Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

PROGRAMMING SERIES SPECIAL EDITION

PROGRAMMING SERIES SPECIAL EDITION

PROGRAM IN PYTHON

Volume Seven

Parts 39-43

Full Circle Magazine Specials

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 39-43** from issues #68 through #72, by peerless Python professor Gregg Walters.

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.

Written by Greg Walters

Programming In Python: Pt 39

Many, many months ago, we worked with API calls for Weather Underground. Actually, it was in part 11 which was back in issue #37. Well, we are going to deal with APIs again, this time for a website named TVRage (http://tvrage.com). If you aren't familiar with this site, it deals with television shows. So far, every TV show that I could think of has been in their system. In this series of articles, we are going to revisit XML, APIs, and ElementTree to create a wrapper library that will allow us to create a small library which simplifies our retrieval of TV information on our favorite shows.

Now, I mentioned a wrapper library. What's that? In simple terms, when you create or use a wrapper library, you are using a set of code that "wraps" the complexity of the website's API into an easy-to-use library. Before we get started, I need to make a few things clear. First, this is a free service. However, they do request donations for use of their API. If you feel that this is a worthwhile service, please consider donating $10 US or more. Second, you should register at their website and get your own API key. It's free, so there's really no reason not to, especially if you are going to use the information provided here. In addition, you have access to a few other fields of information like series and episode summaries that are not included in the unregistered version. Third, they are hard at work at updating the API. This means that when you get to seeing this article, their API might have changed. We'll be using the public feeds, which are free for everyone to use as of December 2012. The API website is located at http://services.tvrage.com/info.php?page=main and shows a few examples of the types of information that are available.

Now, let's begin looking at the API and how we can use it.

Using their API, we can get very specific information about the show itself and/or we can get episode level information. There are basically three steps to finding information about TV Shows. Here are the steps:

• Search their database looking for the show name to get the specific Show ID which must be used to get more data. Think of the showid value as a key directly into a record set in a database, which in this case it is.
• Once you have the Show ID, obtain the show level information.
• Finally, gather the information about a specific episode. This comes from a list of each and every episode that the show has had to date.

There are three basic web calls we will make to get this information. First is the search call, second the show information call, and finally the the episode list call.

Here are the base calls that we will use...

• Search for ShowID based on a show name - `http://services.tvrage.com/feeds/search.php?show={SomeShow}`

• Pull the show level data based on the Show ID (sid) - `http://services.tvrage.com/feeds/showinfo.php?sid={SomeShowID}`

• Pull the episode list for Show ID (sid) - `http://services.tvrage.com/feeds/episode_list.php?sid={SomeShowID}`

What gets returned is a stream of data in XML format. Let's take a moment to review what XML looks like. The first line should always be similar to the one shown below to be considered a proper XML data stream (below).

```
<?xml version="1.0" encoding="UTF-8" ?>
<ROOT TAG>
   <PARENT TAG>
      <CHILD TAG 1>DATA</CLOSING CHILD TAG 1>
      <CHILD TAG 2>DATA</CLOSING CHILD TAG 2>
      <CHILD TAG 3>DATA</CLOSING CHILD TAG 3>
   </CLOSING PARENT TAG>
</CLOSING ROOT TAG>
```

Every piece of data is enclosed within a defining tag and end-tag. Sometimes you will have a child tag that is a parent tag in itself like this...

```
<CHILD PARENT TAG>

<CHILD TAG 1>DATA</CLOSING
CHILD TAG 1>

</CLOSING CHILD PARENT TAG>
```

You also may see a tag that has an attribute associated with it:

```
<TAG INFORMATION = VALUE>

<CHILD TAG>DATA</CLOSING
CHILD TAG>

</CLOSING TAG>
```

Sometimes, you might see a tag with no data associated with it. It would come across like this...

```
<prodnum/>
```

Sometimes, if there is no information for a specific tag, the tag itself just won't be there. Your program will have to deal with these possibilities.

So, when we go through and deal with the XML data, we start with the root tag, and parse each tag – looking for the data we care about. In some instances we want everything; in others, we care about only certain pieces of the information.

Now, let's look at the first call and see what gets returned. Assume the show we are looking for is Buffy the Vampire Slayer. Our search call would look like this:

```
http://services.tvrage.com/fe
eds/search.php?show=buffy
```

The returned XML file would look like this: http://pastebin.com/Eh6ZtJ9N.

Note that I put the indents in myself to make it easier for you to read. Now let's break down the XML file to see what we actually have.

<Results> - This is the ROOT of the XML data. The last line of the stream we get back should be the closing tag </Results>. Basically, this marks the beginning and end of the XML stream. There could be zero results or fifty results.
<show> This is the parent node that says "What follows (until the end show tag) is the information about a single tv show". Again, it's

ended by its end tag </show>. Anything within these two tags should be considered one show's worth of information.
<showid>2930</show> This is the showid tag. This holds the sid that we have to use to get the show information, in this case 2930.
<name>Buffy the Vampire Slayer</name> This is the name of the show
<link>...</link> This would be the link to the show itself (or, in the case of an episode, the episode information) on the TVRage website.
<country>...</country> The country of origin for the show.
...
</show>
</Results>

In the case of our program, we would be really interested in only the two fields <showid> and <name>. We might also consider paying attention to the <started> field as well. This is because we rarely get back just one set of data, especially if we didn't give the absolutely complete show name. For example, if we were interested in the show "The Big Bang Theory," and searched using only the string "Big Bang", we would get twenty or so data sets back because anything that even remotely matched "big" or "bang" would be returned. However, if we were interested in the show "NCIS," and we searched for that, we would get back many responses. Some not what we would expect right away. Not only would we get "NCIS", "NCIS: Los Angeles", "The Real NCIS", but also "The Streets of San Francisco" and "Da Vinci's Inquest", and many more, since the letters "N" "C" "I" and "S" are in all of those, pretty much in that order.

Once we know the show id that we want, then we can request the show information for that ID. The data is similar to the data we just got back in the search response, but more detailed. Again, using Buffy as our example request, here (next page, right) is an abbreviated version of the XML file.

You can see that much of the data is included in the original search response stream. However, things like network, network country, runtime, air day and time, are specific to this response set.

Next, we would request the episode list. If the show is only one season long and has/had only six episodes, this stream would be short. However, let's take the case of one of my favorite TV shows, Doctor Who. Doctor Who is a British TV show that, in its original form, started in 1963 and ran for 26 seasons ('series' for our friends in the UK) until 1989. Its first season alone had 42 episodes, while other seasons/series have around 24 episodes. You can see where you might have a HUGE stream to parse through.

What we get back from the episode list request is as shown on the next page (again using Buffy as our example); I'm going to just use part of the stream so you get a good idea of what comes back.

So to recap, the information we really want/need in the search for show id by name stream would be...

```
<showid>
<name>
<started>
```

In the Show Information stream we would (normally) want...

```
<seasons>
<started>
<start date>
<origin_country>
<status>
<genres>
<runtime>
<network>
<airtime>
<airday>
<timezone>
```

and from the episode list stream...

```
<Season>
<episode number>
<season number>
<production number>
<airdate>
<link>
<title>
```

A word of "warning" here. Season number and Episode number data are not what you might think right away. In the case of

```
<Showinfo>
   <showid>2930</showid>
   <showname>Buffy the Vampire Slayer</showname>
   <showlink>http://tvrage.com/Buffy_The_Vampire_Slayer</showlink>
   <seasons>7</seasons>
   <started>1997</started>
   <startdate>Mar/10/1997</startdate>
   <ended>May/20/2003</ended>
   <origin_country>US</origin_country>
   <status>Canceled/Ended</status>
   <classification>Scripted</classification>
   <genres>
      <genre>Action</genre>
      <genre>Adventure</genre>
      <genre>Comedy</genre>
      <genre>Drama</genre>
      <genre>Mystery</genre>
      <genre>Sci-Fi</genre>
   </genres>
   <runtime>60</runtime>
   <network  country="US">UPN</network>
   <airtime>20:00</airtime>
   <airday>Tuesday</airday>
   <timezone>GMT-5 -DST</timezone>
   <akas>
      <aka country="SE">Buffy &amp; vampyrerna</aka>
      <aka country="DE">Buffy – Im Bann der Dämonen</aka>
      <aka country="NO">Buffy – Vampyrenes skrekk</aka>
      <aka country="HU">Buffy a vámpírok réme</aka>
      <aka country="FR">Buffy Contre les Vampires</aka>
      <aka country="IT">Buffy l'Ammazza Vampiri</aka>
      <aka country="PL">Buffy postrach wampirów</aka>
      <aka country="BR">Buffy, a Caça-Vampiros</aka>
      <aka country="PT">Buffy, a Caçadora de Vampiros</aka>
      <aka country="ES">Buffy, Cazavampiros</aka>
      <aka country="HR">Buffy, ubojica vampira</aka>
      <aka country="FI">Buffy, vampyyrintappaja</aka>
      <aka country="EE">Vampiiritapja Buffy</aka>
      <aka country="IS">Vampírubaninn Buffy</aka>
   </akas>
</Showinfo>
```

the data from TVRage, the season number is the number of the episode within the season. The episode number is the number for that episode within the total life span of the series. The production number is a number that was used internally to the series, that, for many people, means little if anything.

Now that we have refreshed our memory on XML file structures and examined the TVRage API calls, we are ready to start our coding, but that will have to wait until next time.

Until then, have a good holiday season.

```
<Show>
   <name>Buffy the Vampire Slayer</name>
   <totalseasons>7</totalseasons>
   <Episodelist>
      <Season no="1">
         <episode>
            <epnum>1</epnum>
            <seasonnum>01</seasonnum>
            <prodnum>4V01</prodnum>
            <airdate>1997-03-10</airdate>
            <link>http://www.tvrage.com/Buffy_The_Vampire_Slayer/episodes/28077</link>
            <title>Welcome to the Hellmouth (1)</title>
         </episode>
         <episode>
            <epnum>2</epnum>
            <seasonnum>02</seasonnum>
            <prodnum>4V02</prodnum>
            <airdate>1997-03-10</airdate>
            <link>http://www.tvrage.com/Buffy_The_Vampire_Slayer/episodes/28078</link>
            <title>The Harvest (2)</title>
         </episode>
         <episode>
            <epnum>3</epnum>
            <seasonnum>03</seasonnum>
            <prodnum>4V03</prodnum>
            <airdate>1997-03-17</airdate>
            <link>http://www.tvrage.com/Buffy_The_Vampire_Slayer/episodes/28079</link>
            <title>Witch</title>
         </episode>
         ...
      </Season>
   </Episodelist>
</Show>
```

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: Pt 40

Last time, we had a gross discussion about the TVRAGE web API. Now we will start to look at writing code to work with it.

The goal of this part is to begin the process of creating code that will be a reusable module that can be imported into any python program and will provide access to the API easily.

While the TVRAGE API gives us a number of things we can do, and the registered version even more, we will concentrate on only three calls:
1 - Search for show by show name, and get the ShowID
2 - Get show information based on ShowID
3 - Get episode specific information based on ShowID

Last time, I showed you the "unregistered" and accessible-by-anyone API calls. This time we will use the registered calls – based on a registration key I have. I'm going to share this key with you (TVRAGE knows that I'm going to do this). However, I ask that, if you are going to use the API, that you please register and get your own key, and that you don't abuse the site. Please also consider donating to them to support their continuing efforts.

We will create three main routines to make the calls and return the information, three routines that will be used to display the returned information (assuming that we are running in the "stand alone" mode), and a main routine to do the work – again assuming that we are running in the "stand alone" mode.

Here is the list of routines we will be creating (although not all of them this time. I want to leave room for others in this issue.)

```
def FindIdByName(self,
shownname, debug = 0)

def GetShowInfo(self, showid,
debug = 0)

def GetEpisodeList(self,
showid, debug = 0)

def DisplaySearchResult(self,
ShowListDict)

def DisplayShowInfo(self,
dict)

def DisplayEpisodeList(self,
SeriesName, SeasonCount,
EpisodeList)

def main()
```

The routine FindIdByName takes a string (showname), makes the API call, parses the XML response, and returns a list of shows that match with the information in a dictionary, so this will be a list of dictionaries. GetShowInfo takes the showid from the above routine and returns a dictionary of information about the series. GetEpisodeList also uses the showid from the above routine and returns a list of dictionaries containing information for each episode.

We will use a series of strings to hold the key and the base URL, and then append to those what we need. For example consider the following code (we'll expand these later).

```
self.ApiKey =
"Itnl8IyY1hsR9n0IP6zI"

self.FindSeriesString =
"http://services.tvrage.com/m
yfeeds/search.php?key="
```

The call we need to send (to get back a list of series information with the series id) would be:

http://services.tvrage.com/myfeeds/search.php?key=Itnl8IyY1hsR9n0IP6zI&show={ShowName}

We combine the string like this...

```
strng = self.FindSeriesString
+ self.ApiKey + "&show=" +
showname
```

For the purposes of testing, I will be using a show named "Continuum" which, if you've never seen it, is a wonderful science fiction show on the Showcase network out of Canada. I'm using this show for a few reasons. First, there are only (as of this writing) two shows that match the search string "Continuum", so that makes your debug easy, and secondly, there's currently only one season of 10 episodes for you to deal with.

contents ^

You should have an idea what you will be looking for in your parsing routines, so I've placed the full URL calls below for you to test, before you get started with your coding.

Search using a show name...
http://services.tvrage.com/myfeeds/search.php?key=Itnl8IyY1hsR9n0IP6zI&show=continuum

Retrieve Series information using the ShowID (sid)
http://services.tvrage.com/myfeeds/showinfo.php?key=Itnl8IyY1hsR9n0IP6zI&sid=30789

Retrieve Episode list and information using the ShowID (sid)
http://services.tvrage.com/myfeeds/episode_list.php?key=Itnl8IyY1hsR9n0IP6zI&sid=30789

Now that we have all that out of the way, let's get started with our code.

You'll create a file with the name of "tvrage.py". We'll be using this for the next issue or two.

We'll start with our imports shown above right.

```
#======================================================
#                         IMPORTS
#======================================================
from xml.etree import ElementTree as ET
import urllib
import sys
```

You can see that we will be using ElementTree to do the XML parsing, and urllib for the internet communication. The sys library is used for sys.exit.

We'll set up the main loop now so we can test things as we go (bottom right). Remember this is the last thing in our source file.

As I said earlier, the first four lines are our partial strings to build the URL for the function that we want to use. (GetEpisodeListString should all be on one line.) The last four lines are the initialization of the lists we will be using later.

First (middle right), we set up the string that will be used as the URL. Next, we set up the socket with an 8 second default timeout. Then we call urllib.urlopen with our generated URL and (hopefully)

```
    def FindIdByName(self,showname,debug = 0):
        strng = self.FindSeriesString + self.ApiKey + "&show=" + showname
        urllib.socket.setdefaulttimeout(8)
        usock = urllib.urlopen(strng)
        tree = ET.parse(usock).getroot()
        usock.close()
        foundcounter = 0
        self.showlist = []
```

```
#============================================================
#     Main loop
#============================================================
if __name__ == "__main__":
     main()
```

Now we start our class. The name of the class is "TvRage". We'll also make our __init__ routine now.

```
class TvRage:
    def __init__(self):
        self.ApiKey = "Itnl8IyY1hsR9n0IP6zI"
        self.FindSeriesString = "http://services.tvrage.com/myfeeds/search.php?key="
        self.GetShowInfoString = "http://services.tvrage.com/myfeeds/showinfo.php?key="
        self.GetEpisodeListString =
"http://services.tvrage.com/myfeeds/episode_list.php?key="
        self.ShowList = []
        self.ShowInfo = []
        self.EpisodeList = []
        self.EpisodeItem = []
```

receive our xml file in the usock object. We call ElementTree setup so we can parse the xml information. (If you are lost here, please re-read my articles on XML (parts 10, 11 and 12 appearing in FCM #36, 37 and 38)). Next, we close the socket, and initialize the counter for the number of matches found, and reset the list 'showlist' to an empty list.

Now we will step through the xml information using the tag 'show' as the parent for what we want. Remember the returned information looks something like that shown top right.

We will be going through each group of information for the parent 'show' and parsing out the information. In practice, all we really need is the show name (<name>) and the showid (<showid>) shown bottom left, but we'll handle all of the results.

I'll discuss the first one and you'll understand the rest. As we go through the information, we look for tags (bottom right) that match what we want. If we find any, we assign each to a temporary variable and then put that into the dictionary as a value with a key that matches what we are putting in. In the case of the above, we are looking for the tag 'showid' in the XML data. When we find it, we assign that as a value to the dictionary key 'ID'.

The next portion (next page, top right) deals with the genre(s) of the show. As you can see from the above XML snippet, this show has four different genres that it fits into. Action, Crime, Drama, and Sci-Fi. We need to handle each.

Finally, we increment the foundcounter variable, and append this dictionary into the list 'showlist'. Then we start the entire thing over until there is no more

```
        for node in tree.findall('show'):
            showinfo = []
            genrestring = None
            dict = {}
            for n in node:
                if n.tag == 'showid':
                    showid = n.text
                    dict['ID'] = showid
```

```
<Results>
    <show>
        <showid>30789</showid>
        <name>Continuum</name>
        <link>http://www.tvrage.com/Continuum</link>
        <country>CA</country>
        <started>2012</started>
        <ended>0</ended>
        <seasons>2</seasons>
        <status>Returning Series</status>
        <classification>Scripted</classification>
        <genres>
            <genre>Action</genre>
            <genre>Crime</genre>
            <genre>Drama</genre>
            <genre>Sci-Fi</genre>
        </genres>
    </show>
    ...
</Results>
```

```
    elif n.tag == 'name':
         showname = n.text
         dict['Name'] = showname
     elif n.tag == 'link':
         showlink = n.text
         dict['Link'] = showlink
     elif n.tag == 'country':
         showcountry = n.text
         dict['Country'] = showcountry
     elif n.tag == 'started':
         showstarted = n.text
         dict['Started'] = showstarted
     elif n.tag == 'ended':
         showended = n.text
         dict['Ended'] = showended
     elif n.tag == 'seasons':
         showseasons = n.text
         dict['Seasons'] = showseasons
     elif n.tag == 'status':
         showstatus = n.text
         dict['Status'] = showstatus
     elif n.tag == 'classification':
         showclassification = n.text
         dict['Classification'] = showclassification
```

XML data. Once everything is done, we return the list of dictionaries (bottom right).

Most of the code is pretty self explanatory. We'll concentrate on the for loop we use to print out the information. We loop through each item in the list of dictionaries and print a counter variable, the show name (c['Name']), and the id. The result looks something like this...

```
Enter Series Name ->
continuum
2 Found
------------------------
1 - Continuum - 30789
2 - Continuum (Web series) -
32083
Enter Selection or 0 to exit
->
```

Please remember that the list of items is zero based, so when the user enters '1', they are really asking for dictionary number 0. We do this, because "regular" people think that counting should start with '1' not 0. And we can then use 0 to escape the routine and not make them use 'Q' or 'q' or '-1'.

Now, the "main" routine that pulls it all together for us.

For today, we'll just start the routine (middle right) and continue it next time.

Next time, we'll add the other routines. For now, the code can be found at http://pastebin.com/6iw5NQrW

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.

```
elif n.tag == 'genres':
    for subelement in n:
        if subelement.tag == 'genre':
            if subelement.text != None:
                if genrestring == None:
                    genrestring = subelement.text
                else:
                    genrestring += " | " + subelement.text
    dict['Genres'] = genrestring
```

```
def main():
    tr = TvRage()
    #---------------------
    # Find Series by name
    #---------------------
    nam = raw_input("Enter Series Name -> ")
    if nam != None:
        sl = tr.FindIdByName(nam)
        which = tr.DisplayShowResult(sl)
        if which == 0:
            sys.exit()
        else:
            option = int(which)-1
            id = sl[option]['ID']
            print "ShowID selected was %s" % id
```

```
                foundcounter += 1
                self.showlist.append(dict)
        return self.showlist
#=========================================================
```

The next thing we will do is create the routine to display all of our results.

```
        def DisplayShowResult(self, ShowListDict):
            lcnt = len(ShowListDict)
            print "%d Found" % lcnt
            print "------------------------"
            cntr = 1
            for c in ShowListDict:
                print "%d - %s - %s" % (cntr,c['Name'],c['ID'])
                cntr += 1
            sel = raw_input("Enter Selection or 0 to exit -> ")
            return sel
```

Written by Greg Walters

Programming In Python: Pt 41

```
def GetShowInfo(self,showid,debug=0):
    showidstr = str(showid)
    strng = self.GetShowInfoString + self.ApiKey + "&sid=" + showidstr
    urllib.socket.setdefaulttimeout(8)
    usock = urllib.urlopen(strng)
    tree = ET.parse(usock).getroot()
    usock.close()
    dict = {}
```

Last month, we started our command line version of a library to talk to the TVRAGE web API. This month we will continue adding to that library. If you don't have the code from last month, please get it now from pastebin (http://pastebin.com/6iw5NQrW) because we will be adding to that code.

The way we left the code, you would run the program and enter in the terminal window the name of a TV show you want information on. Remember, we used the show Continuum. Once you pressed <Enter>, the program would call the api and search by the name of the show, and then return a list of show names that matches your input. You then would select from the list by entering a number and it would show "ShowID selected was 30789". Now, we will create the code that will use that ShowID to get the series information. One other thing to keep in mind: the display routines are there pretty much to prove the routine works. The ultimate goal here is to create a reusable library that can be used in something like a GUI program. Feel free to modify the display routines if you want to do more with the standalone capabilities of the library.

The last routine we created in the class was "DisplayShowResult". Right after that, and before the routine "main," is where we will put our next routine. The information that will be returned (there is other information, but we will use only the list below) will be in a dictionary and will contain (if available):

- Show ID
- Show Name
- Show Link
- Origin Country of network
- Number of seasons
- Series image
- Year Started
- Date Started
- Date Ended
- Status
(canceled, returning, current, etc)
- Classification
(scripted, reality, etc)
- Series Summary
- Genre(s)
- Runtime in minutes
- Name of the network that originally aired the show
- Network country
(pretty much the same thing as Origin Country)
- Air time
- Air Day (of week)
- TimeZone

Shown above is the beginning of the code.

You should recognize most of the code from last time. There's really not much changed. Here's more code (shown below).

```
for child in tree:
    if child.tag == 'showid':
        dict['ID'] = child.text
    elif child.tag == 'showname':
        dict['Name'] = child.text
    elif child.tag == 'showlink':
        dict['Link'] = child.text
    elif child.tag == 'origin_country':
        dict['Country'] = child.text
    elif child.tag == 'seasons':
        dict['Seasons'] = child.text
    elif child.tag == 'image':
        dict['Image'] = child.text
    elif child.tag == 'started':
        dict['Started'] = child.text
    elif child.tag == 'startdate':
        dict['StartDate'] = child.text
        elif child.tag == 'ended':
            dict['Ended'] = child.text
        elif child.tag == 'status':
            dict['Status'] = child.text
        elif child.tag == 'classification':
            dict['Classification'] = child.text
        elif child.tag == 'summary':
            dict['Summary'] = child.text
```

As you can see (above), there's nothing really new in this bit of code either, if you've been keeping up with the series. We are using a for loop, checking each tag in the XML file for a specific value. If we find it, we assign it to a dictionary item.

Now things get a bit more complicated. We are going to check for the tag "genres". This has child tags underneath it with the name of "genre". For any given show, there can be multiple genres. We'll have to append the genres to a string as they come up and separate them with a vertical bar and two spaces like this " | " (shown top right).

Now we are pretty much back to "normal" code (shown middle right) that you've already seen. The only thing that's a bit different is the tag "network" which has an attribute "country". We grab the attribute data by looking for "child.attrib['attributetag']" instead of "child.text".

That's the end of this routine. Now (below) we'll need some way to display the information we worked so hard to get. We'll create a routine called "DisplayShowInfo".

Now, we must update the "main" routine (next page, shown top right) to support our two new routines. I'm giving the entire routine below, but the new code is shown **in black**.

```
        elif child.tag == 'genres':
            genrestring = None
            for subelement in child:
                if subelement.tag == 'genre':
                    if subelement.text != None:
                        if genrestring == None:
                            genrestring = subelement.text
                        else:
                            genrestring += " | " + subelement.text
            dict['Genres'] = genrestring
```

```
        elif child.tag == 'runtime':
            dict['Runtime'] = child.text
        elif child.tag == 'network': # has attribute
            dict['NetworkCountry'] = child.attrib['country']
            dict['Network'] = child.text
        elif child.tag == 'airtime':
            dict['Airtime'] = child.text
        elif child.tag == 'airday':
            dict['Airday'] = child.text
        elif child.tag == 'timezone':
            dict['Timezone'] = child.text
    return dict
```

Next page, bottom left, is what the output of "DisplayShowInfo" should look like, assuming you chose "Continuum" as the show.

Please notice that I'm not displaying the time zone information here, but feel free to add it if you wish.

```
    def DisplayShowInfo(self,dict):
        print "Show: %s" % dict['Name']
        print "ID: %s  Started: %s  Ended: %s  Start Date: %s  Seasons: %s" %
(dict['ID'],dict['Started'],dict['Ended'],dict['StartDate'],dict['Seasons'])
        print "Link: %s" % dict['Link']
        print "Image: %s" % dict['Image']
        print "Country: %s  Status: %s  Classification: %s" %
(dict['Country'],dict['Status'],dict['Classification'])
        print "Runtime: %s  Network: %s  Airday: %s  Airtime: %s" %
(dict['Runtime'],dict['Network'],dict['Airday'],dict['Airtime'])
        print "Genres: %s" % dict['Genres']
        print "Summary: \n%s" % dict['Summary']
```

Next, we need to work on the episode list routines for the series. The “worker” routine will be called “GetEpisodeList” and will provide the following information...

• Season
• Episode Number
• Season Episode Number (the number of the episode within the season)
• Production Number
• Air Date
• Link
• Title
• Summary
• Rating
• Screen Capture Image of Episode (if available)

Before we start with the code, it would be helpful to revisit what the episode list request to the API returns. It looks something like that shown on the next page, top right.

The information for each episode is in the “episode” tag – which is a child of “Season” – which is a child of “Episodelist” – which is a child of “Show”. We have to be careful how we parse this. As with most of our “worker” routines this time, the first few lines (below) are fairly easy to understand by now.

Now we need to look for the “name” and “totalseasons” tags below the “root” tag “Show”. Once we’ve dealt with them, we look for

```
ShowID selected was 30789
Show: Continuum
ID: 30789  Started: 2012  Ended: None  Start Date:
May/27/2012  Seasons: 2
Link: http://www.tvrage.com/Continuum
Image: http://images.tvrage.com/shows/31/30789.jpg
Country: CA  Status: Returning Series   Classification:
Scripted
Runtime: 60  Network: Showcase   Airday: Sunday
Airtime: 21:00
Genres: Action | Crime | Drama | Sci-Fi
Summary:
Continuum is a one-hour police drama centered on Kiera
Cameron, a regular cop from 65 years in the future who
finds herself trapped in present day Vancouver. She is
alone, a stranger in a strange land, and has eight of the
most ruthless criminals from the future, known as Liber8,
loose in the city.

Lucky for Kiera, through the use of her CMR (cellular
memory recall), a futuristic liquid chip technology
implanted in her brain, she connects with Alec Sadler, a
seventeen-year-old tech genius. When Kiera calls and Alec
answers, a very unique partnership begins.

Kiera's first desire is to get "home." But until she
figures out a way to do that, she must survive in our
time period and use all the resources available to her to
track and capture the terrorists before they alter
history enough to change the course of the future. After
all, what's the point of going back if the future isn't
the one you left?
```

```
def main():
    tr = TvRage()
    #--------------------
    # Find Series by name
    #--------------------
    nam = raw_input("Enter Series Name -> ")
    if nam != None:
        sl = tr.FindIdByName(nam)
        which = tr.DisplayShowResult(sl)
        if which == 0:
            sys.exit()
        else:
            option = int(which)-1
            id = sl[option]['ID']
            print "ShowID selected was %s" % id
    #------------------------------
    # Get Show Info
    #------------------------------
    showinfo = tr.GetShowInfo(id)
    #------------------------------
    # Display Show Info
    #------------------------------
    tr.DisplayShowInfo(showinfo)
```

```
        def GetEpisodeList(self,showid,debug=0):
            showidstr = str(showid)
            strng = self.GetEpisodeListString + self.ApiKey
+ "&sid=" + showidstr
            urllib.socket.setdefaulttimeout(8)
            usock = urllib.urlopen(strng)
            tree = ET.parse(usock).getroot()
            usock.close()
            for child in tree:
                if child.tag == 'name':
                    ShowName = child.text
                elif child.tag == 'totalseasons':
                    TotalSeasons = child.text
                elif child.tag == 'Episodelist':
                    for c in child:
                        if c.tag == 'Season':
                            dict = {}
                            seasonnum = c.attrib['no']
                            for el in c:
```

the “Episodelist”, “Season” tags. Notice above that the “Season” tag has an attribute. You might notice (in the code above) that we aren’t including the “Showname” or “Totalseasons” data in the dictionary. We are assigning them to a variable that will be returned at the end of the routine to the calling code.

Now that we have that portion of the data, we deal with the episode specific information (shown below).

All that’s left now (bottom right) is to append the episode specific information (that we’ve put into the dictionary) to our list, and keep going. Once we are done with all the episodes, we return to the calling routine and, as I stated earlier, return three items of data, “ShowName”, “TotalSeasons” and the list of dictionaries.

```
    if el.tag == 'episode':
        dict={}
        dict['Season'] = seasonnum
        for ep in el:
            if ep.tag == 'epnum':
                dict['EpisodeNumber'] = ep.text
            elif ep.tag == 'seasonnum':
                dict['SeasonEpisodeNumber'] = ep.text
            elif ep.tag == 'prodnum':
                dict['ProductionNumber'] = ep.text
            elif ep.tag == 'airdate':
                dict['AirDate'] = ep.text
            elif ep.tag == 'link':
                dict['Link'] = ep.text
            elif ep.tag == 'title':
                dict['Title'] = ep.text
            elif ep.tag == 'summary':
                dict['Summary'] = ep.text
            elif ep.tag == 'rating':
                dict['Rating'] = ep.text
            elif ep.tag == 'screencap':
                dict['ScreenCap'] = ep.text
```

```
<Show>
<name>Continuum</name>
<totalseasons>2</totalseasons>
<Episodelist>
<Season no="1">
<episode>
<epnum>1</epnum>
<seasonnum>01</seasonnum>
<prodnum/>
<airdate>2012-05-27</airdate>
<link>
http://www.tvrage.com/Continuum/episodes/1065162187
</link>
<title>A Stitch in Time</title>
<summary>
Inspector Kiera Cameron loses everything she has and finds
herself on a new mission when she and eight dangerous
terrorists are transported from their time in 2077 back to
2012 during the terrorist's attempt to escape execution.
She takes on a new identity and joins the VPD in order to
stop the terrorists' reign of violence. Along the way, she
befriends Alec Sadler, the 17 year old who will one day
grow up to create the technology her world is built upon.
</summary>
<rating>8.8</rating>
<screencap>
http://images.tvrage.com/screencaps/154/30789/1065162187.p
ng
</screencap>
</episode>
```

Next, we need to create our display routine. Again, it’s fairly straightforward. The only thing that you might not recognize is the “if e.has_key(‘keynamehere’)“ lines. This is a check to make sure that there is actually data in the “Rating” and “Summary” variables.

```
self.EpisodeItem.append(dict)
        return ShowName,TotalSeasons,self.EpisodeItem
```

Some shows don't have this information, so we include the check to make our print-to-screen data a little prettier (shown above right).

All that's left is to update our "main" routine (next page, shown top right). Once again, I'm going to provide the full "main" routine with the newest code **in black bold**.

Now, if you save and run the program, the output of the "GetEpisodeList" and "DisplayEpisodeList" will work. Shown bottom right is a snippet of the Episode information.

That's it for this month. As always, you can find the full source code on pastebin at http://pastebin.com/kWSEfs2E. I hope you enjoy playing with the library. There is additional data available from the API that you can include. Please remember, TVRage provides this information for free, so consider donating to them to help their efforts at updating the API and for all their hard work.

I'll see you next time. Enjoy.

```
    def DisplayEpisodeList(self,SeriesName,SeasonCount,EpisodeList):
        print "--------------------------------------"
        print "Series Name: %s" % SeriesName
        print "Total number of seasons: %s" % SeasonCount
        print "Total number of episodes: %d" % len(EpisodeList)
        print "--------------------------------------"
        for e in EpisodeList:
            print "Season: %s" % e['Season']
            print "   Season Episode Number: %s - Series Episode Number: %s" %
(e['SeasonEpisodeNumber'],e['EpisodeNumber'])
            print "   Title: %s" % e['Title']
            if e.has_key('Rating'):
                print "   Airdate: %s   Rating: %s" % (e['AirDate'],e['Rating'])
            else:
                print "   Airdate: %s   Rating: NONE" % e['AirDate']
            if e.has_key('Summary'):
                print "   Summary: \n%s" % e['Summary']
            else:
                print "   Summary: NA"
            print "============================"
        print "------------ End of episode list ------------"
```

```
--------------------------------------
Series Name: Continuum
Total number of seasons: 2
Total number of episodes: 10
--------------------------------------
Season: 1
   Season Episode Number: 01 - Series Episode Number: 1
   Title: A Stitch in Time
   Airdate: 2012-05-27   Rating: 8.8
   Summary:
Inspector Kiera Cameron loses everything she has and finds herself on a new mission when
she and eight dangerous terrorists are transported from their time in 2077 back to 2012
during the terrorist's attempt to escape execution. She takes on a new identity and
joins the VPD in order to stop the terrorists' reign of violence. Along the way, she
befriends Alec Sadler, the 17 year old who will one day grow up to create the technology
her world is built upon.
============================
def main():
    tr = TvRage()
    #--------------------
    # Find Series by name
    #--------------------
    nam = raw_input("Enter Series Name -> ")
    if nam != None:
        sl = tr.FindIdByName(nam)
        which = tr.DisplayShowResult(sl)
        if which == 0:
            sys.exit()
        else:
            option = int(which)-1
            id = sl[option]['ID']
            print "ShowID selected was %s" % id
    #------------------------------
    # Get Show Info
    #------------------------------
    showinfo = tr.GetShowInfo(id)
    #------------------------------
    # Display Show Info
    #------------------------------
    tr.DisplayShowInfo(showinfo)
    #-----------------------------
    # Get Episode List
    #-----------------------------
    SeriesName,TotalSeasons,episodelist = tr.GetEpisodeList(id)
    #-----------------------------
    # Display Episode List
    #-----------------------------
    tr.DisplayEpisodeList(SeriesName,TotalSeasons,episodelist)
    #-----------------------------
```

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.

no starch press
the finest in geek entertainment

Catalog

Catalog
- Art, Photography, Design
- Business
- For Kids
- General Computing
- Hardware and DIY
- LEGO®
- Linux, BSD, Unix
- Mac
- Manga
- Programming
- Science & Math
- Security
- System Administration

Free ebook edition with print book purchase from nostarch.com!

Shopping cart
View your shopping cart.

User login
- Log in
- Create account

Bestsellers

New!

Whether you're just getting started with GIMP or working to master GIMP's more complex features, you'll find the answers you're looking for in **The Book of GIMP**.

Learn You Some Erlang for Great Good! is a hilariously illustrated guide to the concurrent functional programming language.

Full of fun examples and color illustrations, **Python For Kids** is a playful introduction to Python that will help any beginner get started with programming.

Master Your Mac teaches the fearless user to harness the many powerful features that lie beneath OS X's glossy surface.

Whether you're brand new to LEGO or have been building for years, unleash your imagination with **The LEGO Adventure Book**. Learn to build robots, trains, medieval villages, and much more.

The Unofficial LEGO Technic Builder's Guide is filled with building techniques and tips for creating strong yet elegant machines and mechanisms.

Coming Soon (see all)

Blender Master Class is a practical, hands-on guide to the potential of the popular open-source 3D graphics tool. Chapters walk through the steps in the modeling process, from concept art to that final polish.

Absolute OpenBSD, 2nd Edition is a practical and straightforward guide for the experienced UNIX user who wants to add OpenBSD to his or her repertoire.

The Modern Web deftly guides you through the technologies web developers will need now and in the years to come.

Arduino Workshop takes you through 65 electronics projects that show the full range of cool stuff you can do with Arduino.

In **Realm of Racket**, you'll learn to wield Racket's mighty yet mind-bending power by reading comics and programming games.

The BrickGun Book offers step-by-step building instructions for five ultra-realistic LEGO® handgun models.

Written by Greg Walters

Programming In Python: Pt 42

Let's assume that you have decided to create a multimedia center for your family room. You have a dedicated computer for the wonderful program called XBMC. You've spent days ripping your DVD movies and TV series onto the computer. You have done the research and named the files the correct way. But let's say that one of your favorite shows is "NCIS," and you have every episode that you can get on DVD. You found a place that provides the current episodes as well. You want to find out what the next episode is and when it will be broadcast. Plus, you want to create a list of all the TV episodes that you have to impress your friends.

This is the project we will be starting this month. Our first task is to dig through the folder containing your TV shows, grabbing the series name, and each episode – including the name and season number, and the episode number. All this information will go into a database for easy storage.

According to XBMC, you should name each of your tv episode files like this:

```
Tv.Show.Name.SxxExx.Episode
name here if you
care.extension
```

So, let's use the very first episode of NCIS as an example. The filename for an AVI file would be:

```
NCIS.S01E01.Yankee White.avi
```

and the very latest episode would be:

```
NCIS.S10E17.Prime Suspect.avi
```

If you have a show name that has more than one word, it could look like this:

```
Doctor.Who.2005.S07E04.The
Power of Three.mp4
```

The directory structure should be as follows:

```
TVShows
    2 Broke Girls
       Season 1
          Episode 1
          Episode 2
             ...
       Season 2
             ...
     Doctor Who 2005
        Season 1
             ...
        Season 2
             ...
```

and so on. Now that we know what we will be looking for and where it will be, let's move on.

A very long time ago, we created a program to make a database of our MP3 files. That was back in issue #35 I believe, which was part number 9 of this series. We used a routine called WalkThePath to recursively dig through all the folders from a starting path, and pull out the filenames that had a ".mp3" extension. We will reuse most of that routine and modify it for our purposes. In this version, we will be looking for video files that have one of the following extensions:

```
.avi
.mkv
.m4v
.mp4
```

Which are very common extensions for video files in the media PC world.

Now we will get started with the first part of our project. Create a file called "tvfilesearch.py". Be sure to save it when we are done this month, because we will be building on it next month.

Let's start with our imports:

```
import os
from os.path import join,
getsize, exists
import sys
import apsw
import re
```

As you can see, we are importing the os, sys and apsw libraries. We've used them all before. We are also importing the re library to support Regular Expressions. We'll touch on that quickly this time, but more in the next article.

Now, let's do our last two routines next (next page). All our other code will go in between the imports and these last two routines.

This (next page, bottom right) is our main worker routine. In it, we

contents ^

create a connection to the SQLite database provided by apsw. Next we create a cursor to interact with it. Then we call the MakeDatabase routine which will create the database if it doesn't exist.

```
#==========================================
def main():
    global connection
    global cursor
    # Create the connection and cursor.
    connection = apsw.Connection("TvShows.db3")
    cursor = connection.cursor()
    MakeDataBase()
```

My TV files are located on two hard drives. So I created a list to hold the path names. If you have only one location, you can change the three lines to be as follows:

```
    startfolder =
"/filepath/folder/"

    WalkThePath(startfolder)
```

Next, we create our "standard" if __name__ routine.

```
#=============================
if __name__ == '__main__':
    main()
```

Now all the dull stuff is done, so we can move on the the meat and potatoes of our project. We'll start with the MakeDataBase routine (middle right). Put it right after the imports.

```
#==========================================
def MakeDataBase():
    # IF the table does not exist, this will create the table.
    # Otherwise, this will be ignored due to the 'IF NOT EXISTS' clause
    sql = 'CREATE TABLE IF NOT EXISTS TvShows (pkID INTEGER PRIMARY KEY, Series TEXT,
RootPath TEXT, Filename TEXT, Season TEXT, EPISODE TEXT);'
    cursor.execute(sql)
```

We discussed this routine before when we dealt with the MP3 scanner, so I'll just remind you that, in this routine, we check to see if the table exists, and if not, we create it.

Now we'll create the WalkThePath routine (right, second from bottom).

When we enter the routine (as we talked about way back when), we give the filepath that we are going to search through. We clear the showname variable, which we will use later, and open an error log file. Then we let the routine do its thing. We get back from the call (os.walk) a 3-tuple (directory path, directory names, filenames). The directory path is a string which is the path to the directory, directory names is a list of the names of subdirectories in the path, and the filenames is a list of non-directory names. We then parse through the list of filenames, checking to see if the filename ends with one of our target extensions.

```
#==========================================
def WalkThePath(filepath):
    showname = ""
    # Open the error log file
    efile = open('errors.log',"w")
    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'))]
:
```

```
#==========================================
# Set your video media paths
#==========================================
startfolder = ["/extramedia/tv_files/","/media/freeagnt/tv_files_2/"]
for cntr in range(0,2):
    WalkThePath(startfolder[cntr])
    # Close the cursor and the database
cursor.close()
connection.close()
print("Finished")
```

Now, we split the filename into the extension and the filename (without the extension). Next, we call the GetSeasonEpisode routine to pull out the Season/Episode information that is embedded in the filename, assuming it is correctly formatted.

```
OriginalFilename,ext =
os.path.splitext(file)

fl = file

isok,data =
GetSeasonEpisode(fl)
```

GetSeasonEpisode returns a boolean and a list (in this case "data") which holds the name of the series, the season, and the episode numbers. If a filename doesn't have the correct format, the "isok" boolean variable (top right) will be false.

Next (middle right), we will check to see if the file is in the database. If so, we don't want to duplicate it. We simply check for the filename. We could go deeper and make sure the path is the same as well, but for this time, this is enough.

If everything works as it should, the response from the query should only be a 1 or a 0. If it's a 0, then it's not there, and we will write the information to the database. If it is, we just move past. Notice the Try Except commands above and below. If something goes wrong, like some character that the database doesn't like, it will keep the program from aborting. We will, however, log the error so we can check it out later on.

```
                     if isok:
                 showname = data[0]
                 season = data[1]
                 episode = data[2]
                 print("Season {0} Episode {1}".format(season,episode))
            else:
                 print("No Season/EPisode")
                 efile.writelines('---------------------------\n')
                 efile.writelines('{0} has no series/episode information\n'.format(file))
                 efile.writelines('---------------------------\n\n')
```

```
            sqlquery = 'SELECT count(pkid) as rowcount from TvShows where Filename =
"%s";' % fl
            try:
                for x in cursor.execute(sqlquery):
                    rcntr = x[0]
                if rcntr == 0:  # It's not there, so add it
```

```
                    try:
                        sql = 'INSERT INTO TvShows (Series,RootPath,Filename,Season,Episode)
VALUES (?,?,?,?,?)'
                        cursor.execute(sql,(showname,root,fl,season,episode))
                    except:
                        print("Error")
                        efile.writelines('---------------------------\n')
                        efile.writelines('Error writing to database...\n')
                        efile.writelines('Filename = {0}\n'.format(file))
                        efile.writelines('---------------------------\n\n')
                except:
                    print("Error")
                print('Series - {0} File - {1}'.format(showname,file))
```

We are simply inserting a new record into the database or writing to the error file.

```
                     # Close the log
file
                 efile.close
          # End of WalkThePath
```

Now, let's look at the GetSeasonEpisode routine.

```
#=============================
=============
def
```

```
GetSeasonEpisode(filename):
    filename =
filename.upper()
    resp =
re.search(r'(.*).S\d\dE\d\d(\
.*)', filename, re.M|re.I)
```

The re.search portion of the code is part of the re library. It uses a pattern string, and, in this case, the filename that we want to parse. The re.M|re.I are parameters that say that we want to use a multiline type search (re.M) combined with an ignore-case (re.I). As I said earlier, we'll deal with the regular expressions more next month, since our routine will match only one type of series|episode string. As for the search pattern we are looking for: ".S", followed by two decimal numbers, followed by an uppercase "E", then two more numbers, then a period. If our filename looked like "tvshow.S01E03.avi", this would match. However, some people encode their shows like this "tvshow.s01e03.avi", or "tvshow.103.avi", which makes it harder to deal with. We'll modify this routine next month to cover the majority of the instances. The "r'" allows for a raw string to be used within the search.

Continuing on, the search returns a match object that we can look at. "resp" is a response that is empty if there is no match, and, in this case, two groups of returned information. The first one will give us the characters up to the match, and the second including the match. So, in the case above, group(1) would be "tvshow", and the second group would be "tvshow.S01E03.". This is specified by the parens in the search "(.*)" and "(\.*)".

```
    if resp:
        showname =
resp.group(1)
```

We take the show name from group number one. Then we get the length of that so we can grab the series and episode string with a substring command.

```
        shownamelength =
len(showname) + 1
        se =
filename[shownamelength:shown
amelength+6]
        season = se[1:3]
        episode = se[4:6]
```

Next, we replace any periods in the showname with a space – to be more "Human Readable".

```
        showname =
showname.replace("."," ")
```

We create a list to include the show name, season and episode, and return it along with the True boolean to say things went well.

```
        ret =
[showname,season,episode]
        return True,ret
```

Otherwise, if we didn't find a match, we create our list containing no show name and two "-1" numbers, and this gets returned with a boolean False.

```
    else:
        ret = ["",-1,-1]
        return False,ret
```

That's all the code. Now let's see what the output would look like. Assuming your file structures are exactly like mine, some of the output on the screen would look like this...

```
Season 02 Episode 04
SELECT count(pkid) as
rowcount from TvShows where
Filename =
"InSecurity.S02E04.avi";
Series - INSECURITY File -
InSecurity.S02E04.avi
Season 01 Episode 08
SELECT count(pkid) as
rowcount from TvShows where
Filename =
"Prime.Suspect.US.S01E08.Unde
rwater.avi";
Series - PRIME SUSPECT US
File -
Prime.Suspect.US.S01E08.Under
water.avi
```

and so on. You can shorten the output to keep the screen from driving you crazy if you would like. As we said earlier, each entry we find gets put to the database. Something like this:

```
pkID | Series | Root Path
      | Filename
      | Season | Episode
2526 | NCIS    |
/extramedia/tv_files/NCIS/Sea
son
7|NCIS.S07E04.Good.Cop.Bad.Co
p.avi | 7          | 4
```

As always, the full code listing is available on PasteBin.com at http://pastebin.com/txmmagkL

Next time, we will deal with more Season|Episode formats, and do some other things to flesh out our program.

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: Pt 43

Last time, we started a project that would eventually use the TvRage module that we created the month before that. Now we will continue the project. This time we will be adding functionality to our program: tweaking the filename parse routine and adding two fields (TvRageId and Status) to the database. So, let's jump right in.

First, we will make the changes to our import lines. For those who are just joining us, I'll include the ones from last time (shown top right).

The lines after 'import re' are the new ones for this time.

The next thing we will do is rewrite the GetSeasonEpisode routine. We are going to throw out pretty much everything we did last month, and make it more flexible across the possible season/episode schemes. In this iteration, we will be able to support the following schemes...

```
Series.S00E00

Series.s00e00

Series.S00E00.S00E01

Series.00x00

Series.S0000

Series.0x00
```

We will also fix any 'missing leading zero' issues before we write to the database.

Our first pattern tries to catch multi-episode files. There are various naming schemes, but the one we will support is similar to 'S01E03.S01E04'. We use the pattern string "(.*)\.s(\d{1,2})e(\d{1,2})\.s(\d{1,2})e(\d{1,2})". This returns (hopefully) five groups which consist of: the series name (S[1]), season(S[2]), episode number 1 (S[3]), season (S[4]), and episode number 2 (S[5]). Remember that the parens create each group for returns. In the case above, we group anything from the first character up to the ".s", then two numbers, skip the "e", then two numbers, and repeat. So the filename "Monk.S01E05.S01E06.avi" returns the following groups...

```
import os
from os.path import join, getsize, exists
import sys
import apsw
import re
#-----------------------------
#   NEW LINES START HERE
#-----------------------------
from xml.etree import ElementTree as ET
import urllib
import string
from TvRage import TvRage
```

```
S[1] = Monk

S[2] = 01

S[3] = 05

S[4] = 01

S[5] = 06
```

We are using only groups S[1], S[2] and S[3] in this code, but you can see where we are going with this. If we find a match, we set a variable named "GoOn" to true. This allows us to know what we should do after we've fallen through the various If lines.

So, next page (top right) is the code for the GetSeasonEpisode routine.

When we get to this point, (next page, bottom left) we prepare the show name by removing any periods in the show name, and then pull the season and episode information from the various groups, and return them. For the season information, if we have a pattern like "S00E00", the season number will have a leading zero. However if the pattern is like "xxx", then the season is assumed to be the first character, and the trailing two are the episode. In order to be forward thinking, we want to make the season a two-digit number with a leading zero if needed.

Next, in our MakeDatabase routine, we will change the create

contents ^

SQL statement to add the two new fields (next page, top).

Again, the only thing that has changed from last time is the last two field definitions.

In our WalkThePath routine, the only changes are the lines that actually insert into the database. This is to support the new structure. If you remember from last time, we pass the folder that holds our TV files to this routine. In my case, there are two folders, so it's set into a list and we use a for loop to pass each into the routine. As we go through the routine, we walk through each directory looking for files with extensions of .avi, .mkv, .mp4 and .m4v. When we find a file that matches, we send it to the GetSeasonEpisode routine. We then check to see if we already

```
def GetSeasonEpisode(filename):
    GoOn = False
    filename = filename.upper()
```

This is our first pattern check.

```
    #Should catch multi episode .S01E01.S01E02 type filenames
    resp = re.search(r'(.*)\.s(\d{1,2})e(\d{1,2})\.s(\d{1,2})e(\d{1,2})',filename, re.I)
    if resp:
        showname = resp.group(1)
        GoOn = True
    else:
```

Our second pattern check looks for SddEdd or sddedd...

```
    # Should catch SddEdd or sddedd
        resp = re.search(r'(.*).S(\d\d?)E(\d\d?)(\.*)', filename, re.I)
        if resp:
            showname = resp.group(1)
            GoOn = True
        else:
```

The next pattern looks for ddxdd.

```
       #check for ddxdd
       resp = re.search(r'(.*)\.(\d{1,2})x(\d{1,2})(.*)', filename, re.I)
       if resp:
           showname = resp.group(1)
           GoOn = True
       else:
```

This pattern checks for Sdddd.

```
                #check for Sdddd
                resp = re.search(r'(.*).S(\d\d)(.\d\d?)' , filename, re.I)
               if resp:
                   showname = resp.group(1)
                   GoOn = True
                else:
```

And finally we try for ddd

```
                    # Should catch xxx
                    resp = re.search(r'(.*)(\d)(.\d\d?)',filename,re.I)
                    if resp:
                        showname = resp.group(1)
                        GoOn = True
```

```
    if GoOn:
        shownamelength = len(showname) + 1
        showname = showname.replace("."," ")
        season = resp.group(2)
        if len(season) == 1:
            season = "0" + season
        episode = resp.group(3)
        ret = [showname,season,episode]
        return True,ret
    else:
        ret = ["",-1,-1]
        return False,ret
```

```
def MakeDataBase():
    # IF the table does not exist, this will create the table.
    # Otherwise, this will be ignored due to the 'IF NOT EXISTS' clause
    sql = 'CREATE TABLE IF NOT EXISTS TvShows (pkID INTEGER PRIMARY KEY, Series TEXT, RootPath TEXT, Filename TEXT,
Season TEXT, Episode TEXT, tvrageid TEXT,status TEXT);'
    cursor.execute(sql)
```

have it entered into the database, and, if not, we add it. I'm going to give you (top right) only part of the routine from last month.

The two lines in black are the ones that are new this time.

We are already over halfway done. Next are some support routines that work with our TvRage routine to fill in the database fields. Our first routine runs after the WalkThePath routine, and runs through the database, getting the series name and querying the TvRage server for the id number. Once we have that, we update the database, then use that id number to once again query TvRage to get the current status of the series. This status can be "New Series", "Returning Series", "Canceled", "Ended" and "On Haitus". The reason we want this information is that, when we go to check for new episodes, we don't want to bother with series that won't have any new episodes

```
            sqlquery = 'SELECT count(pkid) as rowcount from TvShows where Filename =
"%s";' % fl
            try:
                for x in cursor.execute(sqlquery):
                    rcntr = x[0]
                if rcntr == 0:  # It's not there, so add it
                    try:
                        sql = 'INSERT INTO TvShows
(Series,RootPath,Filename,Season,Episode,tvrageid) VALUES (?,?,?,?,?,?)'
                        cursor.execute(sql,(showname,root,fl,season,episode,-1))
                    except:
```

```
def WalkTheDatabase():
    tr = TvRage()
    SeriesCursor = connection.cursor()
    sqlstring = "SELECT DISTINCT series FROM TvShows WHERE tvrageid = -1"
```

because they are cancelled. So, now we have the status and can write that to the database (above).

We will pause here in our code and look at the SQL query we are using. It's a bit different from anything we've done before. The string is:

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

Which says, give me just one instance of the series name, no matter how many of them I have, where the field tvrageid is equal to "-1". If, for example, we have 103 episodes of Doctor Who 2005. By using the Distinct, I will get back only one record, assuming that we haven't gotten a TvRageID yet.

```
    for x in
SeriesCursor.execute(sqlstrin
g):

        seriesname = x[0]

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

We are using the capwords routine from the string library to change the series name (x[0]) to a "proper case" from the all-uppercase we currently store the show name in. We do this because TvRage expects something other that all-uppercase entries, and we won't get the results we are looking for. So the series name "THE MAN FROM UNCLE" will be converted to "The Man From

```
def UpdateDatabase(seriesname,id):
    idcursor = connection.cursor()
    sqlstring = 'UPDATE tvshows SET tvrageid = ' + id + ' WHERE series = "' + seriesname + '"'
    try:
        idcursor.execute(sqlstring)
    except:
        print "error"
```

```
def GetShowStatus(seriesname,id):
    tr = TvRage()
    idcursor = connection.cursor()
    dict = tr.GetShowInfo(id)
    status = dict['Status']
    sqlstring = 'UPDATE tvshows SET status = "' + status + '" WHERE series = "' + seriesname + '"'
    try:
        idcursor.execute(sqlstring)
    except:
        print "Error"
```

Uncle". We use that in the call to our TvRage Library FindIdByName. This gets the list of matching shows, and displays them for us to pick the best one. Once we pick one, we update the database with the id number and then call the GetShowStatus routine to get the current show status from TvRage (bottom right).

The UpdateDatabase routine (top) simply uses the series name as the key to update all the records with the proper TvRage ID.

GetShowStatus (above) is also very simple. We call the GetShowInfo routine from the TvRage library by passing the id that we just got to TvRage – to get the series information. If you remember, there is a lot of information provided about the series from TvRage, but all we are concerned about at this point is the show status. Since everything is returned in a dictionary, we just look for the ['Status'] key. Once we have it, we update the database with that and move on.

```
        print("Requesting information on " + searchname)
        sl = tr.FindIdByName(searchname)
        which = tr.DisplayShowResult(sl)
        if which == 0:
            print("Nothing found for %s" % seriesname)
        else:
            option = int(which)-1
            id = sl[option]['ID']
            UpdateDatabase(seriesname,id)
            GetShowStatus(seriesname,id)
```

We are almost done with our code. We finally add one line to our main routine from last month (in black, below) to call the "WalkTheDatabase" routine after we are done getting all our

```
    startfolder = ["/extramedia/tv_files","/media/freeagnt/tv_files_2"]
    #for cntr in range(0,2):
        #WalkThePath(startfolder[cntr])
    WalkTheDatabase()
    # Close the cursor and the database
    cursor.close()
    connection.close()
    print("Finished")
```

filenames. Again, I'm going to give you only part of the Main routine, just so you can find the correct place to put the new line.

That's all our code. Let's mentally go over what happens when we run the program.

First, we create the database if it doesn't exist.

Next, we walk through the predefined paths, looking for files that have any one of the following extensions:

```
.AVI, .MKV, .M4V, .MP4
```

When we find one, we go through and try to parse the filename looking for a series name, Season number, and episode number. We take that information and put it into a database, if it does not already exist there.

Once we are through looking for files, we query the database looking for series names that don't have a TvRage ID associated with them. We then will query the TvRage API and ask for matching files to gather that ID. Each series will go through that step once. The following screenshot shows the options for, in this case, the tv series Midsomer Murders.

```
Requesting information on Midsomer Murders
5 Found
------------------------
1 - Midsomer Murders - 4466
2 - Motives and Murders - 31373
3 - See No Evil: The Moors Murders - 11199
4 - The Atlanta Child Murders - 26402
5 - Motives & Murders: Cracking the Case - 33322
Enter Selection or 0 to exit ->
```

I entered (in this case) 1, which associates that series with the TvRage ID 4466. That's entered into the database, and we then use that ID to request the current status for the series, again from TvRage. In this case, we got back "Returning Series". This is then entered into the database and we move on.

While doing the initial "run" into the database, it will take a while and require your attention, because each and every series needs to ask about the ID number match. The good news is that this has to be done only once. If you are "somewhat normal", you won't have that many to deal with. I had 157 different series to do, so it took a little while. Since I was careful when I set up my filenames (checking TvRage and TheTvDB.com for the proper wording of the series name), the majority of the searches were the #1 option.

Just to let you know, over half of the TV series that I have either ended or have been canceled. That should tell you something about the age group I fall in.

The full code is, as always, available on PasteBin at http://pastebin.com/MeuGyKpX

Next time we will continue with the integration with TvRage. Until then have a great month!

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