Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

PROGRAMMING SERIES SPECIAL EDITION

PROGRAMMING SERIES
SPECIAL EDITION

PROGRAM IN PYTHON

Volume Three

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 17-21** from issues #43 through #47; 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 17

Written by Greg Walters

As I was finishing up the last installment of our series, I got an email about a programming competition. While we don't have time to deal with this one, various sites have programming competitions throughout the year. The competition information can be found at http://www.freiesmagazin.de/third_programming_contest - if you are interested. That made me realize that we haven't talked about true Client/Server programming. So with that in mind, we'll dig into this topic, and see where we can go with it.

So, what is a Client/Server Application? In very simple terms, anytime you use a program (or even a web interface) that accesses data from another application or computer, you are using a client/server system. Let's look at an example that we actually used before. Remember when we made our cookbook program? That was a VERY simple example (and not a very good one) of a client/server application. The SQLite database is the server, the application we wrote is the client. A better example would be the following. There is a database on a computer in another part of your office, floors away. It holds information on the inventory of the store you work at. You use a point of sale register (one of 10) within the store. Each of those registers are a client and the database located somewhere is the server.

While we won't try to create that kind of system here, we can learn some of the basics.

The first thing we need to think about is the location of our server. Many people have only one computer in their house. Some people might have 7 or 8.

To use a client/server system, we have to connect from the client machine to the server machine. We do this with what is called a pipe or socket. If you ever made a “tin can” telephone when you were a kid, you have an idea of what I'm going to be talking about. If not, let me paint you a picture of times gone by. First, you had to get your mother to save you two tin cans from beans or something. Then you cleaned them carefully, and took them out to the garage. You used a small nail and a hammer to poke a small hole in the bottom of each. Then you got about 15 feet of string (again from your loving mother), ran the end of the string through each can, and tied a large knot in each end of the string to hold it inside the can. You then got your best buddy, and stretched the string tightly and yelled into the can while your friend held his can up to his ear. The vibrations from the bottom of the can went through the taut string, and caused the other can bottom to vibrate. Of course, you could hear without the can, but that was beside the point. It was cool. The socket is about the same thing. The client has a direct connection (think of the string) to the server. If many clients are connecting to the server, each client would have a tin can of their own, and the poor server has to have the same number of tin cans all held tightly to each client's string phone. The bottom line here is each client has its own direct line to the server.

Let's make a simple server and client. We'll start with the server first. In pseudo code, here's what happens.

Create a socket
Get name of server machine
Select a port
Bind socket to address and port
Listen for a connection
If connected...
    Accept the connection
    Print we got a connection
    Close the connection

The actual code to our server is shown on the next page, bottom left.

So, we create the socket, get the hostname of the machine we are running the server on, bind the socket to the port, and start to listen. When we get a connection request, we accept it, we print the fact we are connected, send “Hello and Goodbye”, and close the socket.

Now we need to have a client to make the whole thing work (shown bottom right).

The code is almost like the server, but, in this case, we connect, print what we receive, and close the socket.

The output from the programs are very predictable. On the server side of things we get...

```
My hostname is  earth

I'm now connected to
 ('127.0.1.1', 45879)
```

and on the client side we get...

```
Hello and Goodbye
```

So, it's pretty simple. Now let's do something a bit more realistic.

```
#!/usr/bin/env python
#server1.py
import socket
soc = socket.socket()
hostname = socket.gethostname()
print "My hostname is ", hostname
port = 21000
soc.bind((hostname,port))
soc.listen(5)
while True:
     con,address = soc.accept()
     print "I'm now connected to ",address
     con.send("Hello and Goodbye")
     con.close()
```

We'll create a server that actually will do something. The code for server version 2 can be found at: http://fullcirclemagazine.pastebin.com/Az8vNUv7

Let's break it down. After our imports, we set up some variables. BUFSIZ holds the size of the buffer that we will use to hold the information that we receive from the client. We also set up the port we will listen on, and a list holding the host and port number.

We next create a class called ServCmd. In the __init__ routine, we create a socket, and bind the interface to that socket. In the run routine, we start listening, and wait for a command from the client.

When we do get a command from the client, we use the os.popen() routine. This basically creates a command shell and runs the command.

Next the client (above right), which is a good deal easier.

```
#!/usr/bin/env python
# client2.py

from socket import *
from time import time
from time import sleep
import sys
BUFSIZE = 4096

class CmdLine:
    def __init__(self,host):
        self.HOST = host
        self.PORT = 29876
        self.ADDR = (self.HOST,self.PORT)
        self.sock = None

    def makeConnection(self):
        self.sock = socket( AF_INET,SOCK_STREAM)
        self.sock.connect(self.ADDR)

    def sendCmd(self, cmd):
        self.sock.send(cmd)

    def getResults(self):
        data = self.sock.recv(BUFSIZE)
        print data

if __name__ == '__main__':
    conn = CmdLine('localhost')
    conn.makeConnection()
    conn.sendCmd('ls -al')
    conn.getResults()
    conn.sendCmd('BYE')
```

```
#!/usr/bin/python
# client1.py
#====================
import socket

soc = socket.socket()
hostname = socket.gethostname()
port = 21000

soc.connect((hostname, port))
print soc.recv(1024)
soc.close
```

We'll skip everything here except the send command, since you now have enough information to figure it out on your own. The **conn.sendCmd()** line (line 31) sends a simple **ls -al** request. Here's what my responses look like. Yours will be somewhat different.

Server:

```
python server2.py
...listening
...connected:  ('127.0.0.1',
42198)
Command received -  ls -al
Command received -  BYE
...listening
```

Client:

```
python client2a.py
total 72
drwxr-xr-x 2 greg greg 4096
2010-11-08 05:49 .
drwxr-xr-x 5 greg greg 4096
2010-11-04 06:29 ..
-rw-r--r-- 1 greg greg  751
2010-11-08 05:31 client2a.py
-rw-r--r-- 1 greg greg  760
2010-11-08 05:28 client2a.py~
-rw-r--r-- 1 greg greg  737
2010-11-08 05:25 client2.py
-rw-r--r-- 1 greg greg  733
2010-11-08 04:37 client2.py~
-rw-r--r-- 1 greg greg 1595
2010-11-08 05:30 client2.pyc
-rw-r--r-- 1 greg greg  449
2010-11-07 07:38 ping2.py
-rw-r--r-- 1 greg greg  466
2010-11-07 10:01
python_client1.py
-rw-r--r-- 1 greg greg  466
2010-11-07 10:01
python_client1.py~
-rw-r--r-- 1 greg greg  691
2010-11-07 09:51
python_server1.py
-rw-r--r-- 1 greg greg  666
2010-11-06 06:57
python_server1.py~
-rw-r--r-- 1 greg greg  445
2010-11-04 06:29 re-test1.py
-rw-r--r-- 1 greg greg 1318
2010-11-08 05:49 server2a.py
-rw-r--r-- 1 greg greg 1302
2010-11-08 05:30 server2a.py~
-rw-r--r-- 1 greg greg 1268
2010-11-06 08:02 server2.py
-rw-r--r-- 1 greg greg 1445
2010-11-06 07:50 server2.py~
-rw-r--r-- 1 greg greg 2279
2010-11-08 05:30 server2.pyc
```

We can also connect from another machine without changes anywhere - with the single exception of the conn = CmdLine('localhost') (line 29) in the client program. In this case, change the 'localhost' portion to the IP address of the machine that the server is running on. For my home setup, I use the following line:

```
conn =
CmdLine('192.168.2.12')
```

So, now we are able to send information back and forth from one machine (or terminal) to another.

Next time, we'll make our client/server applications much more robust.

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.

Ideas & Writers Wanted

We've created Full Circle project and team pages on LaunchPad. The idea being that non-writers can go to the project page, click 'Answers' at the top of the page, and leave your article ideas, but **please be specific with your idea!** Don't just put 'server article', please specify what the server should do!

Readers who fancy writing an article, but aren't sure what to write about, can register on the Full Circle team page, then assign article ideas to themselves, and get writing! We do ask that **if you can't get the article written within several weeks (a month at most) that you reopen the question** to let someone else grab the idea.

Project page, **for ideas**:
https://launchpad.net/fullcircle
Team page **for writers**:
https://launchpad.net/~fullcircle

Program In Python - Part 18

Written by Greg Walters

Last time, we created a very simple client/server system. This time, we are going to extend it a bit. The server is a tic-tac-toe (or naughts and crosses) board and checker. The client portion acts as the input/output.

We'll start by using the same server code as last time, and modifying it as we go. If you didn't save the code from then, go to http://fullcirclemagazine.pastebin.com/UhquVK4N, get the source code for this time, and follow along. The first change comes in the __init__ routine where we initialize two new variables, self.player and self.gameboard. The gameboard is a simple list of lists or a basic array. We can access it as follows (more visual than just the flat list). This list will hold our data. There are three possible entries per cell. “-” means the cell is empty. “X” means the cell is occupied by player 1 and “O” means the cell is occupied by player 2. The grid looks like this when put in two dimensions:

```
[0][0] | [0][1] | [0][2]
[1][0] | [1][1] | [1][2]
[2][0] | [2][1] | [2][2]
```

So starting with the server code from last month, in the routine __init__ routine, add the following lines:

```
# The next three lines are new...
    self.player = 1

    self.gameboard = [['-','-','-'],['-','-','-'],['-','-','-']]

    self.run()
```

The run, listen, and servCmd routines have no changes, so we'll concentrate on the changes to the procCmd routine next.

In last time's article, the server waited for a command from the client, then sent it to the os.popen routine. This time, we will parse the command sent in. In this case, we have three separate commands we will listen for. They are 'Start', 'Move', and 'GOODBYE'. When we receive the 'Start' command, the server should initialize the game board to all “-” and then send a “print out” of the board to the client.

The 'Move' command is a compound command, in that it contains the command, and the position that the player wants to move to. For example, 'Move A3'. We parse the command to get three parts, the 'move' command itself, and the the row and column. Finally the 'GOODBYE' command simply resets the game board for another game.

So, we receive the command from the client in the procCmd routine. We then check the command to see what we are supposed to do. Within the procCmd routine, find the 5th line down, and, after the line that says “if self.processingloop:”, remove the rest of that set of code. Now we'll set up the commands as we laid the out. Here's the code for the Start command:

```
if self.processingloop:
  if cmd == 'Start':
    self.InitGameBoard()
    self.PrintGameBoard(1)
```

Next, let's look at the Move portion of the routine (shown below). We first check the first four characters of the passed-in command to see if they match 'Move'. If they match, we then pull the rest of the string starting at position 5 (since things are 0 based), and assign that to a variable named position. We then check to see if the first character is either an 'A', 'B', or 'C'. These represent the row that the client has sent. We then take the integer value of the next character and that's our column:

```
if cmd[:4] == 'Move':
print "MOVE COMMAND"
position = cmd[5:]
if position[0] == 'A':
    row = 0
elif position[0] == 'B':
    row = 1

elif position[0] == 'C':
    row = 2
else:
    self.cli.send('Invalid position')
    return
col = int(position[1])-1
```

Next, we make a quick check to verify that the row position is within the allowable positions:

```
if row < 0 or row > 2:

   self.cli.send('Invalid
position')

   return
```

Finally, we verify that the position is empty ('-'), and, if the current player is number 1, we put an "X" otherwise we put a "O". We then call the PrintGameBoard routine with a "0" parameter:

```
if self.gameboard[row][col]
== '-':

    if self.player == 1:

       self.gameboard[row][co
l] = "X"

else:

       self.gameboard[row][co
l] = "O"

self.PrintGameBoard(0)
```

That finishes the changes to the procCmd routine. Next we have the "initialize the game board" routine. All it does is to set each position to a "-", which the move logic uses to verify that a space is empty:

```
def InitGameBoard(self):

   self.gameboard = [['-','-
','-'],['-','-','-'],['-','-
','-']]
```

The PrintGameBoard routine (below) prints the game board, calls the checkwin routine, and sets the player number. We build a large string to send to the client so it only has to enter the listen routine once per move. The firsttime parameter is included to send the pretty print of the gameboard when the client first connects or resets the game:

```
def PrintGameBoard(self,firsttime):
    #Print the header row
    outp = ('   1   2   3') + chr(13) + chr(10)
    outp += (" A {0} | {1} | {2}".format(self.gameboard[0][0],self.gameboard[0][1],self.gameboard[0][2])) + chr(13)+chr(10)
    outp += ('  ------------')+ chr(13)+chr(10)
    outp += (" B {0} | {1} | {2}".format(self.gameboard[1][0],self.gameboard[1][1],self.gameboard[1][2]))+ chr(13)+chr(10)
    outp += ('  ------------')+ chr(13)+chr(10)
    outp += (" C {0} | {1} | {2}".format(self.gameboard[2][0],self.gameboard[2][1],self.gameboard[2][2]))+ chr(13)+chr(10)
    outp += ('  ------------')+ chr(13)+chr(10)
```

Next, we check to see if the firsttime parameter is set to 0 or 1 (below). Only if firsttime is set to 0, we check to see if the current player has won, and, if so, add the 'Player X WINS!' text to the output string. If the current player did not win, we then add the "Enter move..." text to the output string. Finally we send the string out to the client with the cli.send routine:

```
if firsttime == 0:
            if self.player == 1:
                ret = self.checkwin("X")
            else:
                ret = self.checkwin("O")
            if ret == True:
                if self.player == 1:
                    outp += "Player 1 WINS!"
                else:
                    outp += "Player 2 WINS!"
            else:
                if self.player == 1:
                    self.player = 2
                else:
                    self.player = 1
                outp += ('Enter move for player %s' %
self.player)
        self.cli.send(outp)
```

Finally, on the next page, we have the server check for a win routine. We have already set the player to either an "X" or "O", so we start by using a simple for loop. If we find a win, we return True from the routine. Our for variable 'C' represents each row in our list of lists. First, we will check each Row for a horizontal win:

```
First, we will check each Row for a horizontal win:
   def checkwin(self,player):
        #loop through rows and columns
        for c in range(0,3):
        #check for horizontal line
            if self.gameboard[c][0] == player and
self.gameboard[c][1] == player and self.gameboard[c][2] ==
player:
              print "*********\n\n%s wins\n\n*********" %
player
              playerwin = True
              return playerwin

Next, we check each Column for a win:
            #check for vertical line
            elif self.gameboard[0][c] == player and
self.gameboard[1][c] == player and self.gameboard[2][c] ==
player:
              print "** %s wins **" % player
              playerwin = True
              return playerwin

Now we check for the diagonal win from left to right...
            #check for diagonal win (left to right)
            elif self.gameboard[0][0] == player and
self.gameboard[1][1] == player and self.gameboard[2][2] ==
player:
              print "** %s wins **" % player
              playerwin = True
              return playerwin

Then from right to left...
            #check for diagonal win (right to left)
            elif self.gameboard[0][2] == player and
self.gameboard[1][1] == player and self.gameboard[2][0] ==
player:
              print "** %s wins **" % player
              playerwin = True
              return playerwin

Finally, if there is no win, we return false:
        else:
            playerwin = False
            return playerwin
```

The Client

Once again, we start with the simple routine that we had last time. The changes start right after the call to conn.makeConnection. We send a Start, various Moves, and finally a Goodbye command. The biggest thing to remember here is that you must send a command, then get a response before sending another command. Think of it as a polite conversation. Make your statement, listen for a response, then make another statement, listen for a response, and so on. In this sample we use raw_input simply so you can see what is going on:

```
if __name__ == '__main__':
    conn =
CmdLine('localhost')
    conn.makeConnection()
    conn.sendCmd('Start')
    conn.getResults()
    conn.sendCmd('Move A3')
    conn.getResults()
    r = raw_input("Press
Enter")
    conn.sendCmd('Move B2')
    conn.getResults()
    r = raw_input("Press
Enter")
```

Continue the sendCmd, getResults, raw_input routine set with the following commands (you already have the code for the A3 and B2 moves), C1, A1, C3, B3, C2, then end with a GOODBYE command.

Moving Forward

So, here is your "homework" assignment. In the client app, remove the hard coded move commands, and use raw_input() to prompt for and get moves from the player(s) in the form of "A3" or "B2", then prepend the command "Move" before sending it to the server.

Next time, we'll modify our server to actually play the other player.

Server and Client Full Source Code can be found at http://fullcirclemagazine.pastebin.com/UhquVK4N or at http://thedesignatedgeek.com

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 19

Written by Greg Walters

This time, we are going to work on finishing our Tic-Tac-Toe program. However, unlike most of my other articles, I won't be providing the code. You will! I will however, be giving you the rules. After 18 months, you have the tools and knowledge to finish this project. I'm sure of it.

First, let's look at the logic of playing Tic-Tac-Toe. We'll look at it in pseudo-code. Let's look first at the game board. It's laid out like this...

```
Corner |   Side   | Corner
-------+----------+-------
 Side  | Center   |  Side
-------+----------+-------
Corner |   Side   | Corner
```

Now, whoever is "X", goes first. Their first best move is to take a corner square. Any corner square, it doesn't matter. We'll deal with the permutations of playing "X" first, these are shown right.

The standpoint of the "O" player is shown below right.

```
IF "O" takes a CORNER square THEN
     # Scenario 1
     "X" should take one of the remaining corner squares. Doesn't matter which.
     IF "O" blocks the win THEN
          "X" takes remaining corner square.
          Finish for win.
     ELSE
          Finish for win.
ELIF "O" takes a SIDE square THEN
     # Scenario 2
     "X" takes CENTER square
     IF "O" blocks win THEN
          "X" takes corner square that is not bordered by any "O"
          Finish for win.
     ELSE
          Finish for win.
ELSE
     # "O" has played in the CENTER square – Scenario 3
     "X" takes corner square diagonally to
original move
     IF "O" plays on corner square
          "X" plays remaining open corner square
          Finish for win.
     ELSE
          # Game will be a draw – Scenario 4
          Block "O" win.
          Block any other possible wins
```

```
IF "X" plays to non-center square
THEN
     "O" takes Center Square
     IF "X" has corner square AND
side square THEN
          #Scenario 5
          "O" takes corner
diagonally from corner "X"
          Block possible wins to a
draw.
     ELSE
          # "X" has two Edge squares
– Scenario 6
          "O" moves to corner
bordered by both "X"s
          IF "X" blocks win THEN
               "O" takes any square.
               Block and force draw
          ELSE
               Finish for win.
```

Some possible play outs are shown on the next page.

As you can see, the logic is somewhat complex, but can easily be broken down in a series of IF statements (notice I used "Then", but in Python, we don't, we use the ":"). You should be able to modify the code from last month to deal with this, or at least write one from scratch to simply be a desktop tic-tac-toe program.

contents ^

```
Scenario 1
 X | - | -    X | - | -    X | - | -    X | - | -    X | - | X    X | - | X    X | X | X
 - | - | -    - | - | -    - | - | -    O | - | -    O | - | -    O | O | -    O | O | -
 - | - | -    - | - | O    X | - | O    X | - | O    X | - | O    X | - | O    X | - | O

Scenario 2
 X | - | -    X | - | -    X | - | -    X | - | -    X | - | X    X | - | X    X | X | X
 - | - | -    O | - | -    O | X | -    O | X | -    O | X | -    O | X | -    O | X | -
 - | - | -    - | - | -    - | - | -    - | - | O    - | - | O    O | - | O    X | - | O

Scenario 3
 X | - | -    X | - | -    X | - | -    X | - | X    X | O | X    X | O | X    X | O | X
 - | - | -    - | O | -    - | O | -    - | O | -    - | O | -    O | - | -    O | - | X
 - | - | -    - | - | -    - | - | X    O | - | X    O | - | X    O | - | X    O | - | X

Scenario 4
 X | - | -    X | - | -    X | - | -    X | - | -    X | - | -    X | - | X    X | O | X
 - | - | -    - | O | -    - | O | O    X | O | O    X | O | O    X | O | O    X | O | O
 - | - | -    - | - | -    - | - | X    - | - | X    O | - | X    O | - | X    O | - | X

Scenario 5
 X | - | -    X | - | -    X | - | -    X | - | -    X | - | -    X | - | -    X | - | X
 - | - | -    - | O | -    - | O | X    - | O | X    X | O | X    X | O | X    X | O | X
 - | - | -    - | - | -    - | - | -    - | - | O    - | - | O    O | - | O    O | - | O

Scenario 6
 - | - | -    - | - | -    - | - | -    - | - | -    - | - | X    O | - | X    O | - | X
 X | - | -    X | O | -    X | O | -    X | O | -    X | O | -    X | O | -    X | O | -
 - | - | -    - | - | -    - | X | -    O | X | -    O | X | -    O | X | -    O | X | O
```

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.

Ideas & Writers Wanted

We've created Full Circle project and team pages on LaunchPad. The idea being that non-writers can go to the project page, click 'Answers' at the top of the page, and leave your article ideas, but **please be specific with your idea!** Don't just put 'server article', please specify what the server should do!

Readers who fancy writing an article, but aren't sure what to write about, can register on the Full Circle team page, then assign article ideas to themselves, and get writing! We do ask that **if you can't get the article written within several weeks (a month at most) that you reopen the question** to let someone else grab the idea.

Project page, **for ideas**:
https://launchpad.net/fullcircle
Team page **for writers**:
https://launchpad.net/~fullcircle

Program In Python - Part 20

Welcome back. This time we will re-address GUI programming, but this time we will be using the pyGTK library. We won't be working with a GUI designer right now, we'll just be working with the library.

Use Synaptic to install python-gtk2, python-gtk2-tutorial, and python-gtk2-doc.

Let's jump right in and make our first program using pyGTK, it's shown above right.

```
# simple.py
import pygtk
pygtk.require('2.0')
import gtk

class Simple:
     def __init__(self):
          self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
          self.window.show()
     def main(self):
          gtk.main()
if __name__ == "__main__":
     simple = Simple()
     simple.main()
```

For awhile, we will be building on this simple code set. On line #3 is a new command. The line "pygtk.require('2.0')" means that the application will not run unless the pygtk module is at least version 2.0. In the __init__ routine, we assign a window to the self.window variable (line 8), and then show it (line 9). Remember that the __init__ routine is run as soon as we instantiate the class (line 13). Save this code as "simple1.py".

Run it in a terminal. You'll see a simple window show up somewhere on your desktop. On mine, it shows up in the upper left corner of my desktop. In order to end the program, you have to hit Ctrl-C in the terminal. Why? We haven't added any code to destroy and actually end the app. That's what we'll do next. Add the following line before the self.window.show() line...

```
self.window.connect("delete_event", self.delete_event)
```

Then after the gtk.main() call, add the following routine...

```
def delete_event(self, widget, event, data=None):
     gtk.main_quit()
     return False
```

Now save your app as "simple2.py", and, once again, run it from a terminal. Now, when you click the "X" on the title bar, the application will exit. What is actually happening here? The first line we added (self.window.connect...) connects the delete event to a callback routine, in this case self.delete_event. By returning "False" to the system, it also destroys the actual window from system memory.

Now, I don't know about you, but I prefer my applications to open in the center of the screen, not someplace random, or in a corner - where it might be obscured by something else. Let's modify the code to do this. All we need to do is add the following line before the self.window.connect line in the __init__ function:

```
self.window.set_position(gtk.WIN_POS_CENTER)
```

As you might guess, this sets the position of the window in the center of the screen. Save the app as "simple3.py" and run it.

That's much nicer, but there's not much there. So, let's try to add a widget. If you remember WAY back when we worked with Boa Constructor, widgets are simply predefined controls that we can add to our window to do things. One of the simplest controls to add is a button. We will add the following code right after the self.window.connect line in our previous code in the __init__ routine:

```
self.button =
gtk.Button("Close Me")

self.button.connect("clicked"
,self.btn1Clicked,None)

self.window.add(self.button)

self.button.show()
```

The first line defines the button, and the text on the button surface. The next line is the connector to the click event. The third line adds the button to the window, and the fourth line shows the button on the window surface. Looking at the self.button.connect line, you'll see that there are three arguments. The first is the event we want to connect to, the second is the routine that will be called when the event is triggered, in this case "self.btn1Clicked", and the third is the argument (if any) that will be passed to the routine we just defined.

Next, we need to create the self.btn1Clicked routine. Put this after the self.delete_event routine:

```
def
btn1Clicked(self,widget,data=
None):

    print "Button 1 clicked"

    gtk.main_quit()
```

As you can see, the routine doesn't do much. It prints in the terminal "Button 1 clicked", and then calls the gtk.main_quit() routine. This will close the window and terminate the application - just as if you had clicked the "X" on the title bar. Again, save this as "simple4.py", and run it in a terminal. You'll see our centered window with a button that says "Close me". Click on it, and the application closes, as designed. Notice, however, that the window is much smaller than it was in the simple3.py application. You can resize the application, but the button resizes with it. Why is this? Well, we simply shoved a button into the window and the window resized to fit the control.

We sort of broke the rules of GUI programming by putting the button directly on the form, without using a container. Remember back when we did our first series on GUI programming using Boa Constructor - we used sizer boxes (containers) to hold our controls. We should do this, even if we only have just one control. For our next example, we'll add a HBox (Horizontal box) to hold our button, and add another button. If we wanted a vertical container, we would use a VBox.

To start, use "simple4.py" as our base code. Delete everything between the lines self.window.connect(...) and self.window.show(). This is where we will add our new lines. The code for the HBox and our first button are...

```
self.box1 = gtk.HBox(False,0)

self.window.add(self.box1)

self.button =
gtk.Button("Button 1")

self.button.connect("clicked"
,self.btn1Clicked,None)

self.box1.pack_start(self.but
ton,True,True,0)

self.button.show()
```

Breaking down this code, we add a HBox, naming it self.box1. The parameters we pass to the HBox are homogeneous (True or False), and a spacing value:

```
HBox =
gtk.HBox(homogeneous=False,
spacing=0)
```

Ideas & Writers Wanted

We've created Full Circle project and team pages on LaunchPad. The idea being that non-writers can go to the project page, click 'Answers' at the top of the page, and leave your article ideas, but **please be specific with your idea!** Don't just put 'server article', please specify what the server should do!

Readers who fancy writing an article, but aren't sure what to write about, can register on the Full Circle team page, then assign article ideas to themselves, and get writing! We do ask that **if you can't get the article written within several weeks (a month at most) that you reopen the question** to let someone else grab the idea.

Project page, **for ideas**:
https://launchpad.net/fullcircle
Team page **for writers**:
https://launchpad.net/~fullcircle

The homogeneous parameter controls whether each widget in the box has the same size (width in the case of an HBox and height in the case of a VBox.) In this case, we pass it false, and a spacing value of 0. Next, we add the box to the window. Now, we create the button as before, and connect the clicked event to our routine.

Now, we come to a new command. The self.box1.pack_start command is used to add the button to the container (HBox). We use this command instead of the self.window.add command for the widgets we want to be in the container. The command (as above) is...

```
box.pack_start(widget,expand=True, fill=True, padding=0)
```

The pack_start command has the following parameters. First is the widget, next is expand (True or False), then fill (True or False), and a padding value. Spacing for the containers is the amount of space in between the widgets, and padding is for the right/left side of the widgets. The expand argument allows you to choose whether the widgets in the box will fill all the extra space in the box (True), or if the box shrinks to fit the widgets (False). The fill argument has an effect only if the expand argument is True. Finally we show the button. Next is the code for the second button:

```
self.button2 = gtk.Button("Button 2")

self.button2.connect("clicked",self.btn2Clicked,None)

self.box1.pack_start(self.button2,True,True,0)

self.button2.show()

self.box1.show()
```

Notice that this code is pretty much the same thing as the first button widget. The last line of this new code shows the box.

Now, we have to add the self.btn2Clicked routine. After the self.btn1Clicked routine, add the following code...

```
def btn2Clicked(self,widget,data=None):

    print "Button 2 clicked"
```

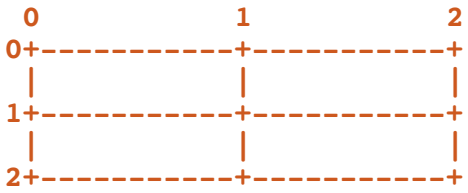
and in the btn1Clicked routine, comment out the line:

```
gtk.main_quit()
```

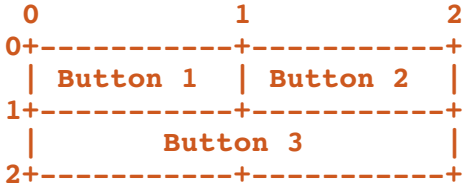
We want both buttons to print their “Button X clicked” response without closing the window.

Save this as “simple4a.py”. Run it in a terminal. What you will see is a centered window with two buttons (right up to the edges of the window) marked “Button 1” and “Button 2”. Click on them and notice that they properly respond to the click event as we have discussed. Now, before closing the window, resize it (drag at the bottom right of the window), and notice that the buttons grow and shrink equally as you resize the window. To understand the expand parameter, change the code for the self.box1.pack_start from True to False in both lines. Re-run your program and see what happens. This time, the window starts out looking the same, but when you resize the window, the buttons stay the same width, and there is empty space to the right as you expand the window. Next, change the expand parameter back to True and set the fill parameter to False. Re-run and notice that the buttons stay the same width, but there is empty space to the left and right of the buttons as you resize the window. Remember the fill parameter doesn't do anything if the expand parameter is set to False.

Another way to pack widgets is by using a table. Many times, if everything you have can fit into a grid-like structure easily, then a table is your best (and easiest) bet. You can think of a table like a spreadsheet grid with rows and columns holding widgets. Each widget can take up one or more cells - as your application requires. Maybe the following diagram will help visualize the possibility. Here is a 2x2 grid:



Into the first row, we will place two buttons. One in column 1 and one in column 2. Into the second row, we will place one button spanning both columns. Like this...



To set up a table, we create a table object and add it into the window. The call to create the table is...

```
Table =
gtk.Table(rows=1,columns=1,ho
mogeneous=True)
```

If the homogeneous flag is set to True, the size of the table boxes are resized to the largest widget in the table. If set to False, the size of the table boxes will be dictated by the tallest widget in the same row and the widest widget in its column. We then create a widget (like a button above), then attach that widget into the table in the proper row and column. The attach call is as follows...

```
table.attach(widget,left
point,right point,top
point,bottom
point,xoptions=EXPAND|FILL,yo
ptions=EXPAND|FILL,
xpadding=0,ypadding=0)
```

The only required parameters are the first 5. So, to attach a button to the table in row 0 column 0, we would use the following command...

```
table.attach(buttonx,0,1,0,1)
```

If it were to be placed into row 0 column 1 (remember this is zero based) as button 2 is above, the call would be...

```
table.attach(buttonx,1,2,0,1)
```

Hopefully, this is as clear as mud for you now. Let's get started with our code, and you'll understand better. First the common part...

```
# table1.py
import pygtk
pygtk.require('2.0')
import gtk
class Table:
    def __init__(self):

        self.window =
gtk.Window(gtk.WINDOW_TOPLEVE
L)

        self.window.set_posit
ion(gtk.WIN_POS_CENTER)

        self.window.set_title
("Table Test 1")

        self.window.set_borde
r_width(20)

        self.window.set_size_
request(250, 100)

        self.window.connect("
delete_event",
self.delete_event)
```

There are a couple of new things here that we need to discuss before we move on. Line 9 sets the title of the window to "Table Test 1". We use the "set_border_width" call to give a border of 20 pixels around the entire window before any widgets are placed. Finally, we are forcing the window to 250 x 100 pixels using the "set_size_request" funciton. Makes sense so far?

Now, we create the table and add it to the window...

```
table = gtk.Table(2, 2,
True) # Create a 2x2 grid

self.window.add(table)
```

Next, we create our first button, set up the event connection, attach it to the table grid point, and show it...

```
button1 = gtk.Button("Button
1")

button1.connect("clicked",sel
f.callback,"button 1")

table.attach(button1,0,1,0,1)

button1.show()
```

Now button number 2...

```
button2 = gtk.Button("Button
2")

button2.connect("clicked",sel
f.callback,"button 2")

table.attach(button2,1,2,0,1)

button2.show()
```

Almost exactly the same as button number 1, but notice the change in the table.attach call. Also notice that the routine we will be using for the event handling is called "self.callback", and is the same for both buttons. That's good for now. You'll understand what we're doing in a moment.

Now for the third button. This will be our "Quit" button:

```
button3 = gtk.Button("Quit")

button3.connect("clicked",sel
f.ExitApp,"button 3")

table.attach(button3,0,2,1,2)

button3.show()
```

Finally, show the table and the

window. Also here is the main routine and the delete routine we have used before:

```
    table.show()

    self.window.show()

def main(self):

    gtk.main()

def delete_event(self,
widget, event, data=None):

    gtk.main_quit()

    return False
```

Now for the fun part. For both button 1 and button 2, we set the event handler routine to "self.callback". Here's the code for that.

```
def
callback(self,widget,data=Non
e):

     print "%s was pressed"
% data
```

What happens is that when the user clicks on the button, the click event is triggered, and the data that was provided when we set the event connection is sent in. For button 1, the data that will be sent is "button 1", and for button 2 it is "button 2". All we are doing here is printing "button x was pressed" into the terminal. I'm sure you can see that this could be a very useful tool when combined with a nicely structured IF | ELIF | ELSE routine.

Now to finish up, we have to define the "ExitApp" routine for when the "Quit" button is clicked...

```
def ExitApp(self, widget,
event, data=None):

     print "Quit button was
pressed"

     gtk.main_quit()
```

And now the final main code...

```
if __name__ == "__main__":

    table = Table()

    table.main()
```

Combine all this code into a single app called "table1.py". Run it in a terminal.

So to recap, when we want to use pyGTK to create a GUI program, the steps are...

- Create the window.
- Create HBox(s), VBox(s) or Table(s) to hold your widgets.
- Pack or attach the widgets (depending on box or table).
- Show the widgets.
- Show the box or table.
- Show the window.

Now we have many of the tools and knowledge to go forward. All code is up on Pastebin at http://fullcirclemagazine.pastebin.com/wnzRsXn9. See you next time.

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.

Full Circle Podcast

In episode #15: Brainstorms, FUD and Media Players

* **Review**: FCM#44.
* **News**: Brainstorm ideas, Software Centre ratings, Fuduntu, Unity, Android, and more!
* **Gaming**: Humble Indie Bundle 2, Mass Effect, FreeCiv, and Dropbox.

File Sizes:
OGG - 46.9Mb
mp3 - 40.4Mb

Runtime: 1hr 24min 34sec
Released: 13th Jan. 2011

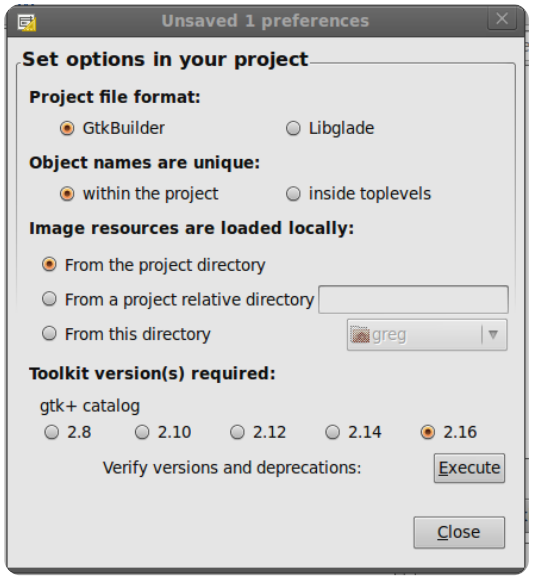
http://fullcirclemagazine.org/

Program In Python - Part 21

If you've been with me for a long while, you might remember back to parts 5 and 6. We talked about using Boa Constructor to design our GUI application. Well, this time, we are going to deal with Glade Designer. Different, but similar. You can install it from the Ubuntu Software Center: search for glade, and install GTK+ 2 User Interface Builder.

Just to let you know, this will be an application that we'll need multiple parts of these tutorials to cover. The ultimate goal is to build a playlist maker for our MP3, and other media files. This portion of the tutorial will be focusing on the design portion. Next time, we'll deal with the code that glues all the parts of the GUI together.

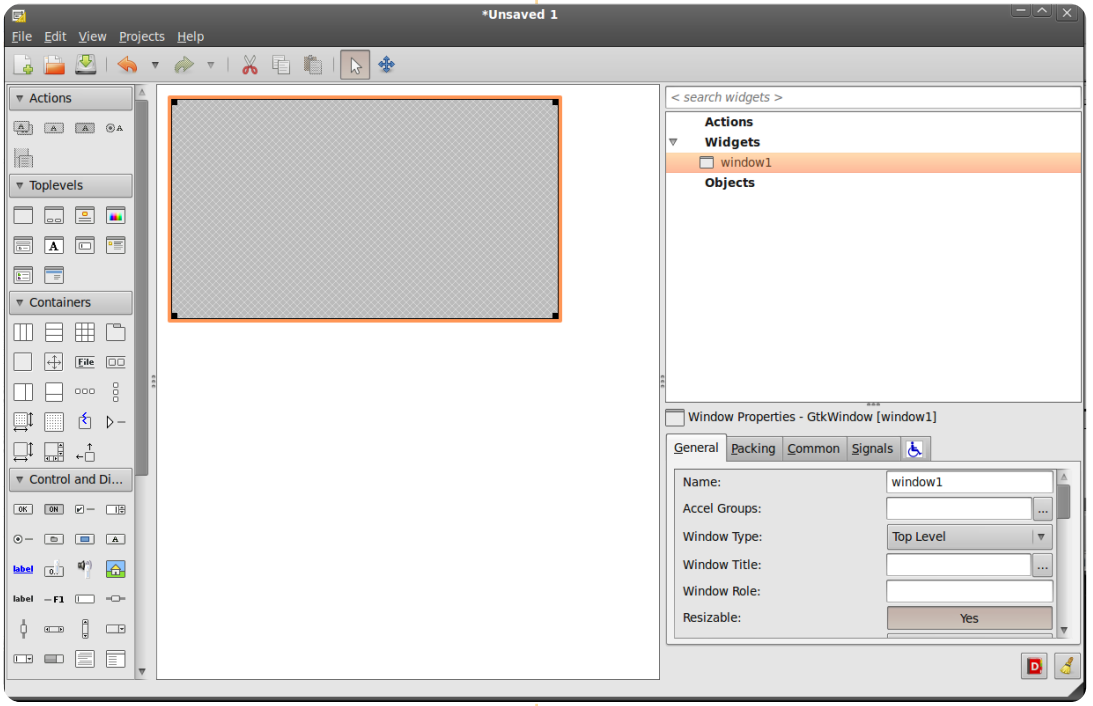
Now to start designing our application. When you first start the Glade designer, you will have a preferences window open (above). Select Libglade, and "inside toplevels", then click close. This will give us our designer main window.



Let's take a look at the main window (right). On the left is our toolkit, in the middle is the designer area, and on the right is our attribute and hierarchy areas.

In the toolkit area, find the group marked "Toplevels", and click on the first tool there (if you hover over it, it should show "Window"). This will give us our blank window "canvas" that we will be working with.

Notice that, in the hierarchy area, you see window1 under the



Widgets section. Now move down to the attributes section, change the name from window1 to MainWindow, and set the Window Title to "Playlist Maker v1.0". Save what you have as "PlaylistMaker.glade". Before we can move on, in the attributes section of the General tab, find the Window Position pulldown and set it to Center. Click the check box for Default Width, and set this to 650. Do the same for Default Height, but set it to 350. Next, click on the

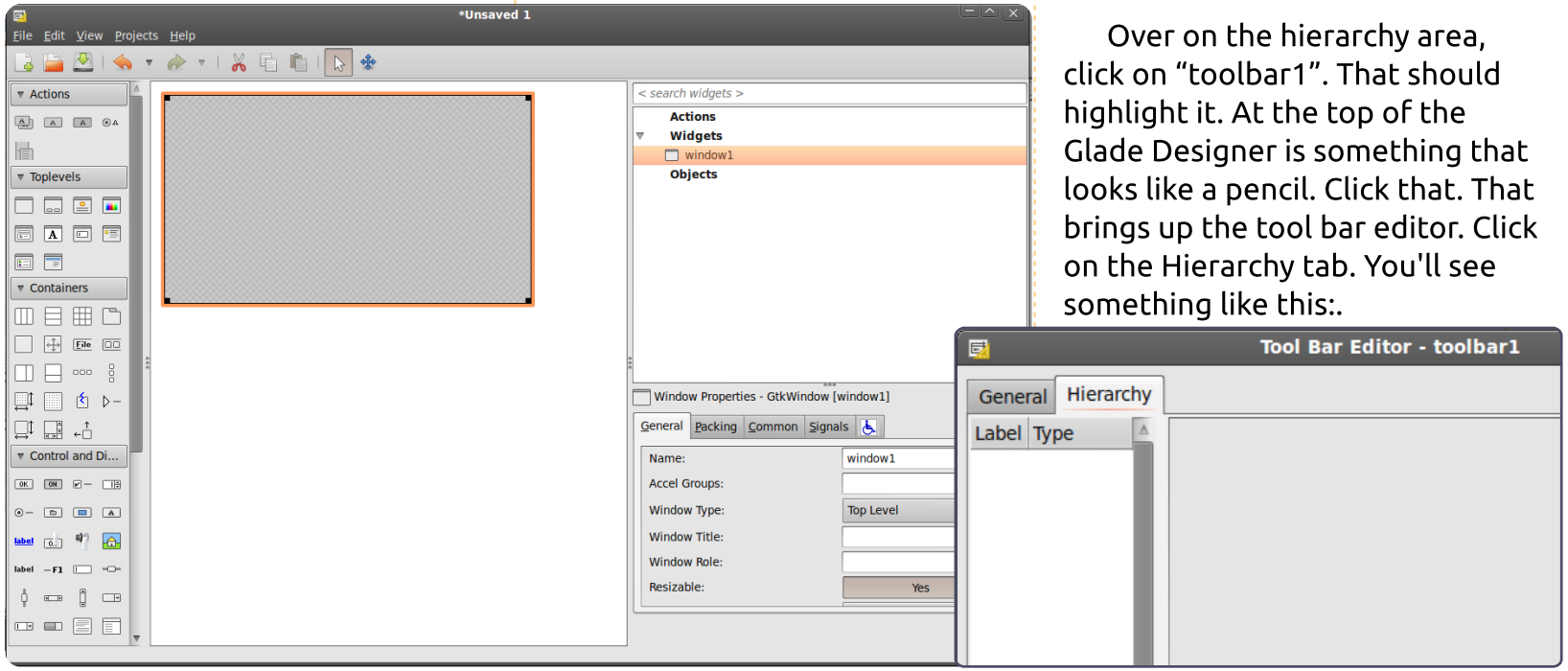
Common tab, and scroll down to the entry marked "Visible". BE SURE TO SET THIS TO "YES" - otherwise your window won't show. Finally, select the Signals tab, scroll down to the GtkObject section, and click the arrow pointing to the right. Under destroy, click the pulldown in the Handler column, and select "on_MainWindow_destroy" setting. This gives us an event that gets raised when the user closes our window by clicking on the "X"

in the titlebar. One word of warning... After setting the destroy event, click somewhere above or below to make the change take. This seems to be a bug in Glade Deisgner. Again, save your project.

Just as before when we were doing GUI design, we need to put our widgets in vboxes and hboxes. This is the hardest thing to remember when doing GUI programming. We will be adding a vertical box to hold our widgets in the window, so, on the toolbox under Containers, select Vertical Box (second icon from the left on the top row), and click in our blank window in the designer section. You will be presented with a pop up window that asks how many slots or items you want. The default is three, but we need five. The layout, from top to bottom, will be a toolbar, an area for a treelist control, two horizontal areas for labels, buttons and text entry boxes, and a status bar.

Now we can start adding our widgets. First, add a toolbar from the toolbox. It's the (in my setup) fourth icon on the second line under containers. Click in the topmost slot of the vbox. That slot will shrink and almost disappear. Don't worry, we'll get it back in a few minutes.

Next, we need to add a Scrolled Window to the next slot down to hold our treelist. This will allow us to scroll within the treelist. So, find the Scrolled Window icon under the Containers section of the toolbox (second icon from the left on the fifth row on my setup), and click that into the second slot of the vbox. Next, we will add two Horizontal boxes, one to each of the next slots. Each needs three slots. Finally, add a Status Bar to the bottom slot. This is under the Control and Display section of the toolbox near the bottom. Now your designer should look something like the image below.



Last, but not least, add a Tree View widget from the Control and Display section of the toolbox into the scrolled window widget. You'll get a pop-up asking which TreeView model you wish to use. Just click the "OK" button for now. We'll set that up later.

Now we need to concentrate on the Scroll Window for a second. Click on it in the hierarchy area. Scroll down in the General tab to the entry marked "Horizontal Scrollbar Policy". Change that to 'Always', and then do the same for the Vertical Scrollbar Policy. Save again.

OK, now let's concentrate on our toolbar. This area will be at the top of our application right under the title bar. It will hold various buttons for us that will do the majority of the work. We will use eleven buttons in the toolbar, and, from left to right, they are...

Add, Delete, Clear List, a Separator, Move To Top, Move Up, Move Down, Move to bottom, another Separator, About, and Exit.

Over on the hierarchy area, click on "toolbar1". That should highlight it. At the top of the Glade Designer is something that looks like a pencil. Click that. That brings up the tool bar editor. Click on the Hierarchy tab. You'll see something like this:.

We will be adding all of our toolbar buttons from here. The steps will be:

- Click the Add Button.
- Change the name of the button.
- Modify the label of the button.
- Select the image.

This will be repeated for all eleven of our widgets. So, Click Add, then in the name box, type “tbtnAdd”. Scroll down to the Edit Label portion and type “Add” in the Label box, then a little further down under Edit Image, in the text box for Stock ID, use the pulldown to select “Add”. That takes care of our Add button. We named it “tbtnAdd” so we can reference it in our code later. The “tbtn” is shorthand for 'Toolbar Button'. This way, in our code, it's easy to find and is fairly self documenting.

Now, we need to add the rest of the widgets to our tool bar. Add another button for Delete. This one will be named (as you might guess) “tbtnDelete”. Again, set the label and the icon. Next, add another button naming it “tbtnClearAll” and use the Clear icon. Now we want a Separator. So, click Add, under name type “Sep1” and in the pulldown for type, select Separator.

Add the rest of the widgets naming them “tbtnMoveToTop”, “tbtnMoveUp”, ”tbtnMoveDown”, ”tbtnMoveToBottom”, “Sep2”, “tbtnAbout” and “tbtnQuit”. I'm sure you can find the correct icons. Once you are finished, you can quit the hierarchy window and save your work. You should have something that looks like the image below.

Now, we need to set the event handlers for all the buttons we created. In the hierarchy area, select the tbtnAdd widget. This should highlight both the entry in the hierarchy and the button itself. Go back to the attributes section, select the Signals tab, and expand the GtkToolButton to reveal the clicked event. Under handler in the clicked event, as before, select “on_tbtnAdd_clicked”, then click above or below to force the change. Do this for all the other buttons we created - selecting the “on_tbtnDelete_clicked” event and so on. Remember to click off of it to force the change, and save your project. Our separators don't need events, so just pass over them.

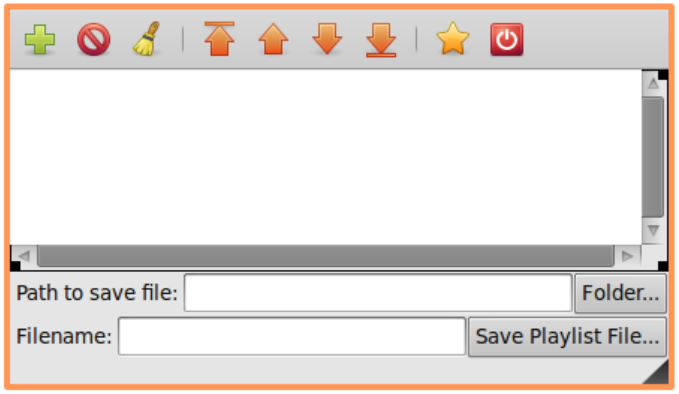
Next, we need to fill in our hboxs. The top hbox will contain (from left to right) a label, a text widget, and a button. In the toolbox, select the label widget (not the blue one), and put it in the left slot. Now put a Text Entry widget in the center slot and a button in the right slot. Do the same for the second hbox.

It's now time to set our attributes for the widgets we just added. In the hierarchy area, select label1 under hbox1. In the attributes section, select the General tab, scroll down to “Edit label appearance” area, and set the label to read “Path to save file:”. Next, go to the Packing tab and set Expand to “No”. You might remember the discussion on packing from last month. Set the padding to 4, which gives a little bit of room on the left and right side of our label. Now select button1 and set the Expand under the Packing tab to “No” also. Go back to the General tab and set the name of our button to “btnGetFolder”. Notice that since this isn't a toolbar button, we didn't preface it with a 't'. Scroll down to the Label entry and enter “Folder...”. Then click on the Signals tab and set the button event of GtkButton/clicked to “on_btnGetFolder_clicked”. Before we set the attributes of the next set of widgets in the next hbox, we need to do one more thing. Select the hbox1 in the hierarchy area and under the Packing tab, set expand to “No”. This makes the hbox take up less space. Finally, set the name of the Text Entry widget to “txtPath”.

Now, do the same thing for hbox2, setting its Expand to “No”,

then set the label text to "Filename:", expand to "No", padding to 4. Set the name of the button to "btnSavePlaylist", its text to "Save Playlist File...", its Expand attribute to "No", set up its clicked event, and set the name of the Text Entry widget here to "txtFilename". Once again, save everything.

So now our window should look something like the image below left.



All that is wonderful, but what did we really do? We can't run this as a program, since we don't have any code. What we have done is create an XML file called "playlistmaker.glade". Don't let the extension fool you. It's really an XML file. If you are very careful, you can open it with your favorite editor (gedit in my case) and look at it.

```
<widget class="GtkWindow" id="MainWindow">
    <property name="visible">True</property>
    <property name="title" translatable="yes">Playlist Maker v1.0</property>
    <property name="window_position">center</property>
    <property name="default_width">650</property>
    <property name="default_height">350</property>
    <signal name="destroy" handler="on_MainWindow_destroy"/>
```

You'll see plain text describing our window and each widget with their attributes. For example, let's look at the code (above) for the main widget, the actual window itself.

You can see that the name of the widget is "MainWindow", its title is "Playlist Maker v1.0", the event handler, and so on.

Let's take a look the code (shown below) for one of our toolbar buttons.

```
<child>
   <widget class="GtkToolButton" id="tbtnAdd">
      <property name="visible">True</property>
      <property name="label" translatable="yes">Add</property>
      <property name="use_underline">True</property>
      <property name="stock_id">gtk-add</property>
      <signal name="clicked" handler="on_tbtnAdd_clicked"/>
   </widget>
   <packing>
       <property name="expand">False</property>
       <property name="homogeneous">True</property>
   </packing>
</child>
```

Hopefully this is starting to make sense to you. Now, we need to write some code to allow us to see our hard work actually do something. Bring up your code editor and start with this...

So, we have created our imports pretty much like we did last month. Notice we are importing "sys" and "MP3" from mutagen.mp3. We installed mutagen back in article number 9, so if you don't have that on your system, refer back to that one.

We'll need the mutagen import for next time, and the sys import is set so the system can exit properly on the last exception.

Next, we need to create our class that will define our window. This is shown above right.

Pretty much the same kind of thing we've done before. Notice the last two lines here. We are defining the glade file (self.gladfile) to be the name of the file we created in the Glade

designer. Notice also that we didn't include a path, just a file name. If your glade file is going to reside somewhere away from your actual code, you must put a path as well. However, it's always smart

```
#!/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)
```

to keep them together. Next, we define our window as self.wTree. We'll be referring to that every time we need to refer to the window. We are also saying that the file is an XML file, and the window we will be using is the one named "MainWindow". You can have multiple windows defined in a single glade file. More on that another time.

Now we need to deal with our events. Last month we used button.connect or window.connect calls to refer to our event handler routines. This time we are going to do something a bit differently. We will use a dictionary. A dictionary is like an array, except rather than being referenced by its index, it's referenced by a key and then has a data element. Key and Data. Here's the code that will probably make it easier to understand. I'm only going to give you two events for now (shown below)...

So we have two events: "on_MainWindow_destroy" and "on_tbtnQuit_clicked" are the keys in our dictionary. The data for our dictionary is "gtk.main_quit" for both entries. Whenever an event is triggered by our GUI, the system uses the event to find the key of our dictionary, then knows what routine to call - from the data segment. Next we need to connect the dictionary to the signal handler of our window. We do it

```
    #================================================
    #                  Create Event Handlers
    #================================================
    dict = {"on_MainWindow_destroy": gtk.main_quit,
            "on_tbtnQuit_clicked": gtk.main_quit}
```

```
class PlayListMaker:
    def __init__(self):
        #================================================
        #                        Window Creation
        #================================================
        self.gladefile = "playlistmaker.glade"
        self.wTree =
gtk.glade.XML(self.gladefile,"MainWindow")
```

with the following line of code.

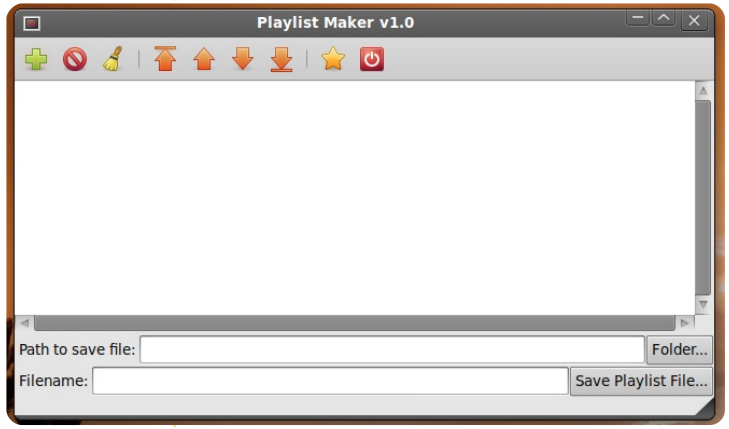
```
self.wTree.signal_a
utoconnect(dict)
```

We're almost ready. We still need our main routine code:

```
if __name__ ==
"__main__":
    plm = PlayListMaker()
    gtk.main()
```

Save this file as "playlistmaker.py". Now you can run it (shown above right).

It doesn't do much right now, other than open and close properly. The rest is for next time. Just to whet your appetite, we'll be discussing the use of the TreeView, Dialog boxes, and adding a bunch more code. So tune in next time.



Glade file:
http://fullcirclemagazine.pastebin.com/YM6U0Ee3

Python source:
http://fullcirclemagazine.pastebin.com/wbfDmmBh

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.