a[pos+1:]
        elif a == 'Debug':
            Debug = True
        elif a == 'SchemaOnly':
            SchemaOnly = True
        elif a == '-Help' or a == '-H' or a == '-?':
            Help = True
    if Help == True:
        usage()
    r = MySQL2SQLite()
    r.SetUp(SourceFile,OutputFile,Debug,SchemaOnly)
    r.DoWork()
```

HOWTO - PROGRAM IN PYTHON - PART 29

When we start the program, we need to provide at least two variables on the command line. These are the Input file, and the Output file. We also will provide support for the user to see what is happening as the program is running, an option to just create the tables and not stuff the data, and for the user to call for help. Our “normal” command line to start the program looks like this:

```
MySQL2SQLite Infile=Foo
Outfile=Bar
```

where “Foo” is the name of the MySQL dump file, and “Bar” is the name of the SQLite sql file we want the program to create.

You can also call it like this:

```
MySQL2SQLite Infile=Foo
Outfile=Bar Debug SchemaOnly
```

Which will add the option to show the debug messages and to ONLY create the tables and not import the data.

Finally if the user asks for help, we just go to the usage portion of the program.

```
def usage():
    message = (
       '===============================================================================\n'
        'MySQL2SQLite - A database converter\n'
        'Author: Greg Walters\n'
        'USAGE:\n'
        'MySQL2SQLite Infile=filename [Outfile=filename] [SchemaOnly] [Debug] [-H-Help-?\n'

        '   where\n'
        '         Infile is the MySQL dump file\n'
        '         Outfile (optional) is the output filename\n'
        '            (if Outfile is omitted, assumed direct to SQLite\n'
        '         SchemaOnly (optional) Create Tables, DO NOT IMPORT DATA\n'
        '         Debug (optional) - Turn on debugging messages\n'
        '         -H or -Help or -? - Show this message\n'
        'Copyright (C) 2011 by G.D. Walters\n'
        '===============================================================================\n'
        )
    error(message)
    sys.exit(1)

if __name__ == "__main__":
    DoIt()
```

Before we continue, let's take another look at how the command line argument support works.

When a user enters the program name from the command line (terminal), the operating system keeps track of the information entered and passes it to the program just in case there are any options entered. If no options (also called arguments) are entered, the number of arguments is one, which is the name of the application - in our case MySQL2SQLite.py. We can access these arguments by calling the sys.arg command. If the count is greater than one, we will access them in a for loop. We will step through the list of arguments and check each one. Some programs require you to enter the arguments in a specific order. By using the for loop approach, the arguments can be entered in any order. If the user doesn't supply any arguments, or uses the help arguments, we show the usage screen. Shown above is the routine for that.

Moving on, once we have parsed the argument set, we instantiate the class, call the setup routine, which fills certain variables and then call the DoWork routine. We'll start our class now (which is shown on the next page, bottom right).

This (next page, top right) is the definition and the __init__ routine. Here we setup the variables that we will need as we go through the code. Remember that right before we call the DoWork routine, we call the Setup routine. We take our empty variables and assign the correct values to them here. Notice that there is the ability to not write to a file, useful for debugging

contents ^

purposes. We also have the ability to simply write the schema, or database structure, without writing the data. This is helpful if you are taking a database and starting a new project without wanting to use any existing data.

We start off by opening the SQL Dump file, then setting some internal scope variables. We also define some strings to save us typing later on. Then, if we are to write to an output file, we open it and then we start the entire process. We will read each line of the input file, process it, and potentially write it to the output file. We use a forced while loop to assist reading each line, with a break command when there is nothing left in the input file. We use f.readline() to get the line to work, and assign it to the variable “line”. Some lines, we can safely ignore. We'll simply use an if/elif statement followed by a pass statement to accomplish this (below).

Next we can stop ignoring things and actually do something. If we have a CreateTable statement,

```
    while 1:
        line = f.readline()
        cntr += 1
        if not line:
            break
        # Ignore blank lines, lines that start with
  "--" or comments (/*!)
        if line.startswith("--"): #Comments
            pass
        elif len(line) == 1: # Blank Lines
            pass
        elif line.startswith("/*!"): # Comments
            pass
        elif line.startswith("USE"):
            #Ignore USE lines
            pass
        elif line.startswith("CREATE DATABASE "):
            pass
```

```
    def SetUp(self, In, Out = '', Debug = False, Schema = 0):
        self.InputFile = In
        if Out == '':
            self.writeFile = 0
        else:
            self.WriteFile = 1
            self.OutputFile = Out
        if Debug == True:
            self.DebugMode = 1
        if Schema == 1:
            self.SchemaOnly = 1
```

Now, we'll deal with the DoWork routine, which is where the actual “magic” happens.

```
    def DoWork(self):
        f = open(self.InputFile)
        print "Starting Process"
        cntr = 0
        insertmode = 0
        CreateTableMode = 0
        InsertStart = "INSERT INTO "
        AI = "auto_increment"
        PK = "PRIMARY KEY "
        IPK = " INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL"
        CT = "CREATE TABLE "
        # Begin
        if self.WriteFile == 1:
            OutFile = open(self.OutputFile,'w')
```

```
#=====================================
#     BEGIN CLASS MySQL2SQLite
#=====================================
class MySQL2SQLite:
    def __init__(self):
        self.InputFile = ""
        self.OutputFile = ""
        self.WriteFile = 0
        self.DebugMode = 0
        self.SchemaOnly = 0
```

we'll start that process. Remember we defined CT to be equal to "Create Table". Here (above right), we set a variable "CreateTableMode" to be equal to 1, so we know that's what we are doing, since each field definition is on a separate line. We then take our line, remove the carriage return, and get that ready to write to our out file, and, if required, write it.

```
    elif line.startswith(CT):
        CreateTableMode = 1
        l1 = len(line)
        line = line[:l1-1]
        if self.DebugMode == 1:
            print "Starting Create Table"
            print line
        if self.WriteFile == 1:
            OutFile.write(line)
```

Now (middle right) we need to start dealing with each line within the create table statements - manipulating each line to keep SQLite happy. There are many things that SQLite won't deal with. Let's look at a Create Table statement from MySQL again.

```
CREATE TABLE `categoriesmain` (
  `idCategoriesMain` int(10) unsigned NOT NULL auto_increment,
  `CatText` char(100) NOT NULL default '',
  PRIMARY KEY  (`idCategoriesMain`)
) ENGINE=InnoDB AUTO_INCREMENT=40 DEFAULT CHARSET=latin1;
```

One thing that SQLite will absolutely have an issue with is the entire last line after the closing parenthesis. Another is the line just above that, the Primary Key line. Yet another thing is the unsigned keyword in the second line. It will take a bit of code (below) to work around these issues, but we can make it happen.

```
        elif CreateTableMode == 1:
            # Parse the line...
            if self.DebugMode == 1:
                print "Line to process – {0}".format(line)
```

First, (third down on the right) we check to see if the line contains "auto increment". We will assume that this will be the primary key line. While this might be true 98.6% of the time, it won't always be. However, we'll keep it simple. Next we check to see if the line starts with ") ". This will signify this is the last line of the create table section. If so, we simply set a string to close the statement properly in the variable "newline", turn off the CreateTableMode variable, and, if we are writing to file, write it out.

```
            p1 = line.find(AI)
            if line.startswith(") "):
                CreateTableMode = 0
                if self.DebugMode == 1:
                    print "Finished Table Create"
                newline = ");\n"
                if self.WriteFile == 1:
                    OutFile.write(newline)
                    if self.DebugMode == 1:
                        print "Writing Line {0}".format(newline)
```

Now (bottom right) we use the information we found about the auto increment key word. First, we strip the line of any spurious spaces, then check to see where (we are assuming it is there) the phrase " int(" is within the line. We will be replacing this with the phrase " INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL". The length of the integer doesn't matter to SQLite. Again, we write it out if we should.

```
            elif p1 != -1:
                # Line is primary key line
                l = line.strip()
                fnpos = l.find(" int(")
                if fnpos != -1:
                    fn = l[:fnpos]
                newline = fn + IPK #+ ",\n"
                if self.WriteFile == 1:
                    OutFile.write(newline)
                    if self.DebugMode == 1:
                        print "Writing Line {0}".format(newline)
```

contents ^

Now we look for the phrase “PRIMARY KEY “ within the line. Notice the extra space at the end - that's on purpose. If it arises, we ignore the line.

```
elif
line.strip().startswith(PK):

    pass
```

Now (top right) we look for the phrase “ unsigned “ (again keep the extra spaces) and replace it with “ “.

```
elif line.find(" unsigned ") != -1:
    line = line.replace(" unsigned "," ")
    line = line.strip()
    l1 = len(line)
    line = line[:l1-1]
    if self.WriteFile == 1:
        OutFile.write("," + line)
        if self.DebugMode == 1:
            print "Writing Line {0}".format(line)
```

Otherwise, we can deal with the line.

```
else:
    l1 = len(line)
    line = line.strip()
    line = line[:l1-4]
    if self.DebugMode == 1:
        print "," + line
    if self.WriteFile == 1:
        OutFile.write("," + line)
```

That's the end of the create table routine. Now (below) we move on to the insert statements for the data. The InsertStart variable is the phrase “INSERT INTO “. We check for that because MySQL allows for multiple insert statements in a single command, but SQLite does not. We need to make separate statements for each block of data. We set a variable called “insertmode” to 1, pull the “INSERT INTO {Table} {Fieldlist} VALUES (“ into a reusable variable (which I'll call our prelude), and move on.

```
elif line.startswith(InsertStart):
    if insertmode == 0:
        insertmode = 1
        # Get tablename and field list here
        istatement = line
        # Strip CR/LF from istatement line
        l = len(istatement)
        istatement = istatement[:l-2]
```

Now, we check to see if we are only supposed to work the schema. If so, we can safely ignore any portions of the insert statements. If not, we need to deal with them.

```
elif self.SchemaOnly == 0:
    if insertmode == 1:
```

We check to see if there is either “');” or “'),” in our line. In the case of “');”, this would be the last line in our insert statement set.

```
posx = line.find("');")
pos1 = line.find("'),")
l1 = line[:pos1]
```

This line checks for escaped single quotes and replaces them.

```
line =
line.replace("\\'","''")
```

```
if posx != -1:
    l1 = line[:posx+3]
    insertmode = 0
    if self.DebugMode == 1:
        print istatement + l1
        print "-----------------------------"
    if self.WriteFile == 1:
        OutFile.write(istatement + l1+"\n")
```

Otherwise, we join the prelude to the value statement and end it with a semicolon.

```
elif pos1 != -1:
    l1 = line[:pos1+2]
    if self.DebugMode == 1:
        print istatement + l1 + ";"
    if self.WriteFile == 1:
        OutFile.write(istatement + l1 + ";\n")
```

If we have a closing statement (");"), that is the end of our insert set, and we can create the statement by joining the prelude to the actual value statement. This is shown on the previous page, bottom right.

This all works (top right) if the last value we have in the insert statement is a quoted string. However, if the last value is a numeric value, we have to deal with things a bit differently. You'll be able to pick out what we are doing here.

Finally, we close our input file, and, if we are writing an output file, we close that as well.

```
f.close()
if self.WriteFile == 1:
   OutFile.close()
```

Once you have your converted file, you can use SQLite Database Browser to fill in the database structure and data.

This code should work over 90% of the time as is. There might be somethings we missed due to other issues, hence the reason for the debug mode. However, I've tested this on multiple files and had no problems.

As always, the code is up at PasteBin at http://pastebin.com/cPvzNT7T.

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.com.

```
else:
    if self.DebugMode == 1:
        print "Testing line {0}".format(line)
    pos1 = line.find("),")
    posx = line.find(");")
    if self.DebugMode == 1:
        print "pos1 = {0}, posx = {1}".format(pos1,posx)
    if pos1 != -1:
        l1 = line[:pos1+1]
        if self.DebugMode == 1:
            print istatement + l1 + ";"
        if self.WriteFile == 1:
            OutFile.write(istatement + l1 + ";\n")
    else:
        insertmode = 0
        l1 = line[:posx+1]
        if self.DebugMode == 1:
            print istatement + l1 + ";"
        if self.WriteFile == 1:
            OutFile.write(istatement + l1 + ";\n")
```

Written by Greg D. Walters

Beginning Python - Part 30

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.

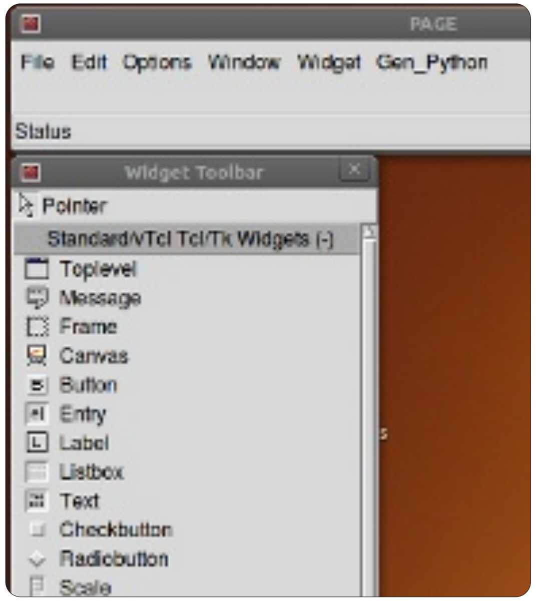
Installation

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.

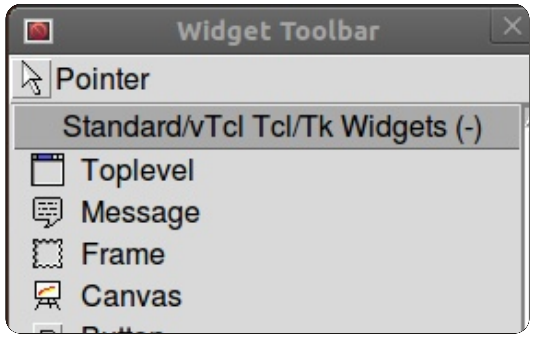
Learning Page

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.

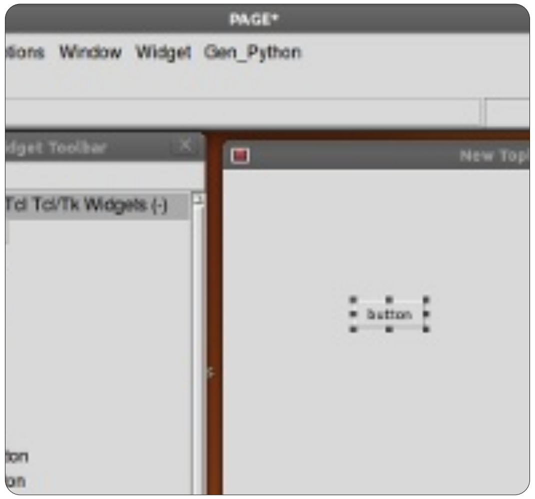
To start a new project, click on the Toplevel button in the toolbox.



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.



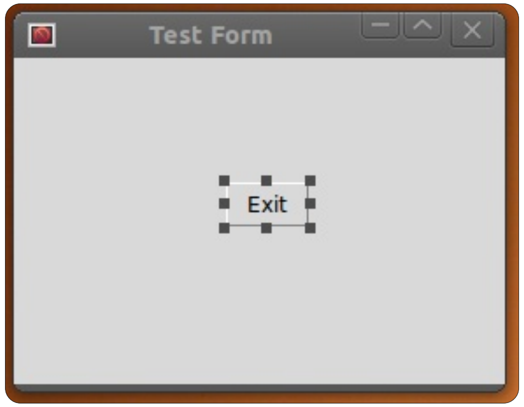
For now, let's do a button. Click on the Button button on the toolbox, and then click somewhere on the main form.



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.

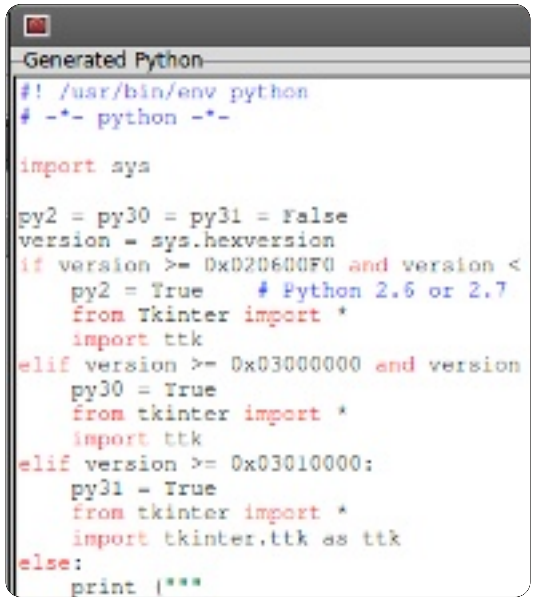
Next click in the main form someplace where the button isn't. The attribute editor form now shows the attributes for the main form. Find the "title" field and change this from "New Toplevel 1" to "Test Form".



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.

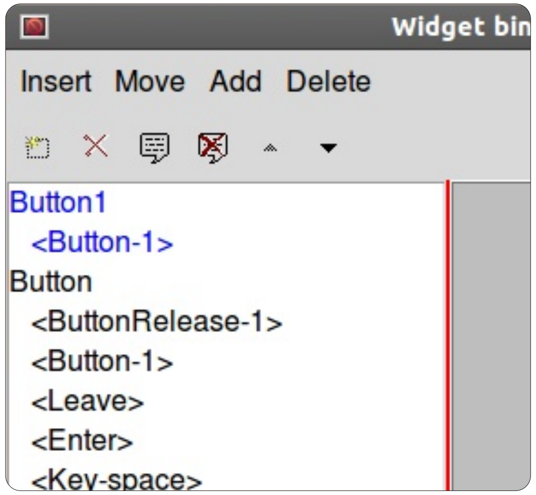
Page has generated (as the name suggests) our python code for us and placed it in a window for us to view. At the bottom of this form, are three buttons...Save, Run, and Close.



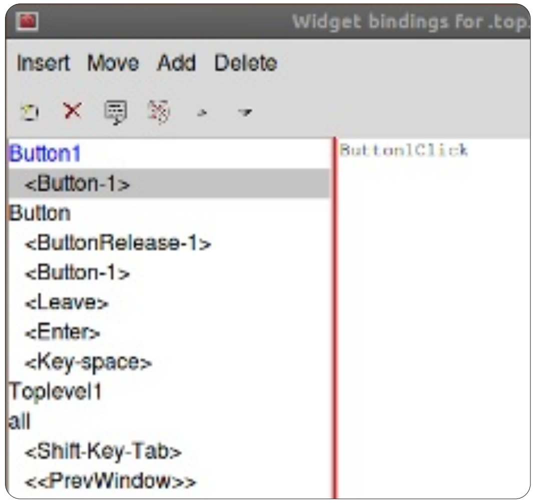
```
#! /usr/bin/env python
# -*- python -*-

import sys

py2 = py30 = py31 = False
version = sys.hexversion
if version >= 0x020600F0 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.

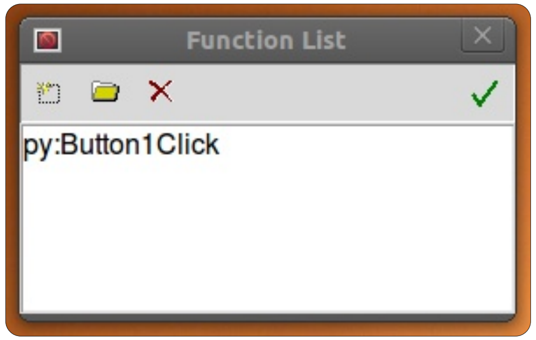


The first on the left allows you to create a new binding. Click on "Button-1". This allows us to enter the binding for the left mouse button. In the window on the right, type "Button1Click".



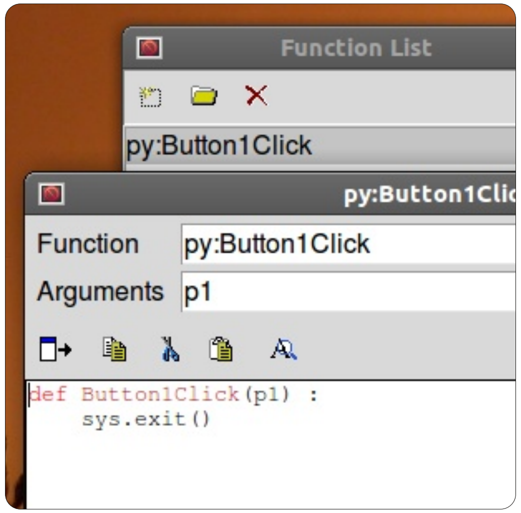
Save and generate the python code again. Scroll down in the Python Console to the bottom of the file. Above the "class Test_Form" code is the function we just asked to be created. Notice that at this point, it simply is passed. Look further down and you'll see the code that creates and controls our button. 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()
```



Function: py:Button1Click
Arguments: p1

```
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...

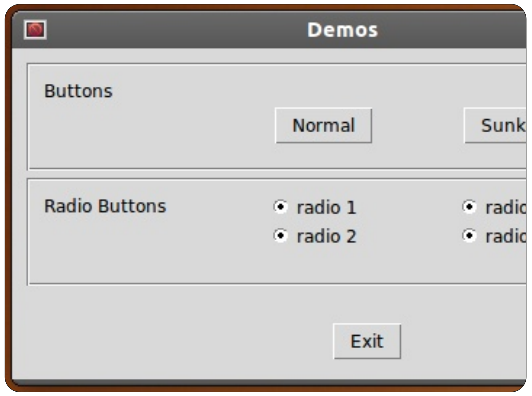
```
self.Button1.bind('<Button-
1>',Button1Click)
```

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.

Written by Greg D. Walters

Beginning Python - Part 31

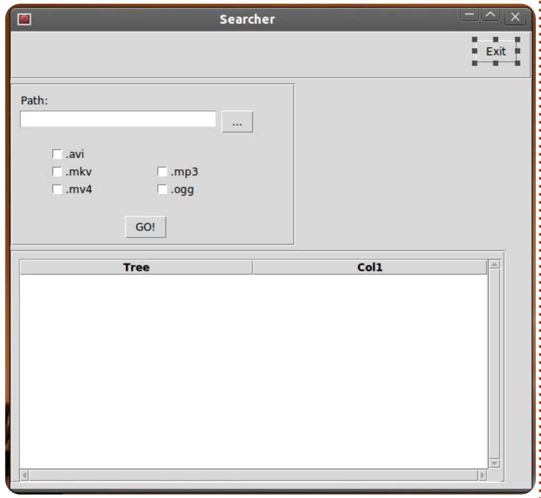
After our last meeting you should have a fairly good idea of how to use Page. If not, please read last month's article. We'll continue this time by creating a file list application with a GUI. The goal here is to create a GUI application that will recursively walk through a directory, looking for files 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
```

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.

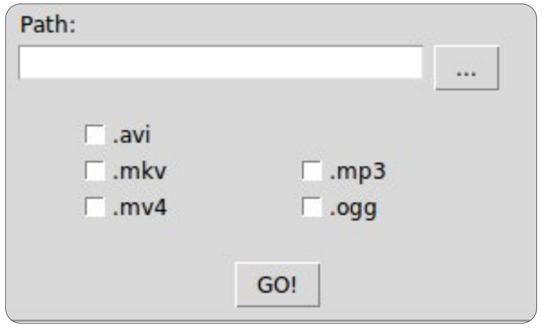
```
Alias: btnExit
Text: Exit
```

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...

contents ^

```
x
x   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

contents ^

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)
```

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 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

```
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")
```

```
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()))
```

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 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.

```
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()
```