Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

PROGRAMMING SERIES SPECIAL EDITION

PROGRAMMING SERIES SPECIAL EDITION

PROGRAM IN PYTHON

Volume Four

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.

Please note: this Special Edition is provided with absolutely no warranty whatsoever; neither the contributors nor Full Circle Magazine accept any responsibility or liability for loss or damage resulting from readers choosing to apply this content to theirs or others computers and equipment.

Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

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 22-26** from issues #48 through #52; nothing fancy, just the facts.

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

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

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

Program In Python - Part 22

Correction

Last month, in part 21, you were told to save what you have as "PlaylistMaker.glade", but, in the code, it was referred to as: "playlistmaker.glade". I'm sure you noticed that one has capitals and the other does not. The code will run only if you use both the call and file name with, or both without, the capitals.

To start off on the right foot, you need to have the playlistmaker.glade and playlistmaker.py from last month. If you don't, jump over to the last issue and get the goodies. Before we get to the code, let's take a look at what a playlist file is. There are multiple versions of play lists, and they all have different extensions. The one we will be creating will be a *.m3u type playlist. In its simplest form, it's just a text file that starts with "#EXTM3U", and then has an entry for each song file you want to play - including the full path. There's also an extension that can be added before each entry that includes the length of the song, the album name the song comes from, the track number, and the song name. We'll bypass the extension for now and just concentrate on the basic version.

Here is an example of a M3U playlist file..

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

All path names are relative to the location of the playlist file.

OK...now let's get to coding. Shown right is the opening of the source code from last month.

Now, we need to create an event handler routine for each of our events that we have set up. Notice that on_MainWindow_destroy and

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

next the class definition

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

and the main routine

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

Next, we have our dictionary which should go after the __init__ routine.

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

on_tbtnQuit_clicked are already done for us, so we need to have only 10 more (shown top right). Just make stubs for now.

We'll modify these stubbed routines in a few minutes. For now, this should get us up and running with an application, and we can test things as we go. But, we need to add one more line to the __init__ routine before we can run the app. After the self.wTree line, add...

```
self.SetEventDictionary()
```

Now, you can run the application, see the window, and click the Quit toolbar button to exit the application properly. Save the code as "playlistmaker-1a.py" and give it a try. Remember to save it in the same folder as the glade file we created last time, or copy the glade file into the folder you saved this code in.

We also need to define a few variables for future use. Add these after the SetEventDictionary call in the __init__ function.

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

Now, we will create a function that allows us to display a popup dialog box whenever we need to give some information to our user. There is a built-in set of routines that we will use, but we'll make a routine of our own to make it easier for us. It is the gtk.MessageDialog routine, and the syntax is as follows...

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

Some discussion is needed before we go too much further. The message type can be one of the following...

```
GTK_MESSAGE_INFO - Informational message
GTK_MESSAGE_WARNING - Nonfatal warning message
GTK_MESSAGE_QUESTION - Question requiring a choice
GTK_MESSAGE_ERROR - Fatal error message
```

And the button types are...

```
GTK_BUTTONS_NONE - no buttons at all
GTK_BUTTONS_OK - an OK button
GTK_BUTTONS_CLOSE - a Close button
GTK_BUTTONS_CANCEL - a Cancel button
GTK_BUTTONS_YES_NO - Yes and No buttons
GTK_BUTTONS_OK_CANCEL - OK and Cancel Buttons
```

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

Normally, you would use the following code , or similar, to create the dialog, display it, wait for a response, and then destroy it.

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

However, if you want to display a message box to the user more than once or twice, that's a LOT of typing. The general rule of thumb is that if you write a series of lines-of-code more than once or twice, it's usually better to create a function and then call that. Think of it this way: If we want to display a message dialog to the user, say ten times in your application, that's 10 X 3 (or 30) lines of code. By making a function to do this for us (using the example I just presented), we would have 10 + 3 (or 13) lines of code to write. The more we call a dialog, the less code we actually have to type, and the more readable our code is. Our

function (top right) will allow us to call any of the four message dialog types with just one routine using different parameters.

This is a very simple function that we would then call like this...

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

Notice that if we choose to use the MESSAGE_QUESTION type of dialog, there are two possible responses that will be returned by the message dialog - a "Yes" or a "No". Whichever button the user clicks, we will receive the information back in our code. To use the question dialog, the call would be something like this...

```
response = self.MessageBox("question","Are you sure you want to do this now?")

if response == gtk.RESPONSE_YES:

   print "Yes was clicked"

elif response == gtk.RESPONSE_NO:

   print "NO was clicked"
```

You can see how you can check the value of the button returned. So now, replace the "pass" call in each of our event handler routines with something like that shown below right.

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

We won't keep it like this, but this gives you a visual indication that the buttons work the way we want. Save the code now as "playlistmaker-1b.py", and test your program. Now we are going to create a function to set our widget references. This routine is going to be called only once, but it will make our code much more manageable and readable. Basically, we want to create local variables that reference the widgets in our glade window - so we can make calls to them whenever (if ever) we need

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

to. Put this function (above right) below the SetEventDictionary function.

Please notice that there is one thing that isn't referenced in our routine. That would be the treeview widget. We'll make that reference when we set up the treeview itself. Also of note is the last line of our routine. In order to use the status bar, we need to refer to it by its context id. We'll be using this later on.

Next, let's set up the function that displays the “about” dialog when we click the About toolbar button. Again, there is a built-in routine to do this provided by the GTK library. Put this after the MessageBox function. Here's the code, below right.

Save your code and then give it a try. You should see a pop-up box, centered in our application, that displays everything we have set. There are more attributes that you can set for the about box (which can be found at http://www.pygtk.org/docs/pygtk/class-gtkaboutdialog.html), but these are what I would consider a minimum set.

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

and then add a call to it right after the self.SetEventDictionary() call in the __init__ function.

```
        self.SetWidgetReferences()
```

Before we go on, we need to discuss exactly what will happen from here. The general idea is that the user will click on the "Add" toolbar button, we'll pop up a file dialog box to allow them to add files to the playlist, and then display the file information into our treeview widget. From there, they can add more files, delete single file entries, delete all file entries, move a file entry up, down, or to the top or down to the bottom of the

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

Now, comment out (or simply remove) the messagebox call in the on_tbtnAbout_clicked routine, and replace it with a call to the ShowAbout function. Make it look like this.

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

treeview. Eventually, they'll set the path that the file will be saved to, provide a filename with a "m3u" extension, and click the save file button. While this seems simple enough, there's a lot that happens behind the scenes. The magic all happens in the treeview widget, so let's discuss that. This will get pretty deep, so you might want to read carefully, since an understanding of this will keep you from making mistakes later on.

A treeview can be something as simple as a columnar list of data like a spreadsheet or database representation, or it could be more complex like a file-folder listing with parents and children, where the folder would be the parent and the files in that folder would be the children, or something even more complex. For this project, we'll use the first example, a columnar list. In the list, there will be three columns. One is for the name of the music file, one is for the extension of the file (mp3, ogg, wav, etc) and the final column is for the path. Combining this into a string (path, filename, extension) gives us the entry into the playlist we will be writing. You could, of course, add more columns as you wish, but for now, we'll deal with just three.

A treeview is simply a visual storage container that holds and displays a model. The model is the actual "device" that holds and manipulates our data. There are two different pre-defined models that are used with a treeview, but you can certainly create your own. That having been said, for 98% of your work, one of the two pre-defined models will do what you need. The two types are GTKListStore and GTKTreeStore. As their names suggest, the ListStore model is usually used for lists, the TreeStore is used for Trees. For our application, we will be using a GTKListStore. The basic steps are:

- Create a reference to the TreeView widget.
- Add the columns.
- Set the type of renderer to use.
- Create the ListStore.
- Set the model attribute in the Treeview to our model.
- Fill in the data.

The third step is to set up the type of renderer the column will use to display the data. This is simply a routine that is used to draw the data into the tree model. There are many different cell renderers that come with GTK, but most of the ones that you would normally use include GtkCellRenderText and GtkCellRendererToggle.

So, let's create a function (shown above) that sets up our TreeView widget. We'll call it SetupupTreeview. First we'll define some variables for our columns, set the variable reference of the TreeView itself, add the columns, set up the ListStore, and set the model. Here's the code for the function. Put it after the SetWidgetReferences function.

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

The variables cFName, cFType and cFPath define the column numbers. The variables sFName, sFType and sFPath will hold the column names in our displayed view. The seventh line sets the variable reference of the treeview widget as named in our glade file.

Next we call a routine (next page, top right), which we'll create in just a moment, for each column we want. Then we define our GTKListStore with three text fields, and finally set the model attribute of our TreeView widget to our GTKListStore. Let's create the AddPlaylistColumn function next. Put it after the SetupTreeview function.

Each column is created with this

function. We pass in the title of the column (what's displayed on the top line of each column) and a columnID. In this case, the variables we set up earlier (sFName and cFname) will be passed here. We then create a column in our TreeView widget giving the title, what kind of cell renderer it will be using, and, finally, the id of the column. We then set the column to be resizable, set the sort id, and finally append the column into the TreeView.

Add these two functions to your code. I choose to put them right after the SetWidgetReferences function, but you can put it anywhere within the PlayListCreator class. Add the following line after the call to SetWidgetReferences() in the __init__ function to call the function.

```
self.SetupTreeview()
```

Save and run your program, and you will see that we now have three columns with headers in our TreeView widget.

There are so many things left to do. We have to have a way to get the music filenames from the user and put them into the TreeView as rows of data. We have to create our Delete, ClearAll, movement functions, save routine, and file path routines, plus a few "pretty" things that will make our application look more professional. Let's start with the Add routine. After all, that's the first button on our toolbar. When the user clicks the Add button, we want to pop up a "standard" open-file dialog that allows for multiple selections. Once the user has made their selection, we then want to take this data and add it into the treeview, as I stated above. So the first logical thing to do is work on the File Dialog. Again, GTK provides us a way to call a "standard" file dialog in code. We could hard code this as just lines in the on_tbtnAdd_clicked event handler, but let's make a separate class to handle this. While we are at it, we can make this class handle not only a file OPEN dialog, but a folder SELECT dialog as well. As before with the MessageBox function, you can pull this into a snippet file that has all kinds of reusable routines for later use.

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

We'll start by defining a new class called FileDialog which will have only one function called ShowDialog. That function will take two parameters, one called 'which' (a '0' or a '1'), that designates whether we are creating an open-file or select-folder dialog, and the other is the path that should be used for the default view of the dialog called CurrentPath. Create this class just before our main code at the bottom of the source file.

```
class FileDialog:

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

The first part of our code should be an IF statement

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

Before going any further, let's explore how the file/folder dialog is actually called and used. The syntax of the dialog is as follows

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

and returns a dialog object. Our first line (under if which == 0) will be the line shown below.

As you can see, the title is "Select files to add...", the parent is set to None. We are requesting a File Open type dialog (action), and we want a Cancel and an Open button, both using "stock" type icons. We

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

are also setting the return codes of gtk.RESPONSE_CANCEL and gtk.RESPONSE_OK for when the user makes their selections. The call for our Folder Chooser under the Else clause is similar.

Basically, the only thing that changed between the two definitions are the title (shown above right) and the action type. So our code for the class should now be the code shown middle right.

These set the default response to be the OK button, and then to turn on the multiple select feature so the user can select (you guessed it) multiple files to add. If we didn't set this, the dialog would only allow one file to be selected at a time, since set_select_multiple is set to False by default. Our next lines are setting the current path, and then displaying the dialog itself. Before we type in the code, let me explain why we want to deal with the current path. Every time you pop up a file dialog box, and you DON'T set a path, the default is to the folder where our application resides. So, let's say that the music files that the user would be looking for are in /media/music_files/, and are then broken down by genre, and further by artist, and further by album. Let's further assume that the user has installed our application in /home/user2/playlistmaker. Each time we pop up the dialog, the starting folder would be /home/user2/playlistmaker. Quickly, the user would become frustrated by this, wanting the last folder he was in to be the starting folder next time. Make sense? OK. So, bottom right are our next lines of code.

Here we check the responses sent back. If the user clicked the 'Open' button which sends back a gtk.RESPONSE_OK, we get the name or names of the files the user selected, set the current path to the folder we are in, destroy the dialog, and then return the data back to the calling routine. If, on the other hand, the user clicked on the 'Cancel' button, we simply destroy the dialog. I put the print

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

```
class FileDialog:
    def ShowDialog(self,which,CurrentPath):
        if which == 0: #file chooser
            #gtk.FileChooserDialog(title,parent,action,buttons,backend)
            dialog = gtk.FileChooserDialog("Select files to add...",None,
                                   gtk.FILE_CHOOSER_ACTION_OPEN,
                                   (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
                                    gtk.STOCK_OPEN, gtk.RESPONSE_OK))
        else:           #folder chooser
            dialog = gtk.FileChooserDialog("Select Save Folder..",None,
                                    gtk.FILE_CHOOSER_ACTION_SELECT_FOLDER,
                                   (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
                                    gtk.STOCK_OPEN, gtk.RESPONSE_OK))
```

The next two lines will be (outside of the IF/ELSE statement)...

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

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

Next, we need to handle the response from the dialog.

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

statement in there just to show you that the button press worked. You can leave it or take it out. Notice that when we return from the Open button part of the routine, we are returning two sets of values. 'fileselection' is a list of the files selected by the user, as well as the CurrentPath.

In order to get the routine to do something, add the following line under the on_tbtnAdd_click routine...

```
fd = FileDialog()

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

Here we retrieve the two return values that are sent from our return call. For now, add the following code to see what the information returned will look like.

```
for f in selectedfiles:

   print "User selected %s" % f

print "Current path is %s" % self.CurrentPath
```

When you run the program, click on the 'Add' button. You'll see the file dialog. Now move to somewhere where you have some files and select them. You can hold down the [ctrl] key and click on multiple files to select them individually, or the [shift] key to select multiple contiguous files. Click on the 'Open' button, and look at the response in your terminal window. Please note that if you click on the 'Cancel' button right now, you'll get an error message. That's because the above code assumes that there are no files selected. Don't worry about that right now - we'll handle that in a little bit. I just wanted to let you see what comes back if the 'Open' button is pressed. One thing we should do is add a filter to our file-open dialog. Since we expect the user to normally select music files, we should (1) give the option to display only music files, and (2) give the option to show all files just-in-case. We do this by using the filefilter attributes of the dialog. Here's the code for that which should go in the which == 0 section right after the dialog set line.

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

We are setting up two "groups", one for music files (filter.set_name("Music Files")), and the other for all files. We use a pattern to define the types of files we want. I have defined three patterns, but you can add or delete any that you wish. I put the music filter first, since that's what we will assume the user is going to be mainly concerned with. So the steps are...

- Define a filter variable.
- Set the name.
- Add a pattern.
- Add the filter to the dialog.

You can have as many or as few filters as you wish. Also notice that once you have added the filter to the dialog, you can re-use the variable for the filter.

Back in the on_tbtnAdd_clicked routine, comment out the last lines we added and replace them with this one line.

```
self.AddFilesToTreeview(selectedfiles)
```

so our routine now looks like the code shown on the next page.

So, when we get the response back from file dialog, we will send the list containing the selected files to this routine. Once here, we set up a counter variable (how many files we are adding), then parse the list. Remember that each entry contains the fully qualified filename with path and extension. We'll want to split the filename into path, filename, and extension. First we get the very last 'period' from the filename and assume that is the beginning of the extension and assign its position in the string to extStart. Next we find the very last '/' in the filename to determine the beginning of the filename. Then we break up the string into extension, filename and file path. We then stuff these values into a list named 'data' and append this into our playlist ListStore. We increment the counter since we have done all the work. Finally we increment the variable RowCount which holds the total number of rows in our ListStore, and then we print a message to the status bar.

Now you can run the application

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

We now have to create the function that we just put the call to. Put this function after the on_btnSavePlaylist_clicked routine.

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

and see the data in the TreeView.

As always, the full code can be found at http://pastebin.com/JtrhuE71.

Next time, we'll finalize our application, filling in the missing routines, etc.

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

Program In Python - Part 23

This time, we are going to finish our playlistmaker program. Last time, we got a good bit done, but we left some things incomplete. We can't save the playlist, we don't have the movement functions done, we can't select the file path to store the file in, and so on. However, there are a few things that we need to do before we start coding. First, we need to find an image for the logo for our application in the about box, and for when the application is minimized. You can dig around in the /usr/share/icons folder for an icon you like, or you can go on the web and get one, or create one yourself. Whatever you get, put it into your code folder with the glade file and the source code from last month. Name it logo.png. Next, we need to open the glade file from last month and make a few changes.

First, using the MainWindow, go to the General tab, and scroll down until you find Icon. Using the browse tool, find your icon and select that. Now the text box should contain "logo.png". Next, in the hierarchy box, select treeview1, go to the signal tab, and, under GtkTreeView | cursor-changed, add a handler for on_treeview1_cursor_changed. Remember, as I told you last month, to click off that to make the change stick. Finally, again in the hierarchy box, select txtFilename, and go to the signal tab. Scroll down until you find 'GtkWidget', and scroll down further until you get to 'key-press-event'. Add a handler for 'on_txtFilename_key_press_event'. Save your glade project and close glade.

Now it's time to complete our project. We'll start from where we left off using last month's code.

The first thing I want to do is modify the code in class FileDialog. If you remember from last time, if the user clicked the 'Cancel' button, there was an error raised. We will fix that first. At the end of the routine, you have the code shown above.

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

Notice that we aren't returning anything. This is what caused the error. So to fix this, we want to add the following line of code after the dialog.destroy() line.

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

This will keep the error from happening. Next, let's add the text box event handler we created in glade. To our dictionary, add the following line.

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

As you remember, this creates a function to handle the keypress event. We'll next create the function.

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

You might imagine, this simply looks at the value of each key that is pressed when the user is in the txtFilename text box, and compares it to the value 65293, which is the code that is assigned to the return key (enter key). If it matches, then it calls the SavePlaylist function. The user doesn't have to even click the button.

Now on to new code. Let's deal with the toolbar button ClearAll. When the user clicks this button, we want the treeview and the ListStore to be cleared. This is a simple one-liner that we can put

into the on_tbtnClearAll_clicked routine.

```
def
on_tbtnClearAll_clicked(self,
widget):

    self.playList.clear()
```

We are simply telling the playList ListStore to clear itself. That was easy. Now we'll deal with the Delete toolbar button. Much harder, but once we get into it, you'll understand.

First we have to discuss how we get a selection from the treeview widget and the ListStore. This is complicated, so go slowly. In order to get data back from the ListStore, we first have to get a gtk.TreeSelection which is a helper object that manages the selection within a treeview. Then we use that helper object to retrieve the model type, and an iterator that contains the selected rows.

I know that you are thinking "What the heck is an iterator?" Well you already have used them and don't even know it. Think about the following code (above right) from the AddFilesToTreeview function from last month.

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

Look at the 'for' statement portion. We use an iterator to walk through the list called FileList. Basically, in this case, the iterator simply goes through each entry in the list returning each item separately. What we are going to do is create an iterator, fill that with the selected rows in the treeview, and use that like a list. So the code (middle right) for on_tbtnDelete_clicked will be.

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

The first line creates the TreeSelection object. We use that to get the rows selected (which is only one because we didn't set the model to support multiple selections), fill that into a list called iters, and then walk it removing (like the .clear method). We also decrement the variable RowCount, and then display the number of files in the status bar.

Now, before we get to the move functions, let's deal with the save-file-path function. We'll use our FileDialog class as before. We'll do all the code (bottom right)for this in the on_btnGetFolder_clicked routine.

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

The only thing really different from before is the last line of this code. We are putting the name of the path returned by the FileDialog into the textbox that we set up previously using the set_text method. Remember that the data returned to us is in the form of a list, even though there is only one entry. That's why we use 'filepath[0]'.

Let's do the file-save function. We can safely do that before we deal with the move functions. We'll create a function called

SavePlaylist. The first thing we need to do (above right) is check to see if there is anything in the txtPath text box. Next we need to check to see if there is a filename in the txtFilename text box. For both of those instances, we use the .get_text() method of the text box.

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

Now check the values...

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

Now that we know that we have a path (fp) and a filename (fn), we can open the file, print our M3U header, and walk the playList. The path is stored (if you will remember) in column 2, the filename in column 0, and the extension in column 1. We simply (right) create a string and then write it to the file and finally close the file.

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

Lastly, we pop up a message box informing the user that the file has been saved.

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

We now need to put in a call to this routine in our on_btnSavePlaylist_clicked event handler routine.

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

Save your code and test it. Your play list should save properly and look something like the sample I gave you last month.

We can now start work on the move functions. Let's start with the Move To Top routine. Like we did when we wrote the delete function, we get the selection and then the selected row. Next we have to step through the rows to get two variables. We will call them path1 and path2. Path2, in this case will be set to 0, which is the "target" row. Path1 is the row the user has selected. We finally use the model.move_before() method to move the selected row up to row 0, effectively pushing everything down. We'll put the code (below right) directly in the on_tbtnMoveToTop_clicked routine.

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

For the MoveToBottom

function, we will use almost exactly the same code as the MoveToTop routine, but, in place of the model.move_before() method, we will use the model.move_after() method, and, instead of setting path2 to 0, we set it to self.RowCount-1. Now you understand why we have a RowCount variable. Remember the counts are zero based, so we have to use RowCount-1 (above right).

Now let's take a look at what it will take to do the MoveUp routine. Once again, it is fairly similar to the last two functions we created. This time, we get path1 which is the selected row and then assign that row number–1 to path2. Then IF path2 (the target row) is greater than or equal to 0, we use the model.swap() method (second down, right).

The same thing applies for the MoveDown function. This time however, we check to see if path2 is LESS than or equal to the value of self.RowCount-1 (third down, right).

Now let's make some changes to the abilities of our play list. In last month's article, I showed you the basic format of the play list file (bottom).

However, I did say that there was an extended format as well. In the extended format, there is an extra line that can be added to the file before each song file entry that contains extra information about the song. The format of this line is as follows...

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

You might have wondered why we included the mutagen library from the beginning since we never used it. Well, we will now. To refresh your memory, the mutagen library is for accessing ID3 tag information from inside of MP3 files. To get the full discussion about this, please refer to issue 35 of Full Circle which has my part 9 of this series. We'll create a function to deal with the reading of the MP3 file and return the Artist name, the Song Title, and the length of the song in seconds, which are the three things we need for the extended information line. Put the function after the ShowAbout function within the PlaylistCreator class (next page, top right).

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

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

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

Again, to refresh your memory, I'll walk through the code. First we

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

clear the three return variables so that if anything happens they are blank upon return. We then pass in the filename of the MP3 file we are going to look at. Next we pull the keys into (yes, you guessed it) an iterator, and walk through that iterator looking for two specific tags. They are 'TPE1' which is the artist name, and 'TIT2' which is the song title. Now, if the key doesn't exist, we would get an error, so we wrap each get call with a 'try|except' statement. We then pull the song length from the audio.info.length attribute, and return the whole shebang.

Now, we will want to modify the SavePlaylist function to support the extended information line. While we are there, let's check to see if the filename exists, and, if so, flag the user and exit the routine. Also, to make things a bit easier for the user, since we don't support any other filetype, let's automatically append the extension '.m3u' to the path and filename if it doesn't exist. First add an import line at the top of the code importing os.path between the sys import and the mutagen import (bottom right).

Just like in the AddFilesToTreeview function, we will use the 'rfind' method to find the position of the last period ('.') in the filename fn. If there isn't one, the return value is set to -1. So we check to see if the return value is -1, and, if so, we append the extension and then put the filename back in the text box just to be nice.

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

```
import os.path
```

Then, go ahead and comment out your existing SavePlaylist function and we'll replace it.

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

Up to this point, the routine is the same. Here's where the changes start.

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

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

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

Next, we want to wrap the rest of the function with an IF|ELSE clause (top right) so if the file already exists, we simply fall out of the routine. We use os.path.exists(filename) to do this check.

The rest of the code is mostly the save as before, but let's look at it anyway.

Line 2 opens the file we are going to write. Line 3 puts the M3U header in. Line 4 sets up for a walk through the playList ListStore. Line 5 creates the filename from the three columns of the ListStore. Line 6 calls GetMP3Info and stores the return values into variables. Line 7 then checks to see if we have values in all three variables. If so, we write the extended information line in line 8, otherwise we don't try. Line 9 writes the filename line as before. Line 10 closes the file gracefully, and line 11 pops up the message box letting the user know the process is all done.

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

Go ahead and save your code and give it a test drive.

At this point about the only thing that should be added would be some tool tips for our controls when the user hovers the mouse pointer over them. It adds that professional flair (below). Let's create a function to do that now.

We are using the widget references we set up earlier, and then setting the text for the tooltip via the (you guessed it) set_tooltip_text attribute. Next we need to add the call to the routine. Back in the __init__ routine, after the self.SetWidgetReferences line, add:

```
self.SetupToolTops()
```

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

Last, but certainly not least, we want to put our logo into our About box. Just like everything else there, there's an attribute for that. Add the following line to the ShowAbout routine.

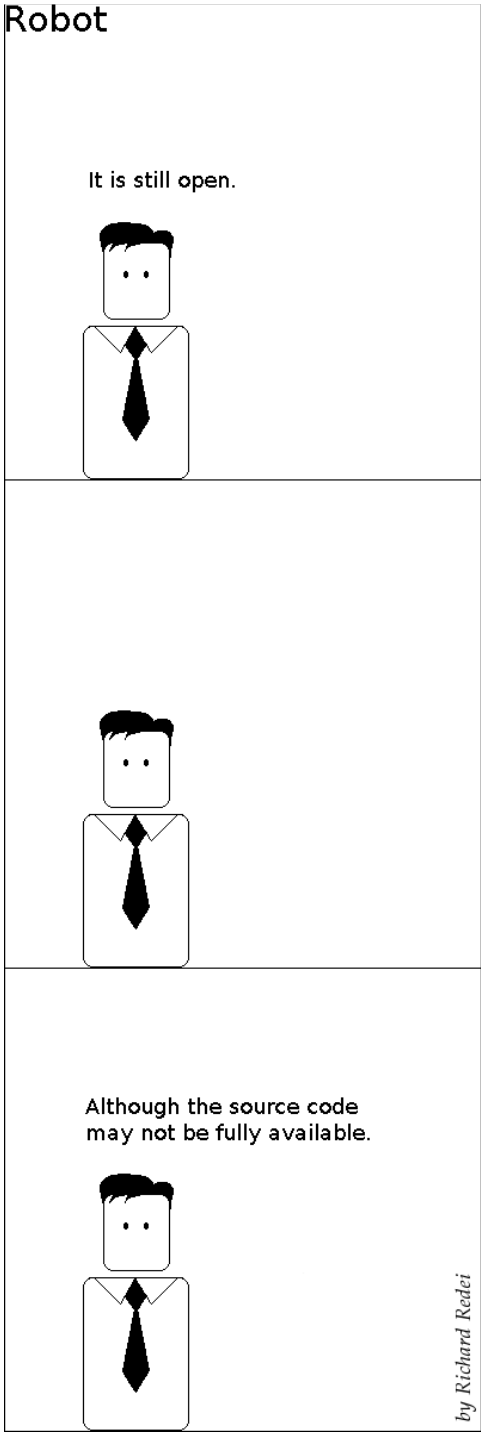
```
about.set_logo(gtk.gdk.pixbuf_new_from_file("logo.png"))
```

That's about it. You now have a fully functioning program that looks good, and does a wonderful job of creating a playlist for your music files.

The full source code, including the glade file we created last month, can be found at pastebin: http://pastebin.com/tQJizcwT

Until next time, enjoy your new found skills.

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.



Program In Python - Part 24

Written by Greg Walters

WOW! It's hard to believe that this is the 24th issue already. Two years we've been learning Python! You've come a very long way.

This time we are going to cover two topics. The first is printing to a printer, the second is creation of RTF (Rich Text Format) files for output.

Generic Printing under Linux

So let's start with printing to a printer. The idea to cover this came from an email sent by Gord Campbell. It's actually easy to do most printing from Linux, and easier than that other operating system that starts with "WIN" - and I won't deal with that OS.

As long as all you want to print is straight text, no bold, italics, font changes, etc, it's fairly easy. Here's a simple app that will print directly to your printer...

```
import os

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

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

pr.write('Print finished\n')

pr.close()
```

This is fairly easy to understand as long as you expand your mind just a bit. In the above code, 'lpr' is the print spooler. The only requirement is that we have already configured 'lpd' and that it's running. More than likely, when you use a printer under Ubuntu, it's already done for you. 'Lpd' is usually referred to as a "magic-filter" that can automatically convert different types of documents to something the printer can understand. We are going to print to the 'lpr' device/object. Think of it simply as a file. We open the file. We have to import 'os'. Then in line 2, we open the 'lpr' with write access - assigning it to the object variable 'pr'. We then do a 'pr.write' with anything we want to print. Finally (line 5) we close the file, which will send the data out to the printer.

We can also create a text file, then send it out to the printer like this...

```
import os

filename = 'dummy.file'

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

In this case, we are still using the lpr object, but we are using the 'os.system' command to basically create a command that looks to linux like we sent it from a terminal.

I'll leave you to play with this for now.

PyRTF

Now let's deal with RTF files. RTF format (that's kind of like saying PIN number since PIN stands for Personal Identification Number, so that translates to Personal-Identification-Number Number. Something from the department of redundancy department, huh?) was originally created by the Microsoft Corporation in 1987, and its syntax was influenced by the TeX typesetting language. PyRTF is a wonderful library that makes it easy to write RTF files. You have to do some planning up front on how you want your files to look, but the results will be well worth it.

> **Wow! It's hard to believe that this is the 24th issue already. Two years we've been learning Python!**

First, you need to download and install the PyRTF package. Go to http://pyrtf.sourceforge.net and get the PyRTF-0.45.tar.gz package. Save it someplace and use archive manager to unpack it. Then using terminal, go to where you unpacked it. First we need to install the package, so type "sudo python setup.py install" and it will

be installed for you. Notice there is an examples folder there. There's some good information there on how to do some advanced things.

Here we go. Let's start as we usually do, creating the stub of our program which is shown on the next page, top right.

Before going any further, we'll discuss what's going on. Line 2 imports the PyRTF library. Note that we are using a different import format than normal. This one imports everything from the library.

Our main working routine is MakeExample. We've stubbed for now. The OpenFile routine creates the file for us with the name we pass into it, appends the extension "rtf", puts it into the "write" mode, and returns a file handle.

We've already discussed the if __name__ routine before, but to refresh your memory, if we are running the program in a standalone mode, the internal variable __name__ is set to "__main__". If we call it as an import from another program, then it will just ignore that portion of the code.

Here, we create an instance of the Renderer object, call the MakeExample routine, getting the returned object doc. We then write the file (in doc) using the OpenFile routine.

Now for the meat of our worker routine MakeExample. Replace the pass statement with the code shown below.

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

def MakeExample():
    pass

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

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

Let's look at what we have done. In the first line we create an instance of Document. Then we create an instance of the style sheet. Then we create an instance of the section object and append it to the document. Think of a section as a chapter in a book. Next we create a paragraph using the Normal style. The author of PyRTF has preset this to be 11-point Arial font. We then put whatever text we want into the paragraph, append that to the section, and return our doc document.

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

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

return doc
```

That is very easy. Again, you need to plan your output fairly carefully, but nothing too onerous.

Save the program as "rtftesta.py" and run it. When it's completed, use openoffice (or LibreOffice) to open the file and look at it.

Now let's do some neat things. First, we'll add a header. Once again, the author of PyRTF has given us a predefined style called Header1. We'll use that for our header. In between the doc.Sections.append line and the p = Paragraph line, add the

following.

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

p.append('Example Heading 1')

section.append(p)
```

Change the name of the rtf file to "rtftestb". It should look like this:

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

Save this as rtftestb.py and run it. So now we have a header. I'm sure your mind is going down many roads thinking about what more can we do. Here's a list of what the author has given us as the predefined styles.

Normal, Normal Short, Heading 1, Heading 2, Normal Numbered, Normal Numbered 2. There's also a List style, which I will let you play with on your own. If you want to see more, on this and other things, the styles are defined in the file Elements.py in the distribution you installed.

> **Let's look at how to change fonts, font sizes and attributes (bold, italic, etc) on the fly.**

While these styles are good for many things, we might want to use something other than the provided styles. Let's look at how to change fonts, font sizes and attributes (bold, italic, etc) on the fly. After our paragraph and before we return the document object, insert the code shown top right, and change the output filename to rtftestc. Save the file as rtftestc.py. And run it. The new portion of our document should look like this...

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

It is also possible to provide overrides for elements of a style. For example you can change just the font size to 24 point or typeface to Impact or even more Attributes like BOLD or Italic or BOTH.

Now what have we done? Line 1 creates a new paragraph. We then start, as we did before, putting in our text. Look at the fourth line (TEXT(' size to 24 point', size = 48),). By using the TEXT qualifier, we are telling PyRTF to do something different in the middle of the sentence, which in this case is to change the size of the font (Arial at this point) to 24-point, followed by the 'size = ' command. But, wait a moment. The 'size =' says 48, and what we are printing says 24 point, and the output is actually in 24-point text. What's going on here? Well the size command is in half points. So if we want an 8-point font we have to use size = 16. Make sense?

Next, we continue the text and then change the font with the 'font =' command. Again, everything within the inline TEXT command between the single quotes is going to be affected and nothing else.

Ok. If that all makes sense, what else can we do?

We can also set the color of the text within the TEXT inline command. Like this.

```
p = Paragraph()

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

    TEXT('RED',colour=ss.Colo
urs.Red),
```

```
        ' in Red text.')

section.append(p)
```

Notice that we didn't have to restate the paragraph style as Normal, since it sticks until we change it. Also notice that if you live in the U.S., you have to use the "proper" spelling of colour.

Here are the colors that are (again) predefined: Black, Blue, Turquoise, Green, Pink, Red, Yellow, White, BlueDark, Teal, GreenDark, Violet, RedDark, YellowDark, GreyDark and Grey.

And here is a list of all the predefined fonts (in the notation you must use to set them):

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

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

section.append(p)
```

So now you must be thinking that this is all well and good, but how do we make our own styles? That's pretty easy. Move to the top of our file, and before our header line, add the following code.

```
result = doc.StyleSheet

NormalText =
TextStyle(TextPropertySet(res
ult.Fonts.CourierNew,16))

ps2 =
ParagraphStyle('Courier',Norm
alText.Copy())

result.ParagraphStyles.append
(ps2)
```

Before we write the code to actually use it, let's see what we have done. We are creating a new stylesheet instance called result. In the second line, we are setting the font to 8-point Courier New, and then "registering" the style as Courier. Remember, we have to use 16 as the size since the font size is in half-point values.

Now, before the return line at the bottom of the routine, let's include a new paragraph using the Courier style.

So now you have a new style you can use anytime you want. You can use any font in the list above and create your own styles. Simply copy the style code and replace the font and size information as you wish. We can also do this...

```
NormalText =
TextStyle(TextPropertySet(res
ult.Fonts.Arial,22,b
old=True,colour=ss.C
olours.Red))

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

result.ParagraphStyles.append
(ps2)
```

And add the code below...

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

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

section.append(p)

to print the ArialBoldRed
style.
```

Tables

Many times, tables are the only way to properly represent data in a

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

document. Doing tables in text is hard to do, and, in SOME cases, it's pretty easy in PyRTF. I'll explain this statement later in this article.

Let's look at a standard table (shown below) in OpenOffice/LibreOffice. It looks like a spreadsheet, where everything ends up in columns.

Rows go left to right, columns go down. Easy concept.

Let's start a new application and call it rtfTable-a.py. Start with our standard code (shown on the next page) and build from there.

We don't need to discuss this since it's basically the same code that we used before. Now, we'll flesh out the TableExample routine. I'm basically using part of the example file provided by the author of PyRTF. Replace the pass statement in the routine with the following code...

```
doc = Document()

ss = doc.StyleSheet

section = Section()

doc.Sections.append(section)
```

This part is the same as before, so we'll just gloss over it.

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

      TabPS.DEFAULT_WIDTH * 3,

      TabPS.DEFAULT_WIDTH * 3)
```

This line (yes, it's really one line, but is broken up for easy viewing) creates our basic table. We are creating a table with 3 columns, the first is 7 tabs wide, the second and third are three tabs wide. We don't have to deal with tabs alone, you can enter the widths in twips. More on that in a moment.

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

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

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

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

Here we are setting the data that goes into each cell in the first row.

```
c1 =
Cell(Paragraph(ss.ParagraphSt
yles.Heading2,'Heading2
Style'))

c2 =
Cell(Paragraph(ss.ParagraphSt
yles.Normal,'Back to Normal
Style'))

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

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

This group of code sets the data for row number two. Notice we can set a different style for a single or multiple cells.

```
c1 =
Cell(Paragraph(ss.ParagraphSt
yles.Heading2,'Heading2
Style'))

c2 =
Cell(Paragraph(ss.ParagraphSt
yles.Normal,'Back to Normal
Style'))

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

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

This sets the final row.

```
section.append(table)

return doc
```

This appends the table into the section and returns the document for printing.

Save and run the app. You'll notice that everything is about what you would expect, but there is no border for the table. That can make things difficult. Let's fix that. Again, I'll mainly use code from the

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

from PyRTF import *

def TableExample():
    pass

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

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

example file provided by the PyRTF author.

Save your file as rtftable-b.py. Now, delete everything between 'doc.Sections.append(section)' and 'return doc' in the TableExample routine, and replace it with the following...

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

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

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

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

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

Here we are setting up the edge and frame definitions for borders and frames.

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

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

c2 = Cell( Paragraph( 'R1C2'
) )

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

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

In row one, the cells in column one (thin frame) and column 3 (thick frame) will have a border around them.

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

c2 = Cell( Paragraph( 'R2C2'
) )

c3 = Cell( Paragraph( 'R2C3'
) )

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

None of the cells will have a border in the second row.

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

c2 = Cell( Paragraph( 'R3C2'
) )

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

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

Once again, cells in column 1 and three have a mixed frame in row three.

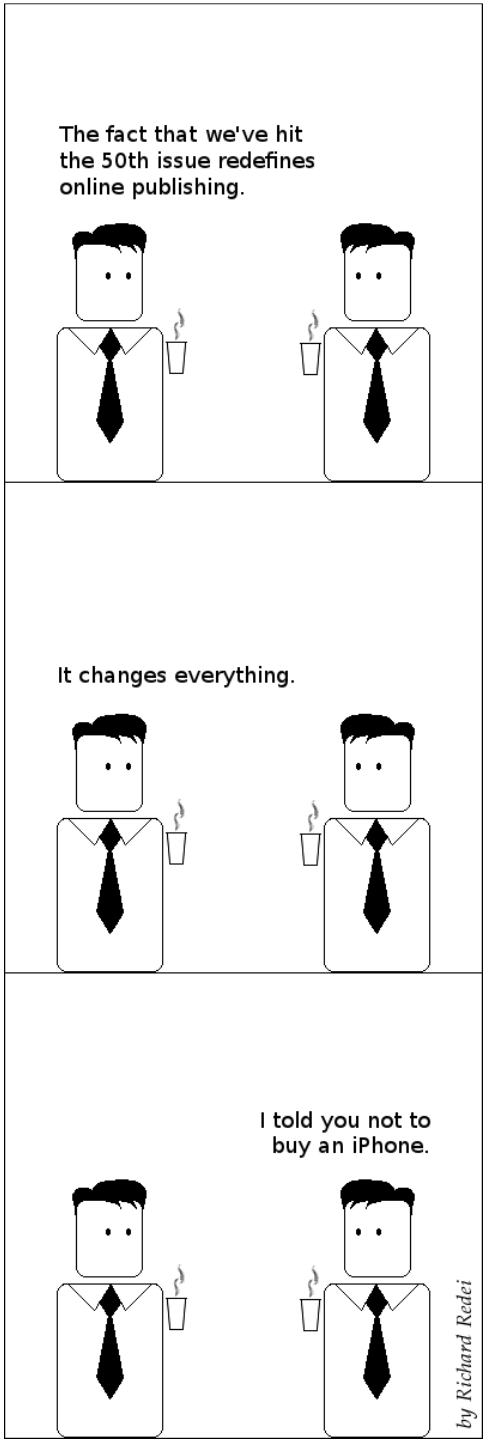
```
section.append( table )
```

So. You have just about everything you need to create, through code, RTF documents.

See you next time!

Source code can be found at pastebin as usual. The first part can be found at http://pastebin.com/3Rs7T3D7 which is the sum of rtftest.py (a-e), and the second rtftable.py (a-b) is at http://pastebin.com/XbaE2uP7.

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.



Program In Python - Part 25

Written by Greg Walters

A number of you have commented about the GUI programming articles and how much you've enjoyed them. In response to that, we will start taking a look at a different GUI toolkit called Tkinter. This is the "official" way to do GUI programming in Python. Tkinter has been around for a long time, and has gotten a pretty bad rap for looking "old fashioned". This has changed recently, so I thought we'd fight that bad thought process.

PLEASE NOTE – All of the code presented here is for Python 2.x only. In an upcoming article, we'll discuss how to use tkinter in Python 3.x. If you MUST use Python 3.x, change the import statements to "from tkinter import *".

A Little History And A Bit Of Background

Tkinter stands for "**Tk inter**face". Tk is a programming language all on its own, and the Tkinter module allows us to use the GUI functions there. There are a number of widgets that come natively with the Tkinter module. Some of them are Toplevel (main window) container, Buttons, Labels, Frames, Text Entry, CheckButtons, RadioButtons, Canvas, Multiline Text entry, and much more. There are also many modules that add functionallity on top of Tkinter. This month, we'll focus on four widgets. Toplevel (from here I'll basically refer to it as the root window), Frame, Labels, and Buttons. In the next article, we'll look at more widgets in more depth.

Basically, we have the Toplevel container widget which contains (holds) other widgets. This is the root or master window. Within this root window, we place the widgets we want to use within our program. Each widget, other than the Toplevel root widget container, has a parent. The parent doesn't have to be the root window. It can be a different widget. We'll explore that next month. For this month, everything will have a parent of the root window.

In order to place and display the child widgets, we have to use what's called "geometry management". It's how things get put into the main root window. Most programmers use one of three types of geometry management, either Packer, Grid, or Place management. In my humble opinion, the Packer method is very clumsy. I'll let you dig into that on your own. The Place management method allows for extremely accurate placement of the widgets, but can be complicated. We'll discuss the Place method in a future article set. For this time, we'll concentrate on the Grid method.

Think of a spreadsheet. There are rows and columns. Columns are vertical, rows are horizontal. Here's a simple text representation of the cell addresses of a simple 5-column by 4-row grid (above right).

```
    COLUMNS - >
ROWS | 0,0 | 1,0 | 2,0 | 3,0 | 4,0 |
  |  | 0,1 | 1,1 | 2,1 | 3,1 | 4,1 |
     | 0,2 | 1,2 | 2,2 | 3,2 | 4,2 |
     | 0,3 | 1,3 | 2,3 | 3,3 | 4,3 |
```

So parent has the grid, the widgets go into the grid positions. At first glance, you might think that this is very limiting. However, widgets can span multiple grid positions in either the column direction, the row direction, or both.

Our First Example

Our first example is SUPER simple (only four lines), but shows a good bit.

```
from Tkinter import *

root = Tk()

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

root.mainloop()
```

Now, what's going on here? Line one imports the Tkinter

library. Next, we instantiate the Tk object using root. (Tk is part of Tkinter). Here's line three.

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

We create a button called button, set its parent to the root window, set its text to "Hello FullCircle," and set it into the grid. Finally, we call the window's main loop. Very simple from our perspective, but there's a lot that goes on behind the scenes. Thankfully, we don't need to understand what that is at this time.

Run the program and let's see what happens. On my machine the main window shows up at the lower left of the screen. It might show up somewhere else on yours. Clicking the button doesn't do anything. Let's fix that in our next example.

Our Second Example

This time, we'll create a class called App. This will be the class that actually holds our window. Let's get started.

```
from Tkinter import *

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

This is the import statement for the Tkinter library.

We define our class, and, in the __init__ routine, we set up our widgets and place them into the grid.

The first line in the __init__ routine creates a frame that will be the parent of all of our other widgets. The parent of the frame is the root window (Toplevel widget). Next we define a label, and two buttons. Let's look at the label creation line.

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

We create the label widget and call it self.lblText. That's inherited from the Label widget object. We set its parent (frame), and set the text that we want it to display (text = "this is a label widget"). It's that simple. Of course we can do much more than that, but for now that's all we need. Next we set up the two Buttons we will use:

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

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

We name the widgets, set their parent (frame), and set the text we want them to show. Now btnQuit has an attribute marked fg which we set to "red". You might have guessed this sets the foreground color or text color to the color red. The last attribute is to set the callback command we want to use when the user clicks the button. In the case of btnQuit, it's frame.quit, which ends the program. This is a built in function, so we don't need to actually create it. In the case of btnHello, it's a routine called self.SaySomething. This we have to create, but we have a bit more to go through first.

We need to put our widgets into the grid. Here's the lines again:

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

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

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

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

First, we assign a grid to the frame. Next, we set the grid attribute of each widget to where we want the widget to go. Notice the columnspan line for the label (self.lblText). This says that we

want the label to span across two grid columns. Since we have only two columns, that's the entire width of the application. Now we can create our callback function:

```
def SaySomething(self):

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

This simply prints in the terminal window the message "Hello to FullCircle Magazine Readers!!"

Finally, we instantiate the Tk class - our App class - and run the main loop.

```
root = Tk()

app = App(root)

root.mainloop()
```

Give it a try. Now things actually do something. But again, the window position is very inconvenient. Let's fix that in our next example.

Our Third Example

Save the last example as example3.py. Everything is exactly the same except for one line. It's at the bottom in our main routine calls. I'll show you those lines with our new one:

```
root = Tk()

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

app = App(root)

root.mainloop()
```

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

What this does is force our initial window to be 150 pixels wide and 75 pixels high. We also want the upper left corner of the window to be placed at X-pixel position 550 (right and left) and the Y-pixel position at 150 (top to botton). How did I come up with these numbers? I started with some reasonable values and tweaked them from there. It's a bit of a pain in the neck to do it this way, but the results are better than not doing it at all.

Our Fourth Example - A Simple Calculator

Now, let's look at something a bit more complicated. This time, we'll create a simple "4 banger" calculator. If you don't know, the phrase "4 banger" means four functions: Add, Subtract, Multiply, and Divide. Right is what it looks like in simple text form.

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

```
from Tkinter import *

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

We'll dive right into it and I'll explain the code (middle right) as we go.

Outside of the geometry statement, this (left) should be pretty easy for you to understand by now. Remember, pick some reasonable values, tweak them, and then move on.

We begin our class definition and set up our __init__ function. We set up three variables as follows:

- CurrentValue – Holds the current value that has been input into the calculator.
- HolderValue – Holds the value that existed before the user clicks a function key.

• CurrentFunction – This is simply a “bookmark” to note what function is being dealt with.

Next, we define the CurrentDisplay variable and assign it to the StringVar object. This is a special object that is part of the Tkinter toolkit. Whatever widget you assign this to automatically updates the value within the widget. In this case, we will be using this to hold whatever we want the display label widget to... er... well... display. We have to instantiate it before we can assign it to the widget. Then we use the built in 'set' function. We then define a boolean variable called DecimalNext, and a variable DecimalCount, and then call the DefineWidgets function, which creates all the widgets, and then call the PlaceWidget function, which actually places them in the root window.

```
def
DefineWidgets(self,master):

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

Now, we have already defined a label earlier. However, this time we are adding a number of other attributes. Notice that we aren't using the 'text' attribute. Here, we assign the label to the parent (master), then set the anchor (or, for our purposes, justification) for the text, when it gets written. In this case, we are telling the label to justify all text to the east or on the right side of the widget. There is a justify attribute, but that's for multiple lines of text. The anchor attribute has the following options... N, NE, E, SE, S, SW, W, NW and CENTER. The default is CENTER. You should think compass points for these. Under normal circumstances, the only really usable values are E (right), W (left), and Center.

Next, we set the relief or visual style of the label. The “legal” options here are FLAT, SUNKEN, RAISED, GROOVE, and RIDGE. The default is FLAT if you don't specify anything. Feel free to try the other combinations on your own after we're done. Next, we set the background (bg) to white in order to set it off from the rest of the window a bit. We set the height to 2 (which is two text lines high, not in pixels), and finally assign the variable we just defined a moment ago (self.CurrentDisplay) to the textvariable attribute. Whenever the value of self.CurrentDisplay changes, the label will change its text to match automatically.

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

Shown above, we'll create some of the buttons.

I've shown only 4 buttons here. That's because, as you can see, the code is almost exactly the same. Again, we've created buttons earlier in this tutor, but let's take a closer look at what we are doing here.

We start by defining the parent (master), the text that we want on the button, and the width and height. Notice that the width is in characters and the height is in text lines. If you were doing a graphic in the button, you would use pixels to define the height and width. This can become a bit confusing until you get your head firmly wrapped around it. Next, we are setting the bind attribute. When we did the buttons in the previous examples, we used the 'command=' attribute to define what function should be called when the user clicks the button. This time, we are using the '.bind' attribute. It's almost the same thing, but this is an easier way to do it, and to pass information to the callback routine that is static. Notice that here we are using '<ButtonRelease-1>' as the trigger for the bind. In this case, we want to make sure that it's only after the user clicks AND releases the left mouse button that we make our callback. Lastly, we define the callback we want to call, and what we are going to pass to it. Now,

those of you who are astute (which is each and every one of you) will notice something new. The 'lambda e:' call.

In Python, we use Lambda to define anonymous functions that will appear to interpreter as a valid statement. This allows us to put multiple segments into a single line of code. Think of it as a mini function. In this case, we are setting up the name of the callback function and the value we want to send as well as the event tag (e:). We'll talk more about Lambda in a later article. For now, just follow the example.

I've given you the first four buttons. Copy and paste the above code for buttons 5 through 9 and button 0. They are all the same with the exception of the button name and the value we send the callback. Next steps are shown right.

The only thing that hasn't been covered before are the columnspan and sticky attributes. As I mentioned before, a widget can span more than one column or row. In this case, we are "stretching" the label widget across all four columns. That's

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

The btnDash sets the value to the absolute value of the value entered. 523 remains 523 and -523 becomes 523. The btnDot button enters a decimal point. These examples, and the ones below, use the callback funcFuncButton.

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

Here are the four buttons that do our math functions.

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

Finally, here is the clear button. It, of course, clears the holder variables and the display. Now we place the widgets in the PlaceWidget routine. First, we initialize the grid, then start putting the widgets into the grid. Here's the first part of the routine.

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

what the “columnspan” attribute does. There's a “rowspan” attribute as well. The “sticky” attribute tells the widget where to align its edges. Think of it as how the widget fills itself within the grid. Above left is the rest of our buttons.

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

Before we go any further let's take a look at how things will work when the user presses buttons.

Let's say the user wants to enter 563 + 127 and get the answer. They will press or click (logically) 5, then 6, then 3, then the “+,” then 1, then 2, then 7, then the “=” buttons. How do we handle this in code? We have already set the callbacks for the number buttons to the funcNumButton function. There's two ways to handle this. We can keep the information entered as a string and then when we need to convert it into a number, or we can keep it as a number the entire time. We will use the latter method. To do this, we will keep the value that is already there (0 when we start) in a variable called “self.CurrentValue”, When a number comes in, we take the variable, multiply it by 10 and add the new value. So, when the user enters 5, 6 and 3, we do the following...

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

```
User clicks 5 – 0 * 10 + 5
(5)

User clicks 6 – 5 * 10 + 6
(56)

User clicks 3 – 56 * 10 + 3
(563)
```

Of course we then display the “self.CurrentValue” variable in the label.

Next, the user clicks the “+” key. We take the value in “self.CurrentValue” and place it into the variable “self.HolderValue,” and reset the “self.CurrentValue” to 0. We then repeat the process for the clicks on 1, 2 and 7. When the user clicks the “=” key, we then add the values in “self.CurrentValue” and “self.HolderValue”, display them, then clear both variables to continue.

Above is the code to start defining our callbacks.

The “funcNumButton routine receives the value we passed from the button press. The only thing that is different from the example above is what if the user pressed the decimal button (“.”). Below, you'll see that we use a boolean variable to hold the fact they pressed the decimal button, and, on the next click, we deal with it. That's what the “if self.DecimalNext == True:” line is all about. Let's walk through it.

The user clicks 3, then 2, then the decimal, then 4, to create “32.4”. We handle the 3 and 2 clicks through the “funcNumButton” routine. We check to see if self.DecimalNext is True (which in this case it isn't until the user clicks the “.” button). If not, we simply multiply the held value (self.CurrentValue) by 10 and add the incoming value. When the user clicks the “.”, the callback “funcFuncButton” is called with the “Dec” value. All we do is set the boolean variable

"self.DecimalNext" to True. When the user clicks the 4, we will test the "self.DecimalNext" value and, since it's true, we play some magic. First, we increment the self.DecimalCount variable. This tells us how many decimal places we are working with. We then take the incoming value, multiply it by (10**-self.DecimalCount). Using this magic operator, we get a simple "raised to the power of" function. For example 10**2 returns 100. 10**-2 returns 0.01. Eventually, using this routine will result in a rounding issue, but for our simple calculator, it will work for most reasonable decimal numbers. I'll leave it to you to work out a better function. Think of this as your homework for this month.

The "funcClear" routine simply clears the two holding variables, then sets the display.

```
def funcClear(self):

self.CurrentValue = 0

self.HolderValue = 0

self.DisplayIt()
```

Now the functions. We've already discussed what happens with the function 'Dec'. We set this

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

The ABS function simply takes the current value and multiplies it by -1.

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

The Add function copies "self.CurrentValue" into "self.HolderValue", clears "self.CurrentValue", and sets the "self.CurrentFunction" to "Add". The Subtract, Multiply and Divide functions do the same thing with the proper keyword being set in "self.CurrentFunction".

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

The "Eq" function (Equals) is where the "magic" happens. It will be easy for you to understand the following code by now.

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

one up first with the “if” statement. We go to the “else,” and if the function is anything else, we clear the “self.DecimalNext” and “self.DecimalCount” variables.

The next set of steps are shown on the previous page (right hand box).

The DisplayIt routine simply sets the value in the display label. Remember we told the label to “monitor” the variable “self.CurrentDisplay”. Whenever it changes, the label automatically changes the display to match. We use the “.set” method to change the value.

```
def DisplayIt(self):

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

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

Finally we have our startup lines.

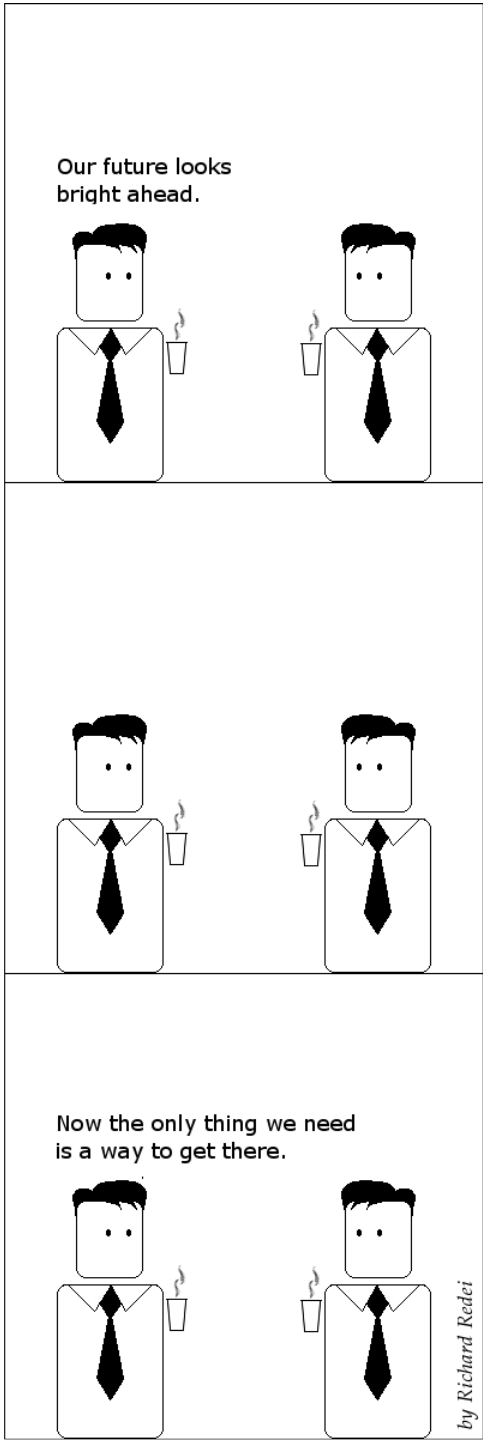
```
if __name__ == '__main__':

StartUp()
```

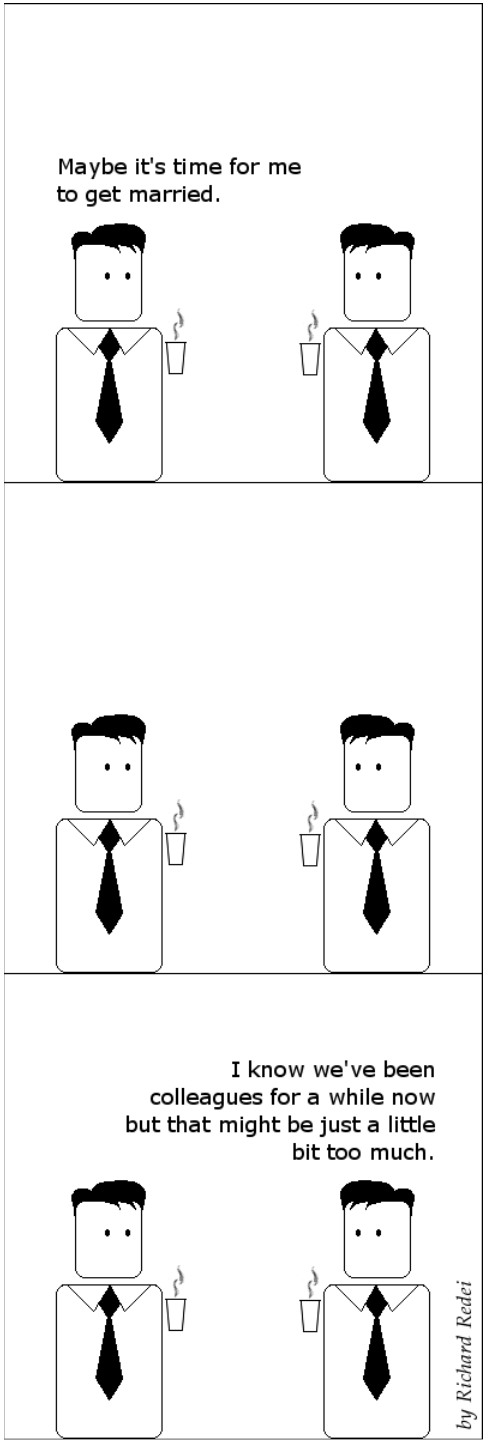
Now you can run the program and give it a test.

As always, the code for this article can be found at PasteBin. Examples 1, 2 and 3 are at: http://pastebin.com/mBAS1Umm and the Calc.py example is at: http://pastebin.com/LbMibF0u

Next month, we will continue looking at Tkinter and its wealth of widgets. In a future article, we'll look at a GUI designer for tkinter called PAGE. In the meantime, have fun playing. I think you'll enjoy Tkinter.

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.





Program In Python - Part 26

Last month we discussed tkInter and four of the widgets available: TopLevel, Frames, Buttons, and Labels. I also told you last month, I'd discuss how to have a widget as a parent other than the Toplevel widget.

So, this month, we'll discuss more on Frames, Buttons, and Labels, and introduce Checkboxes, Radio buttons, Textboxes (Entry widgets), Listboxes with a vertical scrollbar, and Messageboxes. Before we get started, let's examine some of these widgets.

Checkboxes are considered a many of many type selection widget that has two options, checked or not checked, or you could consider it on or off. They are usually used to provide a series of options where any, many, or all of those options may be selected. You can set an event to inform you when the checkbox has been toggled, or just query the value of the widget at any time.

Radiobuttons are considered a one of many type selection widget. It also has two options, on and off. However, they are grouped together to provide a set of options that logically can have only one selection. You can have multiple groups of Radiobuttons that, if properly programmed, won't interact with each other.

A Listbox provides a list of items for the user to select from. Most times, you want the user to select only one of the items at a time, but there can be occasions that you will allow the user to select multiple items. A scroll bar can be placed either horizontally or vertically to allow the user to easily look through all the items available.

Our project will consist of a main window and seven main frames that visually group our widget sets:

• The first frame will be very basic. It simply consists of various labels, showing the different relief options.

• The second will contain buttons, again pretty simple, that use the different relief options.

• In this frame, we'll have two checkboxes and a button that can programmatically toggle them, and they will send their state (1 or 0) back to the terminal window when clicked or toggled.

• Next, we'll have two groups of three radio buttons, each sending a message to the terminal window when clicked. Each group is separate.

• This has some text or entry boxes, which aren't new to you, but there's also a button to enable and disable one of them. When disabled, no entry can be made to that textbox.

• This is a list box with a vertical scroll bar that sends a message to the terminal whenever an item is selected, and will have two buttons. One button will clear the list box and the other will fill it with some dummy values.

• The final frame will have a series of buttons that will call various types of message boxes.

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

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

So now, we'll start our project. Let's name it “widgetdemo1.py”. Be sure to save it because we will be writing our project in little pieces, and build on them to make our full app. Each piece revolves around one of the frames. You'll notice that I'm including a number of comments as we go, so you can refer back to what's happening. Above are first few lines.

The first two lines (comments) are the name of the application and what we are concentrating on in this part. Line three is our import statement. Then we define our class. The next line starts our __init__ routine, which you all should be familiar with by now,

but, if you are just joining us, it's the code that gets run when we instantiate the routine in the main portion of the program. We are passing it the Toplevel or root window, which comes in as master here. The last three lines (so far), call three different routines. The first (DefineVars) will set up various variables we'll need as we go. The next (BuildWidgets) will be where we define our widgets, and the last (PlaceWidgets) is where we actually place the widgets into the root window. As we did last time, we'll be using the grid geometry manager. Notice that BuildWidgets will return the object "f" (which is our root window), and we'll pass that along to the PlaceWidgets routine.

Above right is our BuildWidgets routine. Each of the lines that start with "self." have been split for two reasons. First, it's good practice to keep the line length to 80 characters or less. Secondly, it makes it easier on our wonderful editor. You can do two things. One, just make each line long, or keep it as is. Python lets us split lines as long as they are within parentheses or brackets. As I said earlier, we are defining the widgets before we place them in the grid. You'll notice when we do the next routine, that we can also define a widget at the time we place it in the grid, but defining it before we put it in the grid in a routine like this makes it easier to keep track of everything, since we are doing (most of) the definitions in this routine.

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

So, first we define our master frame. This is where we will be putting the rest of our widgets. Next, we define a child (of the master frame) frame that will hold five labels, and call it lblframe. We set the various attributes of the frame here. We set the relief to 'SUNKEN', a padding of 3 pixels on left and right (padx), and 3 pixels on the top and bottom (pady). We also set the borderwidth to 2 pixels so that its sunken relief is noticeable. By default, the borderwidth is set to 0, and the effect of being sunken won't be noticed. Finally, we set the total width of the frame to 500 pixels.

Next, we define each label widget that we will use. We set the parent as self.lblframe, and not to frame. This way all the labels are children of lblframe, and lblframe is a child of frame. Notice that each definition is pretty much the same for all five of the labels except the name of the widget (lbl1, lbl2, etc), the text, and the relief or visual effect. Finally, we return the frame back to the calling routine (__init__).

The following page (top right) shows our PlaceWidgets routine.

We get the frame object in as a parameter called master. We assign that to 'frame' to simply be consistent with what we did in the BuildWidgets routine. Next, we set our main grid up (frame.grid(column = 0, row = 0)). If we don't do this, nothing works correctly. Then we start putting our widgets into the grid locations. First we put the frame (lblframe)

that holds all our labels, and set its attributes. We put it in column 0, row 1, set the padding to 5 pixels on all sides, tell it to span 5 columns (left and right), and finally use the "sticky" attribute to force the frame to expand fully to the left and right ("WE", or West and East). Now comes the part that sort of breaks the rule that I told you about. We are placing a label as the first widget in the frame, but we didn't define it ahead of time. We define it now. We set the parent to lblframe, just like the other labels. We set the text to "Labels |", the width to 15, and the anchor to east ('e'). If you remember from last time, using the anchor attribute, we can set where in the widget the text will display. In this case, it's along the right border. Now the fun part. Here we define the grid location (and any other grid attributes we need to), simply by appending ".grid" at the end of the label definition.

Next, we lay out all of our other labels in the grid - starting at column 1, row 0.

Here is our DefineVars routine. Notice that we simply use the pass statement for now. We'll be filling it in later on, and we don't need it for this part:

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

And lastly we put in our main routine code:

```
root = Tk()
root.geometry('750x40+1
50+150')
root.title("Widget
Demo 1")
demo = Demo(root)
root.mainloop()
```

First, we instantiate an instance of Tk. Thern we set the size of the main window to 750 pixels wide by 40 pixels high, and locate it at 150 pixels from the left and top of the screen. Then we set the title of the window and instantiate our Demo object, and finally call the Tk mainloop.

Give it a try. You should see the five labels plus the "last minute" label in various glorious effects.

Buttons

Now save what you have as widgetdemo1a.py, and let's add some buttons. Since we built our base program to be added to, we'll simply add the parts that apply. Let's start with the BuildWidgets routine. After the labels definitions, and before the "return frame" line, add what is shown on the next page, top right.

Nothing really new here. We've defined the buttons, with their attributes, and set their callbacks via the .bind configuration. Notice that we are using lambda to send the values 1 through 5 based on which button is clicked. In the callback, we'll use that so we know which button we are dealing with. Now we'll work in the PlaceWidgets routine. Put the code below after the last label placement.

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

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

Once again, nothing really new here, so we'll move on. Bottom right is our callback routine. Put it after the DefineVars routine.

Again, nothing really fancy here. We just use a series of IF/ELIF routines to print what button was clicked. The main thing to look at here (when we run the program) is that the sunken button doesn't "move" when you click on it. You would not usually use the sunken relief unless you were making a button that stays "down" when you click it. Finally, we need to tweak the geometry statement to support the extra widgets we put in:

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

Ok. All done with this one. Save it and run it.

Now save this as widgetdemo1b.py, and we'll move on to checkboxes.

Checkboxes

As I said earlier, this part of the demo has a normal button and two checkboxes. The first checkbox is what you would normally expect a checkbox to look like. The second is more like a "sticky" button - when it's not selected (or checked), it looks like a normal button. When you select it, it looks like a button that is stuck down. We can do this by simply setting the indicatoron attribute to False. The "normal" button will toggle the checkboxes from checked to unchecked, and vice versa, each time you click the button. We get to do this programmatically by calling the .toggle method attached to the checkbox. We bind the left mouse button click event (button release) to a function so we can send a message (in this case) to the terminal. In addition to all of this, we are setting two variables (one for each of the checkboxes) that we can query at any time. In this case, each time the checkbox is clicked we query this value and print it. Pay attention to the variable portion of the code. It is used in many widgets.

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

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

Under the BuildWidget routine, after the button code we just put in and before the return statement, put the code shown on the next page, top right.

Again, you have seen all of this before. We create the frame to hold our widgets. We set up a button and two check boxes. Let's place them now using the code on the next page, middle right.

Now we define the two

variables that we will use to monitor the value of each check box. Under DefineVars, comment out the pass statement, and add this...

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

After the button callback return, put the text shown bottom right.

And finally replace the geometry statement with this:

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

Save and run. Save it as widgetdemo1c.py, and let's do radio buttons.

Radiobuttons

If you are old enough to remember car radios with push buttons to select the station presets, you'll understand why these are called Radiobuttons. When using radiobuttons, the variable attribute is very important. This is what groups the radiobuttons together. In this demo, the first group of buttons is grouped by the variable named self.RBVal. The second is grouped by the variable self.RBValue2. We also need to set the value attribute at design time. This ensures that the buttons will return a value that makes sense whenever they are clicked.

Back to BuildWidgets, and, just before the return statement, add the code shown on the following page.

One thing of note here. Notice the “last minute” label definitions in the

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

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

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

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

PlaceWidget routine. These long lines are broken up to show how to use parens to allow our long lines to be formatted nicely in our code, and still function correctly.

In DefineVars add:

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

Add the click routines:

```
def RBClick(self):

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

```
def RBClick2(self):

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

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

In PlaceWidgets, add this:

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

and, finally, rework the geometry statement as follows.

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

Save the project as widgetdemo1d.py, and run it. Now, we'll start working on standard textboxes (or entry widgets).

Entry

Again, we've used textboxes or entry widgets in various GUI flavors before. However this time, as I said earlier, we will show how to keep the user from making changes to the textbox by disabling it. This is helpful if you are showing some data, and allowing the user to change it only when in the "edit" mode. By now, you should be pretty sure that the first thing we need to do is add code (shown right) to the BuildWidget routine.

Listbox

Next we'll work our listbox. Starting in BuildWidgets, add the code from the following page, right side.

As usual, we create our frame. Then we create our vertical scroll bar. We do this before we create the list box, because we have to reference the scrollbar '.set' method. Notice the attribute 'height = 5'. This forces the listbox to show 5 items at a time. In the .bind statement, we use '<<ListboxSelect>>' as the event. It's called a virtual event, since it's not really an "official" event.

Now, we'll deal with the additional code for the PlaceWidgets routine, and that's shown on the following page, left side.

Message Dialogs

This section is

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

Next, add this code to the PlaceWidget routine:

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

Add this line to the bottom of the DefineVars routine:

```
        self.Disabled = False
```

Now, add the function that responds to the button click event:

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

And finally, rework the geometry statement:

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

Save it as widgetdemo1d.py, and run it.

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

In DefineVars add this...

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

And add the following support routines:

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

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

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

Finally, update the geometry line.

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

Save this as widgetdemo1e.py, and run it. Now we will do our last modifications to our application.

```
        # List Box Stuff
      self.lstframe = Frame(frame,
            relief = SUNKEN,
            padx = 3,
            pady = 3,
            borderwidth = 2,
            width = 500
        )
        # Scrollbar for list box
        self.VScroll = Scrollbar(self.lstframe)
        self.lbox = Listbox(self.lstframe,
            height = 5,
            yscrollcommand = self.VScroll.set)
      # default height is 10
        self.lbox.bind('<<ListboxSelect>>',self.LBox
Select)
        self.VScroll.config(command =
self.lbox.yview)
        self.btnClearLBox = Button(
            self.lstframe,
            text = "Clear List",
            command = self.ClearList,
            width = 11
        )
        self.btnFillLBox = Button(
            self.lstframe,
            text = "Fill List",
            command = self.FillList,
            width = 11
        )
        # <<ListboxSelect>> is virtual event
        # Fill the list box
        self.FillList()
```

simply a series of "normal" buttons that will call various types of Message Dialogs. We've done them before in a different GUI toolkit. We will explore only 5 different types, but there are more. In this section, we'll look at Info, Warning, Error, Question, and Yes/No dialogs. These are very useful when you need to pass some information to your user in a rather big way. In the BuildWidgets routine add the code shown below.

Here is the support routine. For the first three (Info, Warning, and Error), you simply call 'tkMessageBox.showinfo', or whichever you need, with two parameters. First is the title for the message dialog, and second is the actual message you want to show. The icon is handled for you by tkinter. For the dialogs that provide a response (question, yes/no), we provide a variable that receives the value of which button was clicked. In the case of the question dialog, the response is either "yes" or "no", and, in the case of the yes/no dialog, the response is either "True" or "False".

Finally, modify the geometry line:

```
root.geometry('750x490+550+15
0')
```

Save this as widgetdemo1f.py, and play away.

I've put the code for widgetdemo1f.py on pastebin at http://pastebin.com/ZqrgHcdG.

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

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

Now, add the code for the PlaceWidgets routine:

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