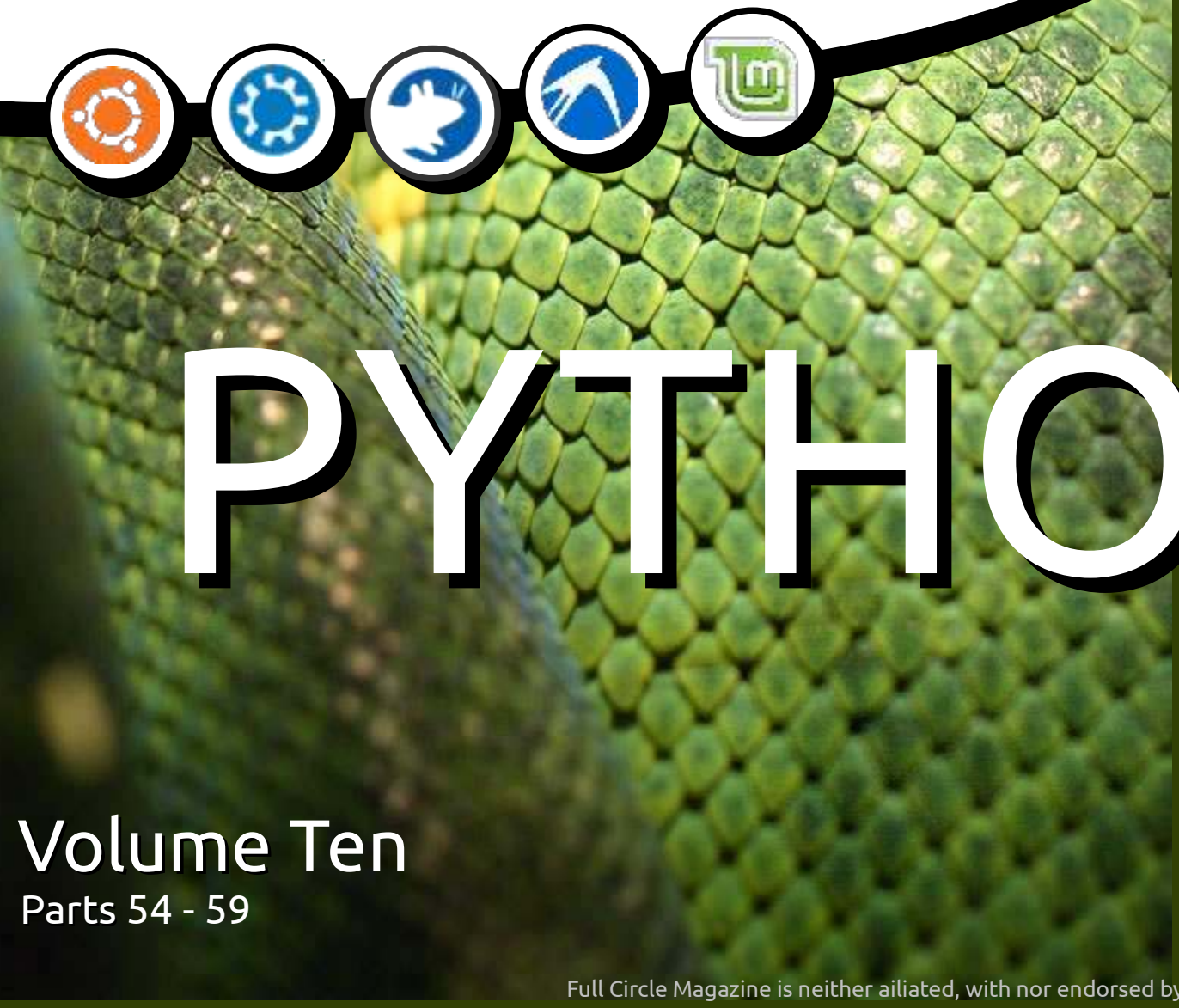
Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

PROGRAMMING SERIES SPECIAL EDITION

PROGRAMMING SERIES SPECIAL EDITION



PYTHON

Volume Ten

Parts 54 - 59

Program In Python - Part 54

Written by Greg D. Walters

Many years ago, I was dealing with high blood pressure issues. My doctor suggested that I do something that allowed me to concentrate on something fairly useful, but rather trivial. I dealt with it by trying to do counted cross stitch. It's creative, focused, and keeps your mind occupied on what you are doing, not what is bothering you. I find myself in that position again, so I broke out the hoop and needles and started again.

In case you aren't familiar with counted cross stitch, I'll give you a gross overview of what it is. Cross stitch is a type of needlework that uses tiny 'x' patterns of thread that eventually make up a picture. The thread is called "floss" and the fabric that you use is called "aida". According to Wikipedia, aida is a special fabric that has tiny squares that have small holes at regular intervals that form the squares. This facilitates the placement of the "x" patterns that make the image. There are two types of cross stitch. One has an image printed on the aida (sort of like paint by numbers), and the other uses totally blank aida that you count stitches from the pattern. The second is much harder than the first. Go to your favorite fabric store or craft section of your local mega-mart and you'll get the idea.

Also a while back, I started playing with creating a program that would take an image and convert it into a cross stitch pattern. One thing lead to another, and I had to shelve the program for other things. I've now dusted off the idea and started anew.

We will spend the next few articles dealing with this project. It will take a while, since some things are fairly complex and have many parts to them. Here is the "game plan":

- Create a database for the pixel colors to floss colors.
- Create a GUI using Tkinter for the application.
- Flesh out the application to do the manipulation of the image files.
- Create a PDF file that will be the ultimate pattern for the project.

What you will learn

- Revisitation of database and XML manipulation.
- Revisitation of Tkinter GUI programming. If you missed the previous articles on this, please refer to FCM issues 51 thru 54.
- Image manipulation using PIL (http://pillow.readthedocs.org/en/latest/).
- PDF creation using pyFPDF (https://code.google.com/p/pyfpdf).

Getting Started

The first thing in our list of tasks is to create the database that will hold the DMC(™) floss colors and reference them to the closest approximation to the RGB (Red, Green, Blue) values that are used in images on the computer. At the same time, the database will hold the hex value and the HSV (Hue, Saturation, Value) representations for each floss color. It seems that HSV is the easiest way to find the "closest" representation of a color that will match the floss colors. Of course, the human eye is the ultimate decision maker. If you are not familiar with HSV color representations, there is a rather complex writeup on Wikipedia at http://en.wikipedia.org/wiki/HSL_and_HSV. It might help, but it might make things less clear.

The first thing we need is an XML file that has the DMC floss colors with a RGB conversion. The best one I found is at http://sourceforge.net/p/kxstitch/feature-requests/9/. The file you want is dmc.xml. Download it and put it in a folder that you will use to hold the Python code.

Now we will be using apsw (below) to do our database manipulation, which you should

```
# makedb.py
# DMC.xml to SQLite database
# For Full Circle Magazine #85

import apsw
from xml.etree import ElementTree as ET
tablename = "DMC"
```

already have and ElementTree to do the XML parsing (which is included in Python version 2.7+).

As always, we start with our imports. In this program, we have only the two. We also set the name of the table.

The next portion should be familiar if you have been reading the articles for a while. We create a function that will read the XML file, and parse it for us. We then can use the information to load the database. A snippet of the XML file is shown top right.

We are looking for the <floss> tag for each line of information. To do this, we use the .findall('floss') command. Once we have the information line, we break each tag (name, description, etc.) into separate variables to place into the database. When it comes to the <color> tag, we use the .floss.findall('color') command to get each value of Red, Green and Blue.

We start by telling the function that we will be using the global variables connection and cursor. We then set the filename of the XML file, parse the XML file, and get started. We also use a counter variable to show that something is happening while the parsing and database inserts are going on.

Now that we have all our data, we need to create the SQL insert statement and execute it. Notice the "\" after the word VALUES in the SQL statement. That is a line-continuation character to make it easier for printing here in the magazine. We will be creating the database and table in a few moments.

```
    SQL = "INSERT INTO DMC
(DMC,Description,Red,Green,Bl
ue) VALUES \

('%s','%s',%s,%s,%s)" %
(name,desc,red,green,blue)

 cursor.execute(SQL)
```

Now, we print to the terminal window that something is going on:

```
    print "Working record
{0}".format(cntr)

    cntr += 1
```

Now we create and/or open the database in the OpenDB routine (bottom right). If you've been with us when we have done database work before, you will notice that we are using two cursors this time. The cursor variable is used for the "normal" inserts, and later on in the select statement for the update to set the hex and HSV values. We have to use two cursors, since if you modify a cursor in the middle of a logic statement, you lose everything with the new command. By using 'ucursor', we can use that for the update statements. Other than that, it is our normal OpenDB routine.

```
<floss>
    <name>150</name>
    <description>Dusty Rose Ultra VDK</description>
    <color>
         <red>171</red>
         <green>2</green>
        <blue>73</blue>
    </color>
</floss>
```

```
def ReadXML():
    global connection
    global cursor
    fn = 'dmc.xml'
    tree = ET.parse(fn)
    root = tree.getroot()
    cntr = 0
    for floss in root.findall('floss'):
        name = floss.find('name').text
        desc = floss.find('description').text
        for colour in floss.findall('color'):
            red = colour.find('red').text
            green = colour.find('green').text
            blue = colour.find('blue').text
```

```
def OpenDB():
    global connection
    global cursor
    global ucursor
    global dbname
    connection = apsw.Connection("floss.db3")
    cursor = connection.cursor()
    ucursor = connection.cursor()
```

Now that the database is created and/or opened, we can set up our table (top right). Notice that the SQL statement below uses the triple quote to allow for the line to break neatly for viewing.

```
def MakeTables():
    sql = '''CREATE TABLE IF NOT EXISTS DMC
        (pkID INTEGER PRIMARY KEY, DMC INTEGER,
         Description TEXT, Red INTEGER, Green INTEGER, Blue INTEGER,
         HEX TEXT,H INTEGER,S INTEGER,V INTEGER)'''
    cursor.execute(sql)
```

The EmptyTables routine (middle right) is there just to make sure that if we want to or need to run the application more than once, we start with a clean and empty table if it exists.

```
def EmptyTables():
    sql="DELETE FROM %s" % tablename
    cursor.execute(sql)
```

IF we were to stop here, we would have a reasonable working database with the DMC color, color name and the RGB values associated with each. However, as I alluded to before, it is easier to pick the closest floss color by using the HSV data.

We next create the hex value from the RGB values (middle left).

```
def rgb2hex(rgb):
    return '%02x%02x%02x' % rgb
```

The next function creates the HSV values from the RGB values. I found the algorithm on the internet. You can research it there.

```
def rgb2hsv(r, g, b):
    r, g, b = r/255.0, g/255.0, b/255.0
    mx = max(r, g, b)
    mn = min(r, g, b)
    df = mx-mn
    if mx == mn:
        h = 0
    elif mx == r:
        h = (60 * ((g-b)/df) + 360) % 360
    elif mx == g:
        h = (60 * ((b-r)/df) + 120) % 360
    elif mx == b:
        h = (60 * ((r-g)/df) + 240) % 360
    if mx == 0:
        s = 0
    else:
        s = df/mx
        v = mx
    return int(round(h,0)), int(round(s*100,0)),
int(round(v*100,0))
```

Finally, we create the UpdateDB function (next page, top left). We use the SELECT * FROM DMC command and use the "standard" cursor variable to hold the data. We then step through the returned data, and read the RGB values, and pass them to the rgb2hex function as a tuple and to the rgb2hsv function as three separate values. Once we get the return values, we use the update SQL command to match the proper record by using the primary key (pkID). As I stated before, we have to use a separate cursor for the update statement.

The last thing we do is call each of the functions in order to create the database, and, at the end, we print "Finished" so the user knows everything is done.

```
OpenDB()
MakeTables()
EmptyTables() # Just to be
safe
ReadXML()
UpdateDB()
print "Finished"
```

I named this program "MakeDB". The database should be created in the same folder where the code and XML file are located. As always, the full code can be found at http://pastebin.com/Zegqw3pi.

Next time, we will work on the GUI. We use Tkinter for the GUI, so, in the meantime, you might want to refresh your memory by looking at FCM issues 51 thru 54 where I take you through Tkinter.

Until next time, have a good month.

Program In Python - Part 55

Written by Greg D. Walters

This is the second in a multi-part tutorial on creating a Cross Stitch pattern generator. In the first part (FCM85), we created a database containing the DMC™ floss colors with their closest RGB values. In this part, we will create the GUI using Tkinter. We will also use PIL (Python Imaging Library) and PMW (Python Mega Widgets). You'll need to download those libraries and install them before we go too far. For PIL, go to the Pillow fork at https://github.com/python-imaging/Pillow and download the latest version. For PMW, go to http://pmw.sourceforge.net/ and download from there.

You will also need two image files. One is a simple grey rectangle 500x400 pixels. You can use GIMP or some other image manipulating program to create it. Name it default.jpg, and place it into your source code directory along with the database. The other is an image of a folder for the open image button. I got one from open clipart and searched for the word "folder". I found a reasonable one at https://openclipart.org/detail/177890/file-folder-by-thebyteman-177890. Open it in GIMP, resize it to 30x30 and save it in the same directory as the other two files as "open.gif".

Above is a screenshot of what the finished GUI will look like. There are four main frames in the GUI. Three on the left side and one on the right. When we go through the build widget process, I refer to them as Top Frame, Middle Frame, Bottom Frame and Side Frame. The top frame deals with the original image. The middle frame deals with the processing of the image. The bottom frame shows the original image on the left and the processed image on the right, and the side frame displays the colors and floss required. It seems from first glance there is a lot of wasted space here, but when you see the program run, it doesn't really have that much empty space, once we get through the processing portion.

Now we are ready to start working on the code. Here is our long list of imports...

```
from Tkinter import *
import tkFileDialog
import tkCommonDialog
import tkMessageBox
import ttk
from PIL import
Image,ImageTk,ImageOps
import Pmw
import apsw  # Database
Access
import math  # Math library
import sys
```

From the sheer number of imports, you can tell this is going to be a long program. In fact, the UI portion of the code will be over 300 lines, including comments. The "good" news is that about 200 of the lines of code deal with the Tkinter portion of the program, the actual GUI itself. The majority of the remaining lines of code in this portion are stubs for functions needed for the next article.

We'll create a class to hold all of our UI processing code (next page,

top right).

First, we have the class definition and next we have the __init__ function which we pass the TopLevel "root" window into. We create the TopLevel window in the last four lines of the program. Within the __init__ function we are defining all the global variables and doing some initial assignments before we start the other functions. The first thing we do is create a list of Tuples that hold the picture file formats that we need when we call the OpenFile dialog. The next two lines below, define and ready the two image files we just created (open folder GIF file, and the grey rectangle – which will be used as placeholders for our images used to create the pattern.

```
self.openimage =
PhotoImage(file='open.gif')

self.DefaultImage
=ImageTk.PhotoImage(self.Thum
bnail("default.jpg",450,450))
```

Now we get into the global definitions (middle right). If you remember, when you use Tkinter, if you have a widget like a text entry box or combo box that you want to retrieve the information selected or entered, you define a global variable and then assign it to a Variable Class (BooleanVar, DoubleVar, IntVar or StringVar). This will then "track" changes to the values within the widget values so you can access them with the .get() or .set() methods. In the next lines of code, we create the global variable name, then assign it to the proper wrapper class. I put some comments into the code to try to help you keep track of what we are doing.

As you can see, we are setting a variable called OriginalFilename, which holds the image that we want to create the pattern from, OriginalColorCount which holds the number of colors in the original image file, and OriginalSize which holds the size in pixels of the original image. As they say on tv... "***BUT WAIT! THERE'S MORE!***" (bottom right):

The ComboStitch variable is set by a combobox, and handles the stitch size of the aida that you wish to use for your project. The ComboSize variable is also set by a combo box and holds the size of the aida fabric. FabricHeight and FabricWidth are the breakdowns from the aida size. MaxColors is a value from an entry box to set the number of colors, and BorderSize is a floating point value that specifies the amount of unused aida for framing.

```
class XStitch:
    def __init__(self, master):
        self.picFormats = [
        ('JPEG / JFIF','*.jpg'),
        ('Portable Network Graphics','*.png'),
        ('CompuServer GIF','*.gif'),
        ('Windows Bitmap','*.bmp'),
        ('All File Types *.*','*.*'),
        ]
```

```
        #-----------------------------------------
        #                Global Definitions
        #-----------------------------------------
        #                    UI Required
        global OriginalFilename
        OriginalFilename = StringVar()
        global OriginalColorCount
        OriginalColorCount = StringVar()
        global OriginalSize
        OriginalSize = StringVar()
```

```
        global ComboStitch
        ComboStitch = IntVar()
        global ComboSize
        ComboSize = StringVar()
        global FabricWidth
        FabricWidth = DoubleVar()
        global FabricHeight
        FabricHeight = DoubleVar()
        global MaxColors
        MaxColors = IntVar()
        global BorderSize
        BorderSize = DoubleVar()
```

```
global ProcessedColors

ProcessedColors = StringVar()
```

```
global ProcessedSize

ProcessedSize = StringVar()

global DmcColor

DmcColor = StringVar()
```

The final 'variable class' variables are used for information once we have processed the original image to the desired parameters.

The next set of globals is (top right) used for easy access throughout the program. For the most part, they are either obvious by their name, or will become obvious once we use them. There are three not-so-obvious variables here. backgroundColor1 and backgroundColor2 are tuples that are used in the gridding process, and the ReadyToProcess variable is used to designate that the original image is loaded and everything is ready to go – just in case the user presses the Process button too early.

Finally we have assigned all our globals, and now have the code that actually creates the GUI. We open the database, create the menu, set up the widgets, and finally place the widgets into the proper places. Just to give you a heads-up, we will be using the Grid geometry placement manager. More on that later.

```
#------------------------

self.OpenDB()

self.MakeMenu(master)

frm =
self.BuildWidgets(master)

self.PlaceWidgets(frm)
```

The next portion of our code (middle right) will set up the menu bar. I've tried to lay it out logically so it will be easy to understand.

We define a function called MakeMenu, and pass in the TopLevel window. We then define the three menu sets we will be creating. One for File, one for Process, and one for Help.

```
menu.add_cascade(label="F
ile", menu=filemenu)

menu.add_cascade(label="P
rocess",menu=process)

menu.add_cascade(label="H
elp",menu=help)
```

Now we set up the File menu options (bottom right). Open will open our image and uses a function called "GetFileName". Save will create the output PDF file and uses the FileSave function. We add a separator and finally an Exit function.

```
        #----------------------------------------
        global ShowGrid
        ShowGrid = True
        global ProcessedImage
        ProcessedImage = ""
        global GridImage
        GridImage = ""
        global backgroundColor1
        backgroundColor1 = (120,)*3
        global backgroundColor2
        backgroundColor2 = (0,)*3
        global ReadyToProcess
        ReadyToProcess = False
```

```
#=========================================================
#                    BEGIN UI DEFINITION
#=========================================================
    def MakeMenu(self,master):
            menu = Menu(master)
            root.config(menu=menu)
            filemenu = Menu(menu, tearoff=0)
            process = Menu(menu,tearoff=0)
            help = Menu(menu,tearoff=0)
```

```
#----------------------------------------
    #                  File Menu
    #----------------------------------------
    filemenu.add_command(label="New")
    filemenu.add_command(label="Open", command=self.GetFileName)
    filemenu.add_command(label="Save", command=self.FileSave)
    filemenu.add_separator()
    filemenu.add_command(label="Exit", command=self.DoExit)
```

Now we have the Process option and the Help functions (next page, top right).

All of the options in the menu bar are also available from various buttons within the program.

Now we will make our BuildWidgets function. This is where we create all the widgets that will be used on the GUI.

```
def
BuildWidgets(self,master):
```

```
        self.frame =
Frame(master,width=900,height
=850)
```

We start with the function (bottom right) definition, passing in the TopLevel window (master) and placing a frame that holds all of our other widgets. I've added comments to help realize which part of code deals with which frame. We'll deal with the top frame first.

Assuming you remember or refreshed your memory on Tkinter, it should be fairly straight-forward. Let's look at the first label as a discussion item.

```
self.label1 =
Label(self.frm1,text =
"Original Filename: ")
```

First, we define the name of the widget (self.label1 =). Next we set that variable to which widget type we want to use; in this case Label. Finally we set the parameters we want to apply to that widget starting with the parent widget (self.frm1), and in this case, the text that will show up in the label. Now let's take a moment to look at the button self.btnGetFN.

```
 self.btnGetFN =
Button(self.frm1, width=28,
image=self.openimage,

command=self.GetFileName)
```

First thing to notice is that this is broken into two lines. You can safely place everything on one line...it is just too long to fit into a 72-character line. We'll really pay attention to the parameters we use here. First the parent (frm1), next the width which is set at 28. When we use a widget that has the option of text or an image, we have to be careful setting the width. If it will contain text, the width parameter is the number of characters it will hold. If it is to display an image, it will be set at the number of pixels. Finally, we set the command parameter, which tells the system what function to call when the button is clicked.

```
#-----------------------------------------
#                   Process Menu
#-----------------------------------------
process.add_command(label="All",command=self.Process)
#-----------------------------------------
#                    Help Menu
#-----------------------------------------
help.add_command(label="Help",command=self.ShowHelp)
help.add_separator()
help.add_command(label="About",command=self.ShowAbout)
```

One more thing to look at is the textvariable parameter. This tells us what variable will hold the information that will be displayed in the widget. We set these in the __init__ function earlier. One other thing to mention is that the frame itself has two parameters you might not remember. The Relief parameter sets the border type of

```
# ---------------Middle Frame --------------------
self.frm2 = Frame(self.frame,width=900,height=160,bd=4,relief=GROOVE)
self.lbl4 = Label(self.frm2,text="Aida Stitch Size: ")
self.lbl5 = Label(self.frm2,text="Aida Fabric Size: ")
self.TCombobox1 = ttk.Combobox(self.frm2,textvariable=ComboStitch,width=8)
self.TCombobox1.bind('<<ComboboxSelected>>', self.StitchSizeSelect)
self.TCombobox1['values'] = (7,10,11,12,14,16,18,22)
self.TCombobox2 = ttk.Combobox(self.frm2,textvariable=ComboSize,width = 8)
self.TCombobox2.bind('<<ComboboxSelected>>',self.AidaSizeSelect)
self.TCombobox2['values'] = ("12x18","15x18","30")
```

```
# ---------------- TOP FRAME --------------------
self.frm1 = Frame(self.frame,width=900,height=100,bd=4,relief=GROOVE)
self.label1 = Label(self.frm1,text = "Original Filename: ")
self.entFileName = Entry(self.frm1,width=50,textvariable=OriginalFilename)
self.btnGetFN = Button(self.frm1, width=28, image=self.openimage,
     command=self.GetFileName)
self.label2 = Label(self.frm1,text = "Original Colors: ")
self.lblOriginalColorCount = Label(self.frm1,text="",width=10,
     textvariable=OriginalColorCount)
self.label3 = Label(self.frm1,text = "Original Size: ")
self.lblOriginalSize = Label(self.frm1,text="",width=10,
     textvariable=OriginalSize)
```

the frame, which in this case is GROOVE, and the bd parameter sets the border width. Border width defaults at 0 so if you want to see the effect, you have to set the border width (bd is a shortcut).

Now we'll deal with the middle frame widgets.

The last 6 lines of this section (previous page, middle right) deal with the two combo boxes in the UI. Each combo box uses three lines (the way I programmed it to make it easy to understand). In the first line, we set the basic parameters. The next line, we bind the combobox selection-changed event to the function StitchSizeSelect, and the last line has a list of the values that will be available for the pulldown.

Everything else above is pretty "normal" stuff. Now we set our defaults for the widgets that need them. Again, we are using the global variables that we set up in the __init__ function and wrapped to the widget variable class.

```
ComboStitch.set(14)

ComboSize.set("15x18")

FabricWidth.set(15)

FabricHeight.set(18)

MaxColors.set(50)

BorderSize.set(1.0)
```

Now we deal with the bottom frame. This is really simple, since we have to set up only the frame and two labels which we will use to hold our images.

Finally we deal with the side frame. The side frame will hold a ScrolledFrame from the PMW library. It's really easy to use and provides a nice interface to the information about the floss that should be used. You can research the ScrolledFrame on your own, since we still have a lot to cover here.

```
        self.lbl6 = Label(self.frm2,text="Max Colors: ")
        self.entMaxColors = Entry(self.frm2,textvariable=MaxColors,width=3)
        self.lbl7 = Label(self.frm2,text="Border Size: ")
        self.entBorderSize = Entry(self.frm2,textvariable=BorderSize,width = 8)
        self.frmLine = Frame(self.frm2,width=6,height=80,bd=3,relief="raised")
            self.lbl8 = Label(self.frm2,text="          Processed Image Colors: ")
        self.lbl9 = Label(self.frm2,text="Processed Image Stitch Count: ")
        self.lblProcessedColors = Label(self.frm2, width=10,textvariable=ProcessedColors,
             justify=LEFT)
        self.lblProcessedSize = Label(self.frm2, width=10, textvariable=ProcessedSize,
             justify=LEFT)
        self.btnDoIt = Button(self.frm2,text="Process",width=11,command = self.Process)
        self.btnShowGrid = Button(self.frm2,text="Hide Grid", width=11,
             command=self.ShowHideGrid)
        self.btnCreatePDF = Button(self.frm2, text="Create PDF", width=11,
             command=self.CreatePDF)
```

```
        # --------------- Bottom Frame ------------------
        self.frm3 = Frame(self.frame,width=450,height=450,bd=4,relief=GROOVE)
        self.lblImageL = Label(self.frm3, image=self.DefaultImage,
             height=400, width=400, borderwidth=2, relief=GROOVE)
        self.lblImageR = Label(self.frm3, image=self.DefaultImage, height=400,
             width=400,borderwidth=2, relief=GROOVE)
```

```
        #---------------- Side Frame ------------------
        self.frm4 = Frame(self.frame,width = 300,height=580,bd=4,relief=GROOVE)
        # Create the ScrolledFrame.
        self.sf = Pmw.ScrolledFrame(self.frm4,
            labelpos = 'n', label_text = 'Processed Color List',
            usehullsize = 1,
            hull_width = 300,
            hull_height = 567,)
        return self.frame
```

That's all for the widgets. Now we have to place them. As I said earlier, we will be using the Grid geometry manager, rather than the absolute or pack managers.

The Grid method places the widgets in (you guessed it) a grid, referenced by row and column designations. I'll use the top frame as an example (shown top right).

First we place the frame.

You can see that we place the widget by using the {widgetname}.grid command, then the row and column positions. Notice that we are telling the entry widget to span 5 columns. Padx and pady values will place some extra space on both the right and left sides (padx) or the top and bottom (pady). The sticky parameter is similar to a justify command for text.

The middle frame is a bit more complicated, but basically the same as the top frame. You might notice an extra frame in the middle of the code (self.frmLine). This gives us a nice divider between the options section and the display section. Since there is no horizontal or vertical line widget, I cheated and used a frame with a width of 6 pixels and border width of 3, making it just look like a fat line.

```
ROW | Col 0   | Col 1 - Col 6           |Col 7    | Col 9 | Col 10                  |
    ------------------------------------------------------------------------------
 0  | Label1  | entFileName             |btnGenFN| Label2|lblOriginalColorCount    |
 1  |                                            | Label3|lblOriginalSize          |
    ------------------------------------------------------------------------------
```

```
    def PlaceWidgets(self,frame):
        frame.grid(column = 0, row = 0)
        # ---------------- TOP FRAME ---------------------
        self.frm1.grid(column=0,row=0,rowspan=2,sticky="new")
        self.label1.grid(column=0,row=0,sticky='w')
        self.entFileName.grid(column=1,row=0,sticky='w',columnspan = 5)
        self.btnGetFN.grid(column=7,row = 0,sticky='w')
        self.label2.grid(column=9,row=0,sticky='w',padx=10)
        self.lblOriginalColorCount.grid(column=10,row=0,sticky='w')
        self.label3.grid(column=9,row=1,sticky='w',padx=10,pady=5)
        self.lblOriginalSize.grid(column=10,row=1,sticky='w')
```

```
        # ---------------- MIDDLE FRAME ---------------------
        self.frm2.grid(column=0,row=2,rowspan=2,sticky="new")
        self.lbl4.grid(column=0,row=0,sticky="new",pady=5)
        self.lbl5.grid(column=0,row=1,sticky="new")
        self.TCombobox1.grid(column=1,row=0,sticky="new",pady=5)
        self.TCombobox2.grid(column=1,row=1,sticky="new")
        self.lbl6.grid(column=2,row = 0,sticky="new",padx=5,pady=5)
        self.entMaxColors.grid(column=3,row=0,sticky="new",pady=5)
        self.lbl7.grid(column=2,row=1,sticky='new',padx=5)
        self.entBorderSize.grid(column=3,row=1,sticky='new')
        self.frmLine.grid(column=4,row=0,rowspan=2,sticky='new',padx=15)
        self.lbl8.grid(column=5,row=0,sticky='new',pady=5)
        self.lbl9.grid(column=5,row=1,sticky='new')
        self.lblProcessedColors.grid(column=6,row=0,sticky='w')
        self.lblProcessedSize.grid(column=6,row=1,sticky='new')
        self.btnDoIt.grid(column=7,row=0,sticky='e',padx=5,pady = 5)
        self.btnShowGrid.grid(column=7,row=1,sticky='e',padx=5,pady = 5)
        self.btnCreatePDF.grid(column=8,row=0,rowspan=2,sticky='ew',padx=10)
```

```
    # ---------------- BOTTOM FRAME ---------------------
    self.frm3.grid(column=0,row=4,sticky="nsew")
    self.lblImageL.grid(column=0,row=0,sticky="w")
    self.lblImageR.grid(column=1,row=0,sticky="e")
```

The bottom frame is simple since we have only the frame and the two labels to hold the images.

The side frame is pretty much the same thing, except the ScrolledFrame allows for a frame to be set to the interior of the scrolled frame widget. We then create three widgets here and place them in their grids as column headers. We do this since we assigned the interior frame for the scroll frame here and we have to assign the parent (self.sfFrame) after we have created it.

That's all the hard work for now. At this point, we will create all of the functions that we need to get the GUI to run, stubbing most of them until next month. There are a few we will go ahead and complete, but they are fairly short.

The first function will be the Exit option from the menu bar. It's under the File menu option.

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

The only other one is the Thumbnail function. We need this to fill the grey rectangles into the labels in the bottom frame. We pass the filename and the width and height that we want the thumbnail image to be.

Since this article is so long, I'm going to give you a list of function names and all you have to do is stub it out by using the pass command. We'll fill them in next month. I'll give you the first one as an example, but you should already know how to do it.

```
def GetFileName(self):
    pass
```

For the rest of the functions, I'll just give you the def lines. Be sure to include them all in your code.

You can see, we have a large amount of work to do next month. We still have four more lines to write to finish up for this month. This is out of our class code.

```
root = Tk()

root.title("Cross Stitch Pattern Creator")
```

```
# --------------- SIDE FRAME ---------------------
self.frm4.grid(column=2,row=0,rowspan=12,sticky="new")
self.sf.grid(column=0,row=1)
self.sfFrame = self.sf.interior()
self.lblch1 = Label(self.sfFrame,text="        Original")
self.lblch2 = Label(self.sfFrame,text="        DMC")
self.lblch3 = Label(self.sfFrame,text="Name/Number")
self.lblch1.grid(column=0,row=0,sticky='w')
self.lblch2.grid(column=1,row=0,sticky='w')
self.lblch3.grid(column=2,row=0,sticky="w")
```

```
    def Thumbnail(self,file,hsize,wsize):
        size = hsize,wsize
        extpos = file.rfind(".")
        outfile = file[:extpos] + ".thumbnail"
        im = Image.open(file)
        im.thumbnail(size)
        im.save(outfile,"JPEG")
        return im
```

```
test = XStitch(root)

root.mainloop()
```

The first line sets up the root TopLevel window. The next line sets the title on the top line. The third line instantiates our XStitch class, and the last line starts the main loop that shows the UI and gives control over to it.

Well that's a lot for this month, but we are finally done. You can actually run the program to see the GUI.

As always, the code is available on Pastebin at http://pastebin.com/XtBawJps.

Next month we will flesh out the code. See you then.

```
def ShowHelp(self):, def ShowAbout(self):, def OpenDB(self):, def ShowHideGrid(self):
def StitchSizeSelect(self,p):, def AidaSizeSelect(self,p):, def Process(self):
def CreatePDF(self):, def OriginalInfo(self,file):, def GetColorCount(self,file):
def GetHW(self,file):, def GetHW2(self,file):, def GetColors(self,image):
def Pixelate(self,im,pixelSize):, def ReduceColours(self,ImageName):
def MakeLines(self,im,pixelSize):, def MakeLines2(self,im,pixelSize):
def Rgb2Hex(self,rgb):, def FillScrolledList(self,filename):
def GetBestDistance(self,r1,g1,b1):
```

Program In Python - Part 56

We've been working on a Cross Stitch pattern generator. Last month we did the UI portion, and now it's time to do the code that does the most of the work. Next month we will start working on the PDF file output portion.

We'll work on the menu items first. The code is shown below.

The global ReadyToProcess variable is used to make sure that if the user presses the Process button, the system doesn't try to process things without anything to process. We use the tkFileDialog askopenfilename built-in dialog routine to get the filename of the original image. We then get the number of colors in the original image as well as the width and height. We save those values and display them in the GUI. We then open the image and create a thumbnail image to display in the left image in the bottom frame. See the text box to the right.

Next we do the ShowHideGrid function. This simply exchanges two images in the right image label based on the global variable ShowGrid. If False, we change the text on the show/hide button, then set the ShowGrid variable to true and set the image to the one with the grid. Otherwise we change the text on the show/hide button to "Show Grid", set the ShowGrid variable to False and put up the ungridded image. Code is on the next page, top left.

The StitchSizeSelect function is fired whenever the stitch size combobox is changed. We get the value from the combo box and assign it to a local variable.

```
def StitchSizeSelect(self,p):

selection = ComboStitch.get()
```

```
        OriginalFilename.set(fileName)
        OriginalColorCount.set(self.GetColorCount(fileName))
        OriginalSize.set(self.GetHW(fileName))
        masterimage=Image.open(fileName)
        masterimage.thumbnail((400,400))
        self.img = ImageTk.PhotoImage(masterimage)
        self.lblImageL['image'] = self.img
        ReadyToProcess = True
```

The FileSave menu option will simply call the CreatePDF routine, once it's finished.

```
    def FileSave(self):
        self.CreatePDF()
```

We'll stub out the ShowHelp and ShowAbout routines with a dialog box saying that those options are not yet available.

```
    def ShowHelp(self):
        tkMessageBox.showinfo(title="Help",message='Sorry,
but help is not yet available.')

    def ShowAbout(self):
        tkMessageBox.showinfo(title="About",message='Sorry,
but the About function is not yet available.')
```

We've written the OpenDB routine a dozen times before, so you should know what it does.

```
    def OpenDB(self):
        global connection
        global cursor
        #-------------------------------
        connection = apsw.Connection("floss.db3")
        cursor = connection.cursor()
```

```
    def GetFileName(self):
        global ReadyToProcess
        #-------------------------------
        fileName = tkFileDialog.askopenfilename(parent=root,filetypes=self.picFormats ,title="Select File to open...")
```

```
    def ShowHideGrid(self):
        global ShowGrid
        #-------------------------------
        if ShowGrid == False:
            self.btnShowGrid['text'] = 'Hide Grid'
            ShowGrid = True
            self.im2=Image.open(self.GridImage)
            self.im2.thumbnail((400,400))
            self.img3 = ImageTk.PhotoImage(self.im2)
            self.lblImageR['image'] = self.img3
        else:
            self.btnShowGrid['text'] = 'Show Grid'
            ShowGrid = False
            self.im2=Image.open(self.ProcessedImage)
            self.im2.thumbnail((400,400))
            self.img3 = ImageTk.PhotoImage(self.im2)
            self.lblImageR['image'] = self.img3
```

```
    def AidaSizeSelect(self,p):
        selection = ComboSize.get()
        if selection != "30":
            pos = selection.find("x")
            width = int(selection[:pos])
            height=int(selection[pos+1:])
        else:
            width = 30
            height = 30
        FabricWidth.set(width)
        FabricHeight.set(height)
```

The AidaSizeSelect function (top right) is very similar to the StitchSizeSelect function. We set the FabricWidth and FabricHeight globals based on the selection on the combo box. We also default to 30x30 if they select 30.

We have a variable called ReadyToProcess (below) just in case the user tries to run the process function before the image is loaded.

We pixelate the original file to a 5x5 pixel matrix This allows us to group that 5x5 matrix to a single color. We then reduce the colors, get the width and height of the processed image and set the size so the user can see how big the resulting image will be.

```
# Place image

self.im2=Image.open(Reduced)

self.im2.thumbnail((400,400))

self.img3 =
ImageTk.PhotoImage(self.im2)

self.lblImageR['image'] =
self.img3

self.ProcessedImage =
'im1.png'
```

The above set of code places the processed image into the image that will hold the processed image. The next set of code will create a grid so that the user will have the grid to do the cross stitching.

```
self.MakeLines(Reduced,5)

self.MakeLines2('output.png',
50)

self.im2 =
Image.open('output2.png')

self.im2.thumbnail((400,400))

self.img3 =
ImageTk.PhotoImage(self.im2)

self.lblImageR['image'] =
self.img3

self.FillScrolledList('output
.png')
```

```
    def Process(self):
        global ReadyToProcess
        #-------------------------------
        if ReadyToProcess == False:
            tkMessageBox.showinfo(title="ERROR...",message='You must load an original imaage first.')
        else:
            newimage = self.Pixelate(OriginalFilename.get(),5)
            Reduced = self.ReduceColors(newimage)
            W,H = self.GetHW2(Reduced)
            siz = "{0}x{1}".format(W/5,H/5)
            ProcessedSize.set(siz)
```

```
self.GridImage =
'output2.png'
```

We stub the CreatePDF function until we finish the PDF function next month.

```
    def CreatePDF(self):

tkMessageBox.showinfo(title="
Create PDF",message='Sorry,
but the Create PDF function
is not yet available.')
```

The OriginalInfo() routine gets and sets variables based on the original image format, size and mode.

```
def OriginalInfo(self,file):
  im = Image.open(file)
  imFormat = im.format
  imSize = im.size
  imMode = im.mode

  self.size = imSize
  self.imformat = imFormat
  self.immode = imMode
```

The GetColorCount function uses the .getcolors method to get the number of colors in the image file. We have to use 1600000 as the maxcolors parameter because if the image contains more than 256 colors (or whatever is in the parameter, the method returns 'None'. This function is similar to the GetColors function except the GetColors works with an already opened image file. If you use GetColorCount, you have to pass an unopened file.

```
def GetColorCount(self,file):
  im = Image.open(file)
  numColors =
im.getcolors(1600000)
  self.colors =
len(numColors)
  return self.colors
```

The next two functions return the height and width of the image file in pixels. The difference between the two is that GetHW returns a string like 1024x768 and GetHW2 returns two integers.

```
def GetHW(self,file):
  im = Image.open(file)
  tmp =
"{0}x{1}".format(im.size[0],i
m.size[1])
  return tmp

def GetHW2(self,file):
  im = Image.open(file)
  return
im.size[0],im.size[1]
```

GetColors will get the number of colors in the passed image file. We use 1.6 million colors as the parameter, because the image.getcolors() routine defaults to 0 over color count over 256.

```
def GetColors(self,image):
  numColors =
image.getcolors(1600000)
  colors = len(numColors)
```

```
def Pixelate(self,im,pixelSize):
    image = Image.open(im)
    self.GetColors(image)
    image = image.resize((image.size[0]/pixelSize, image.size[1]/pixelSize), Image.NEAREST)
    image = image.resize((image.size[0]*pixelSize, image.size[1]*pixelSize), Image.NEAREST)
    self.GetColors(image)
    #image.show()
    image.save('newimage.png')
    return 'newimage.png'
```

The Pixelate function (above) takes two parameters, image filename (im) and the size of pixels you want. The work is done by the image.resize method. I found this routine on the web in a number of places. In this instance we will be passing a pixel size of 5, which works well for Cross Stitch projects. We also tell the method to take the color of the nearest neighbor. This returns a new image, which we save as a file and return the filename.

The ReduceColors routine (below) basically uses the Image.ADAPTIVE pallet so we can

```
def ReduceColors(self,ImageName):
    #Reduce colors
    numcolors=MaxColors.get()
    image = Image.open(ImageName)
    output = image.convert('P', palette=Image.ADAPTIVE, colors=numcolors)
    x = output.convert("RGB")
    self.GetColors(x)
    numcolors = x.getcolors()
    ProcessedColors.set(len(numcolors))
    x.save('im1.png')
    return 'im1.png'
```

get a much smaller number of colors.

There are two MakeLines (top right) routines. They create the grid we spoke of earlier.

Rgb2Hex() returns a hex value of the RGB value that is passed in. We will use this to try to compare the colors in the database with the colors in the image.

```
def Rgb2Hex(self,rgb):
  return '#%02x%02x%02x' %
rgb
```

The ScrollList (below) on the right side holds the colors that will be used to get the proper floss colors. We simply create labels to hold the colors (visual) and text.

This (next page) is the routine that we use to try to find the closest match between the color in the image and the color in the database. There are many different algorithms on the web that you can look at and try to understand the logic behind it. It gets rather complicated.

Ok. That's all for this month. Next time, we will start creating the PDF output file so the cross stitcher has something to work

```
    def MakeLines(self,im,pixelSize):
        global backgroundColor1
        #--------------------------------
        image = Image.open(im)
        pixel = image.load()
        for i in range(0,image.size[0],pixelSize):
            for j in range(0,image.size[1],pixelSize):
                for r in range(pixelSize):
                    pixel[i+r,j] = backgroundColor1
                    pixel[i,j+r] = backgroundColor1
        image.save('output.png')

    def MakeLines2(self,im,pixelSize):
        global backgroundColor2
        #--------------------------------
        image = Image.open(im)
        pixel = image.load()
        for i in range(0,image.size[0],pixelSize):
            for j in range(0,image.size[1],pixelSize):
                for r in range(pixelSize):
                    try:
                        pixel[i+r,j] = backgroundColor2
                        pixel[i,j+r] = backgroundColor2
                    except:
                        pass
        image.save('output2.png')
```

```
    def FillScrolledList(self,filename):
        im = Image.open(filename)
        numColors = im.getcolors()
        colors = len(numColors)
        cntr = 1
        for c in numColors:
            hexcolor = self.Rgb2Hex(c[1])
            lblColor=Label(self.sfFrame,text="              ",bg=hexcolor,relief=GROOVE)
            lblColor.grid(row = cntr, column = 0, sticky = 'nsew',padx=10,pady=5)
            pkID = self.GetBestDistance(c[1][0],c[1][1],c[1][2])
            sql = "SELECT * FROM DMC WHERE pkID = {0}".format(pkID)
            rset = cursor.execute(sql)
            for r in rset:
                hexcolor2 = r[6]
                dmcnum = r[1]
                colorname = r[2]
            lblColor2=Label(self.sfFrame,text="              ",bg="#" + hexcolor2,relief=GROOVE)
            lblColor2.grid(row = cntr,column = 1,sticky = 'w',padx=5,pady=5)
            lblColor3=Label(self.sfFrame,text = str(dmcnum) + "-" + colorname,justify=LEFT)
            DmcColor.set(dmcnum)
            lblColor3.grid(row = cntr, column = 2,sticky = "w",padx=1,pady=5)
            cntr += 1
```

with.

As always, the code is available on PasteBin at http://pastebin.com/DmQ1GeUx. We will continue in the next month or so. I'm facing some surgery soon so I'm not sure how soon I will be able to sit for any long periods of time. Until then, enjoy.

```
def GetBestDistance(self,r1,g1,b1):
    # dist = math.sqrt(((r1-r2)**2) + ((g1-g2)**2) + ((b1-b2)**2))
    sql = "SELECT * FROM DMC"
    rset = cursor.execute(sql)
    BestDist = 10000.0
    for r in rset:
        pkID = r[0]
        r2 = r[3]
        g2 = r[4]
        b2 = r[5]
        dist = math.sqrt(((r1-r2)**2) + ((g1-g2)**2) + ((b1-b2)**2))
        if dist < BestDist:
            BestDist = dist
            BestpkID = pkID
    return BestpkID
```

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. His website is www.thedesignatedgeek.net.

Cross Stitch Pattern Generator - Part 4 - Understanding pyfPDF

Sorry for missing so many months. I still can't sit for long periods of time, so this article might be shorter than what you are used to. My original plan was to jump right into the PDF output portion of the program, but there is so much to understand about this library, I decided to use this installment as a tutorial on pyfPDF and then tackle the PDF output next time. So let's get started.

FPDF stands for Free PDF. A VERY minimal example would be as follows:

```
from fpdf import FPDF

pdf = FPDF()

pdf.add_page()

pdf.set_font('Arial','B',16)

pdf.cell(40,10,'Hello From Python')

pdf.output('example1.pdf','F')
```

The first line imports the library file. The next creates an instance of the FPDF object. We use the default values for this example, which are:
• Portrait
• Measure Unit = Millimeters.
• Format = A4

If you need to use 'US' standards, you could do it this way:

```
pdf=FPDF('P','in','Letter)
```

Notice the parameters are FPDF(orientation, units, format):
• Possible values for orientation are "P" for Portrait and "L" for Landscape.
• Possible values for units are: 'pt' (poiints), 'mm' (millimeter), 'cm' (centimeter), 'in' (inches).
• Possible values for format are: 'A3', 'A4', 'A5', 'Letter', 'Legal' or a tuple containing the width and height expressed in the unit given in the unit parameter.

The third line creates a page to enter data into. Notice a page is not automatically created when we create the instance of the object. The origin of the page is the upper-left corner, and the current position defaults to 1 cm from the margin. The margin can be changed with the SetMargins function.

Before you can actually print any text, you must call pdf.set_font() to define a font. In the line above, we are defining Arial Bold 16 point. Standard valid fonts are Arial, Times, Courier, Symbol and ZapfDingbats.

Now we can print a cell with the pdf.cell() call. A cell is a rectangular area, possibly framed, which contains some text. Output is at the current position which is specified (40,10 cm) in the above example. The parameters are:

```
pdf.cell(Width, Height, text, border, line, align, fill, link)
```

Where:
• Width is length of cell. If 0, width extends to the right margin.
• Height is the height of the cell.
• Text is the string of text you want to print.
• Border is either 0 (no border(default)), 1 is border, or a string of any or all of the following characters: "L","T","B","R"
• Line is where the current position should go after printing the text. Values are 0 (to the right), 1 (to the beginning of the next line, 2 (below). Default is 0, and putting 1 is equivalent to putting 0 and calling ln() immediatly after.
• Align allows to center or align the text within the cell. Values are "L" (left), "C" (center), "R" (right).
• Fill sets the background to be painted (true) or transparent (false). Default is false.
• Link is a url or identifier returned by addlink().

Finally, the document is closed and sent to the file with Output. The parameters are fpdf.output(name,dest). If file is not specified, the output will be sent to the browser. Options for destination are "I" (inline to browser(default)), "F" (local file given by name), "D" (to the browser and force a file download with the name passed), and "S" (return the document as a string).

Since we will be sending our cross stitch images to the pdf file, we will have to understand the image function.

The function is called like this:

```
pdf.image(name,x=None,y=None,
w=0,h=0,type="",link="")
```

This function puts the image. The size it will take on the page can be specified in different ways:
• Explicit width and height or
• One explicit dimension

Supported formats are JPEG, PNG, and GIF. If you wish to use GIF files, you must get the GD extension.

For JPEGs, all flavors are allowed:
• gray scale
• true colours (24 bits)
• CMYK (32 bits)

For PNGs, the following are allowed:
• gray scales on at most 8 bits (256 levels)
• indexed colors
• true colors (24 bits)

Note: interlacing is not allowed, and if you are using a version of FPDF prior to 1.7, Alpha channel is not supported.

I stole this example (shown right) from the pyFPDF tutorial.

You have been around long enough that you should be able to look at the program and understand what is going on. But in this example the line we are REALLY interested in is the fourth line:

```
this.image('img1.png',10,8,33
)
```

In this instance, we are calling the image function with the filename, the x position of where the picture will go on the page, the y position, and the width of the picture.

Now that you have a gross grasp of the library, we will start our PDF code next time.

Until then, have a good month. See you soon.

```
from fpdf import FPDF

class PDF(FPDF):
    def header(this):
        # Logo - replace with a small png of your own
        this.image('img1.png',10,8,33)
        # Arial bold 15
        this.set_font('Arial','B',15)
        # Move to the right
        this.cell(80)
        # Title
        this.cell(30,10,'Title',1,0,'C')
        # Line break
        this.ln(20)

# Instantiation of inherited class
pdf=PDF()
pdf.alias_nb_pages()
pdf.add_page()
pdf.set_font('Times','',12)
for i in range(1,41):
    pdf.cell(0,10,'Printing line number '+str(i),0,1)
pdf.output('example2.pdf','F')
```

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. His website is www.thedesignatedgeek.net.

Program in Python - Part 57

Written by Greg D. Walters

First, let me thank all the readers who sent me emails of hope and wishes for a quick recovery. They were very kind and helpful. I also want to thank Ronnie, our wonderful editor, for his support and patience during that painful period. I still have issues with sitting for long periods of time, so this is being done over the course of a number of days, so I hope the continuity that I'm trying for works. Now on with "the show"…

Not too long ago, I was walking to the time clock and the General Manager of my "day job" called me into his office. Hoping it was just a "how's it going" talk, I went in and sat down. He then started the meeting with "I'm having a problem with my spreadsheet program, and was hoping you could help me".

As my vision darkened and the three-note ominous orchestral string hits "Da Da DAAAAAAAAA" that we all know from the horror flicks of the 70's and 80's rang through my mind, rather than running screaming from the room, I innocently asked what was wrong. He responded that there was something wrong with one of the macros and "the thing just quits in the middle of the calculations". As I whipped out my white cowboy hat, I said in my best hero voice "Don't worry citizen. We'll have you up and running in no time." Within a short while, I discovered the reason the spreadsheet was unceremoniously crashing was that one cell in one of 35 workbooks was getting a divide by zero error due to an expected value not being entered in another cell in yet another one of the 35 workbooks. Let me make this perfectly clear, it was not my boss's fault. All he had asked for was a simple way to get the higher-up values from the data. (The previous two sentences have absolutely nothing to do with the fact that my boss may read this article! Or maybe it does.)

As I walked back to my work area, brushing the spurious bits of computer code from my white hat, I realized that this would be an excellent teaching moment. So, here we are. But first, let's revert back to 1979 when Apple introduced Visicalc. That was the first "Free Form Calculation type system" to really make a hit in the marketplace. While there were many bugs in the software, the world loved the idea and clones (bugs and all) began to pop up on other computer systems, like the Commodore Pet and other Apple competitors (including Microsoft in 1981 with a program called Multiplan). Finally, in 1983, a company called Lotus Development Corp. introduced Lotus 1-2-3. While very close to Visicalc in many aspects, including the menu structure, it was written completely in x86 assembly language, which made it very fast, and many of the bugs of Visicalc were fixed. Lotus 1-2-3 was so popular that it became a common benchmark to test a machine for "PC Compatibility".

The advent of the Free Form Calculation systems, allowed the "normal" person to deal with numbers in a way that previously was in the realm of the programmer. Almost anyone could, in a few hours or so, make sense of numbers, create charts and graphs, and share that information with coworkers. Shortly after that, the ability to automate some portions of the spreadsheet through Macros and Basic-like embedded languages gave these non-programmer users even more power over their destiny. They could get the answers themselves, and pretty charts and graphs as well, without having to wait in the queue for I.T. assistance. However, as we all learned from Peter Parker's uncle Ben…

With great power, comes great responsibility.

Soon the spreadsheet was taken into areas that were better suited for databases than spreadsheets. We now had workbooks upon workbooks that relied on other workbooks, and if one little number along the way didn't happen to get updated… well, we had the old "house of cards" effect.

While I don't think that every spreadsheet is evil, there are some (read this to say 'many') that should have been converted to databases many years ago. They just became too large and unwieldy for their own good. If someone had just sat down with the programmers and said, "Please help", the world would be a kinder, gentler place.

Now as I step down from my soapbox, we come to the real reason for this month's article. Every good Python programmer should have a way to deal with spreadsheets in their arsenal of tools. You never know when you will be called upon to pull data from a spreadsheet and manipulate it. While there are many ways to get data from spreadsheets like using CSV files, which has its own drawbacks, sometimes you need to read and write directly from and to a 'live' spreadsheet. After looking around, I settled on a very nice library to access my boss's problematical spreadsheet.

We will be adding the library called XLRD, which one might imagine stands for eXceL ReaD. This library allows us to easily read data from Excel files (.xls , .xlsx and .xlsm) from versions 2.0 onward.

Let's create an excel spreadsheet that we can use to examine the functionality of XLRD. Either open excel, or openoffice or libreoffice calc. In the first column (A), enter the numbers 1 to 5 going down. In the next column (B), enter 6 to 10. It should look something like this:

	A	B
1	1	6
2	2	7
3	3	8
4	4	9
5	5	10

Now save the spreadsheet as "example1.xls" in the folder you will use to save the test code. This way, we won't have to worry about paths.

Now download and install XLRD (https://pypi.python.org/pypi/xlrd). We can use it like is shown below.

Save the file as example1.py in the same folder as the spreadsheet. Since the code is so short, we will simply discuss it here. Of course, the first line imports the library. Then we create a function called OpenFile and pass the name (and path if needed) of the spreadsheet to the function.

Now we call the open_workbook method and get back a 'book' object. Then we use the nsheets attribute to return the number of ACTIVE workbooks. We can also get the name of the workbooks. In this case, they are the default. We use the sheet_by_index method to get Sheet1 into the first_sheet object. Now we can start getting data. We get the information from the cell at position (1,1) which translates to cell position B2 (it's Zero based, so cell A1 would be (0,0)). We print the data from there, both what the cell contains and the value, so we could use it in a calculation if we wish.

That was really easy, wasn't it? Now, let's do something a bit more useful. Enter the code shown on the next page (top right) and save it as 'example2.py'. This example will print out the contents of the workbook.

Since we already used the first four lines of code in the first

```
import xlrd
def OpenFile(path):
      # Open and read excel file
      book = xlrd.open_workbook(path)
      # Get number of active workbooks
      print "Number of workbooks: ",book.nsheets
      # Get the names of those workbooks
      print "Workbook names: ",book.sheet_names()
      first_sheet = book.sheet_by_index(0)
      cell = first_sheet.cell(1,1)
      print "Cell at 1,1: ",cell
      print "Cell Value at 1,1: ",cell.value

if __name__ == "__main__":
      path = "example1.xls"
      OpenFile(path)
```

example, we'll skip them. By using the 'sheet.nrows' and 'sheet.ncols' attributes, we get the number of rows and columns. This can be helpful, not only so we know what we are dealing with; we can write "generic" routines that use those values in our calculations as you will see. In fact, we use 'rows' in a for loop to obtain each row's information.

Notice the line that has 'first_sheet.row_slice'. This gets a block of cells of a given row. The syntax is as follows:

```
X =
first_sheet.row_slice(RowInQu
estion, Start_Column,
End_Column)
```

So we have used the number of rows and the number of columns in calculations. The output from our program should look something like this…

```
There are 5 rows in this
workbook.
There are 2 cols in this
workbook.
[number:1.0, number:6.0]
[number:2.0, number:7.0]
[number:3.0, number:8.0]
[number:4.0, number:9.0]
[number:5.0, number:10.0]
Press any key to continue . .
.
```

```
import xlrd
def OpenFile(path):
      book = xlrd.open_workbook(path)
      first_sheet = book.sheet_by_index(0)
# Get the number of rows in this workbook
rows = first_sheet.nrows
      # get the number of columns in this workbook
      cols = first_sheet.ncols
      print "There are %d rows in this workbook." % rows
      print "There are %d cols in this workbook." % cols
      for r in range(0,rows):
      cells = first_sheet.row_slice(rowx=r,start_colx=0,end_colx=cols)
      print cells

if __name__ == "__main__":
      path = "example1.xls"
      OpenFile(path)
```

We'll do one more example before we end this month's article. Go to the spreadsheet and in column C put some dates. Here's what my spreadsheet looks like now:

1	6	1/10/2014
2	7	4/15/2015
3	8	6/24/1986
4	9	9/30/1963
5	10	3/3/2000

You can use any dates you like. Now let's re-run our example2.py program. Here is the output from mine.

```
There are 5 rows in this
workbook.
There are 3 cols in this
workbook.
[number:1.0, number:6.0,
xldate:41649.0]
[number:2.0, number:7.0,
xldate:42109.0]
[number:3.0, number:8.0,
xldate:31587.0]
[number:4.0, number:9.0,
xldate:23284.0]
[number:5.0, number:10.0,
xldate:36588.0]
Press any key to continue ...
```

Well, that's not what we expected. It seems that excel holds dates as a value that is simply formatted for whatever we ask it to. This might be helpful for sorting and calculations, but, for showing the actual data, this won't do. Luckily, the writers of the library already thought of this. Delete the line that says "print cells" and replace it with the code shown below.

```
      for c in cells:
          if c.ctype == xlrd.XL_CELL_DATE:
          date_value = xlrd.xldate_as_tuple(c.value,book.datemode)
          dt = str(date_value[1]) + "/" + str(date_value[2]) + "/" + str(date_value[0])
          print dt
          else:
          print c.value
```

Here, we go through each cell in the cells list and check the type of the cell to see if it is considered a XL_CELL_DATE. If it is, then we convert it to a tuple. It is stored as YYYY,MM,DD. We simply pretty it up to print it as MM/DD/YYYY. Here is the output of our new program…

```
There are 5 rows in this
workbook.
There are 3 cols in this
workbook.
1.0
6.0
1/10/2014
2.0
7.0
4/15/2015
3.0
8.0
6/24/1986
4.0
9.0
9/30/1963
5.0
10.0
3/3/2000
Press any key to continue ...
```

Just for your information, there is a library from the same wonderful people called XLWT, which allows you to write to excel files. There is a wonderful tutorial and documentation on these two libraries at http://www.python-excel.org/.

The source code for example3.py is on pastebin at http://pastebin.com/bWz7beBw.

Hopefully, I'll see you next month.

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. His website is www.thedesignatedgeek.net.

Python Special Editions:

http://fullcirclemagazine.org/issue-py01/

http://fullcirclemagazine.org/issue-py02/

http://fullcirclemagazine.org/python-special-edition-issue-three/

http://fullcirclemagazine.org/python-special-edition-volume-four/

http://fullcirclemagazine.org/python-special-edition-volume-five/

http://fullcirclemagazine.org/python-special-edition-volume-six/

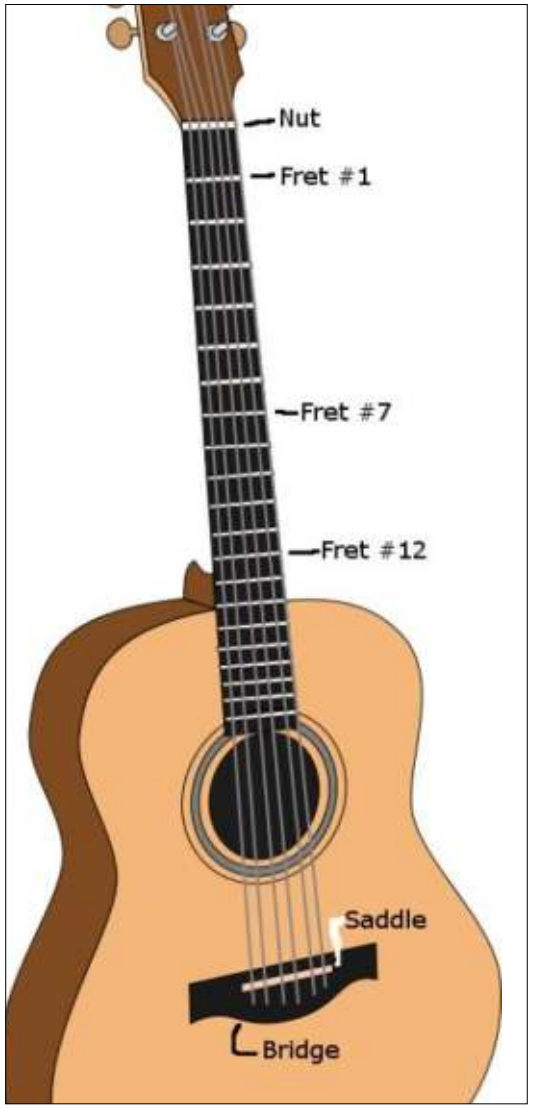
Program in Python Pt. 59

Written by Greg. D. Walters

First, let me say Happy 100 to Ronnie and the crew. It's a privilege to be part of this milestone.

This time I thought that I'd share some information on my new obsession. I've started repairing and building stringed musical instruments like guitars and violins. Believe it or not, there is a lot of math involved in musical instruments. Today, we will look at some of the math involved with the length of strings and where the frets should be placed on the fretboard.

Take a look at the picture of the guitar. I annotated various items in the image. The important things to look at are the Nut near the top of the fingerboard, the Frets, the Bridge near the bottom, and the white "line" within the bridge called the Saddle. The purpose of the frets is to create a perfect spot to change the length of the string to create a note that is in tune. The positions of these frets are not arbitrary, but mathematically determined.



Now, the physics of vibrating strings tells us that if you half the vibrating string length of a theoretically perfect string, you will double the frequency of the vibrations. In the case of a guitar, this string length is between the nut and the saddle. This distance is referred to the Scale Length of the guitar. The half-point that allows for the doubled frequency is fret # 12. If correctly done, just by lightly placing your finger on the string at this location, you get a pleasing tone. There are a few other positions that this will happen, but the 12th fret should be the perfect location for this doubling, making the note go up one octave.

Different scale lengths will create different feel and tones. For example, guitars like the Fender Stratocasters® have a scale length of 25½", which produces a rich and strong bell-like tone. On the other hand, Gibson guitars often use a scale length of 24¾". This creates a lower string tension which makes an easier playing feel and a warmer tone. Other guitar manufacturers decided that a scale length of 25" makes a clearer tone than either of the other two "standard" scale lengths.

So with the ability of a guitar maker to come up with their own scale length, the spacing of the frets will have to be recalculated. This is something that luthiers (guitar makers) have been dealing with for hundreds of years.

In the past, there was a technique called the rule of 18 which involves successively dividing the scale length minus the offset to the previous fret by 18. While this kind of worked, the tones were off, the higher up the fingerboard the player went. These days, we use a different constant. This constant is 17.817. By using this "new" constant, the 12th fret or octave is at the exact position to be half the scale length of the string.

Now, these calculations are easy enough to do by paper and pencil or even a simple calculator, it's just as easy to create a Python program to do the calculations for us in just a second. Once you have the positions, you simply saw a slot for the fret at the correct positions and then hammer in the frets.

So, let's take a look at the program.

We want to create a program that will prompt for the scale length of the guitar (or bass), do the calculations and then print out the distances. The calculations and all returned lengths are all in inches, so all our friends that use metric measurements, please add the proper conversion calculations. After almost 5 years, you should be able to do this with ease.

We don't need to import any libraries for this so we will start off by defining a couple of variables.

```
ScaleLength = 0

CumulativeLength = 0
```

Next we will create a routine (top right) that will be called repeatedly as we "travel down" the fingerboard. We will pass two values into this routine. One is the scale length and the other is the cumulative distance from the nut to the previous fret.

In this routine, we take the scale length, subtract the cumulative distance and assign that value to BridgeToFret. We then take that value, divide it by our constant (17.817), add back in the cumulative distance and then return that value to our calling routine. Remember, we could simply have returned the calculated value without assigning it to a variable name. However, if we ever want to inspect the calculated values, it's easier to do if we assign the value before we return it.

Now we will make our worker routine. We've done this kind of thing many times in the past. We will pass it the scale length and it will loop for up to 24 frets (range(1,25)). Even if your project has less than 24 frets, you will have the correct positions of all the frets you do have. I chose 24 because that's the maximum of frets for most guitars. When we get into the loop, we check the fret number (x) and if it is 1, we pass the cumulative length as 0, since this is the first calculation. Otherwise, we pass the last cumulative length in and it becomes the result from the calculation routine. Finally, we print each fret number followed by a formatted version of the cumulative length.

```
def CalcSpacing(Length,NTF):
    BridgeToFret = Length-NTF
    NutToFret = (BridgeToFret/17.817) + NTF
    return NutToFret
```

Finally, we have the code that does the prompting for the scale length. I'm sure you will remember the format for the raw_input routine, since we have used it so many times before. Something you might not remember: that raw_input always returns a string, so when we pass it off to the DoWork routine, we have to pass it as a floating point number so the routine will work correctly. Of course, we could simply pass it as a string, but we would have to deal with the conversion in the DoWork routine.

```
ScaleLength =
raw_input("Please enter Scale
Length of guitar -> ")

DoWork(float(ScaleLength))
```

You might wonder what good this program will do if you aren't going to build a guitar from scratch. It can be valuable when you're looking at buying a used guitar or trying to tweak a guitar with a floating bridge. Also, if you are a guitar player, this might have been something you didn't know about guitars.

Of course, the code is available from pastebin at http://pastebin.com/A2RNECt5.

```
def DoWork(ScaleLength):
    CumulativeLength = 0
    for x in range(1,25):
        FretNumber = x
        if FretNumber == 1:
            CumulativeLength = CalcSpacing(ScaleLength,0)
        else:
            CumulativeLength = CalcSpacing(ScaleLength,CumulativeLength)
        print("Fret=%d,NutToFret=%.3f" % (FretNumber,CumulativeLength))
```

HOW TO CONTRIBUTE

FULL CIRCLE NEEDS YOU!

A magazine isn't a magazine without articles and Full Circle is no exception. We need your opinions, desktops, stories, how-to's, reviews, and anything else you want to tell your fellow *buntu users.
Send your articles to: articles@fullcirclemagazine.org

We are always looking for new articles to include in Full Circle. For help and advice please see the Official Full Circle Style Guide: http://url.fullcirclemagazine.org/75d471

Send your comments or Linux experiences to: letters@fullcirclemagazine.org
Hardware/software reviews should be sent to: reviews@fullcirclemagazine.org
Questions for Q&A should go to: questions@fullcirclemagazine.org
Desktop screens should be emailed to: misc@fullcirclemagazine.org
... or you can visit our site via: fullcirclemagazine.org

Please note:

Special editions are compiled from originals and may not work with current versions.

Full Circle Team

Editor - Ronnie Tucker
ronnie@fullcirclemagazine.org

Webmaster - Lucas Westermann
admin@fullcirclemagazine.org

Special Editions - Jonathan Hoskin

Editing & Proofreading
Mike Kennedy, Gord Campbell, Robert Orsino, Josh Hertel, Bert Jerred, Jim Dyer and Emily Gonyer

Our thanks go to Canonical, the many translation teams around the world and Thorsten Wilms for the FCM logo.

For the Full Circle Weekly News:

You can keep up to date with the Weekly News using the RSS feed: http://fullcirclemagazine.org/feed/podcast

Or, if your out and about, you can get the Weekly News via Stitcher Radio (Android/iOS/web):
http://www.stitcher.com/s?fid=85347&refid=stpr

and via TuneIn at: http://tunein.com/radio/Full-Circle-Weekly-News-p855064/

Getting Full Circle Magazine:

EPUB Format - Most editions have a link to the epub file on that issues download page. If you have any problems with the epub file, email: mobile@fullcirclemagazine.org

Issuu - You can read Full Circle online via Issuu: http://issuu.com/fullcirclemagazine. Please share and rate FCM as it helps to spread the word about FCM and Ubuntu.

Magzster - You can also read Full Circle online via Magzster: http://www.magzter.com/publishers/Full-Circle. Please share and rate FCM as it helps to spread the word about FCM and Ubuntu Linux.