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!
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
<?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>
```
HOWTO - PROGRAMMING PYTHON Pt39

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...

```xml
<!-- 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:

```xml
<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...

```xml
<prodnum/>
```

You also may see a tag that has no information for a specific tag, the tag itself just won’t be there. Your program will have to deal with these possibilities.

```xml
<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</showid> 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...

```xml
<showid>
<name>
<started>
</started>
</name>
</showid>
```

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

```xml
<seasons>
<started>
<start_date>
<origin_country>
<status>
<genres>
<runtime>
<network>
<airtime>
<airday>
<timezone>
</timezone>
</airtime>
</airday>
</airtime>
</airday>
</airtime>
</timezone>
</network>
</airtime>
</airday>
</airtime>
</airday>
</airtime>
</airday>
</airtime>
<runtime>
<network country="US">UPN</network>
</runtime>
</network>
<airtime>20:00</airtime>
</airdate>
<link>
<title>
</title>
</link>
</showinfo>
```

and from the episode list stream...

```xml
<Season>
<episode_number>
<season_number>
<production_number>
<airdate>
</airdate>
</production_number>
</season_number>
</episode_number>
</Season>
```

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

```xml
<akas>
<aka country="SE">Buffy &amp; vampyrerna</aka>
<aka country="DE">Buffy - Im Bann der Dämonen</aka>
<aka country="NO">Buffy - Vampyrenes skrek</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">Vampirubaninn 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.

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](http://www.thedesignatedgeek.net).
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.)

```python
def FindIdByName(self, showname, 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).

```python
def main()

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...

```python
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.
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.

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.

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

```python
class TvRage:
    def __init__(self): 
        self.ApiKey = "Itnl8IyY1hsR9n0IP6zI"
        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

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

```
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.
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

![Code Snippet]

```python
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 = {}  
    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['Country'] = 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
```

• 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).
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).

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.

```python
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 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.
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

```python
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)
```
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.

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.

```python
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
        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](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 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)
According to XBMC, you should name each of your tv episode files like this:

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:


The directory structure should be as follows:

TVShows
  2 Broke Girls
    Season 1
    Episode 1
    Episode 2
    ...  
  Season 2

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:

```python
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...
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.

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:

```python
startfolder = "/filepath/folder/"
WalkThePath(startfolder)
```

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

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

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

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.

```
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)
```

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

```
def WalkThePath(filepath):
    showname = ""
    # Open the error log file
    efile = open('errors.log',"w")
    for root, dirs, files in os.walk(filepath,topdown=True):
        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.
```

```python
    for file in [f for f in files if f.endswith(('avi','mkv','mp4','m4v'))]:
```

```
    # 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.

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

Now, let's look at the GetSeasonEpisode routine.
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 parameters 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.

    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:shownamelength+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.Underwater.avi";
    Series - PRIME SUSPECT US File -

    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/Season
    7|NCIS.S07E04.Good.Cop.Bad.Coop.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.

    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.
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.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...

```
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.

```
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
```

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
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:
        # Should catch SddEd or sddd
        resp = re.search(r'(.*).S\(\d\d\?)E\(\d\d\?)\(.*\)', filename, re.I)
        if resp:
            showname = resp.group(1)
            GoOn = True
        else:
            #check for ddd
            resp = re.search(r'(.*)(\d)(\d\d?)', filename, re.I)
            if resp:
                showname = resp.group(1)
                GoOn = True
            else:
                shownamelength = len(showname) + 1
                showname = showname.replace('.', ' ').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

    Our second pattern check looks for SddEd or sddd...
    # Should catch SddEd or sddd
    resp = re.search(r'(.*).S\(\d\d\?)E\(\d\d\?)\(.*\)', filename, re.I)
    if resp:
        showname = resp.group(1)
        GoOn = True
    else:
        #check for ddxdd
        resp = re.search(r'(.*\d\d\d)', filename, re.I)
        if resp:
            showname = resp.group(1)
            GoOn = True
        else:
            #check for Sdddd
            resp = re.search(r'(.*).S\(\d\d\\d)', filename, re.I)
            if resp:
                showname = resp.group(1)
                GoOn = True
            else:
                ret = ['-', -1, -1]
                return False, ret

    The next pattern looks for ddxdd.
    #check for ddxdd
    resp = re.search(r'(.*\d\d\d)', filename, re.I)
    if resp:
        showname = resp.group(1)
        GoOn = True
    else:
        #check for Sdddd
        resp = re.search(r'(.*).S\(\d\d\)', filename, re.I)
        if resp:
            showname = resp.group(1)
            GoOn = True
        else:
            ret = ['-', -1, -1]
            return False, ret

    This pattern checks for Sdddd.
    #check for Sdddd
    resp = re.search(r'(.*).S\(\d\d\\)', filename, re.I)
    if resp:
        showname = resp.group(1)
        GoOn = True
    else:
        ret = ['-', -1, -1]
        return False, ret

    And finally we try for ddd
    # Should catch xxx
    resp = re.search(r'(.*\d)', filename, re.I)
    if resp:
        showname = resp.group(1)
        GoOn = True
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 Hiatus". 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 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:

```sql
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.

```python
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)
```
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 fromTvRage, 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.

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

```python
startfolder = ["/extramedia/tv_files","/media/freeagent/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.

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](http://pastebin.com/MeuGyKpX)

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