Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

PROGRAMMING SERIES SPECIAL EDITION

PROGRAMMING SERIES SPECIAL EDITION

PROGRAM IN PYTHON

Volume Eight

Parts 44-48

Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

About Full Circle

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

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

Welcome to another 'single-topic special'

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

For now, this is a straight reprint of the series **'Programming in Python', Parts 44-48** from issues #73 through #78, allowing peerless Python professor Gregg Walters #74 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.

Programming In Python: Pt 44

Written by Greg Walters

We are going to take a short detour this month from our TVRage program to partially answer a question from a reader. I was asked to talk about QT Creator, and how to use it to design user interfaces for Python programs.

Unfortunately, from what I can tell, the support for QT Creator isn't ready yet for Python. It IS being worked on, but is not “ready for prime time” quite yet.

So, in an effort to get us ready for that future article, we will work with QT4 Designer. You will need to install (if they aren't already) python-qt4, qt4-dev-tools, python-qt4-dev, pyqt4-dev-tools and libqt4-dev.

Once that is done, you can find QT4 Designer under Applications | Programming. Go ahead and start it up. You should be presented with something like the following:

Make sure that 'Main Window' is selected, and click the 'Create' button. Now you will have a blank form that you can drag and drop controls onto.

The first thing we want to do is resize the main window. Make it about 500x300. You can tell how big it is by looking at the Property Editor under the geometry property on the right side of the designer window. Now, scroll down on the property editor list box until you see 'windowTitle'. Change the text from 'MainWindow' to 'Python Test1'. You should see the title bar of our design window change to 'Python Test1 – untitled*'. Now is a good time to save our project. Name it 'pytest1.ui'. Next, we will put a button on our form. This will be an exit button to end the test program. On the left side of the designer window you will see all of the controls that are available. Find the 'Buttons' section and drag and drop the 'Push Button' control onto the form. Unlike the GUI designers we have used in the past, you don't have to create grids to contain your controls when you use QT4 Designer. Move the button to near center-bottom of the form. If you look at the Property Editor under geometry, you will see something like this:

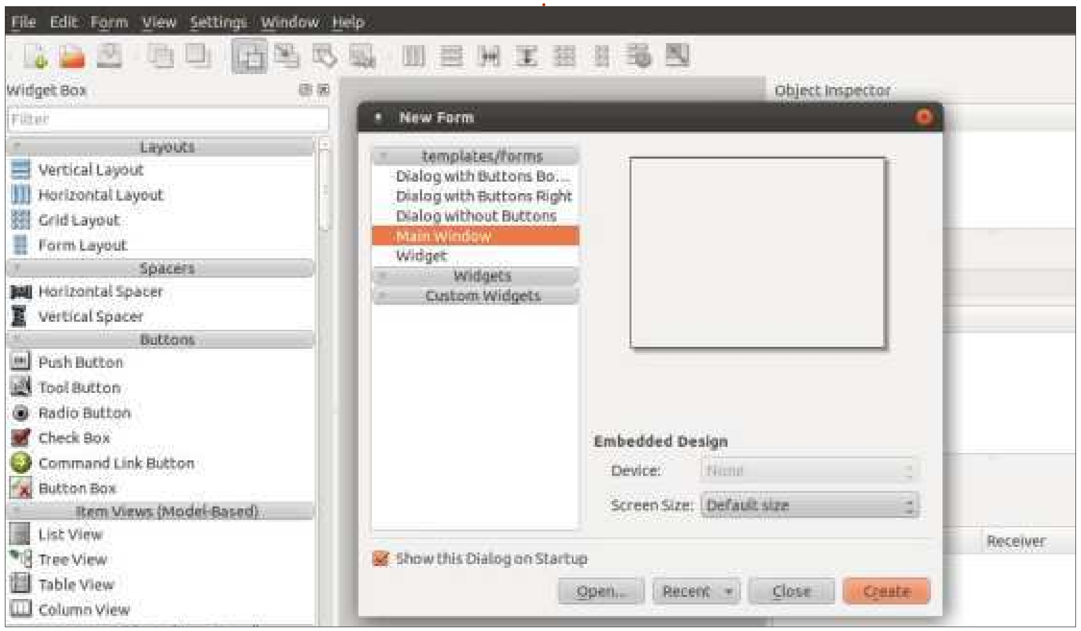
```
[(200,260), 97x27]
```

In the parentheses are the X and Y positions of the object (push-button in this case) on the form, followed by its width and height. I moved mine to 200,260.

Just above that is the objectName property—which, by default, is set to 'pushButton'. Change that to 'btnExit'. Now scroll down on the Property Editor list to the 'QAbstractButton' section, and set the 'text' property to 'Exit'. You can see on our form that the text on the button has changed.

Now, add another button and position it at 200,200. Change its objectName property to 'btnClickMe,' and set the text to 'Click Me!'.

Next add a label. You will find it in the toolbox on the left under 'DisplayWidgets'. Put it close to the center of the form (I put mine

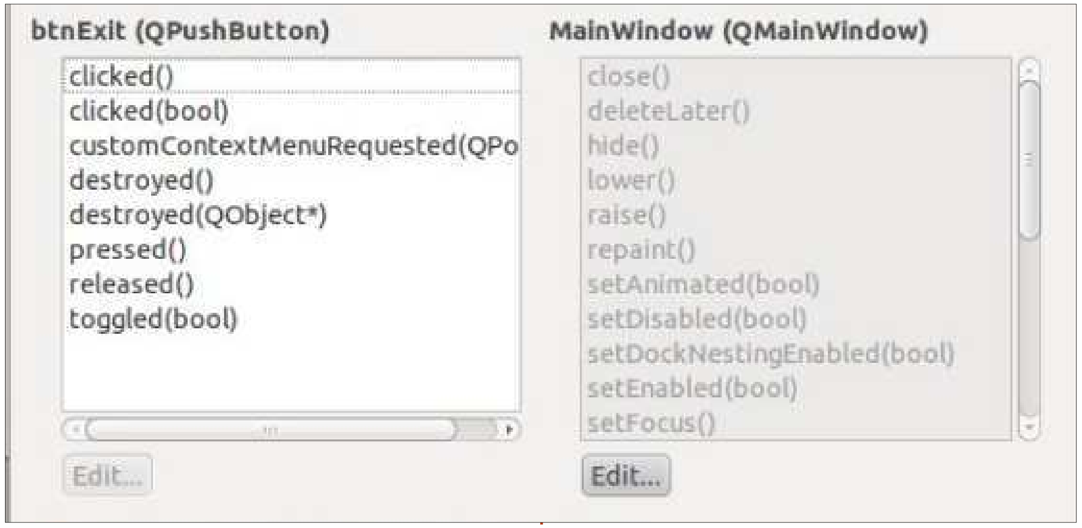


at 210,130), and set its objectName property to lblDisplay. We will want to make it bigger than what it is by default, so set its size to somewhere around 221 x 20. In the property editor, scroll down to the 'Qlabel' section, and set the Horizontal alignment to 'AlignHCenter'. Change the text to blank. We will set the text in code—when the btnClickMe is clicked. Now save the project again.

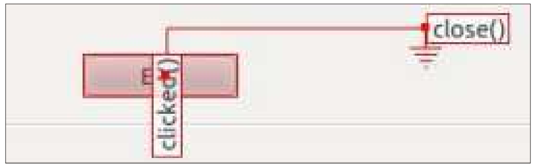
Slots & Signals

This next section might be a bit difficult to wrap your head around, especially if you have been with us for a long time and have dealt with the previous GUI designers. In the other designers, we used events that were raised when an object was clicked, like a button. In QT4 Designer, events are called Signals, and the function that is called by that signal is called a Slot. So, for our Exit button, we use the Click signal to call the Main Window Close slot. Are you totally confused right now? I was when I first dealt with QT, but it begins to make sense after a while.

Fortunately, there is a very easy way to use predefined slots & signals. If you press the F4 button on the keyboard, you will be in the Edit Signals & Slots mode. (To get out of the Edit Signals & Slots mode, press F3.) Now, left click and hold on the Exit button, and drag slightly up and to the right, off the button onto the main form, then release the click. You will see a dialog pop up that looks something like that shown above.



This will give us an easy way to connect the clicked signal to the form. Select the first option on the left which should be 'clicked()'. This will enable the right side of the window and select the 'close()' option from the list, then click 'OK'. You will see something that looks like this:



The click signal (event) is linked to the Close routine of the main window.

For the btnClickMe clicked signal, we will do that in code.

Save the file one more time. Exit QT4 Designer and open a terminal. Change to the directory that you saved the file in. Now we will generate a python file by using the command line tool pyuic4. This will read the .ui file. The command will be:

```
pyuic4 -x pytest1.ui -o
pytest1.py
```

The -x parameter says to include the code to run and display the UI. The -o parameter says to create an output file rather than just display the file in stdout. One important thing to note here. Be SURE to have everything done in QT4 Designer before you create the python file. Otherwise, it will be completely rewritten and you'll have to start over from scratch.

Once you've done this, you will have your python file. Open it up in your favorite editor.

The file itself is only about 65 lines long, including comments. We had only a few controls so, it wouldn't be very long. I'm not going to show a great deal of the code. You should be able to follow most all of the code by now. However we will be creating and adding to the code in order to put the functionality in to set the label text.

The first thing we need to do is copy the signal & slot line and modify it. Somewhere around line 47 should be the following code:

```
QtCore.QObject.connect(self.b
tnExit,
QtCore.SIGNAL(_fromUtf8("clic
ked()")), MainWindow.close)
```

Copy that, and, right below it, paste the copy. Then change it to:

```
QtCore.QObject.connect(self.b
tnClickMe,
QtCore.SIGNAL(_fromUtf8("clic
ked()")), self.SetLabelText)
```

This will then create the signal/slot connection to our routine that will set the label text. Under the retranslateUi routine add the following code:

```
def SetLabelText(self):

self.lblDisplay.setText(_from
Utf8("That Tickles!!!"))
```

I got the label setText information from the initialization line in the setupUi routine.

Now run your code. Everything should work as expected.

Although this is a VERY simple example, I'm sure you are advanced enough to play with QT4 Designer and get an idea of the power of the tool.

Next month, we will return from our detour and start working on the user interface for our TVRage program.

As always, the code can be found on pastebin at http://pastebin.com/98fSasdb for the .ui code, and http://pastebin.com/yC30B885 for the python code.

See you next time.

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

MY STORY QUICKIE

By Anthony Venable

This story begins at the beginning of 2010. I was broke at the time so I was trying to find a free operating system. I needed something I could run on my PCs at home. I had searched on the Internet, but found nothing useful for a long time. But one day I was at Barnes and Noble and I saw a magazine for Linux. (While I had heard of Linux before, I never thought of it as something I would ever be able to use.) When I asked people who I knew were computer professionals, I was told it was for people that were experts, and difficult to use. I never heard anything positive about it. I am so amazed that I hadn't came across it sooner.

When I read the magazine I became exposed to Ubuntu 9.10 - Karmic Koala. It sounded so good, as if it was exactly what I was looking for. As a result, I got very excited took it home, and to my surprise had such an easy time installing it to my PC that I decided to run it along with Windows XP as a dual boot system. All I did was put the live CD in the drive and the instructions were step by step you would have to be pretty slow to not get how to set things up.

Since then I have been very satisfied with Ubuntu in general and I have been able to check out later versions of it such as 10.04 (Maverick Meerkat) and 10.10 Lucid Lynx. I looked forward to future versiobs for how they integrate multi-touch even more than 10.04.

This experience just goes to show once again how I manage to find the coolest stuff by accident.

contents ^

Written by Greg Walters

Programming In Python - Part 45

This time, we are going to rework our database program from the previous few articles (parts 41, 42 and 43). Then, over the next few articles, we will use QT to create the user interface.

First, let's look at how the existing application works. Here's a gross overview:
• Create a connection to the database – which creates the database if needed.
• Create a cursor to the database.
• Create the table if it doesn't exist.
• Assign the video folder(s) to a variable.
• Walk through the folder(s) looking for video files.
• Get the filename, seriesname, season number, episode number.
• Check to see if the episode exists in the database.
• If it is not there, add it to the database with a "-1" as the TvRage ID.
• Then walk through the database getting show id and status if needed, and update database.

We will redesign the database to include another table and modify the existing data table. First, we will create our new table called Series. It will hold all the information about the tv series we have on our system. The new table will include the following fields:
• Pkid
• Series Name
• TvRage Series ID
• Number of seasons
• Start Date
• Ended Flag
• Country of origin
• Status of the series (ended, current, etc)
• Classification (scripted, "reality", etc)
• Summary of the series plot
• Genres
• Runtime in minutes
• Network
• Day of the week it airs
• Time of day it airs
• Path to the series

We can use the existing MakeDataBase routine to create our new table. Before the existing code, add the code shown above right.

```
    sql = 'CREATE TABLE IF NOT EXISTS Series (
            pkid INTEGER PRIMARY KEY AUTOINCREMENT,
            SeriesName TEXT,
            SeriesID TEXT,
            Seasons TEXT,
            StartDate TEXT,
            Ended TEXT,
            OriginCountry TEXT,
            Status TEXT,
            Classification TEXT,
            Summary TEXT,
            Genres TEXT,
            Runtime TEXT,
            Network TEXT,
            AirDay TEXT,
            AirTime TEXT,
            Path TEXT);'
        cursor.execute(sql)
```

The SQL statement ("sql = …") should be all on one line, but is broken out here for ease of your understanding. We'll leave the modification of the existing table for later.

Now we have to modify our WalkThePath routine to save the series name and path into the series table.

Replace the line that says

```
sqlquery = 'SELECT
count(pkid) as rowcount from
TvShows where Filename =
"%s";' % fl
```

with

```
sqlquery = 'SELECT
count(pkid) as rowcount from
series where seriesName =
"%s";' % showname
```

This (to refresh your memory) will check to see if we have already put the series into the table. Now find the two lines that say:

```
sql = 'INSERT INTO TvShows
(Series,RootPath,Filename,Sea
son,Episode,tvrageid) VALUES
(?,?,?,?,?,?)'

cursor.execute(sql,(showname,
root,fl,season,episode,-1))
```

and replace them with

```
sql = 'INSERT INTO Series
(SeriesName,Path,SeriesID)
VALUES (?,?,?)'

cursor.execute(sql,(showname,
root,-1))
```

This will insert the series name (showname), path to the series, and a “-1” as the TvRage id. We use the “-1” as a flag to know that we need the series information from TvRage.

Next we will rework the WalkTheDatabase routine to pull those series that we don’t have any information for (SeriesID = -1) and update that record.

Change the query string from

```
sqlstring = "SELECT DISTINCT
series FROM TvShows WHERE
tvrageid = -1"
```

to

```
sqlstring = "SELECT
pkid,SeriesName FROM Series
WHERE SeriesID = -1"
```

This will create a result-set that we can then use to query TvRage for each series. Now find/replace the following two lines

```
seriesname = x[0]

searchname =
string.capwords(x[0]," ")
```

with

```
pkid = x[0]

seriesname = x[1]

searchname =
string.capwords(x[1]," ")
```

We will use the pkID for the update statement. Next we have to modify the call to the UpdateDatabase routine to include the pkid. Change the line

```
UpdateDatabase(seriesname,id)
```

to

```
UpdateDatabase(seriesname,id,
pkid)
```

and change the line

```
GetShowStatus(seriesname,id)
```

to

```
GetShowData(seriesname,id,pki
d)
```

Which will be a new routine we will create in a moment.

Next, change the definition of the UpdateDatabase routine from

```
def
UpdateDatabase(seriesname,id)
:
```

to

```
def
UpdateDatabase(seriesname,id,
pkid):
```

Next, we need to change the query string from

```
sqlstring = 'UPDATE tvshows
SET tvrageid = ' + id + '
WHERE series = "' +
seriesname + '"'
```

to

```
sqlstring = 'UPDATE Series
SET SeriesID = ' + id + '
WHERE pkID = %d' % pkid
```

Now we need to create the GetShowData routine (top). We’ll grab the information from TvRage and insert it into the Series table.

```
def GetShowData(seriesname,id,pkid):
    tr = TvRage()
    idcursor = connection.cursor()
    dict = tr.GetShowInfo(id)
```

```
    seasons = dict['Seasons']
    startdate = dict['StartDate']
    ended = dict['Ended']
    origincountry = dict['Country']
    status = dict['Status']
    classification = dict['Classification']
    summary = dict['Summary']
```

Just as a memory refresher, we are creating an instance of the TvRage routines and creating a dictionary that holds the information on our series. We will then create variables to hold the data for updating the table (above).

Remember that Genres come in as subelements and contain one or many genre listings. Luckily when we coded the TvRage routines, we created a string that holds all the genres, no matter how many are returned, so we can just use the genre string:

```
genres = dict['Genres']
```

```
runtime = dict['Runtime']

network = dict['Network']

airday = dict['Airday']

airtime = dict['Airtime']
```

Finally, we create the query string to do the update (bottom). Again, this should all be on one line, but I've broken it up here to make it easy to understand.

The {number} portion (just to remind you) is similar to the "%s" formatting option. This creates our query string replacing the {number} with the actual data we want. Since we've already defined all of these fields as text, we want to use the double quotes to enclose the data being added.

And lastly, we write to the database (below).

```
    try:
        idcursor.execute(sqlstring)
    except:
        print "Error Adding Series Information"
```

That is all for this time. Next time, we'll continue as I laid out at the beginning of the article. Until next time, Enjoy.

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.

```
sqlstring = 'Update Series SET Seasons = "{0}", StartDate = "{1}", Ended = "{2}",
OriginCountry = "{3}", Status = "{4}", Classification = "{5}",
Summary = "{6}", Genres = "{7}", Runtime = "{8}", Network = "{9}",
AirDay = "{10}",AirTime = "{11}"
WHERE pkID ={12}'.format(seasons,startdate,ended,
origincountry,status,classification,summary,
genres,runtime,network,airday,airtime,pkid)
```

Programming In Python - Part 46

Written by Greg Walters

Usually, my articles are fairly long. However, due to some medical issues, this will be a fairly short article (in the grand scheme of things) this month. However, we will push through and continue our series on the media manager program.

One of the things our program will do for us is let us know if we have any missing episodes from any given series in the database. Here's the scenario. We have a series, we'll call it “That 80's Show”, that ran for three seasons. In season 2, there were 15 episodes. However, we have only 13 of them in our library. How do we find which episodes are missing – programmatically?

The simplest way is to use lists and sets. We have already used lists in a number of the articles over the last four years, but Sets are a new data type to this series, so we'll examine them for a while. According to the “official documentation” for Python (docs.python.org), here is the definition of a set:

“A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.”

I'll continue to use the example from the documentation page to illustrate the process.

```
>>> Basket = ['apple','orange','apple','pear','orange','banana']

>>> fruit = set(basket)

>>> fruit

set(['orange','pear','apple','banana'])
```

Notice that in the original list that was assigned to the basket variable, apple and orange were put in twice, but, when we assigned it to a set, the duplicates were discarded. Now, to use the set that we just created, we can check to see if an item of fruit (or something else) is in the set. We can use the “in” operator.

```
>>> 'orange' in fruit

True

>>> 'kiwi' in fruit

False

>>>
```

That's pretty simple and, hopefully, you are beginning to see where all this is going. Let's say we have a shopping list that has a bunch of fruit in it, and, as we go through the store, we want to check what we are missing – basically the items in the shopping list but not in our basket. We can start like this.

```
>>> shoppinglist = ['orange','apple','pear','banana','kiwi','grapes']

>>> basket = ['apple','kiwi','banana']

>>> sl = set(shoppinglist)

>>> b = set(basket)

>>> sl-b

set(['orange', 'pear', 'grapes'])

>>>
```

We create our two lists, shoppinglist for what we need and basket for what we have. We assign each to a set and then use the set difference operator (the minus sign) to give us the items that are in the shopping list but not in the basket.

Now, using the same logic, we will create a routine (next page, bottom left) that will deal with our missing episodes. We will call our routine “FindMissing” and pass it two variables. The first is an integer that is set to the number of episodes in that season and the second is a list containing the episode numbers that we have for that season.

The routine, when you run it, prints out [5, 8, 15], which is correct. Now let's look at the code. The first line creates a set called EpisodesNeeded using a list of integers created using the range function. We need to give the range function the start value and end value. We add 1 to the range high value to give us the correct list of values from 1 to 15.

contents ^

Remember the range function is actually 0 based, so when we give it 16 (expected (15) + 1), the actual list that range creates is 0 to 15. We tell the range function to start at 1, so even though the range is 0 to 15 which is 16 values, we want 15 starting at 1.

Next we create a set from the list that is passed into our routine, which contains the episode numbers that we actually have.

Now we can create a list using the set difference operator on the two sets. We do this so we can sort it with the list.sort() method. You can certainly return the list if you wish, but in this iteration of the routine, we'll just print it out.

Well, that's all the time in the chair in front of the computer that my body can stand, so I'll leave you for this month, wondering how we are going to use this in our media manager.

Have a good month and see you soon.

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.

```
def FindMissing(expected,have):
    #====================================
    # 'expected' is the number of episodes we should have
    # 'have' is a list of episodes that we do have
    # returns a sorted list of missing episode numbers
    #====================================
    EpisodesNeeded = set(range(1,expected+1))
    EpisodesHave = set(have)
    StillNeed = list(EpisodesNeeded - EpisodesHave)
    StillNeed.sort()
    print StillNeed

FindMissing(15,[1,2,3,4,6,7,9,10,11,12,13,14])
```

Written by Greg Walters

Programming In Python - Part 47

Last month, we discussed using sets to show us missing episode numbers. Now's the time to put the rough code we presented into practice.

We'll modify one routine and write one routine. We'll do the modification first. In the working file that you've been using the last few months, find the WalkThePath(filepath) routine. The fourth and fifth lines should be:

```
efile =
open('errors.log',"w")

for root, dirs, files in
os.walk(filepath,topdown=True
):
```

In between these two lines, we will insert the following code:

```
lastroot = ''

elist = []

currentshow = ''

currentseason = ''
```

By now, you should recognize that all we're doing here is initializing variables. There are three string variables and one list.

```
for root, dirs, files in os.walk(filepath,topdown=True):
    for file in [f for f in files if f.endswith (('.avi','mkv','mp4','m4v'))]:
```

We will use the list to hold the episode numbers (hence the elist name).

Let's take a quick look and freshen our memory (above) about what we're doing in the existing routine before we modify any further.

The first two lines here set things up for the walk-the-path routine where we start at a given folder in the file system and recursively visit each folder below, and check for files that have the file extension of .avi, .mkv, .mp4 or .m4v. If there are any, we then iterate through the list of those filenames.

In the line above right, we call the GetSeasonEpisode routine to pull the series name, season number and episode number from the filename. If everything parses correctly, the variable isok is set to true, and the data we are looking for is placed into a list and then returned to us.

```
# Combine path and filename to create a single variable.
fn = join(root,file)
OriginalFilename,ext = os.path.splitext(file)
fl = file
isok,data = GetSeasonEpisode(fl)
```

Here (below) we are simply assigning the data passed back from GetSeasonEpisode and putting them into separate variables that we can play with. Now that we know where we were, let's talk about where we are going.

We want to get the episode number of each file and put it into the elist list. Once we are done with all the files within the folder we are currently in, we can then make the assumption that we have been pretty much keeping up with the files and the highest numbered episode is the latest one available. As we discussed last month, we can then create a set that is numbered from 1 to the last episode, and convert the list to a set and pull a difference. While that is great in theory, there is a bit of a "hitch in our git-a-long" when it comes down to actual practice. We don't actually get a nice and neat indication as to when we are done with any particular folder. What we do have though, is the knowledge that when we get done with each file, the code right after the "for file in [...]" gets run. If we know the name of the last folder

```
if isok:
    showname = data[0]
    season = data[1]
    episode = data[2]
    print("Season {0} Episode {1}".format(season,episode))
```

visited, and the current folder name, we can compare the two and, if they are different, we have finished a folder and our episode list should be complete. That's what the 'lastroot' variable is for.

Just after the 'for file in[' line is where we'll put the majority of our new code. It's only seven lines. Here are the seven lines. (The black lines are the existing lines for your convenience.)

```
    for file in [f for f in files if f.endswith (('.avi','mkv','mp4','m4v'))]:
        # Combine path and filename to create a single variable.
        if lastroot != root:
            lastroot = root
            if len(elist) > 0:
                Missing(elist,max(elist),currentseason,currentshow)
            elist = []
            currentshow = ''
            currentseason = ''
        fn = join(root,file)
```

Line by line of the new code, here is the logic:

First, we check to see if the variable lastroot has the same value as root (the current folder name). If so, we are in the same folder, so we don't run any of the code. If not, we then assign the current folder name to the lastroot variable. Next, we check to see if the episode list (elist) has any entries (len(elist) > 0). This is to make sure we weren't in an empty directory. If we have items in the list, then we call the Missing routine. We pass the episode list, the highest episode number, the current season number, and the name of the season, so we can print that out later on. The last three lines clear the list, the current show name, and the current season, and we move on as we did before.

Next we have to change two lines and add one line of code into the if isok: code, a few lines down. Again, right, the black lines are the existing code:

```
        isok,data = GetSeasonEpisode(fl)
        if isok:
            currentshow = showname = data[0]
            currentseason = season = data[1]
            episode = data[2]
            elist.append(int(episode))
        else:
```

Here, we have just come back from the GetSeasonEpisode routine. If we had a parsable file name, we want to get the show name and season number, and add the current episode into the list. Notice, we are converting the episode number to an integer before we add it to the list.

We are done with this portion of the code. Now, all we have to do is add the Missing routine. Just after the WalkThePath routine, we'll add the following code.

Again, it is a very simple set of code and we pretty much went over it last month, but we'll walk through it just in case you missed it.

We define the function and set up four parameters. We will be passing the episode list (eplist), the number of episodes we should expect (shouldhave) which is the highest episode number in the episode list, the season number (season), and the show name (showname).

Next, we create a set that contains a list of numbers using the range built-in function, starting with 1 and going to the value in shouldhave + 1. We then call the difference function – on this set

```
#-----------------------------------
def Missing(eplist,shouldhave,season,showname):
    temp = set(range(1,shouldhave+1))
    ret = list(temp-set(eplist))
    if len(ret) > 0:
        print('Missing Episodes for {0} Season {1} - {2}'.format(showname,season,ret))
```

contents ^

and a converted set from the episode list (temp-set(eplist)) – and convert it back to a list. We then check to see if there is anything in the list – so we don't print a line with an empty list, and if there's anything there, we print it out.

That's it. The one flaw in this logic is that by doing things this way, we don't know if there are any new episodes that we don't have.

I've put the two routines up on pastebin for you if you just want to do a quick replace into your working code. You can find it at http://pastebin.com/XHTRv2dQ.

Have a good month and we'll see you soon.

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.

Written by Greg Walters

Programming In Python - Part 48

Welcome back. It's hard to imagine that it's been 4 years since I began this series. I thought that I'd shelve the media manager project for a bit and return to some basics of Python programming.

This month, I'll revisit the print command. It's one of the most used (at least in my programming) function that never seems to get the detail it deserves. There is a lot of things you can do with it outside of the standard '%s %d'.

Since the print function syntax is different between Python 2.x and 3.x, we'll look at them separately. Remember, however, you can use the 3.x syntax in Python 2.7. Most everything I present this month will be done from the interactive shell. You can follow along as we go. The code will look like this:

```
>>> a = "Hello Python"

>>> print("String a is %s" % a)
```

and the output will be in bold, like this:

```
String a is Hello Python
```

Python 2.x

Of course you remember the simple syntax for the print function in 2.x uses the variable substitution of %s or %d for simple strings or decimals. But many other formatting options are available. For example, if you need to format a number with leading zeros, you can do it this way:

```
>>> print("Your value is %03d" % 4)
Your value is 004
```

In this case, we use the '%03d' formatting command to say, "Display the number to a width of 3 characters and if needed, left pad with zeros".

```
>>> pi = 3.14159

>>> print('PI = %5.3f.' % pi)

PI = 3.142.
```

Here we use the float formatting option. The '%5.3f' says to produce an output with a total width of five and three decimal places. Notice that the decimal point takes up one of the places of the total width.

One other thing that you might not realize is that you can use the keys of a dictionary as part of the format command.

```
>>> info = {"FName":"Fred","LName":"Farkel","City":"Denver"}

>>> print('Greetings %(FName)s %(LName)s of %(City)s!' % info)

Greetings Fred Farkel of Denver!
```

The following table shows the various possible substitution keys and their meanings.

Conversion	Meaning
'd'	Signed integer decimal
'i'	Signed integer decimal
'u'	Obsolete - identical to 'd'
'o'	Signed octal value
'x'	Signed hexadecimal - lowercase
'X'	Signed hexadecimal - uppercase
'f'	Floating point decimal
'e'	Floating point exponential - lowercase
'E'	Floating point exponential - uppercase
'g'	Floating point format - uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise
'G'	Floating point format - uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise
'c'	Single character
'r'	String (converts valid Python object using repr())
's'	String (converts valid Python object using str())
'%'	No argument is converted, results in a '%' character

PYTHON 3.X

With Python 3.x, we have many more options (remember we can use these in Python 2.7) when it comes to the print function.

To refresh your memory, here's a simple example of the 3.x print function.

```
>>> print('{0}
{1}'.format("Hello","Python")
)
Hello Python

>>> print("Python is {0}
cool!".format("WAY"))
Python is WAY cool!
```

The replacement fields are enclosed within curly brackets "{" "}". Anything outside of these are considered a literal and will be printed as is. In the first example, we have numbered the replacement fields 0 and 1. That tells Python to take the first (0) value and put it into the field {0} and so on. However, you don't have to use any numbers at all. Using this option causes the first value to be places in the first set of brackets and so on.

```
>>> print("This version of {}
is
{}".format("Python","3.3.2"))
This version of Python is
3.3.2
```

As they say on the TV ads, "BUT WAIT… THERE'S MORE". If we wanted to do some inline formatting, we have the following options.

```
:<x Left align with a width
of x
:>x Right align with a width
of x
:^x Center align with a width
of x
```

Here is an example:

```
>>>
print("|{:<20}|".format("Left
"))
|Left                |
>>>
print("|{:>20}|".format("Righ
t"))
|               Right|
>>>
print("|{:^20}|".format("Cent
er"))
|       Center       |
```

You can even specify a fill character along with the justification/width.

```
>>>
print("{:*>10}".format(321.40
))
*****321.4
```

If you need to format a date/time output, you can do something like this:

```
>>> d =
datetime.datetime(2013,10,9,1
0,45,1)

>>>
print("{:%m/%d/%y}".format(d)
)
10/09/13

>>>
print("{:%H:%M:%S}".format(d)
)
10:45:01
```

Printing thousands separator using a comma (or any other character) is simple.

```
>>> print("This is a big
number
{:,}".format(7219219281))
This is a big number
7,219,219,281
```

Well, that should give you enough food for thought for this month. I'll see you at the start of the 5th year.

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.

ubuntu
uk podcast

The Ubuntu Podcast covers all the latest news and issues facing Ubuntu Linux users and Free Software fans in general. The show appeals to the newest user and the oldest coder. Our discussions cover the development of Ubuntu but aren't overly technical. We are lucky enough to have some great guests on the show, telling us first hand about the latest exciting developments they are working on, in a way that we can all understand! We also talk about the Ubuntu community and what it gets up to.

The show is presented by members of the UK's Ubuntu Linux community. Because it is covered by the Ubuntu Code of Conduct it is suitable for all.

The show is broadcast live every fortnight on a Tuesday evening (British time) and is available for download the following day.

podcast.ubuntu-uk.org