Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

PROGRAMMING SERIES SPECIAL EDITION

PROGRAMMING SERIES SPECIAL EDITION

PROGRAM IN PYTHON

Volume Six

Parts 32-38

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 27-31** from issues #60 through #67, allowing peerless Python professor Gregg Walters #66 as time off for good behaviour.

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

Enjoy!

Find Us

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

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

IRC: #fullcirclemagazine on chat.freenode.net

Editorial Team

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

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

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

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

Beginning Python - Part 32

Written by Greg D. Walters

I must say, I love my Android tablet. While I use it every day, it's not yet a replacement for my desktop. And I must also admit, most of what I use it for is pretty much what everyone uses theirs for: web browsing, listening to music, watching videos, playing games, and so on. I try to justify it by having apps that deal with grocery and todo lists, finding cheap gas, fun things for our grandson, etc. It's really a toy for me right now. Why use a fancy touch-screen tablet to do your grocery list? Let's face it... it's the cool looks of envy that people give me in the store when they see me rolling the cart down the aisle and I tap my tablet to mark items off the list. Ahh--- the geek factor RULES! Of course, I can use the back of an old envelope to hold my list. But that wouldn't be cool and geeky, now, would it?

Like 99% of geeky married men in the world, I am married to a non-geek woman. A wonderful loving woman, to be sure, but a non-geek who, when I start drooling at the latest gadget, sighs, and says something like "Well, if you REALLY think we need that...". Then she gives me the same look I give her as she is lovingly fondles the 50th pair of shoes at the store.

In all honesty, it wasn't hard to get the first tablet into our house. I bought it for my wife while she was going through chemotherapy. She tried to use a laptop for a while, but the heat and weight on her lap was too much after a while. E-books on a laptop for her wasn't an option, so when she tried to read, she had to juggle the book, and the laptop, and the mp3 player. All while being tied to a recliner with tubes running into her arm filling her with nasty chemicals. When I got her the tablet, it was the best of all worlds. She could read an e-book, listen to music, watch a TV show, browse the web, check her E-mail, update her cancer blog, follow her friends on facebook, and play games - all on a device that was light and cool. If she got tired, she could just slip it off to the side between her and the recliner (or bed when she was home trying to regain strength). MUCH better than a bulky laptop, and book, mp3 player, remote control, and more.

As she was getting pumped full of noxious chemicals, I would commandeer a table and chair in the corner of the treatment room, near a power outlet, and try to work on my six-year old laptop. In between projects, I would do research on Android programming. I found out that most programming for Android is done in Java. I had almost resigned myself to re-learning Java when I stumbled across a few tools that allow Python programming for the Android Operating system. One of these tools is called "SL4A". SL4A stands for Scripting Layer for Android. That's what we will concentrate on in the next couple of articles. We'll really focus on getting SL4A set up on Android in this one.

You might ask, why in the world I would be talking about Android programming in a magazine designed for Linux. Well, the simple reason is that the core of Android is Linux. Everything that Android is, sits on top of Linux!

Many web pages show how to load SL4A into the Android Emulator for Desktops. We'll look at doing that another time, but for now we'll deal with the Android device itself. To install SL4A on your Android device, go to http://code.google.com/p/android-scripting/; you'll find the installation file for SL4A. Don't be absolutely confused here. There's a square High Density barcode that you tap to download the APK. Be sure that you have the "Unknown

Sources" option enabled in the Application settings. It's a quick download. Once you have it downloaded and installed, go ahead and find the icon, and tap it. What you will see is a rather disappointing black screen saying "Scripts...No matches found". That's OK. Hit the menu button and select View. You'll see a menu. Select Interpreters. Then select menu again, and select Add. From the next menu, select Python 2.6.2. This should ask you to start a browser session to download Python for Android. Once this is installed, select Open. You'll get a screen menu with the options to Install, Import Modules, Browse Modules, and Uninstall modules. Select Install. Now Python will download and install along with other extra modules. In addition, you'll get some sample scripts. Finally, tap the back button and you'll see Python 2.6.2 installed in the interpreters screen. Tap again on the back button and you'll see a list of some sample python scripts.

That's all we are going to do this time. All I wanted to do is whet your appetite. Explore Python on Android. You might also want to visit

> **you'll see a rather disappointing black screen [...] That's OK.**

http://developer.android.com/sdk/index.html to get the Android SDK (Software Development Kit) for your desktop. It includes an Android Emulator so you can play along. Setting up the SDK is really pretty easy on Linux, so you shouldn't have too much trouble.

Greg is the 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.

How to Include Accents from the Keyboard

by Barry Smith

If your Linux system is in French, German, or Spanish, and, therefore, requiring accents, or if, occasionally, you need to use accents which do not appear in English words, many users do not know that there is a very easy way to do this from the keyboard. The following applies to only the UK keyboard.

Acute accent
Press Alt Gr + ; (semi-colon) Lift hand then press the desired vowel é

Circumflex
Press Alt Gr + ' (apostrophe) Lift hand then press the desired vowel î

Grave accent
Press Alt Gr + # (hache) Lift hand then press the desired vowel è

Umlaut
Press Alt Gr + [Lift hand then press u ü

ñ - Press Alt Gr +] Lift hand then press n ñ

œ - Press Shift + Alt Gr Lift hand then press o then press e œ
The œ will not appear until after the e is keyed.

To get ¿ and ¡ (inverted exclamation mark) which I use all the time in Spanish before questions, and exclamations, press Alt Gr + Shift, keeping both keys pressed, then hit _ (underscore) for ¿ or hit ! (exclamation mark) for ¡.

If you want any of these in capitals, just press Shift before keying in the letter.

Beginning Python - Part 33

This time, we'll set up the Android SDK on our Linux desktop. We'll also create a virtual Android device, install SL4A and python on it, and do a quick test.

Please be aware, this is not something you would want to do for machines that have less than 1 GB of ram. The emulator eats up a huge amount of memory. I've tried it on a laptop running Ubuntu with only 512 MB of ram. It WILL work, but it is REALLY slow.

Here's a quick list of what we'll do. We'll go step-by-step in a minute.
• Install the Java JDK6.
• Install the Android SDK starter pack.
• Create and setup AVDs.
• Test AVD, and install SL4A and Python.

In reality, we should also install Eclipse and the Android ADT plugin for Eclipse, but, since we won't be dealing with Eclipse in this set of articles, we can bypass that. If you want to include those steps, head over to http://developer.android.com/sdk/installing.html to see all the steps in the suggested order. Let's get started.

STEP 1 - Java JDK 6

From everything I've read and tried, it must be the actual Sun release. OpenJDK is not supposed to work. You can find information on this on the web, but here's the steps that I did. In a terminal, type the following...

```
sudo add-apt-repository ppa:ferramroberto/java

sudo apt-get update

sudo apt-get install sun-java6-jdk
```

Once everything here is done, you will want to edit your .bashrc file to set "JAVA_HOME" so everything runs correctly. I used gedit and, at the bottom of the file, I added the following line...

```
export JAVA_HOME="/usr/lib/jvm/java-6-sun-1.6.0.06"
```

Save the file and move on to step 2.

STEP 2 - Android SDK Starter Pack

Now the actual "fun" begins. You'll want to go to developer.android.com/sdk/index.html. This is where the SDK is located. Download the latest version for Linux, which, at the time of this writing, is android-sdk_r18-linux.tgz. Using Archive Manager, unpack it somewhere convenient. I put it in my home directory. Everything runs directly from this folder, so you really don't have to install anything. So the path for me is /home/greg/android-sdk-linux. Navigate to this folder, then go to the tools folder. There you will find a file called "android". This is what runs the actual SDK. I created a launcher on my desktop to make it easy to get to.

Now the boring part. Run the android file, and the Android SDK Manager will start. It will go out and update the platforms that are available. I will warn you now that this process will take some time, so don't bother if you don't have a lot of time to deal with it. For the sake of brevity, I would suggest you get only one platform to start. A good one to begin with is the Android 2.1 platform, since, for the most part, if you develop for an older platform, there should be no problem running on a newer platform. You also need to get the Tools set as well. Simply check the box next to those two items, then click on the install button. Once you get the platform of your choice, and the tool set, you are almost ready to create your first virtual machine.

STEP 3 - Create and set up your first AVD

Back in the Android SDK Manager, select Tools from the main menu, then select Manage AVDs. This will open a new window. Since this is the first time, there won't be any virtual devices set up. Click on the "New" button. This opens yet another window where

we define the properties of the virtual Android device. Here's the steps that you should use to set up a simple Android emulator device:

- Set the name of the device. This is important if you have more than one device.
- Set the target platform level.
- Set the size of the SD card (see below).
- Set the skin resolution.
- Create the device.

So, In the name text box, type "Test1". Under the target combo-box, select Android 2.1 - API Level 7. In the text box for "SD Card:" enter 512 and make sure the dropdown shows "MiB". Under "Skin", set the resolution to 800x600. (You can play with the other built-in sizes on your own.) Finally, click the "Create AVD" button. Soon, you'll see a message box saying that the AVD was created.

STEP 4 - Testing the AVD and installing SL4A and Python

Now, finally, we can have a bit of fun. Highlight the AVD you just created and click on the Start button. In the dialog box that pops up, simply click the "Launch" button. Now, you have to wait a few minutes for the virtual device to be created in memory, and the Android platform to be loaded and started. (We'll talk about speeding this process up in later runs.)

Once the AVD starts up and you have the "home" screen up, you will install SL4A. Using the browser or the google web search box on the home screen, search for "sl4a". Go to the downloads page, and you'll eventually find the web page for the downloads at http://code.google.com/p/android-scripting/downloads/list.

Scroll down the page until you get to the sl4a_r5 link. Open the link and tap on the "sl4a_r5.apk" link. Notice I said "tap" rather than "click". Start thinking about using your finger to tap the screen rather than clicking the mouse. It will make your programming transition easier. You'll see the download start. You may have to pull down the notification bar at the top to get to the downloaded file. Tap on that, then tap the install button.

Once the file is downloaded, you'll be presented with the option to open the downloaded app or to tap "Done" to exit the installer. Here we will want to tap "Open".

Now SL4A will start. You'll probably see a dialog asking if you will agree to usage tracking. Either accept or refuse this - it's up to you. Before we go any farther, you should know some keyboard shortcuts that will help you move around. Since we don't have a "real" Android device, buttons like Back, Home, and Menu, aren't available. You'll need them to navigate around. Here's a few important shortcuts.

Back - Escape
Home - Home
Menu - F2

Now we will want to download and install python into SL4A. To do this, first tap Menu (press F2). Select "View" from the menu. Now select "Interpreters". It looks like nothing happened, but tap Menu again (F2), then select "Add" from the popup. Now scroll down and select "Python 2.6.2". This will download the base package for Python for Android. Install the package, then open it. You will be presented with four options. Install, Import Modules, Browse Modules, and Uninstall Module. Tap on Install. This will start downloading and installing all the pieces of the latest Python for Android. This can take a few minutes.

Once everything is done, tap Back (escape key) until you get to the SL4A Interpreters screen. Now everything is loaded for us to play in Python on Android. Tap Python 2.6.2, and you'll be in the "standard" Python shell. This is just like the shell on your desktop. Type the following three lines, one at a time, into the shell. Be sure to wait for the ">>>" prompt each time.

```
import android

droid = android.Android()

droid.makeToast("Hello from
Python on Android")
```

After you type the last line and press Enter, you'll see a rounded corner box at the center bottom of the shell that says "Hello from Python on Android". That's what the "droid.makeToast" command does.

You've written your first Python script for Android. Neat, huh?

Now let's create a shortcut on the Android home screen. Tap the Home key (Home button). If you chose the 2.1 platform, you should see a slider bar on the far right of the screen. If you chose another platform, it might be a square or rectangle consisting of small squares. Either way, this gets you to the Apps screen. Tap that, and find the SL4A icon. Now perform a "long tap" (long click), which will create a shortcut on the Home screen. Move the shortcut wherever you want it.

Next, we will create our first saved script. Go back into SL4A. You should be presented with the sample scripts that come with Python 4 Android. Tap the Menu button and select "Add". Select "Python 2.6.2" from the list. You'll be presented with the script editor. At the top is the filename box with ".py" already filled out. Below that is the editor window that already has the first two lines of our program entered for us. (I included them below in italics so you can check it. We also used these two lines in our first sample.)

```
import android

droid = android.Android()
```

Now, enter the following two lines to the python script.

```
uname =
droid.dialogGetInput("What's
your name?")

droid.makeToast("Hello %s
from Python on Android") %
uname.result
```

The first new line will create a dialog box (droid.dialogGetInput()) that asks for the user's name. The response is returned to our program in uname.result. We've already used the droid.makeToast() function.

Name the file andtest1.py, then tap Done, and tap "Save & Run". If everything worked, you should see a dialog box asking for your name. After you enter it, you should see the alert at the bottom of the screen saying "Hello Your Name from Python on Android".

That's all for this time. For now, there's a TON of documentation about SL4A for free on the web. You can play a bit on your own until next time. I'd suggest that you start by going to http://code.google.com/p/android-scripting/wiki/Tutorials.

Greg is the 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.

Beginning Python - Part 34

Written by Greg D. Walters

This time, we'll finish up using SL4A. We'll make a larger program and then send it to the virtual machine via ADB.

Let's deal with our code first. In this, we'll simply be trying out some of the "widgets" that are available to us when using SL4A. Start on your desktop using your favorite editor.

Enter the code shown top right and save it (but don't try to run it) as "atest.py".

```
import android

droid = android.Android()
uname = droid.dialogGetInput("Hello","What's your name?")
droid.makeToast("Hello %s from python on Android!" % uname.result)
```

```
droid.dialogCreateAlert(uname.result,"Would you like to play a game?")
droid.dialogSetPositiveButtonText('Yes')
droid.dialogSetNegativeButtonText('No')
droid.dialogShow()
while True: #wait for events for up to 10 seconds...
      response = droid.eventWait(10000).result
      if response == None:
      break
      if response["name"] == "dialog":
      break
droid.dialogDismiss()
```

The first line imports the android library. We create an instance of it in the second line. Line 3 creates and displays a dialog box with the title "Hello", the prompt of "What's your name?", a text box for the user to enter their name, and two buttons, "OK" and "Cancel". Once the user presses "OK", the response is returned in the variable uname. The last line (so far) then says "Hello {username} from python on Android!". This isn't new, we did this before. Now we'll add more code (above).

Save your code as atest1.py. We'll be sending this to our virtual machine after we discuss what it does.

Take a look at the first four lines we just entered. We create an alert type dialog asking "Would you like to play a game?". In the case of an alert type dialog, there's no text box to enter anything. The next two lines say to create two buttons, one with the text "Yes", which is a "positive" button, and one with the text "No", a "negative" button. The positive and negative buttons refer to the response returned - either "positive" or "negative". The next line then shows the dialog. The next seven lines wait for a response from the user.

We create a simple loop (while True:) then wait for a response for up to 10 seconds by using the droid.eventWait(value) call. The response (either "positive" or "negative") will be returned in - you guessed it - the response variable. If response has the name of "dialog", then we will break out of the loop and return the response. If nothing happens before the timeout occurs, we simply break out of the loop. The actual information returned in the response variable is something like this (assuming the "positive" or "Yes" button is pressed)...

```
{u'data': {u'which':
u'positive'}, u'name':
u'dialog', u'time':
1339021661398000.0}
```

You can see that the value is passed in the 'data' dictionary, the dialog key is in the 'name' dictionary, and there is a 'time' value that we don't care about here.

Finally we dismiss the dialog box.

Before we can send our code to the virtual machine, we have to start the virtual machine. Start your Android emulator. Once it starts up, notice that the title bar has four digits at the start of the title. This is the port that the machine is listening on. In my case (and probably yours) it's 5554.

Now, let's push it to our virtual machine. Open a terminal window and change to the folder you saved the code in. Assuming you have set your path to include the SDK, type

```
adb devices
```

This asks adb to show any devices that are connected. This can include not only the Android emulator but also any smartphones, tablets, or other Android devices. You should see something like this...

```
List of devices attached
emulator-5554    device
```

Now that we are sure that our device is attached, we want to push the application to the device. The syntax is...

```
adb push source_filename
destination_path_and_filename
```

So, in my case it would be...

```
adb push atest1.py
/sdcard/sl4a/scripts/atest1.p
y
```

If everything works correctly, you'll get a rather disappointing message similar to this...

```
11 KB/s (570 bytes in 0.046s)
```

Now, on the Android emulator, start SL4A. You should see all of the python scripts, and, in there you should see atest1.py. Tap (click) on 'atest1.py', and you'll see a popup dialog with 6 icons. From left to right, they are "Run in a dialog window", "Run outside of a window", "Edit", "Save", "Delete", and "Open in an external editor". Right now, tap (click) on the far left icon "Run in a dialog window" so you can see what happens.

You'll see the first dialog asking for your name. Enter something in the box and tap (click) the 'Ok' button. You'll see the hello message. Next, you'll see the alert dialog. Tap (click) on either button to dismiss the dialog. We aren't looking at the responses yet so it doesn't matter which one you choose. Now we'll add some more code (top right).

I'm sure you can figure out that

```
if response==None:
    print "Timed out."
else:
    rdialog=response["data"]
```

this set of code simply checks the response, and, if it's 'None' due to a timeout, we simply print "Timed out." And, if it's actually something we want, then we assign the data to the variable rdialog. Now add the next bit of code (below)...

This part of the code will look at the data passed back by the button-press event. We check to see if the response has a "which" key, and, if so, it's a legitimate button press for us. We then check to see if the result is a "positive" ('Ok' button) response. If so, we'll create another alert dialog, but this time, we will add a list of items for the user to choose from. In this case, we offer the user to select

```
    if rdialog.has_key("which"):
    result=rdialog["which"]
    if result=="positive":
        droid.dialogCreateAlert("Play a Game","Select a game to play")
        droid.dialogSetItems(['Checkers','Chess','Hangman','Thermal
Nuclear War']) # 0,1,2,3
        droid.dialogShow()
        resp = droid.dialogGetResponse()
```

from a list including Checkers, Chess, Hangman, and Thermal Nuclear War, and we assign the values 0 to 3 to each item. (Is this starting to seem familiar? Yes, it's from a movie.) We then display the dialog and wait for a response. The part of the response we are interested in is in the form of a dictionary. Assuming the user tapped (clicked) on Chess, the resulting response comes back like this...

```
Result(id=12,
result={u'item':1},
error=None)
```

In this case, we are really interested in the result portion of

the returned data. The selection is #1 and is held in the 'item' key. Here's the next part of the code (above right)...

```
if resp.result.has_key("item"):
    sel = resp.result['item']
    if sel == 0:
        droid.makeToast("Enjoy your checkers game")
    elif sel == 1:
        droid.makeToast("I like Chess")
    elif sel == 2:
        droid.makeToast("Want to 'hang around' for a while?")
    else:
        droid.makeToast("The only way to win is not to play...")
```

Here we check to see if the response has the key "item", and, if so, assign it to the variable "sel". Now we use an if/elif/else loop to check the values and deal with whichever is selected. We use the droid.makeToast function to display our response. Of course, you could add your own code here. Now for the last of the code (bottom right)...

```
        elif result=="negative":
            droid.makeToast("Sorry.  See you later.")
    elif rdialog.has_key("canceled"):
        print "Sorry you can't make up your mind."
    else:
        print "unknown response=",response
print "Done"
```

As you can see, we simply respond to the other types of button-presses here.

Save, push, and run the program.

As you can see, SL4A gives you the ability to make "GUIfied" applications, but not full gui apps. This however, should not keep you from going forward and starting to write your own programs for Android. Don't expect to put these up on the "market". Most people really want full GUI type apps. We'll look at that next time. For more information on using SL4A, simply do a web search and you'll find lots of tutorials and more information.

By the way, you can push directly to your smartphone or tablet in the same way.

As usual, the code has been put up on pastebin at http://pastebin.com/REkFYcSU

See you next time.

Greg is the 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.

Beginning Python - Part 35

Written by Greg D. Walters

This time, we are going to take a short detour from our exploration of Android programming, and look at a new framework for GUI programming called ***Kivy***. You'll want to head over to http://kivy.org and download and install the package – before getting too far into this month's installment. The Ubuntu installation instructions can be found at http://kivy.org/docs/installation/installation-ubuntu.html.

First off, Kivy is an open source library that makes use of multi-touch displays. If that isn't cool enough, it's also cross-platform, which means that it will run on Linux, Windows, Mac OSX, IOS and Android. Now you can see why we are talking about this. But remember, for the most part, anything you code using Kivy, can run on any of the above platforms without recoding.

Before we go too far, let me make a couple of statements. Kivy is VERY powerful. Kivy gives you a new set of tools to make your GUI programming. All that having been said, Kivy is also fairly complicated to deal with. You are limited to the widgets that they have provided. In addition, there is no GUI designer for Kivy, so you have to do a GREAT deal of pre-planning before you try to do anything complicated. Also remember, Kivy is continually under development so things can change quickly. So far, I haven't found any of my test code that has broken by a new version of Kivy, but that's always a possibility.

Rather than jump in and create our own code this month, we'll look at some of the examples that come with Kivy, and, next month, we'll "roll our own".

Once you've unpacked Kivy into its own folder, use a terminal and change to that folder. Mine is in /home/greg/Kivy-1.3.0. Now change to the examples folder, then to the widgets folder. Let's look at the accordion_1.py example.

It's very simple, but shows a really neat widget. Below is their code.

As you can see, the first three lines are import statements. Any widget you use must be imported, and you must always import App from kivy.app.

The next eight lines are the main application class. The class is defined, then a routine called build is created. You will almost always have a build routine somewhere in your Kivy programs. Next we set a root object from the Accordion widget. Next we create five AccordionItems and set their title. We then add ten labels with the text "Very big content". We then add each label to the root widget (the Accordion) and then finally we return the root object. This, in essence, displays the root object in the window that Kivy creates for

```
from kivy.uix.accordion import Accordion, AccordionItem
from kivy.uix.label import Label
from kivy.app import App

class AccordionApp(App):
     def build(self):
          root = Accordion()
          for x in xrange(5):
               item = AccordionItem(title='Title %d' % x)
               item.add_widget(Label(text='Very big content\n' * 10))
               root.add_widget(item)
          return root

if __name__ == '__main__':
     AccordionApp().run()
```

us. Finally we have the "if __name__" statement and then run the application.

Go ahead and run it to see what it does.

You will see that in a moment or two, a window opens up with five vertical bars in it. Clicking on a bar causes it to open up revealing the ten labels. Of course, each bar has the same text in the ten labels, but you can figure out how to fix that.

The Accordion widget can be used for any number of things, but the thing that has always jumped to my mind is for a configuration screen... each bar being a different configuration set.

Next we'll look at the textalign.py example. It's not as "sexy" as the last one, but it's a good example that gives you some important information for later on.

Before we look at the code, run the program.

What you should see is a label at the top of the window, a set of nine red boxes with text in a 3x3 grid, and four buttons along the bottom of the window. As you click (tap) each of the buttons, the alignment of the text within the red boxes will change. The main reason you would want to pay attention to this example is how to use and control some of the important widgets as well as how to change the alignment in your widgets, which is not completely intuitive.

Above right is their code for this one. I'll break it into pieces. First the import code (above right).

Below is something special. They created a class with no code in it. I'll discuss that in a few minutes:

```
class BoundedLabel(Label):

  pass
```

Next a class called "Selector" (below) is created:

```
from kivy.app import App
from kivy.uix.label import Label
from kivy.uix.gridlayout import GridLayout
from kivy.uix.floatlayout import FloatLayout
from kivy.properties import ObjectProperty
```

```
class Selector(FloatLayout):

app = ObjectProperty(None)
```

Now the Application class is created.

Here the routine select is created. A GridLayout widget is created (called grid) which has 3 rows and 3 columns. This grid is going to hold the nine red boxes.

```
for valign in ('bottom',
'middle', 'top'):

for halign in ('left',
'center', 'right'):
```

Here we have two loops, one inner and one outer.

```
label = BoundedLabel(text='V:
%s\nH: %s' % (valign,
halign),

size_hint=(None, None),

halign=halign, valign=valign)
```

In the code above, an instance of the BoundedLabel widget is created, once for each of the nine red boxes. You might want to stop here and say "But wait! There isn't a BoundedLabel widget. It just has a pass statement in it." Well, yes, and no. We are creating an instance of a custom widget. As I said a little bit above, we'll talk more about that in a minute.

In the code block (top right, next page), we examine the variable 'case' which is passed into the select routine.

```
class TextAlignApp(App):

      def select(self, case):

      grid = GridLayout(rows=3, cols=3, spacing=10, size_hint=(None, None),
                  pos_hint={'center_x': .5, 'center_y': .5})
```

Here, the grid is removed, to clear the screen.

```
if self.grid:

self.root.remove_widget(self.grid)
```

The bind method here sets the size, and the grid is added to the root object.

```
grid.bind(minimum_size=grid.setter('size'))

self.grid = grid

self.root.add_widget(grid)
```

Remember in the last example I said that you will almost always use a build routine. Here is the one for this example. The root object is created with a FloatLayout widget. Next (middle right) we call the Selector class to create a Selector object, then it's added to the root object, and we initialize the display by calling self.select(0).

Finally the application is allowed to run.

```
TextAlignApp().run()
```

Now, before we can go any further, we need to clear up a few things. First, if you look in the folder that holds the .py file, you'll notice another file called textalign.kv. This is a special file that Kivy uses to allow you to create your own widgets and rules. When your Kivy application starts, it looks in the same directory for the .kv helper file. If it is there, then it loads it first. Here's the code in the .kv file.

This first line tells Kivy what minimum version of Kivy that must be used to run this app.

```
#:kivy 1.0
```

Here the BoundedLabel widget is created. Each of the red boxes in the application is a BoundedLabel.

Color sets the background color of the box to red (rgb: 1,0,0). The Rectangle widget creates a (you guessed it) rectangle. When we call the BoundedLabel widget in the actual application code, we are passing a label as the parent. The size and position (here in the .kv file) are set to whatever the size and position of the label are.

Here (right, next page) the Selector widget is created. This is the four buttons that appear at the bottom of the window as well as the label across the top of the window.

```
if case == 0:
                    label.text_size = (None, None)
          elif case == 1:
                    label.text_size = (label.width, None)
          elif case == 2:
          label.text_size = (None, label.height)
          else:
          label.text_size = label.size
                    grid.add_widget(label)
```

```
     def build(self):
     self.root = FloatLayout()
          self.selector = Selector(app=self)
     self.root.add_widget(self.selector)
     self.grid = None
          self.select(0)
     return self.root
```

```
<BoundedLabel>:
     canvas.before:
          Color:
               rgb: 1, 0, 0
     Rectangle:
          pos: self.pos
               size: self.size
```

Notice that the label that makes up the title at the top of the window has a position (pos_hint) as top, has a height of 50 pixels and a font size of 16. Each of the buttons has an alignment for the text of center. The on_release statement is a bind-like statement so that, when the button is released, it calls (in this case) root.app.select with a case value.

Hopefully, this is beginning to make sense now. You can see why Kivy is so powerful.

Let's talk for a moment about two widgets that I have passed over in the discussion of the application code, The GridLayout and the FloatLayout.

The GridLayout is a parent widget that uses a row and column description to allow widgets to be placed in each cell. In this case, it is a 3x3 grid (like a Tic-Tac-Toe (or Naughts and Crosses) board).

When you want to place a widget into a GridLayout, you use the add_widget method. Here lies a problem. You can't specify which control goes into which grid cell other than the order in which you add them. In addition, each widget is added from left to right, top to bottom. You can't have an empty cell. Of course, you can cheat. I'll leave that up to you to figure out.

The FloatLayout widget seems to be just a parent container for other child widgets.

I've glossed over a few points for now. My intent this time was simply to get you somewhat excited about the possibilities that Kivy has to offer. In the next couple of articles, we'll continue to explore what Kivy has for us, how to use various widgets, and how to create an APK to publish our applications to Android.

Until then, explore more of the examples in Kivy, and be sure to go to the documentation page for Kivy at http://kivy.org/docs/.

```
<Selector>:
      Label:
      pos_hint: {'top': 1}
            size_hint_y: None
      height: 50
            font_size: 16
      text: 'Demonstration of text valign and halign'
      BoxLayout:
            size_hint_y: None
      height: 50
            ToggleButton:
                  halign: 'center'
            group: 'case'
                  text: 'label.text_size =\n(None, None)'
                  on_release: root.app.select(0)
                  state: 'down'
      ToggleButton:
                  halign: 'center'
            group: 'case'
                  text: 'label.text_size =\n(label.width, None)'
                  on_release: root.app.select(1)
      ToggleButton:
                  halign: 'center'
                  group: 'case'
            text: 'label.text_size =\n(None, label.height)'
                  on_release: root.app.select(2)
      ToggleButton:
                  halign: 'center'
                  group: 'case'
            text: 'label.text_size =\n(label.width, label.height)'
            on_release: root.app.select(3)
```

Greg is the 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.

Beginning Python - Part 36

Written by Greg D. Walters

Before I begin, I want to note that this article marks three years of the Beginning Programming using Python series. I want to thank Ronnie and the entire staff of Full Circle Magazine for all their support and especially, you, the readers. I NEVER thought that this would continue this long.

I also want to take a second to note that there has been some comments floating around the ether that, after three years, the word "Beginning" might be misplaced in the title of this series. After all, after three years, would you still be a beginner? Well, on some levels, I agree. However, I still get comments from readers saying that they just found the series and Full Circle Magazine, and that they are now running back to the beginning of the series. So, those people ARE beginners. So, as of part 37, we'll drop "Beginning" from the series title.

Now to the actual meat of this article... more on Kivy.

Imagine you play guitar. Not air guitar, but an actual guitar. However, you aren't the best guitar player, and some chords are problematical for you. For example, you know the standard C, E, G, F type chords, but some chords – like F# minor or C# minor – while doable, are hard to get your fingers set in a fast song. What do you do, especially if the gig is only a couple of weeks away and you HAVE to be up to speed TODAY? The workaround for this is to use the Capo (that funny clippy thing that you see sometimes on the neck of the guitar). This raises the key of the guitar and you use different chords to match the rest of the band. This is called transposing. Sometimes, you can transpose on the fly in your head. Sometimes, it's easier to sit down on paper and work out that if, for example, the chord is F# minor and you put the capo on fret 2, you can simply play an E minor. But that takes time. Let's make an app that allows you to simply scroll through the fret positions to find the easiest chords to play.

Our app will be fairly simple. A title label, a button with our basic scale as the text, a scrollview (a wonderful parent widget that holds other controls and allows you to "fling" the inside of the control to scroll) holding a number of buttons that have repositioned scales as the text, and an exit button. It will look SOMETHING like the text below.

Start with a new python file named main.py. This will be important if/when you decide to create an Android app from Kivy. Now we'll add our import statements which are shown on the next page, top right.

Notice the second line, "kivy.require('1.0.8')". This allows you to make sure that you can use the latest and greatest goodies that Kivy provides. Also notice that we are including a system exit (line 3). We'll eventually include an exit button.

Here is the beginning of our class called "Transpose".

```
class Transpose(App):

    def exit(instance):

        sys.exit()
```

Now we work on our build

```
------------------------------------------------------------------
                        Transposer Ver 0.1

     C  C#/Db  D  D#/Eb   E    F   F#/Gb   G  G#/Ab  A   A#/Bb  B    C

 1| C#/Db  D  D#/Eb  E     F  F#/Gb   G   G#/Ab  A  A#/Bb  B       C   C#/Db
 2|   D  D#/Eb  E     F   F#/Gb  G   G#/Ab  A   A#/Bb  B    C  C#/Db D
------------------------------------------------------------------
```

routine (middle right). This is needed for every Kivy app.

This looks rather confusing. Unfortunately, the editor doesn't always keep spaces correct even in a mono-spaced font. The idea is that the text1 string is a simple scale starting with the note "C". Each should be centered within 5 spaces. Like the text shown bottom right.

The text2 string should be the same thing but repeated. We will use an offset into the text2 string to fill in the button text within the scrollview widget.

Now we create the root object (which is our main window) containing a GridLayout widget. If you remember WAY back when we were doing other GUI development for Glade, there was a grid view widget. Well, the GridLayout here is pretty much the same. In this case, we have a grid that has one column and three rows. In each of the cells created in the grid, we can put other widgets. Remember, we can't define which widget goes where other than in the order in which we place the widgets.

```
root =
GridLayout(orientation='verti
cal', spacing=10,
cols=1,rows=3)
```

In this case, the representation is as follows....

```
----------------------------
(0)             title label
----------------------------
(1)             main button
----------------------------
(2)             scrollview
----------------------------
```

The scrollview contains multiple items – in our case buttons. Next, we create the label which will be at the top of our application.

```
lbl = Label(text='Transposer
Ver 0.1',
      font_size=20,
      size_hint=(None,None),
      size=(480,20),
      padding=(10,10))
```

```
import kivy
kivy.require('1.0.8')
from sys import exit
from kivy.app import App
from kivy.core.window import Window
from kivy.uix.button import Button
from kivy.uix.label import Label
from kivy.uix.anchorlayout import AnchorLayout
from kivy.uix.scrollview import ScrollView
from kivy.uix.gridlayout import GridLayout
```

padding and size_hint properties. The padding is the number of pixels around the item in an x,y reference. Taken directly from the Kivy documentation size_hint (for X which is same as Y) is defined as:

X size hint. Represents how much space the widget should use in the direction of the X axis, relative to its parent's width. Only Layout and Window make use of the hint. The value is in percent as a float from 0. to 1., where 1. means the full size of his parent. 0.5 represents 50%.

In this case, size_hint is set to none, which defaults to 100% or 1. This will be more important (and convoluted) later on.

Now we define our "main" button (next page, top right). This is a static reference for the scale.

Again, most of this should be

```
    def build(self):
        #-------------------------------------------------------------
        text1 = "  C  C#/Db  D  D#/Eb  E   F  F#/Gb  G  G#/Ab  A  A#/Bb  B  C"
        text2 = "  C  C#/Db  D  D#/Eb  E   F  F#/Gb  G  G#/Ab  A  A#/Bb  B  C  C#/Db  D
 D#/Eb  E   F  F#/Gb  G  G#/Ab  A  A#/Bb  B    C  C#/Db"
        #-------------------------------------------------------------
```

The properties that are set should be fairly self-explanatory. The only ones that might give you some trouble would be the

```
|    |    |    |    |    |    |    |    |    |    |    |    |
12345678901234567890123456789012345678901234567890123456
  C  C#/Db  E   F  F#/Gb  G  G#/Ab  A  A#/Bb  B   C
```

fairly clear.

Now we add the widgets to the root object, which is the GridLayout widget. The label (lbl) goes in the first cell, the button (btn1) goes in the second.

```
#----------------------------
root.add_widget(lbl)
root.add_widget(btn1)
#----------------------------
```

Now comes some harder-to-understand code. We create another GridLayout object and call it "s". We then bind it to the height of the next widget which, in this case, will be the ScrollView, NOT the buttons.

```
s = GridLayout(cols=1,
spacing = 10, size_hint_y =
None)
s.bind(minimum_height=s.sette
r('height'))
```

Now (middle right) we create 20 buttons, fill in the text property, and then add it to the GridLayout.

Now we create the ScrollView, set its size, and add it to the root GridLayout.

```
sv =
ScrollView(size_hint=(None,
None), size=(600,400))

sv.center = Window.center

root.add_widget(sv)
```

Lastly, we add the GridLayout that holds all our buttons into the ScrollView, and return the root object to the application.

```
sv.add_widget(s)

return root
```

Finally, we have our "if __name__" routine. Notice that we are setting ourselves up for the possibility of using the application as an android app.

```
if __name__ in
('__main__','__android__'):

    Transpose().run()
```

Now you might wonder why I used buttons instead of labels for all our textual objects. That's because labels in Kivy don't have any kind of visible border by default. We will play with this in the next installment. We will also add an exit button and a little bit more.

```
    btn1 = Button(text = "  " + text1,size=(680,40),
            size_hint=(None, None),
            halign='left',
            font_name='data/fonts/DroidSansMono.ttf',
            padding=(20,20))
```

```
for i in range(0,19):
    if i <= 12:
        if i < 10:
            t1 = " " + str(i) + "| "
        else:
            t1 = str(i) + "| "
    else:
        t1 = ''
        text2 = ''
    btn = Button(text = t1 + text2[(i*5):(i*5)+65],
           size=(680, 40),
           size_hint=(None,None),
           halign='left',
           font_name='data/fonts/DroidSansMono.ttf')
    s.add_widget(btn)
#----------------------------------------------------------------
```

The source code can be found on PasteBin at http://pastebin.com/hsicnyt1

Until next time, enjoy and thank you for putting up with me for three years!

Greg is the 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.

Programming in Python - Part 37

Written by Greg D. Walters

This month, we'll finish up the transposer program that we wrote in Kivy. Hopefully, you saved the code from last time, because we'll be building upon it. If not, grab the code from FCM#64.

Let's start by recapping what we did last month. We created an application that allows for a guitarist to quickly transpose from one key to the other. The ultimate goal is to be able to run this app not only on your Linux or Windows box, but on an android device as well. I take mine on my tablet whenever I go to band practice. I was going to deal with packaging our project for Android, but some things have changed in the method to do that, so we'll work on that next month.

The app, as we left it last time, looked like that shown below left.

When we are done, it should look like the screen below right.

The first thing you will notice is that there are blue labels rather than boring gray ones. The next is that there are three buttons. Finally the scrollable labels are closer to the entire width of the window. Other than that, it's pretty much (visually) the same. One of the buttons is an "about" button that will pop up simple information, but it explains how to make a simple popup. One of the buttons is an exit button. The other button will swap the label text to make it easy to transpose from piano to guitar or guitar to piano.

```
#:kivy 1.0
#:import kivy kivy

<BoundedLabel>:
      canvas.before:
      Color:
            rgb: 0, 0, 1
      Rectangle:
            pos: self.pos
            size: self.size
```

Let's get started by creating a .kv file (above right). This is what will give us the colored labels. It's a very simple file.

The first two lines are required. They basically say what version of Kivy to expect. Next we create a new type of label called 'BoundedLabel'. The color is set with RGB values (between 0 and 1, which can be considered as 100 percent), and as you can see the blue value is set at 100 percent. We will also create a rectangle which is the actual label. Save this as "transpose.kv". You must use the name of the class that will be using it.

Now that you have that completed, add the following lines just before the transpose class to the source file from last time:

```
class BoundedLabel(Label):

    pass
```

To make it work, all we need is a definition. Before we go any further, add the following line to the import section:

```
from kivy.uix.popup import Popup
```

This allows us to create the popup later on. Now, in the Transpose class, just inside the def build routine, place the code shown above right.

```
def LoadLabels(w):
    if w == 0:
                tex0 = self.text1
                tex1 = self.text2
            else:
                tex0 = self.text3
                tex1 = self.text4
            for i in range(0,22):
                if i <= 12:
                   if i < 10:
                      t1 = " " + str(i) + "| "
                   else:
                      t1 = str(i) + "| "
                      t = tex1
                else:
                    t1 = ''
                    t = ''
                l = BoundedLabel(text=t1+t[(i*6):(i*6)+78], size=(780, 35),
                    size_hint=(None,None),halign='left',
                    font_name='data/fonts/DroidSansMono.ttf')
                s.add_widget(l)
```

The LoadLabels routine will give us the colored labels (BoundedLabel) and the swap ability. You saw most of this last time. We pass a value to the "w" parameter to determine which text is being displayed. The l=BoundedLabel line is pretty much the same line from last time, with the exception that, this time, we are using a BoundedLabel widget instead of a Button widget. The LoadLabels will mainly be called from the next routine, Swap. Place this code (shown right) below LoadLabels.

```
def Swap(instance):
    if self.whichway == 0:
        self.whichway = 1
        btnWhich.text = "Guitar --> Piano"
        btn1.text = "    " + self.text3
        s.clear_widgets()
        LoadLabels(1)
    else:
        self.whichway = 0
        btnWhich.text = "Piano --> Guitar"
        btn1.text = "    " + self.text1
        s.clear_widgets()
        LoadLabels(0)
```

```
self.whichway=0
self.text1 = "  C  |  B  |A#/Bb|  A  |G#/Ab|  G  |F#/Gb|  F  |  E  |D#/Eb|  D  |C#/Db|  C  |"
self.text2 = "  C  |  B  |A#/Bb|  A  |G#/Ab|  G  |F#/Gb|  F  |  E  |D#/Eb|  D  |C#/Db|  C  |  B  |A#/Bb|  A  |G#/Ab|  G  |F#/Gb|  F  |  E  |D#/Ab|  D  |C#/Db|  C  |"
self.text3 = "  C  |C#/Db|  D  |D#/Eb|  E  |  F  |F#/Gb|  G  |G#/Ab|  A  |A#/Bb|  B  |  C  |"
self.text4 = "  C  |C#/Db|  D  |D#/Eb|  E  |  F  |F#/Gb|  G  |G#/Ab|  A  |A#/Bb|  B  |  C  |C#/Db|  D  |D#/Eb|  E  |  F  |F#/Gb|  G  |G#/Ab|  A  |A#/Bb|  B  |  C  |C#/Db|"
```

You can see that this routine is pretty self explanatory. We use a variable (self.whichway) to determine "which way" the labels are displaying... from Guitar to Piano or Piano to Guitar.

Be sure to save your work at this point, since we are going to be making a lot of changes from here on.

Replace the lines defining text1 and text two with the lines shown above.

We set self.whichway to 0 which will be our default for the swap procedure. Then we define four strings instead of the two we had last time. You might notice that strings text3 and text4 are simple reversals of text1 and text2.

Now we will tweak the root line definition. Change it from...

```
root =
GridLayout(orientation='verti
cal', spacing=10,
cols=1,rows=3)
```

to

```
root =
GridLayout(orientation='verti
cal', spacing=6, cols=1,
rows=4,
row_default_height=40)
```

We've changed the spacing from 10 to 6 and set the default row height to 40 pixels. Change the text for the label (next line) to "text='Transposer Ver 0.8.0'". Everything else stays the same on this line.

Now change the button definition line from...

```
btn1 = Button(text = "   " +
text1,size=(680,40),
  size_hint=(None,None),
  halign='left',
font_name='data/fonts/DroidSa
nsMono.ttf',
  padding=(20,20))
```

to:

```
btn1 = Button(text = "      "
+ self.text1,size=(780,20),
  size_hint=(None, None),
  halign='left',
font_name='data/fonts/DroidSa
nsMono.ttf',
  padding=(20,2),
background_color=[0.39,0.07,.
92,1])
```

Notice that I've changed the formatting of the first definition for clarity. The big changes are the size change from 680,40 to 780,20 and the background color for the button. Remember, we can change the background color for buttons, not "standard" labels.

Next, we will define three AnchorLayout widgets for the three buttons that we will add in later. I named them al0 (AnchorLayout0), al1 and al2. We also add the code for the About Popup, and define our buttons along with the bind statements. This is shown on the next page, top left.

Find the "s = GridLayout" line and change the spacing from 10 to 4. Next, add the following line after the s.bind line (right before the for loop):

```
LoadLabels(0)
```

This calls the LoadLabels routine with our default "which" of 0.

Next, comment out the entire for loop code. This starts with "for i in range(0,19):" and ends with "s.add_widget(btn)". We don't need this since the LoadLabels routine does this for us.

```
al0 = AnchorLayout()
al1 = AnchorLayout()
al2 = AnchorLayout()
popup = Popup(title='About Transposer',
        content=Label(text='Written by G.D. Walters'),
        size_hint=(None,None),size=(400,400))
btnWhich = Button(text = "Piano --> Guitar",
           size=(180,40),size_hint=(None,None))
btnWhich.bind(on_release=Swap)
btnAbout = Button(text="About",size=(180,40),
           size_hint=(None,None))
btnAbout.bind(on_release=ShowAbout)
btnExit = Button(text="Exit", size=(180,40),
          size_hint=(None,None))
btnExit.bind(on_release=exit)
```

Now, save your code and try to run it. You should see a deep purple button at the top, and our pretty blue BoundLabels. Plus, you will notice that the BoundLabels in the scroll window are closer together, which makes it much easier to read.

We are almost through with our code, but we still have a few things to do. After the "sv = ScrollView" line add the following line...

```
sv.size = (720, 320)
```

This sets the size of the ScrollView widget to 720 by 320 – which makes it wider within the root window. Now, before the "return root" line, add the code shown top right.

Here we add the three buttons to the AnchorLayout widgets, create a GridLayout to hold the AnchorLayouts, and then finally add the AnchorLayouts to the GridLayout.

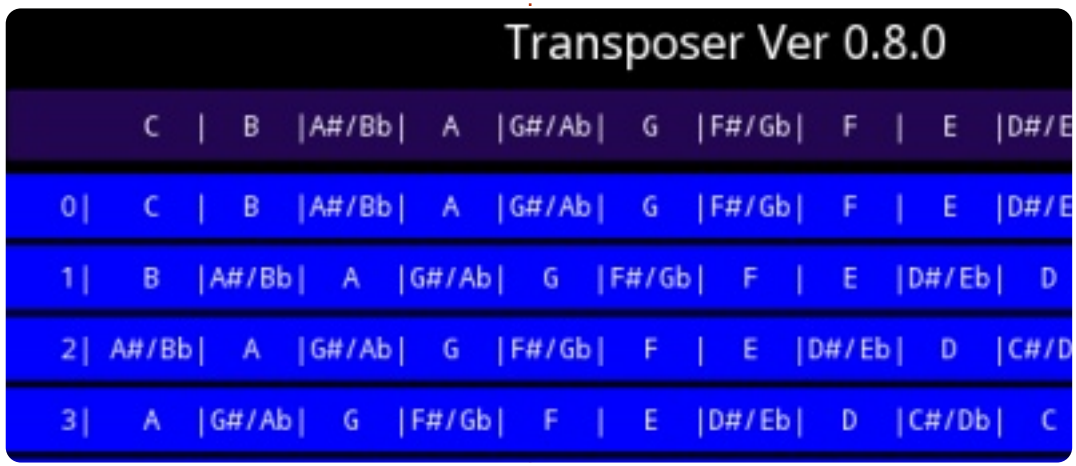
Go back just below the "def Swap" routine and add the following...

```
def ShowAbout(instance):

  popup.open()
```

That's it. Save and run the code. If you click on the About button, you will see the simple popup. Just click anywhere outside of the popup to make it go away.

```
        al0.add_widget(btnWhich)
        al1.add_widget(btnExit)
        al2.add_widget(btnAbout)
        bgl = GridLayout(orientation='vertical',
              spacing=6, cols=3,rows=1,
              row_default_height=40)
        bgl.add_widget(al0)
        bgl.add_widget(al1)
        bgl.add_widget(al2)
```



Now our code is written. You can find the full code at http://pastebin.com/GftmjENs

Next, we need to create our android package... but that will have to wait for next time.

If you want to get set up and try packaging for Android before next month, you should go to http://kivy.org/docs/guide/packaging-android.html for the documentation on this. Be sure to follow the documentation carefully.

See you next month.

Greg is the 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.

Programming In Python: Pt 38

As I promised in part 37, we will take the transposer app that we created, and create an APK to install it on your android device.

Before we get started, let's make sure we have everything ready. First thing we need is the two files we created last time in a folder that you can easily access. Let's call it "transposer". Create it in your home directory. Next, copy the two files (transpose.kv and transpose.py) into that folder. Now rename transpose.py to main.py. This part is important.

Next, we need to reference the kivy packaging instructions in a web browser. The link is http://kivy.org/docs/guide/packaging-android.html. We will be using this for the next steps, but not exactly as the Kivy people intended. You should have the android SDK from our earlier lesson. Ideally, you will go through and get all the software that is listed there, but for our purposes, you can just follow along here. You will need to download the python-for-android software. Open a terminal window and type the following...

```
git clone
git://github.com/kivy/python-for-android
```

This will download and set up the software that we need to continue. Now, in a terminal window, change your directory to the python-for-android/dist/default folder.

Now you will find a file called build.py. This is what will do all the work for us. Now comes the magic.

The build.py program will take various command-line arguments and create the APK for you. Shown above is the syntax for build.py taken directly from the Kivy documentation.

```
./build.py --dir <path to your app>
           --name "<title>"
           --package <org.of.your.app>
           --version <human version>
           --icon <path to an icon to use>
           --orientation <landscape|portrait>
           --permission <android permission like VIBRATE> (multiple allowed)
           <debug|release> <installd|installr|...>
```

For our use, we will use the following command (the "\" is a line continuation character):

```
./build.py --dir ~/transposer
--package
org.RainyDay.transposer \
 --name "RainyDay Transposer"
--version 1.0.0 debug
```

Let's look at the pieces of the command...

./build.py - this is the application
--dir ~/transposer - this is the directory where our application code lives.
--package org.RainyDay.transposer - This is the name of the package
--name "RainyDay Transposer" - this is the name of the application that will show up in the apps drawer.
--version 1.0.0 - the version of our application
debug - this is the level of release (debug or release)

Once you execute this, assuming that everything worked as expected, you should have a number of files in the /bin folder. The one you are looking for is titled "RainyDayTransposer-1.0.0-debug.apk". You can copy this to your android device using your favorite file manager app, and install it just like any other application from the various app stores.

That's all I have time for this month.