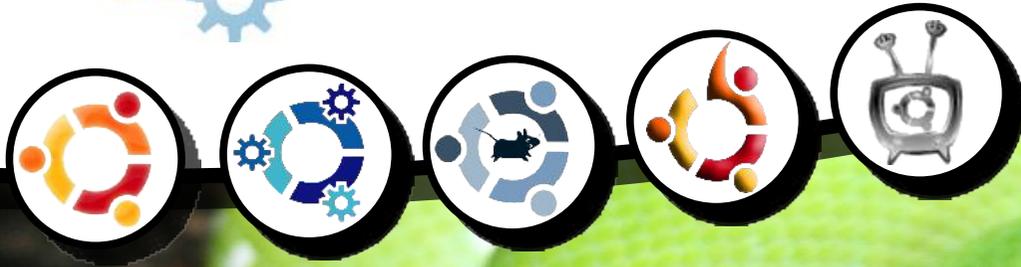




full circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU COMMUNITY
PROGRAMMING SERIES SPECIAL EDITION

PROGRAMMING SERIES
SPECIAL EDITION



PROGRAM IN PYTHON

Volume Two



About Full Circle

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

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

Please note: this Special Edition is provided with absolutely no warranty whatsoever; neither the contributors nor Full Circle Magazine accept any responsibility or liability for loss or damage resulting from readers choosing to apply this content to theirs or others computers and equipment.

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 9-16** from issues #35 through #42; nothing fancy, just the facts.

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)

Podcaster: Robin Catling
(aka RobinCatling)
podcast@fullcirclemagazine.org

Communications Manager:
Robert Clipsham
(aka: mrmonday) -
mrmonday@fullcirclemagazine.org



The articles contained in this magazine are released under the Creative Commons Attribution-Share Alike 3.0 Unported license. This means you can adapt, copy, distribute and transmit the articles but only under the following conditions: You must attribute the work to the original author in some way (at least a name, email or URL) and to this magazine by name ('full circle magazine') and the URL www.fullcirclemagazine.org (but not attribute the article(s) in any way that suggests that they endorse you or your use of the work). If you alter, transform, or build upon this work, you must distribute the resulting work under the same, similar or a compatible license.

Full Circle Magazine is entirely independent of Canonical, the sponsor of Ubuntu projects and the views and opinions in the magazine should in no way be assumed to have Canonical endorsement.



HOW-TO

Written by Greg Walters

Program In Python - Part 9

SEE ALSO:

FCM#27-34 - Python Parts 1 - 8

APPLICABLE TO:

ubuntu kubuntu xubuntu

CATEGORIES:



DEVICES:



If you are anything like me, you have some of your favorite music on your computer in the form of MP3 files. When you have less than 1000 music files, it's rather easy to remember what you have and where it is. I, on the other hand, have many more than that. In a past life, I was a DJ and converted most of my music a number of years ago. The biggest problem that I had was disk space. Now the biggest problem is

remembering what I have and where it is.

In this and the next installment we will look at making a catalog for our MP3 files. We will also take a look at some new python concepts as well as re-visiting our database skills.

First, an MP3 file can hold information about the file itself. The title of the song, the album, artist and more information. This information is held in ID3 tags and is referred to as metadata. Back in the early days, there was only a limited amount of information that could be held inside of the MP3 file.

Originally, it was stored at the very end of the file in a block of 128 bytes. Because of the small size of this block, you could only hold 30 characters for the title of the song, name of the artist, and so on. For many music files, this was fine, but (and this is one of my favorite songs ever) when you

had a song with the name "Clowns (The Demise of the European Circus with No Thanks to Fellini)", you only got the first 30 characters.

That was a BIG frustration for many people. So, the "standard" ID3 tag became known as ID3v1 and a new format was created called, amazingly enough, ID3v2.

This new format allowed for variable length information and was placed at the beginning of the file, while the old ID3v1 metadata was still stuck at the end of the file for the benefit of the older players. Now the metadata container could hold up to 256 MB of data. This was ideal for radio stations and crazies like me. Under ID3v2, each group of information is held in what's called a frame and each frame has a frame identifier. In an earlier version of ID3v2, the identifier was three characters long. The current version (ID3v2.4) uses a four character identifier.

In the early days, we would open the file in binary mode, and dig around getting the information as we needed it, but that was a lot of work, because there were no standard libraries available to handle it. Now we have a number of libraries that handle this for us. We will use one for our project called Mutagen.

You will want to go into Synaptic and install python-mutagen. If you want, you could do a search for "ID3" in Synaptic. You'll find there are over 90 packages (in Karmic), and if you type "Python" in the quick search box, you'll find 8 packages. There are pros and cons with any of them, but for our project, we'll stick with Mutagen. Feel free to dig into some of the other ones for your extended learning.

Now that you have Mutagen installed, we'll start our coding.

Start a new project and name it "mCat". We'll start by doing our imports.



```
from mutagen.mp3 import MP3

import os

from os.path import
join, getsize, exists

import sys

import apsw
```

For the most part, you've seen these before. Next, we want to create our stubbed function headers.

```
def MakeDataBase():
    pass
def S2HMS(t):
    pass
def WalkThePath(musicpath):
    pass
def error(message):
    pass
def main():
    pass
def usage():
    pass
```

Ahhh...something new. We now have a main function and a usage function. What are these for? Let's put one more thing in before we discuss them.

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

What the heck is that? This is a trick that allows our file to be used as either a stand alone application or a re-usable module that gets imported into another app. Basically it says "IF this file is the main app, we should go into the main routine to run, otherwise we are going to use this as a utility module and the functions will be called directly from another program.

Next, we'll flesh out the usage function. Below is the full code for the usage routine.

Here we are going to create a message to display to the user if they don't start our application with a parameter that we need to be able to run

```
def usage():
    message = (
        '=====\n'
        'mCat - Finds all *.mp3 files in a given folder (and sub-folders),\n'
        '\tread the id3 tags, and write that information to a SQLite database.\n\n'
        'Usage:\n'
        '\t{0} <foldername>\n'
        '\t WHERE <foldername> is the path to your MP3 files.\n\n'
        'Author: Greg Walters\n'
        'For Full Circle Magazine\n'
        '=====\n'
    ).format(sys.argv[0])
    error(message)
    sys.exit(1)
```

as a standalone app. Notice we use '\n' to force a new line and '\t' to force a tab. We also use a '%s' to include the application name which is held in the sys.argv[0]. We then use the error routine to output the message, then exit the application (sys.exit(1)).

Next, let's flesh out the error routine. Here is the full error routine.

```
def error(message):
    print >> sys.stderr,
    str(message)
```

We are using something called redirection here (the ">>"). When we use the function "print", we are telling

python we want to output, or stream, to the standard output device, usually the terminal that we are running in. To do this we use (invisibly) stdout.

When we want to send an error message, we use the stderr stream. This is also the terminal. So we redirect the print output to the stderr stream.

Now, let's work on the main routine. Here we will setup our connection and cursor for our database, then look at our system argument parameters, and if everything is good, we'll call our functions to do the actual work we want done. Here's the code:



```

def main():
    global connection
    global cursor
    #-----
    if len(sys.argv) != 2:
        usage()
    else:
        StartFolder = sys.argv[1]
        if not exists(StartFolder): # From os.path
            print('Path {0} does not seem to
exist...Exiting.').format(StartFolder)
            sys.exit(1)
        else:
            print('About to work {0}
folder(s):').format(StartFolder)
            # Create the connection and cursor.
            connection=apsw.Connection("mCat.db3")
            cursor=connection.cursor()
            # Make the database if it doesn't exist...
            MakeDataBase()
            # Do the actual work...
            WalkThePath(StartFolder)
            # Close the cursor and connection...
            cursor.close()
            connection.close()
            # Let us know we are finished...
            print("FINISHED!")

```

As we did last time, we create two global variables called connection and cursor for our database. Next we look at the parameters (if any) passed from the command line in the terminal. We do this with the sys.argv command. Here we are looking for two parameters, first the application name which is

automatic and secondly the path to our MP3 files. If we don't see two parameters, we jump to the usage routine, which prints our message to the screen and exits. If we do, we fall into the else clause of our IF statement. Next, we put the parameter for the starting path into the StartFolder variable. Understand that if

you have a path with a space in it, for example, (/mnt/musicmain/Adult Contemporary), the characters after the space will be seen as another parameter. So, whenever you use a path with a space, make sure you quote it. We then setup our connection and cursor, create the database, then do the actual hard work in the WalkThePath routine and finally close our cursor and connection to the database and then tell the user we are done. The full WalkThePath routine can be found at: <http://pastebin.com/CegsAXjW>.

First we clear the three counters we will be using to keep track of the work that has been done. Next we open a file to hold our error log just in case we have any problems.

Next we do a recursive walk down the path provided by the user. Basically, we start at the provided file path and “walk” in and out of any sub-folders that happen to be there, looking for any files that have a “.mp3” extension. Next we increment the folder counter then the file counter to keep track of how

many files we've dealt with.

Next we step through each of the files. We clear the local variables that hold the information about each song.

We use the join function from os.path to create a proper path and filename so we can tell mutagen where to find the file.

Now we pass the filename to the MP3 class getting back an instance of “audio”. Next we get all the ID3 tags this file contains and then step through that list checking for the tags we want to deal with and assigning them to our temporary variables. This way, we can keep errors to a minimum. Take a look at the portion of code dealing with the track number. When mutagen returns a track number it can be a single value, a value like “4/18” or as _trk[0] and _trk[1] or it can be absolutely nothing. We use the try/except wrappers to catch any errors that will occur due to this. Next, look at the writing of the data records. We are doing things a bit different from last time. Here we create the SQL statement like before, but this time we are replacing the value variables with “?”.

We then put in the values in the cursor.execute statement.

According to the ASPW web site, this is the better way to deal with it, so I won't argue with them. Finally we deal with any other types of errors we come up with. For the most part, these will be TypeErrors or ValueError and will probably occur because of Unicode characters that can't be handled. Take a quick look at the strange way we are formatting and outputting the string. We aren't using the '%' substitution character. We are using a "{0}" type substitution, which is part of the Python 3.x specification. The basic form is:

```
Print('String that will be
printed with {0} number of
statements').format(replaceme
nt values)
```

We are using the basic syntax for the efile.writelines as well.

Finally we should take a look at the S2HMS routine. This routine will take the length of the song which is a floating point value returned by

mutagen and convert it to a string using either "Hour:Minutes:Seconds" format or "Minutes:Seconds" format. Look at the return statements. Once again, we are using the Python 3.x formatting syntax. However, there's something new in the mix. We are using three substitution sets (0, 1 and 2), but what's the ":02n" after numbers 1 and 2? That says that we want leading zeros to two places. So if a song is 2 minutes and 4 seconds, the returned string would be "2:04", not "2:4".

The full code of our program is at:
<http://pastebin.com/rFf4Gm7E>.

Dig around on the web and see what you can find about Mutagen. It does more than just MP3s.



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.

MY STORY QUICKIE

My studio is fully digital with four Windows XP machines in a peer to peer network. My fifth machine runs Linux Ubuntu 9.04 exclusively as my test machine for Linux. I started with Ubuntu 7.04 and have upgraded each time there was a release. I have found it to be very stable, easy to use and configure as each version improves the OS.

At this time it is only my test bed but is linked to my network and shares data with my Windows machines. I have been very happy with the stability of Ubuntu in its upgrades, programs, hardware support, and driver updates. Although it is unfortunate that more major vendors such as Adobe don't port over, but Wine seems to work well. There are graphics programs and professional printers related to my camera equipment that do not work so I will have to wait until Wine gets better or the software gets ported over.

Audio, video, CD/DVD, USB, and Zip drives all seem to work 'out of the box' which is nice. Still some flaws in the software but they appear to be minor annoyances.

All in all Ubuntu has been visually refreshing and fun to play with. I am not a geek so I really do not use the command line unless curious about a tutorial and want to try it, the OS GUI is quite complete for us non-geeks who want to stick to a GUI.

I download Full Circle Magazine every month and have shared it with one of my colleagues to show him what is available. A lot of people still do not know about the OS and how easy it is to use, but as the Microsoft disgruntled get the word out I expect to see more growth. The one thing I absolutely love about this OS is the ability to shut down a misbehaving program. The break button works slickly in Linux and eliminates the frustration of waiting for Windows to unfreeze in XP. Why can't Windows do something as easy as that? I seldom need to use the button in Linux anyway which shows how stable Linux is.

Brian G Hartnell - *Photographer*





SEE ALSO:
FCM#27-35 - Python Parts 1 - 9

APPLICABLE TO:

CATEGORIES:

Dev Graphics Internet M/media System

DEVICES:

CD/DVD HDD USB Drive Laptop Wireless

You probably have heard of the term XML. You may not, however, know what it is. XML will be the focus of our lesson this month. The goal is:

- To familiarize you with what XML is.
- To show you how to read and write XML files in your own applications.
- Get you ready for a fairly large XML project next time.

So... let's talk about XML. XML stands for EXtensible Markup Language, very much like HTML. It was designed to provide a way to store and transport data efficiently over the Internet or other communication path. XML is basically a text file that is formatted using your own tags and should be fairly self-documenting. Being a text file, it can be compressed to allow for faster and easier transfer of the data. Unlike HTML, XML doesn't do anything by itself. It doesn't care how you want your data to look. As I said a moment before, XML doesn't require you to stick to a series of standard tags. You can create your own.

Let's take a look at a generic example of an XML file:

```
<root>
  <node1>Data Here</node1>
  <node2
attribute="something">Node 2
data</node2>
  <node3>
    <node3sub1>more
```

```
data</node3sub1>
  </node3>
</root>
```

The first thing to notice is the indentation. In reality, indentation is simply for human consumption. The XML file would work just as well if it looked like this...

```
<root><node1>Data
Here</node1><node2
attribute="something">Node 2
data</node2><node3><node3sub1
>more
data</node3sub1></node3></root>
```

Next, the tags contained in the "<>" brackets have some rules. First, they must be a single word. Next, when you have a start tag (for example <root>) you must have a matching closing tag. The closing tag starts with a "/". Tags are also case sensitive: <node>, <Node>, <NODE> and <NodeE> are all different tags, and the closing tag must match. Tag names may contain letters, numbers and other characters, but may not start

with a number or punctuation. You should avoid "-", ".", and ":" in your tag names since some software applications might consider them some sort of command or property of an object. Also, colons are reserved for something else. Tags are referred to as elements.

Every XML file is basically a tree - starting from a root and branching out from there. Every XML file MUST have a root element, which is the parent of everything else in the file. Look again at our example. After the root, there are three child elements: node1, node2 and node3. While they are children of the root element, node3 is also a parent of node3sub1.

Now take a look at node2. Notice that in addition to having its normal data inside the brackets, it also has something called an attribute. These days, many developers avoid attributes, since

elements are just as effective and less hassle, but you will find that attributes are still used. We'll look at them some more in a little bit.

Let's take a look at the useful example below.

Here we have the root element named "people", containing two child elements named "person". Each 'person' child has 6 child elements: firstname, lastname, gender, address, city and state. At first glance, you might think of this XML file as a database (remembering the last few lessons), and you would be

```
<people>
  <person>
    <firstname>Samantha</firstname>
    <lastname>Pharoh</lastname>
    <gender>Female</gender>
    <address>123 Main St.</address>
    <city>Denver</city>
    <state>Colorado</state>
  </person>
  <person>
    <firstname>Steve</firstname>
    <lastname>Levon</lastname>
    <gender>Male</gender>
    <address>332120 Arapahoe Blvd.</address>
    <city>Denver</city>
    <state>Colorado</state>
  </person>
</people>
```

correct. In fact, some applications use XML files as simple database structures. Now, writing an application to read this XML file could be done without too much trouble. Simply open the file, read each line and, based on the element, deal with the data as it's read and then close the file when you are done. However, there are better ways to do it.

In the following examples, we are going to use a library module called ElementTree. You can get it directly from Synaptic by installing python-elementtree. However, I chose to go to the ElementTree

website (<http://effbot.org/downloads/#elementtree>) and download the source file directly (elementtree-1.2.6-20050316.tar.gz). Once downloaded, I used the package manager to extract it to a temporary folder. I changed to that folder and did a "sudo python setup.py install". This placed the files into the python common folder so I could use it in either python 2.5 or 2.6. Now we can start to work. Create a folder to hold this month's code, copy the above XML data into your favorite text editor, and save it into that folder as "xmlsample1.xml".

Now for our code. The first thing we want to do is test our install of ElementTree. Here's the code:

```
import
elementtree.ElementTree as ET

tree =
ET.parse('xmlsample1.xml')

ET.dump(tree)
```

When we run the test program, we should get back something like what is shown below right.

All that we did was allow ElementTree to open the file, parse the file into its base

```
/usr/bin/python -u
"/home/greg/Documents/articles/xml/reader1.py"

<people>
  <person>
    <firstname>Samantha</firstname>
    <lastname>Pharoh</lastname>
    <gender>Female</gender>
    <address>123 Main St.</address>
    <city>Denver</city>
    <state>Colorado</state>
  </person>
  <person>
    <firstname>Steve</firstname>
    <lastname>Levon</lastname>
    <gender>Male</gender>
    <address>332120 Arapahoe
Blvd.</address>
    <city>Denver</city>
    <state>Colorado</state>
  </person>
</people>
```

parts, and dump it out as it is in memory. Nothing fancy here.

Now, replace your code with the following:

```
import
elementtree.ElementTree as ET

tree =
ET.parse('xmlsample1.xml')

person =
tree.findall(' ./person')

for p in person:
    for dat in p:
        print "Element: %s -
Data: %s" %(dat.tag,dat.text)
```

and run it again. Now your output should be:

```
/usr/bin/python -u
"/home/greg/Documents/articl
es/xml/reader1.py"
```

```
Element: firstname - Data:
Samantha
Element: lastname - Data:
Pharoh
Element: gender - Data:
Female
Element: address - Data: 123
Main St.
Element: city - Data: Denver
Element: state - Data:
Colorado
Element: firstname - Data:
Steve
Element: lastname - Data:
Levon
```

```
Element: gender - Data: Male
Element: address - Data:
332120 Arapahoe Blvd.
Element: city - Data: Denver
Element: state - Data:
Colorado
```

Now we have each piece of data along with the tag name. We can simply do some pretty printing to deal with what we have. Let's look at what we did here. We had ElementTree parse the file into an object named tree. We then asked ElementTree to find all instances of person. In the sample we are using, there are two, but it could be 1 or 1000. Person is a child of people and we know that people is simply the root. All of our data is

broken down within person. Next we created a simple for loop to walk through each person object. We then created another for loop to pull out the data for each person, and display it by showing the element name (.tag) and the data (.text).

Now for a more real-world example. My family and I enjoy an activity called Geocaching. If you don't know what that is, it's a "geeky" treasure hunt that uses a hand-held GPS device to find something someone else has hidden. They post the gross GPS coordinates on a web site, sometimes with clues, and we enter the

coordinates into our GPS and then try to go find it. According to Wikipedia, there are over 1,000,000 active cache sites world wide, so there are probably a few in your area. I use two websites to get the locations we search for. One is <http://www.geocaching.com/> and the other is <http://navicache.com/>. There are others, but these two are about the biggest.

Files that contain the information for each geocaching site are usually basic XML files. There are applications that will take those data and transfer them to the GPS device. Some of

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<loc version="1.0" src="NaviCache">
  <waypoint>
    <name id="N02CAC"><![CDATA[Take Goofy Pictures at Grapevine Lake by g_phillips
Open Cache: Unrestricted
Cache Type: Normal
Cache Size: Normal
Difficulty: 1.5
Terrain : 2.0]]></name>
    <coord lat="32.9890166666667" lon="-97.0728833333333" />
    <type>Geocache</type>
    <link text="Cache Details">http://www.navicache.com/cgi-
bin/db/displaycache2.pl?CacheID=11436</link>
  </waypoint>
</loc>
```

Navicache file

them act as database programs - that allow you to keep track of your activity, sometimes with maps. For now, we'll concentrate on just parsing the download files.

I went to Navicache and found a recent hide in Texas. The information from the file is shown on the previous page.

Copy the data from that box, and save it as "Cache.loc". Before we start coding, let's examine the cache file.

The first line basically tells us that this is a validated XML file, so we can safely ignore it. The next line (that starts with "loc") is our root, and has the attributes "version" and "src". Remember I said earlier that attributes are used in some files. We'll deal with more attributes in this file as we go on. Again, the root in this case can be ignored. The next line gives us our waypoint child. (A waypoint is a location where, in this case, the cache is to be found.) Now we get the important data that we want. There is the name of the cache, the coordinates in

latitude and longitude, the type of cache it is, and a link to the web page for more information about this cache. The name element is a long string that has a bunch of information that we can use, but we'll need to parse it ourselves. Now let's create a new application to read and display this file. Name it "readacache.py". Start with the import and parse statements from our previous example.

```
import
elementtree.ElementTree as ET
tree = ET.parse('Cache.loc')
```

Now we want to get back just the data within the waypoint tag. To do this, we use the .find function within ElementTree. This will be returned in the object "w".

```
w = tree.find('.//waypoint')
```

Next, we want to go through all the data. We'll use a for loop to do this. Within the loop, we will check the tag to find the elements 'name', 'coord', 'type' and 'link'. Based on which tag we get, we'll pull out the information to print it later on.

```
for w1 in w:
    if w1.tag == "name":
```

Since we will be looking at the 'name' tag first, let's review the data we will be getting back.

```
<name
id="N02CAC"><![CDATA[Take
Goofy Pictures at Grapevine
Lake by g_phillips
```

```
Open Cache: Unrestricted
```

```
Cache Type: Normal
```

```
Cache Size: Normal
```

```
Difficulty: 1.5
```

```
Terrain : 2.0]]></name>
```

This is one really long string. The 'id' of the cache is set as

```
# Get text of cache name up to the phrase "Open Cache: "
CacheName = w1.text[:w1.text.find("Open Cache: ")-1]
# Get the text between "Open Cache: " and "Cache Type: "
OpenCache = w1.text[w1.text.find("Open Cache: ")
+12:w1.text.find("Cache Type: ")-1]
# More of the same
CacheType = w1.text[w1.text.find("Cache Type: ")
+12:w1.text.find("Cache Size: ")-1]
CacheSize = w1.text[w1.text.find("Cache Size: ")
+12:w1.text.find("Difficulty: ")-1]
Difficulty= w1.text[w1.text.find("Difficulty: ")
+12:w1.text.find("Terrain : ")-1]
Terrain = w1.text[w1.text.find("Terrain :")+12:]
```

an attribute. The name is the part after "CDATA" and before the "Open Cache:" part. We will be chopping up the string into smaller portions that we want. We can get part of a string by using:

```
newstring =
oldstring[startposition:endpo
sition]
```

So, we can use the code below to grab the information we need.

Next we need to grab the id that's located in the attribute of the name tag. We check to see if there are any attributes (which we know there are), like this:

PROGRAM IN PYTHON - PART 10

```
if w1.keys():
    for name,value in
w1.items():
    if name == 'id':
        CacheID = value
```

Now, we can deal with the other tags for Coordinates, type, and link the code shown below right. Finally, we print them out to see them using the code at the bottom right. Far right is the full code.

You've learned enough now to read most XML files. As always, you can get the full code for this lesson on my website which is at: <http://www.thedesignatedgeek.com>.

Next time, we will utilize our XML knowledge to get information from a wonderful weather site and display it in a terminal. Have fun!



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.

```
elif w1.tag == "coord":
    if w1.keys():
        for name,value in w1.items():
            if name == "lat":
                Lat = value
            elif name == "lon":
                Lon = value
elif w1.tag == "type":
    GType = w1.text
elif w1.tag == "link":
    if w1.keys():
        for name, value in w1.items():
            Info = value
    Link = w1.text
```

```
print "Cache Name: ",CacheName
print "Cache ID: ",CacheID
print "Open Cache: ",OpenCache
print "Cache Type: ",CacheType
print "Cache Size: ",CacheSize
print "Difficulty: ", Difficulty
print "Terrain: ",Terrain
print "Lat: ",Lat
print "Lon: ",Lon
print "GType: ",GType
print "Link: ",Link
```

```
import elementtree.ElementTree as ET
tree = ET.parse('Cache.loc')
w = tree.find('./waypoint')
for w1 in w:
    if w1.tag == "name":
        # Get text of cache name up to the phrase "Open Cache:
        ""
        CacheName = w1.text[:w1.text.find("Open Cache: ")-1]
        # Get the text between "Open Cache: " and "Cache Type:
        ""
        OpenCache = w1.text[w1.text.find("Open Cache:
")+12:w1.text.find("Cache Type: ")-1]
        # More of the same
        CacheType = w1.text[w1.text.find("Cache Type:
")+12:w1.text.find("Cache Size: ")-1]
        CacheSize = w1.text[w1.text.find("Cache Size:
")+12:w1.text.find("Difficulty: ")-1]
        Difficulty= w1.text[w1.text.find("Difficulty:
")+12:w1.text.find("Terrain : ")-1]
        Terrain = w1.text[w1.text.find("Terrain :")+12:]
        if w1.keys():
            for name,value in w1.items():
                if name == 'id':
                    CacheID = value
    elif w1.tag == "coord":
        if w1.keys():
            for name,value in w1.items():
                if name == "lat":
                    Lat = value
                elif name == "lon":
                    Lon = value
    elif w1.tag == "type":
        GType = w1.text
    elif w1.tag == "link":
        if w1.keys():
            for name, value in w1.items():
                Info = value
        Link = w1.text
print "Cache Name: ",CacheName
print "Cache ID: ",CacheID
print "Open Cache: ",OpenCache
print "Cache Type: ",CacheType
print "Cache Size: ",CacheSize
print "Difficulty: ", Difficulty
print "Terrain: ",Terrain
print "Lat: ",Lat
print "Lon: ",Lon
print "GType: ",GType
print "Link: ",Link
print "="*25

print "finished"
```





HOW-TO

Written by Greg Walters

Program In Python - Part 1 1

SEE ALSO:

FCM#27-36 - Python Parts 1 - 10

APPLICABLE TO:

ubuntu kubuntu xubuntu

CATEGORIES:



DEVICES:



Last time, I promised you that we would use our XML expertise to grab weather information from a website and display it in a terminal. Well, that time has come.

We will use an API from www.wunderground.com. I hear the question "What's an API" rising in your throat. API stands for Application Programming Interface. It's really a fancy phrase for a way to interface

with another program. Think of the libraries we import. Some of those can be run as stand-alone applications, but if we import the application as a library, we can use many of its functions in our own program, and we get to use someone else's code. In this case, we will use specially formatted URL addresses to query the wunderground website for information about the weather - without using a web browser. Some people might say that an API is like a secret back door into another program - that the programmer(s) intentionally put there for our use. Either way, this is a supported extension of one application for its use in other applications.

Sounds intriguing? Well, read on, my dear padawan.

Fire up your favorite browser, and head to www.wunderground.com. Now enter your postal code or city and state (or country) into the

search box. There is a wealth of information here. Now, let's jump to the API web page: http://wiki.wunderground.com/index.php/API_-_XML

One of the first things you will notice is the API Terms of Service. Please read and follow them. They aren't onerous, and are really simple to abide by. The things that are going to be of interest to us are the *GeoLookupXML*, *WXCurrentObXML*, *AlertsXML* and *ForecastXML* calls. Take some time to scan over them.

I'm going to skip the *GeoLookupXML* routine, and let you look at that on your own. We will concentrate on two other commands: *WXCurrentObXML* (Current Conditions) this time, and *ForecastXML* (Forecast) next time.

Here's the link for *WXCurrentObXML*: <http://api.wunderground.com/uto/wui/geo/WXCurrentObXML/i>

[index.xml?query=80013](http://api.wunderground.com/uto/wui/geo/ForecastXML/index.xml?query=80013)

Replace the 80013 U.S. ZIP code with your postal code or if you are outside the U.S. you can try city, country - like Paris, France, or London, England.

And the link for the ForecastXML: <http://api.wunderground.com/uto/wui/geo/ForecastXML/index.xml?query=80013>

Again, replace the 80013 U.S. ZIP code with your postal code or city, country.

Let's start with the current information. Paste the address into your favorite browser. You'll see a great deal of information returned. I'll let you decide what's really important to you, but we'll look at a few of the elements.

For our example, we'll pay attention to the following tags:

display_location

```
observation_time
weather
temperature_string
relative_humidity
wind_string
pressure_string
```

Of course, you can add other tags that are of interest to you. However, these tags will provide enough of an example to take you as far as you would like to go.

Now that we know what we will be looking for, let's start coding our app. Let's look at the gross flow of the program.

First, we check what the user has asked us to do. If she passed a location, we will use that, otherwise we will use the default location we code into the main routine. We then pass that getCurrents routine. We use the location to build the request string to send out to the web. We use urllib.urlopen to get the response from the web, and put that in an object, and pass that object to ElementTree library function parse. We then close the connection to the web and start looking for our tags.

When we find a tag we are interested in, we save that text into a variable that we can use to output the data later on. Once we have all our data, we display it. Fairly simple in concept.

Start by naming your file w_currents.py. Here's the import portion of our code:

```
from xml.etree import
ElementTree as ET
```

```
import urllib
```

```
import sys
```

```
import getopt
```

Next, we'll put a series of help lines (above right) above the imports.

Be sure to use the triple double-quotes. This allows us to have a multi-line comment. We'll discuss this part more in a bit.

Now we'll create our class stubs, below right, and the main routines, which are shown on the following page.

```
""" w_currents.py
Returns current conditions, forecast and alerts for a
given zipcode from WeatherUnderground.com.
Usage: python wonderground.py [options]
Options:
-h, --help Show this help
-l, --location City,State to use
-z, --zip Zipcode to use as location
```

Examples:

```
w_currents.py -h (shows this help information)
w_currents.py -z 80013 (uses the zip code 80013 as
location)
"""
```

```
class CurrentInfo:
```

```
"""
```

```
This routine retrieves the current condition xml data
from WeatherUnderground.com
based off of the zip code or Airport Code...
currently tested only with Zip Code and Airport code
For location,
if zip code use something like 80013 (no quotes)
if airport use something like "KDEN" (use double-quotes)
if city/state (US) use something like "Aurora,%20CO" or
"Aurora,CO" (use double-quotes)
if city/country, use something like "London,%20England"
(use double-quotes)
"""
```

```
def getCurrents(self,debuglevel,Location):
pass
```

```
def output(self):
```

```
pass
```

```
def DoIt(self,Location):
```

```
pass
```

```
=====
# END OF CLASS CurrentInfo()
=====
```

You will remember from



previous articles the "if __name__" line. If we are calling this as a stand alone app, we will run the main routine - otherwise we can use this as part of a library. Once in the main routine, we then check what was passed into the routine, if anything.

If the user uses the "-h" or "--help" parameter, we print out the triple-commented help lines at the top of the program code. This is called by the usage routine telling the app to print __doc__.

If the user uses the "-l" (location) or "-z" (zipcode), that will override the internally set location value. When passing a location, be sure that you use double quotes to enclose the string and that you do not use spaces. For example, to get the current conditions for Dallas, Texas, use -l "Dallas,Texas".

Astute readers will realize that the -z and -l checks are pretty much the same. You can modify the -l to check for spaces and reformat the string before passing it to the routines. That's something you

can do by now.

Finally, we create an instance of our CurrentInfo class that we call currents, and then pass the location to the "DoIt" routine. Let's fill that in now:

```
def DoIt(self,Location):  
  
self.getCurrents(1,Location)  
  
self.output()
```

Very simple. We pass the location and debug level to the getCurrents routine, and then call the output routine. While we could have simply done the output directly from the getCurrents routine, we are developing the flexibility to output in various ways if we need to.

The code for the getCurrents routine is displayed on the next page.

Here we have a parameter called debuglevel. By doing this, we can print out helpful information if things don't seem to be going quite the way we want them to. It's also useful when we are doing our

```
def usage():  
    print __doc__  
    def main(argv):  
        location = 80013  
        try:  
            opts, args = getopt.getopt(argv, "hz:l:", ["help=",  
                "zip=", "location="])  
        except getopt.GetoptError:  
            usage()  
            sys.exit(2)  
        for opt, arg in opts:  
            if opt in ("-h", "--help"):  
                usage()  
                sys.exit()  
            elif opt in ("-l", "--location"):  
                location = arg  
            elif opt in ("-z", "--zip"):  
                location = arg  
        print "Location = %s" % location  
        currents = CurrentInfo()  
        currents.DoIt(location)  
  
        #=====  
        # Main loop  
        #=====  
        if __name__ == "__main__":  
            main(sys.argv[1:])
```

early code. If, when you are all happy with the way your code is working, you can remove anything related to debuglevel. If you are going to release this into the wild, like if you are doing this for someone else, be sure to remove the code and test it again before release.

Now, we use a try/except

wrapper to make sure that if something goes wrong, the app doesn't just blow up. Under the try side, we set up the URL, then set a timeout of eight seconds (urllib.socket.setdefaulttimeout(8)). We do this because, sometimes, wunderground is busy and doesn't respond. This

way we don't just sit there waiting for the web. If you want to get more information on urllib, a good place to start is <http://docs.python.org/library/urllib.html>.

If anything unexpected happens, we fall through to the except section, and print an error message, and then exit the application (sys.exit(2)).

Assuming everything works, we start looking for our tags. The first thing we do is find our location with the tree.findall("//full"). Remember, tree is the parsed object returned by elementtree. What is returned by the website API in part is shown below.

This is our first instance of the tag <full>, which in this

case is "Aurora, CO". That's what we want to use as our location. Next, we are looking for "observation_time". This is the time when the current conditions were recorded. We continue looking for all the data we are interested in - using the same methodology.

Finally we deal with our output routine which is shown top left on the following page.

Here we simply print out the variables.

That's all there is to it. A sample output from my zip code with debuglevel set to 1 is shown bottom left on the next page.

Please note that I chose to use the tags that included both

```
<display_location>
<full>Aurora, CO</full>
<city>Aurora</city>
<state>CO</state>
<state_name>Colorado</state_name>
<country>US</country>
<country_iso3166>US</country_iso3166>
<zip>80013</zip>
<latitude>39.65906525</latitude>
<longitude>-104.78105927</longitude>
<elevation>1706.00000000 ft</elevation>
</display_location>
```

```
def getCurrents(self, debuglevel, Location):
    if debuglevel > 0:
        print "Location = %s" % Location
    try:
        CurrentConditions =
        'http://api.wunderground.com/auto/wui/geo/WXCurrentObXML
        /index.xml?query=%s' % Location
        urllib.socket.setdefaulttimeout(8)
        usock = urllib.urlopen(CurrentConditions)
        tree = ET.parse(usock)
        usock.close()
    except:
        print 'ERROR - Current Conditions - Could not get
        information from server...'
        if debuglevel > 0:
            print Location
            sys.exit(2)
        # Get Display Location
        for loc in tree.findall("//full"):
            self.location = loc.text
        # Get Observation time
        for tim in tree.findall("//observation_time"):
            self.obtime = tim.text
        # Get Current conditions
        for weather in tree.findall("//weather"):
            self.we = weather.text
        # Get Temp
        for TempF in tree.findall("//temperature_string"):
            self.tmpB = TempF.text
        #Get Humidity
        for hum in tree.findall("//relative_humidity"):
            self.relhum = hum.text
        # Get Wind info
        for windstring in tree.findall("//wind_string"):
            self.winds = windstring.text
        # Get Barometric Pressure
        for pressure in tree.findall("//pressure_string"):
            self.baroB = pressure.text
```

getCurrents routine

```
def output(self):
print 'Weather Information From Wunderground.com'
print 'Weather info for %s ' % self.location
print self.obtime
print 'Current Weather - %s' % self.we
print 'Current Temp - %s' % self.tmpB
print 'Barometric Pressure - %s' % self.baroB
print 'Relative Humidity - %s' % self.relhum
print 'Winds %s' % self.winds
```

Fahrenheit and Celsius values. If you wish, for example, to display only Celsius values, you can use the <temp_c> tag rather than the <temperature_string> tag.

The full code can be downloaded from:
<http://pastebin.com/4ibJGm74>

Next time, we'll concentrate on the forecast portion of the API. In the meantime, have fun!

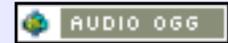
```
Location = 80013
Weather Information From Wunderground.com
Weather info for Aurora, Colorado
Last Updated on May 3, 11:55 AM MDT
Current Weather - Partly Cloudy
Current Temp - 57 F (14 C)
Barometric Pressure - 29.92 in (1013 mb)
Relative Humidity - 25%
Winds From the WNW at 10 MPH
Script terminated.
```



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.



Full Circle Podcast



The **Full Circle Podcast** is back and better than ever!

Topics in episode six include:

- News - Ubuntu 10.04 released
 - Opinions
 - Gaming - Steam coming to Linux?
 - Feedback
- ...and all the usual hilarity.

Your Hosts:

- *Robin Catling*
- *Ed Hewitt*
- *Dave Wilkins*

The podcast and show notes are at:
<http://fullcirclemagazine.org/>





HOW-TO

Written by Greg Walters

Program In Python - Part 12

SEE ALSO:

FCM#27-37 - Python Parts 1 - 11

APPLICABLE TO:

ubuntu kubuntu xubuntu

CATEGORIES:



DEVICES:



In our last session, we looked at the API from wunderground, and wrote some code to grab the current conditions. This time, we will be dealing with the forecast portion of the API. If you haven't had a chance to look at the last two installments about XML, and the last one specifically, you might want to review them before moving forward.

Just as there was a web address for the current conditions, there is one for the forecast. Here is the link to the forecast XML page:

<http://api.wunderground.com/aut/wui/geo/ForecastXML/index.xml?query=80013>

As before, you can change the '80013' to your City/Country, City/State, or postal code. You'll probably get back about 600 lines of XML code. You have a root element of 'forecast', and then four sub elements: 'termsofservice', 'txt_forecast', 'simpleforecast' and 'moon_phase'. We will concentrate on the 'txt_forecast' and 'simpleforecast' elements.

Since we went over the usage, main, and "if __name__" sections last time, I'll leave those to you to deal with and just concentrate on the goodies that we need for this time. Since I showed you a snippet of txt_forecast, let's start with that.

Shown below is a very small portion of the txt_forecast set for my area.

After the txt_forecast parent element, we get the date, a "number" element, then an element that has children of its own called forecastday which includes period, icon, icons, title and something called fcttext...then it repeats itself. The first thing you'll notice is that under txt_forecast, the date isn't a date, but a time value. It turns out that this is

when the forecast was released. The <number> tag shows how many forecasts there are for the next 24 hour period. I can't think of a time that I've seen this value less than 2. For each forecast for the 24 hour period (<forecastday>), you have a period number, multiple icon options, a title option ("Today", "Tonight", "Tomorrow"), and the text of a simple forecast. This is a quick overview of the forecast, usually for the next 12 hours.

```
<txt_forecast>
  <date>3:31 PM MDT</date>
  <number>2</number>
  -<forecastday>
    <period>1</period>
    <icon>nt_cloudy</icon>
    +<icons></icons>
    <title>Tonight</title>
    -<fcttext>
      Mostly cloudy with a 20
      percent chance of thunderstorms in the evening...then
      partly cloudy after midnight. Lows in the mid 40s.
      Southeast winds 10 to 15 mph shifting to the south after
      midnight.
    </fcttext>
  </forecastday>
  +<forecastday></forecastday>
</txt_forecast>
```



Before we start working with our code, we should take a look at the `<simpleforecast>` portion of the xml file which is shown right.

There is a `<forecastday>` tag for each day of the forecast period, usually 6 days including the current day. You have the date information in various formats (I personally like the `<pretty>` tag), projected high and low temps in both Fahrenheit and Celsius, gross condition projection, various icons, a sky icon (sky conditions at the reporting station), and “pop” which stands for “Probability Of Precipitation”. The `<moon_phase>` tag provides some interesting information including sunset, sunrise, and moon information.

Now we'll get into the code. Here is the import set:

```
from xml.etree import  
ElementTree as ET
```

```
import urllib
```

```
import sys
```

```
import getopt
```

Now we need to start our class. We will create an `__init__` routine to setup and clear the variables that we need, this is shown top right on the following page.

If you don't care about carrying the ability of both Fahrenheit and Celsius, then leave out whichever variable set you don't want. I decided to carry both.

Next, we'll start our main retrieval routine to get the forecast data. This is shown bottom right on the next page.

This is pretty much the same as the current conditions routine we worked on last time. The only major difference (so far) is the URL we are using. Now things change. Since we have multiple children that have the same tag within the parent, we have to make our parse calls a bit different. The code is top left on the following page.

Notice we are using `tree.find` this time, and we are using for loops to walk through the data. It's a shame that Python

```
<simpleforecast>  
  --<forecastday>  
    <period>1</period>  
    --<date>  
      <epoch>1275706825</epoch>  
      <pretty_short>9:00 PM MDT</pretty_short>  
      <pretty>9:00 PM MDT on June 04, 2010</pretty>  
      <day>4</day>  
      <month>6</month>  
      <year>2010</year>  
      <yday>154</yday>  
      <hour>21</hour>  
      <min>00</min>  
      <sec>25</sec>  
      <isdst>1</isdst>  
      <monthname>June</monthname>  
      <weekday_short/>  
      <weekday>Friday</weekday>  
      <amp;pm>PM</amp;pm>  
      <tz_short>MDT</tz_short>  
      <tz_long>America/Denver</tz_long>  
    </date>  
    --<high>  
      <fahrenheit>92</fahrenheit>  
      <celsius>33</celsius>  
    </high>  
    --<low>  
      <fahrenheit>58</fahrenheit>  
      <celsius>14</celsius>  
    </low>  
    <conditions>Partly Cloudy</conditions>  
    <icon>partlycloudy</icon>  
    +<icons>  
      <skyicon>partlycloudy</skyicon>  
      <pop>10</pop>  
    </forecastday>  
    ...  
</simpleforecast>
```

```

=====
# Get the forecast for today and (if available)
tonight
=====
fcst = tree.find('.//txt_forecast')
for f in fcst:
    if f.tag == 'number':
        self.periods = f.text
    elif f.tag == 'date':
        self.date = f.text
    for subelement in f:
        if subelement.tag == 'period':
            self.period=int(subelement.text)
        if subelement.tag == 'fcttext':
            self.forecastText.append(subelement.text)
        elif subelement.tag == 'icon':
            self.icon.append( subelement.text)
        elif subelement.tag == 'title':
            self.Title.append(subelement.text)

```

```

class ForecastInfo:
    def __init__(self):
        self.forecastText = [] # Today/tonight forecast
information
        self.Title = [] # Today/tonight
        self.date = ''
        self.icon = [] # Icon to use for conditions
today/tonight
        self.periods = 0
        self.period = 0
        =====
        # Extended forecast information
        =====
        self.extIcon = [] # Icon to use for extended
forecast
        self.extDay = [] # Day text for this forecast
("Monday", "Tuesday" etc)
        self.extHigh = [] # High Temp. (F)
        self.extHighC = [] # High Temp. (C)
        self.extLow = [] # Low Temp. (F)
        self.extLowC = [] # Low Temp. (C)
        self.extConditions = [] # Conditions text
        self.extPeriod = [] # Numerical Period
information (counter)
        self.extpop = [] # Percent chance Of
Precipitation

```

```

def GetForecastData(self,location):
    try:
        forecastdata = 'http://api.wunderground.com/auto/wui/geo/ForecastXML/index.xml?query=%s' % location
        urllib.socket.setdefaulttimeout(8)
        usock = urllib.urlopen(forecastdata)
        tree = ET.parse(usock)
        usock.close()
    except:
        print 'ERROR - Forecast - Could not get information from server...'
        sys.exit(2)

```

doesn't offer a SELECT/CASE command set like other languages. The IF/ELIF routine, however, works well, just a bit clunkier. Now we'll break down the code. We assign the variable `fcst` to everything within the `<txt_forecast>` tag. This gets all the data for that group. We then look for the tags `<date>` and `<number>` - since those are simple "first level" tags - and load that data into our variables. Now things get a bit more difficult. Look back at our xml response example. There are two instances of `<forecastday>`. Under `<forecastday>` are sub-elements that consist of `<period>`, `<icon>`, `<icons>`, `<title>` and `<fcttext>`. We'll loop through these, and again use the IF statement to load them into our variables.

Next we need to look at the extended forecast data for the next X number of days. We are basically using the same methodology to fill our variables; this is shown top right.

Now we need to create our output routine. As we did last

time, it will be fairly generic. The code for this is shown on the right of the following page.

Again, if you don't want to carry both Centigrade and Fahrenheit information, then modify the code to show what you want. Finally, we have a "Dolt" routine:

```
def
DoIt(self, Location, US, Include
Today, Output):

    self.GetForecastData(Loca
tion)

    self.output(US, IncludeTod
ay, Output)

forecast = ForecastInfo()
forecast.DoIt('80013', 1, 0, 0)
# Insert your own postal code
```

That's about it for this time. I'll leave the alert data to you, if you want to go through that.

Here is the complete running code:
<http://pastebin.com/wsSXMxOx>

Have fun until next time.

```
=====
# Now get the extended forecast
=====
fcst = tree.find('./simpleforecast')
for f in fcst:
    for subelement in f:
        if subelement.tag == 'period':
            self.extPeriod.append(subelement.text)
        elif subelement.tag == 'conditions':
            self.extConditions.append(subelement.text)
        elif subelement.tag == 'icon':
            self.extIcon.append(subelement.text)
        elif subelement.tag == 'pop':
            self.extpop.append(subelement.text)
        elif subelement.tag == 'date':
            for child in subelement.getchildren():
                if child.tag == 'weekday':
                    self.extDay.append(child.text)
        elif subelement.tag == 'high':
            for child in subelement.getchildren():
                if child.tag == 'fahrenheit':
                    self.extHigh.append(child.text)
                if child.tag == 'celsius':
                    self.extHighC.append(child.text)
        elif subelement.tag == 'low':
            for child in subelement.getchildren():
                if child.tag == 'fahrenheit':
                    self.extLow.append(child.text)
                if child.tag == 'celsius':
                    self.extLowC.append(child.text)
```



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.



```

def output(self,US,IncludeToday,Output):
    # US takes 0,1 or 2
    # 0 = Centigrade
    # 1 = Fahrenheit
    # 2 = both (if available)
    # Now print it all
    if Output == 0:
        for c in range(int(self.period)):
            if c <> 1:
                print '-----'
                print 'Forecast for %s' %
self.Title[c].lower()
                print 'Forecast = %s' %
self.forecastText[c]
                print 'ICON=%s' % self.icon[c]
                print '-----'
            print 'Extended Forecast...'
            if IncludeToday == 1:
                startRange = 0
            else:
                startRange = 1
            for c in range(startRange,6):
                print self.extDay[c]
                if US == 0: #Centigrade information
                    print '\tHigh - %s(C)' %
self.extHigh[c]
                    print '\tLow - %s(C)' % self.extLow[c]
                elif US == 1: #Fahrenheit information
                    print '\tHigh - %s(F)' %
self.extHigh[c]
                    print '\tLow - %s(F)' % self.extLow[c]
                else: #US == 2 both(if available)
                    print '\tHigh - %s' % self.extHigh[c]
                    print '\tLow - %s' % self.extLow[c]
                if int(self.extpop[c]) == 0:
                    print '\tConditions - %s.' %
self.extConditions[c]
                else:
                    print '\tConditions - %s. %d%% chance
of precipitation.' %
(self.extConditions[c],int(self.extpop[c]))

```



Full Circle Podcast



The **Full Circle Podcast** is back and better than ever!

Topics in episode eight include:

- News - Maverick development
- Ubuntu interview
- Gaming - Ed reviews Osmos
- Feedback

...and all the usual hilarity.

Your Hosts:

- *Robin Catling*
- *Ed Hewitt*
- *Dave Wilkins*

The podcast and show notes are at:

<http://fullcirclemagazine.org/>



This month, we talk about using Curses in Python. No, we're not talking about using Python to say dirty words, although you can if you really feel the need. We are talking about using the Curses library to do some fancy screen output.

If you are old enough to remember the early days of computers, you will remember that, in business, computers were all mainframes - with dumb terminals (screens and keyboards) for input and output. You could have many terminals connected to one computer. The problem was that the terminals were very dumb devices. They had neither windows, colors, or much of anything - just 24 lines of 80 characters (at best). When personal computers became popular, in the old days of DOS and CPM, that is what you had as well. When programmers worked on fancy screens (those days),

especially for data input and display, they used graph paper to design the screen. Each block on the graph paper was one character position. When we deal with our Python programs that run in a terminal, we still deal with a 24x80 screen. However, that limitation can be easily dealt with by proper forethought and preparation. So, go out to your local office supply store and get yourself a few pads of graph paper.

Anyway, let's jump right in and create our first Curses program, shown above right. I'll explain after you've had a look at the code.

Short but simple. Let's examine it line by line. First, we do our imports, which you are very familiar with by now. Next, we create a new Curses screen object, initialize it, and call the object `myscreen`. (`myscreen = curses.initscr()`). This is our canvas that we will paint to. Next, we use the

```
#!/usr/bin/env python
# CursesExample1
#-----
# Curses Programming Sample 1
#-----
import curses
myscreen = curses.initscr()
myscreen.border(0)
myscreen.addstr(12, 25, "See Curses, See Curses Run!")
myscreen.refresh()
myscreen.getch()
curses.endwin()
```

`myscreen.border(0)` command to draw a border around our canvas. This isn't needed, but it makes the screen look nicer. We then use the `addstr` method to "write" some text on our canvas starting on line 12 position 25. Think of the `.addstr` method of a Curses print statement. Finally, the `.refresh()` method makes our work visible. If we don't refresh the screen, our changes won't be seen. Then we wait for the user to press any key (`.getch`) and then we release the screen object (`.endwin`) to allow our terminal to act normally. The `curses.endwin()` command is VERY important, and, if it

doesn't get called, your terminal will be left in a major mess. So, make sure that you get this method called before your application ends.

Save this program as `CursesExample1.py` and run it in a terminal. Some things to note. Whenever you use a border, it takes up one of our "usable" character positions for each character in the border. In addition, both the line and character position count is ZERO based. This means that the first line in our screen is line 0 and the last line is line 23. So, the very top left



position is referred to 0,0 and the bottom right position is 23,79. Let's make a quick example (above right) to show this.

Very simple stuff except the try/finally blocks. Remember, I said that curses.endwin is VERY important and needs to be called before your application finishes. Well, this way, even if things go very badly, the endwin routine will get called. There's many ways of doing this, but this way seems pretty simple to me.

Now let's create a nice menu system. If you remember back a while, we did a cookbook application that had a menu (Programming Python - Part 8). Everything in the

terminal simply scrolled up when we printed something. This time we'll take that idea and make a dummy menu that you can use to pretty up the cookbook. Shown below is what we used back then.

This time, we'll use Curses. Start with the following template. You might want to save this snippet (below right) so you can use it for your own future programs.

Now, save your template again as "cursessmenu1.py" so that we can work on the file and keep the template.

```
=====
                        RECIPE DATABASE
=====
1 - Show All Recipes
2 - Search for a recipe
3 - Show a Recipe
4 - Delete a recipe
5 - Add a recipe
6 - Print a recipe
0 - Exit
=====
Enter a selection ->
```

```
#!/usr/bin/env python
# CursesExample2
import curses
=====
#                               MAIN LOOP
=====
try:
    myscreen = curses.initscr()
    myscreen.clear()
    myscreen.addstr(0,0,"0           1           2           3
                    4           5           6           7")
    myscreen.addstr(1,0,"1234567890123456789012345678901234567890123456
789012345678901234567890123456789012345678901234567890")
    myscreen.addstr(10,0,"10")
    myscreen.addstr(20,0,"20")
    myscreen.addstr(23,0, "23 - Press Any Key to Continue")
    myscreen.refresh()
    myscreen.getch()
finally:
    curses.endwin()
```

```
#!/usr/bin/env python
#-----
# Curses Programming Template
#-----
import curses

def InitScreen(Border):
    if Border == 1:
        myscreen.border(0)

#-----
#                               MAIN LOOP
=====
myscreen = curses.initscr()
InitScreen(1)
try:
    myscreen.refresh()
    # Your Code Stuff Here...
    myscreen.addstr(1,1, "Press Any Key to Continue")
    myscreen.getch()
finally:
    curses.endwin()
```

Before we go any further with our code, we are going to do this in a modular way. Here (above right) is a pseudo-code example of what we are going to do.

Of course this pseudo code is just that...pseudo. But it gives you an idea of where we are going with this whole thing. Since this is just an example, we'll only go just so far here, but you can take it all the way if you want. Let's start with the main loop (middle far right).

Not much in the way of programming here. We have our try|finally blocks just as we had in our template. We initialize the Curses screen and then call a routine named LogicLoop. That code is shown bottom far right.

Again, not much, but this is only a sample. Here we are going to call two routines. One called DoMainMenu and the other MainInKey. DoMainMenu will show our main menu, and the MainInKey routine handles everything for that main menu. The DoMainMenu routine is shown right.

```

curses.initscr
LogicLoop
    ShowMainMenu                # Show the main menu
    MainInKey                   # This is our main input handling routine
        While Key != 0:
            If Key == 1:
                ShowAllRecipesMenu # Show the All Recipes Menu
                Inkey1              # Do the input routines for this
                ShowMainMenu        # Show the main menu
            If Key == 2:
                SearchForARecipeMenu # Show the Search for a Recipe Menu
                InKey2              # Do the input routines for this option
                ShowMainMenu        # Show the main menu again
            If Key == 3:
                ShowARecipeMenu     # Show the Show a recipe menu routine
                InKey3              # Do the input routine for this routine
                ShowMainMenu        # Show the main menu again
            ...                    # And so on and so on
curses.endwin()                # Restore the terminal
    
```

```

def DoMainMenu():
    myscreen.erase()
    myscreen.addstr(1,1,
"=====
myscreen.addstr(2,1, "           Recipe
Database")
    myscreen.addstr(3,1,
"=====
myscreen.addstr(4,1, "  1 - Show All
Recipes")
    myscreen.addstr(5,1, "  2 - Search for a
recipe")
    myscreen.addstr(6,1, "  3 - Show a recipe")
    myscreen.addstr(7,1, "  4 - Delete a recipe")
    myscreen.addstr(8,1, "  5 - Add a recipe")
    myscreen.addstr(9,1, "  6 - Print a recipe")
    myscreen.addstr(10,1, "  0 - Exit")
    myscreen.addstr(11,1,
"=====
myscreen.addstr(12,1, "  Enter a selection: ")
myscreen.refresh()
    
```

```

#    MAIN LOOP
try:
    myscreen = curses.initscr()
    LogicLoop()
finally:
    curses.endwin()
    
```

```

def LogicLoop():
    DoMainMenu()
    MainInKey()
    
```

Notice that this routine does nothing but clear the screen (`mymScreen.erase()`), and then print what we want on the screen. There is nothing here dealing with keyboard handling. That's the job of the `MainInKey` routine, which is shown below.

This is really a simple routine. We jump into a while loop until the key that is

entered by the user equals 0.

Within the loop, we check to see if it's equal to various values, and, if so, we do a series of routines, and finally call the main menu when we are done. You can fill in most of these routines for yourself by now, but we will look at option 2, Search for a Recipe. The menu is short and sweet. The `InKey2` routine (right) is a bit more complicated.

```
def MainInKey():
    key = 'X'
    while key != ord('0'):
        key = myScreen.getch(12,22)
        myScreen.addch(12,22,key)
        if key == ord('1'):
            ShowAllRecipesMenu()
            DoMainMenu()
        elif key == ord('2'):
            SearchForARecipeMenu()
            InKey2()
            DoMainMenu()
        elif key == ord('3'):
            ShowARecipeMenu()
            DoMainMenu()
        elif key == ord('4'):
            NotReady("'Delete A Recipe'")
            DoMainMenu()
        elif key == ord('5'):
            NotReady("'Add A Recipe'")
            DoMainMenu()
        elif key == ord('6'):
            NotReady("'Print A Recipe'")
            DoMainMenu()
    myScreen.refresh()
```

```
def SearchForARecipeMenu():
    myScreen.addstr(4,1, "-----")
    myScreen.addstr(5,1, " Search in")
    myScreen.addstr(6,1, "-----")
    myScreen.addstr(7,1, " 1 - Recipe Name")
    myScreen.addstr(8,1, " 2 - Recipe Source")
    myScreen.addstr(9,1, " 3 - Ingredients")
    myScreen.addstr(10,1, " 0 - Exit")
    myScreen.addstr(11,1, "Enter Search Type -> ")
    myScreen.refresh()

def InKey2():
    key = 'X'
    doloop = 1
    while doloop == 1:
        key = myScreen.getch(11,22)
        myScreen.addch(11,22,key)
        tmpstr = "Enter text to search in "
        if key == ord('1'):
            sstr = "'Recipe Name' for -> "
            tmpstr = tmpstr + sstr
            retstring = GetSearchLine(13,1,tmpstr)
            break
        elif key == ord('2'):
            sstr = "'Recipe Source' for -> "
            tmpstr = tmpstr + sstr
            retstring = GetSearchLine(13,1,tmpstr)
            break
        elif key == ord('3'):
            sstr = "'Ingredients' for -> "
            tmpstr = tmpstr + sstr
            retstring = GetSearchLine(13,1,tmpstr)
            break
        else:
            retstring = ""
            break
    if retstring != "":
        myScreen.addstr(15,1, "You entered - " + retstring)
    else:
        myScreen.addstr(15,1, "You entered a blank string")
    myScreen.refresh()
    myScreen.addstr(20,1, "Press a key")
    myScreen.getch()

def GetSearchLine(row,col,strng):
    myScreen.addstr(row,col,strng)
    myScreen.refresh()
    instrng = myScreen.getstr(row,len(strng)+1)
    myScreen.addstr(row,len(strng)+1,instrng)
    myScreen.refresh()
    return instrng
```

Again, we are using a standard while loop here. We set the variable `doloop = 1`, so that our loop is endless until we get what we want. We use the `break` command to drop out of the while loop. The three options are very similar. The major difference is that we start with a variable named `tmpstr`, and then append whatever option text has been selected...making it a bit more friendly. We then call a routine called `GetSearchLine` to get the string to search for. We use the `getstr` routine to get a string from the user rather than a character. We then return that string back to our input routine for further processing.

The full code is at:
<http://pastebin.com/ELuZ3T4P>

One final thing. If you are interested in looking into Curses programming further, there are many other methods available than what we used this month. Besides doing a Google search, your best starting point is the official docs page at

<http://docs.python.org/library/curses.html>.

See you next time.

OOPS!

It seems that the code for **Python Pt.11** isn't properly indented on Pastebin. The correct URL for Python Pt.11 code is:

<http://pastebin.com/Pk74fLF3>

Please check:

<http://fullcirclemagazine.pastebin.com> for all Python (and future) code.



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.



Full Circle Podcast



The **Full Circle Podcast** is back and better than ever!

Topics in episode ten include:

- News
 - Opinion - Contributing articles with the FCM Editor.
 - Interview - with Amber Graner
 - Feedback
- ...and all the usual hilarity.

Your Hosts:

- *Robin Catling*
- *Ed Hewitt*
- *Ronnie Tucker*

The podcast and show notes are at:
<http://fullcirclemagazine.org/>





Last time we talked about the Curses library. This time we are going to delve further into the curses library, and concentrate on the color commands. Just in case you missed the last article, let's have a quick review. First, you have to import the curses library. Next you have to call `curses.initscr` to get things started. To put text on the screen you call the `addstr` function, and then call `refresh` to show your changes to the screen. Finally, you have to call `curses.endwin` to restore the terminal window to its normal state.

Now, we are going to create a quick and easy program that uses color. It's pretty much the same as what we did before, but we have a few new commands this time. First we use `curses.start_color` to tell the system that we want to use color in our program. Next, we assign a color pair of foreground and background. We can assign many pairs, and use them whenever we want. We do that by using the `curses.init_pair` function. The syntax is:

```
curses.init_pair([pairnumber]
,[foreground
color],[background color])
```

The colors are set up by using "curses.COLOR_" and the color you want. For example, `curses.COLOR_BLUE` or `curses.COLOR_GREEN`. The options here are black, red, green, yellow, blue, magenta, cyan and white. Just add "curses.COLOR_", and the color you want, in upper case. Once we have set up our color pair, we can use it as a final parameter in our `screen.addstr` function like this:

```
myscreen.addstr([row],[column]
,[text],curses.color_pair(X)
)
```

Here X is the color set we wish to use.

Save the following code (above right) as `colortest1.py`, then run it. Don't try to run a curses program in an IDE like SPE or Dr. Python. Run it from a terminal.

What you should see is a grey

```
import curses
try:
    myscreen = curses.initscr()
    curses.start_color()
    curses.init_pair(1, curses.COLOR_BLACK,
curses.COLOR_GREEN)
    curses.init_pair(2, curses.COLOR_BLUE,
curses.COLOR_WHITE)
    curses.init_pair(3,
curses.COLOR_MAGENTA,curses.COLOR_BLACK)
    myscreen.clear()
    myscreen.addstr(3,1," This is a test
",curses.color_pair(1))
    myscreen.addstr(4,1," This is a test
",curses.color_pair(2))
    myscreen.addstr(5,1," This is a test
",curses.color_pair(3))
    myscreen.refresh()
    myscreen.getch()
finally:
    curses.endwin()
```

background, with three lines of text saying " This is a test " in different colors. The first should be black-on-green, the second blue-on-white, and the third magenta on the grey background.

Remember the Try/Finally set. This makes sure that if anything happens, our program will automatically restore our terminal to its normal state. There is another way. There is a curses

command called `wrapper`. `Wrapper` does all the work for you. It does the `curses.initscr`(), the `curses.start_color`(), and the `curses.endwin`(), so that you don't have to. The one thing you have to remember is that you call `curses.wrapper` with your main routine. It passes back your screen pointer. On the following page (top right) is the same program as before, but this time using the

curses.wrapper function.

That's a whole lot easier, and we don't have to worry about calling `curses.endwin()` if something bad happens. All the work is done for us.

Now that we have a bunch of basics, let's put some of the things we've learned over the past year to work, and start making a game. Before we start however, let's lay out what we are going to do. Our game will pick a random uppercase letter, and move it from the right side of the screen to the left side. At a random position, it will drop down to the bottom of the screen. We'll have a "gun" that can be moved using the right and left arrow keys to be positioned below the falling letter. Then, by pressing the space bar, we will shoot it. If we shoot the letter before it gets to our gun, we get a point. If not, our gun explodes. If we lose three guns, the game is over. While on the surface this seems like a simple game, there's a lot of code to it.

Let's get started. We need to do our setup, and create a few routines before we go very far. Create a new project and call it

`game1.py`. Start with the code shown below right:

This code won't do much right now, but it's our starting point. Notice that we have four `init_pair` statements setting the colors that we will use for our random color sets, and one for the explosions (number 5). Now we need to set up some variables and constants that will be used during our game. We will put them in the `__init__` routine of class `Game1`. Replace the pass statement in `__init__` with the code on the following page.

You should be able to figure out what is happening in these definitions. If you are unsure at this precise moment, it should become clearer as we fill in the code.

We are getting closer to having something that will run. We still need to make a few more routines before it will do much. Let's work on the routine that will move a letter from right to left on the screen:

<http://fullcirclemagazine.pastebin.com/z5CgMAGm>

This is our longest routine in the program, and there are some

```
import curses
def main(stdscreen):
    curses.init_pair(1, curses.COLOR_BLACK,
curses.COLOR_GREEN)
    curses.init_pair(2, curses.COLOR_BLUE,
curses.COLOR_WHITE)
    curses.init_pair(3,
curses.COLOR_MAGENTA,curses.COLOR_BLACK)
    stdscreen.clear()
    stdscreen.addstr(3,1," This is a test
",curses.color_pair(1))
    stdscreen.addstr(4,1," This is a test
",curses.color_pair(2))
    stdscreen.addstr(5,1," This is a test
",curses.color_pair(3))
    stdscreen.refresh()
    stdscreen.getch()
curses.wrapper(main)
```

```
import curses
import random

class Game1():
    def __init__(self):
        pass
    def main(self, stdscr):
        curses.init_pair(1, curses.COLOR_BLACK,
curses.COLOR_GREEN)
        curses.init_pair(2, curses.COLOR_BLUE,
curses.COLOR_BLACK)
        curses.init_pair(3, curses.COLOR_YELLOW,
curses.COLOR_BLUE)
        curses.init_pair(4, curses.COLOR_GREEN,
curses.COLOR_BLUE)
        curses.init_pair(5, curses.COLOR_BLACK,
curses.COLOR_RED)

        def StartUp(self):
            curses.wrapper(self.main)

g = Game1()
g.StartUp()
```

new functions in this routine. The `scrn.delch` function deletes the character at the given row | column. The `curses.napms()` tells python to sleep (nap) for X number of milliseconds (ms).

So the logic in this routine is as follows (in pseudocode) on the next page (top right).

You should be able to follow the code by now. We need two new routines to keep everything correct. The first is `Explode`, which we will stub with the `pass` directive. The second is `ResetForNew`. This is where we will reset the current row for the letter to the default letterline, reset the current column, set the `DroppingLetter` flag to 0, pick a random letter, and pick a random drop point. Following page, middle right, are those two routines.

Now we need four more routines to keep up with things (next page, bottom right). One picks a random letter, the other picks a random drop point. Remember we quickly discussed the random module early on in the series.

In `PickALetter`, we generate a

random integer between 65 and 90 ("A" to "Z"). Remember when we use the random integer function we must give a range of minimum-number to maximum-number. The same thing goes for `PickDropPoint`. We also make a call to `random.seed()` in both routines, which sets up the random

generator with a different number every time it's called. The fourth routine is called `CheckKeys`. This routine will look at any keystrokes entered by the user, and deal with them to move our gun. However, we'll stub it out for the moment but we will need it later. We'll also need a routine called `CheckForHit`,

which we will also stub for the time being.

```
def
CheckKeys(self, scrn, keyin):
    pass
def CheckForHit(self, scrn):
    pass
```

We are going to create a small

```
# Line Specific Stuff
self.GunLine = 22                #Row where our gun lives
self.GunPosition = 39           #Where the gun starts on GunLine
self.LetterLine = 2            #Where our letter runs right to left
self.ScoreLine = 1             #Where we are going to display the score
self.ScorePosition = 50        #Where the score column is
self.LivesPosition = 65        #Where the lives column is

# Letter Specific Stuff
self.CurrentLetter = "A"        #A dummy Holder Variable
self.CurrentLetterPosition = 78 #Where the letter will start on the LetterLine
self.DropPosition = 10         #A dummy Holder Variable
self.DroppingLetter = 0        #Flag - Is the letter dropping?
self.CurrentLetterLine = 3     #A dummy Holder Variable
self.LetterWaitCount = 15      #How many times should we loop before actually
                                working?

# Bullet Specific Stuff
self.Shooting = 0              #Flag - Is the gun shooting?
self.BulletRow = self.GunLine - 1
self.BulletColumn = self.GunPosition

# Other Stuff
self.LoopCount = 0             #How many loops have we done in MoveLetter
self.GameScore = 0            #Current Game Score
self.Lives = 3                 #Default number of lives
self.CurrentColor = 1         #A dummy Holder Variable
self.DecScoreOnMiss = 0       #Set to 1 if you want to decrement the
                                #score every time the letter hits the
                                #bottom row
```

routine which will be the “brains” of our game. We'll call it GameLoop (next page, top right).

The logic behind this is to first set our keyboard to nodelay(1). This means that we won't wait for a keystroke to happen, and when it does, we just cache it for latter processing. Then we enter a while loop which we force to always be true (1) so that the game continues until we are ready for it to end. We nap for 40 milliseconds, move our letter and then check to see if the user has pressed a key. If it's a “Q” (notice it's upper case), or the ESC key, then we break out of our loop and end the program. Otherwise, we check to see if it's a left or right arrow key, or the space bar. Later on, you can make the game a bit more difficult by checking the keystroke against the current character and only fire the gun if the user has pressed the same key, ala a simple typing tutor. Just remember to remove the “Q” as a quit key.

We'll also need to create a routine that sets up for each new play of our game. Let's call it NewGame (next page, middle right).

```
IF we have waited the correct number of loops THEN
  Reset the loop counter
  IF we are moving to the left of the screen THEN
    Delete the character at the the current row,column.
    Sleep for 50 milliseconds
    IF the current column is greater than 2 THEN
      Decrement the current column
    Set the character at the current row,column
    IF the current column is at the random column to drop to the bottom THEN
      Set the DroppingLetter flag to 1
  ELSE
    Delete the character at the current row,column
    Sleep for 50 milliseconds
    IF the current row is less than the line the gun is on THEN
      Increment the current row
      Set the character at the current row,column
    ELSE
      IF
        Explode (which includes decrementing the score if you wish) and check to
        see if we continue.
        Pick a new letter and position and start everything over again.
  ELSE
    Increment the loopcounter
    Refresh the screen.
```

We also need the PrintScore routine that will show the current score and the number of lives that are left (next page, bottom right).

Now we only need to add some code (next page, bottom left) to our main routine to start our game loop. The additional code is below. Add it under the last init_pair call.

Now we should have a program that does something. Give it a try. I'll wait.

```
def Explode(self,scrn):
    pass
def ResetForNew(self):
    self.CurrentLetterLine = self.LetterLine
    self.CurrentLetterPosition = 78
    self.DroppingLetter = 0
    self.PickALetter()
    self.PickDropPoint()
```

```
def PickALetter(self):
    random.seed()
    char = random.randint(65,90)
    self.CurrentLetter = chr(char)
```

```
def PickDropPoint(self):
    random.seed()
    self.DropPosition = random.randint(3,78)
```

PROGRAM IN PYTHON - PART 14

Now we have a program that picks a random uppercase letter, moves it from the right side of the screen to the left a random number of columns, then moves that letter down to the bottom(ish) of the screen. However, the first thing you should notice is that every time you run the program the first letter is always "A", and the drop point is always column 10. That's because we set defaults in the `__init__` routine. To fix this, simply call `self.ResetForNew` before you enter the while loop in the Main routine.

At this point, we need to work on our "gun" and supporting routines. Add the code (next page, top right) to the Game1 class.

Movegun will take the current gun position and move it in whichever direction we want it to go. The only thing that is new in this routine is at the end of the `addch` routine. We are calling the `colorpair (2)` to set the color, and, at the same time, we are forcing the gun to have the bold attribute. We are using a bitwise OR ("`|`") to force the attribute on. Next we need to flesh out our `CheckKeys` routine. Replace the pass

```
stdscr.addstr(11,28,"Welcome to Letter Attack")
stdscr.addstr(13,28,"Press a key to begin...")
stdscr.getch()
stdscr.clear()
PlayLoop = 1
while PlayLoop == 1:
    self.NewGame(stdscr)
    self.GameLoop(stdscr)
    stdscr.nodelay(0)
    curses.flushinp()
    stdscr.addstr(12,35,"Game Over")
    stdscr.addstr(14,23,"Do you want to play
again? (Y/N)")
    keyin = stdscr.getch(14,56)
    if keyin == ord("N") or keyin == ord("n"):
        break
    else:
        stdscr.clear()
```

```
def GameLoop(self,scrn):
    test = 1          #Set the loop
    while test == 1:
        curses.napms(20)
        self.MoveLetter(scrn)
        keyin =
scrn.getch(self.ScoreLine,self.ScorePosition)
        if keyin == ord('Q') or keyin == 27: # 'Q'
or <Esc>
            break
        else:
            self.CheckKeys(scrn,keyin)
            self.PrintScore(scrn)
            if self.Lives == 0:
                break
    curses.flushinp()
    scrn.clear()
```

```
def NewGame(self,scrn):
    self.GunChar = curses.ACS_SSBS
    scrn.addch(self.GunLine,self.GunPosition,self.Gun
Char,curses.color_pair(2) | curses.A_BOLD)
    scrn.nodelay(1) #Don't wait for a
keystroke...just cache it.
    self.ResetForNew()
    self.GameScore = 0
    self.Lives = 3
    self.PrintScore(scrn)
    scrn.move(self.ScoreLine,self.ScorePosition)
```

```
def PrintScore(self,scrn):
    scrn.addstr(self.ScoreLine,self.ScorePosition,"S
CORE: %d" % self.GameScore)
    scrn.addstr(self.ScoreLine,self.LivesPosition,"L
IVES: %d" % self.Lives)
```

PROGRAM IN PYTHON - PART 14

statement with the new code (next page, bottom right).

Now we need to make a routine that will move our bullet "up" the screen (below left).

We need a few more routines (next page, top right) before we are finished. Here's the code to fill out the CheckForHit routine and the code to ExplodeBullet.

Finally we need to flesh out our Explode routine. Replace pass with the following code (next page, bottom).

Now we have a working program. You can tweak the value in LetterWaitCount to speed up or slow down the movement of the letter going across the screen to make it easier or harder. You can also use the variable CurrentColor to create a random color choice and set the letter color to one of the 4 color sets we have made and change the color assignment to the random color. I wanted to give you a challenge.

I hope you had fun this time, and will add some additional code to make the game more playable. As always, the full code is available

at www.thedesignedgeek.com,
or at:
<http://fullcirclemagazine.pastebin.com/DeReeh8m>.

```
def MoveGun(self, scrn, direction):
    scrn.addch(self.GunLine, self.GunPosition, " ")
    if direction == 0: # left
        if self.GunPosition > 0:
            self.GunPosition -= 1
    elif direction == 1: # right
        if self.GunPosition < 79:
            self.GunPosition += 1
    scrn.addch(self.GunLine, self.GunPosition, self.GunChar, curses.color_pair(2) | curses.A_BOLD)
```

```
if keyin == 260: # left arrow - NOT on keypad
    self.MoveGun(scrn, 0)
    curses.flushinp() #Flush out the input buffer for safety.
elif keyin == 261: # right arrow - NOT on keypad
    self.MoveGun(scrn, 1)
    curses.flushinp() #Flush out the input buffer for safety.
elif keyin == 52: # left arrow ON keypad
    self.MoveGun(scrn, 0)
    curses.flushinp() #Flush out the input buffer for safety.
elif keyin == 54: # right arrow ON keypad
    self.MoveGun(scrn, 1)
    curses.flushinp() #Flush out the input buffer for safety.
elif keyin == 32: #space
    if self.Shooting == 0:
        self.Shooting = 1
        self.BulletColumn = self.GunPosition
        scrn.addch(self.BulletRow, self.BulletColumn, "|")
        curses.flushinp() #Flush out the input buffer for safety.
```



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.

```
def MoveBullet(self, scrn):
    scrn.addch(self.BulletRow, self.BulletColumn, " ")
    if self.BulletRow > self.LetterLine:
        self.CheckForHit(scrn)
        self.BulletRow -= 1
        scrn.addch(self.BulletRow, self.BulletColumn,
            "|")
    else:
        self.CheckForHit(scrn)
        scrn.addch(self.BulletRow, self.BulletColumn,
            " ")

        self.BulletRow = self.GunLine - 1
        self.Shooting = 0
```

```

def CheckForHit(self,scrn):
    if self.Shooting == 1:
        if self.BulletRow == self.CurrentLetterLine:
            if self.BulletColumn == self.CurrentLetterPosition:
                scrn.addch(self.BulletRow,self.BulletColumn," ")

                self.ExplodeBullet(scrn)
                self.GameScore +=1
                self.ResetForNew()

def ExplodeBullet(self,scrn):
    scrn.addch(self.BulletRow,self.BulletColumn,"X",curses.color_pair(5))
    scrn.refresh()
    curses.napms(200)
    scrn.addch(self.BulletRow,self.BulletColumn,"|",curses.color_pair(5))
    scrn.refresh()
    curses.napms(200)
    scrn.addch(self.BulletRow,self.BulletColumn,"-",curses.color_pair(5))
    scrn.refresh()
    curses.napms(200)
    scrn.addch(self.BulletRow,self.BulletColumn,".",curses.color_pair(5))
    scrn.refresh()
    curses.napms(200)
    scrn.addch(self.BulletRow,self.BulletColumn," ",curses.color_pair(5))
    scrn.refresh()
    curses.napms(200)

```

```

scrn.addch(self.CurrentLetterLine,self.CurrentLetterPosition,"X",curses.color_pair(5))
curses.napms(100)
scrn.refresh()
scrn.addch(self.CurrentLetterLine,self.CurrentLetterPosition,"|",curses.color_pair(5))
curses.napms(100)
scrn.refresh()
scrn.addch(self.CurrentLetterLine,self.CurrentLetterPosition,"-",curses.color_pair(5))
curses.napms(100)
scrn.refresh()
scrn.addch(self.CurrentLetterLine,self.CurrentLetterPosition,".",curses.color_pair(5))
curses.napms(100)
scrn.refresh()
scrn.addch(self.CurrentLetterLine,self.CurrentLetterPosition," ")
scrn.addch(self.GunLine,self.GunPosition,self.GunChar,curses.color_pair(2) | curses.A_BOLD)
scrn.refresh()

```



HOW-TO

Written by Greg Walters

Program In Python - Part 15

This month we are going to explore **Pygame**, a set of modules designed for writing games. The

website is <http://www.pygame.org/>. To quote from the Pygame read-me: "Pygame is a cross-platform library designed to make it easy to write multimedia software, such as games, in Python. Pygame requires the Python language and SDL multimedia library. It can also make use of several other popular libraries."

You can install Pygame through Synaptic as 'python-pygame'. Do this now so we can move forward.

First, we import Pygame (see above right). Next, we set the `os.environ` to make our window centered in our screen. Next, we initialize Pygame, then set the Pygame window to 800x600 pixels, and set the caption. Finally, we display the screen, and go into a loop waiting for a keystroke or mouse-button-down event. The screen is an object that will contain anything we decide to put

on it. It's called a surface. Think of it as a piece of paper that we will draw things onto.

Not very exciting, but it's a start. Let's make it a bit less boring. We can change the background color to something less dark. I found a program called "colorname" that you can install via the Ubuntu Software Center. This allows you to use the "color wheel" to pick a color you like, and it will give you the RGB or Red, Green, Blue values of that color. We must use RGB colors if we don't want to use the predefined colors that Pygame gives us. It's a neat utility that you should consider installing.

Right after the import statements, add...

```
Background = 208, 202, 104
```

This will set the variable `Background` to a tanish color. Next, after the `pygame.display.set_caption` line, add the following lines...

```
#This is the Import
import pygame
from pygame.locals import *
import os
# This will make our game window centered in the screen
os.environ['SDL_VIDEO_CENTERED'] = '1'
# Initialize pygame
pygame.init()
#setup the screen
screen = pygame.display.set_mode((800, 600))
# Set the caption (title bar of the window)
pygame.display.set_caption('Pygame Test #1')
# display the screen and wait for an event
doloop = 1
while doloop:
    if pygame.event.wait().type in (KEYDOWN,
    MOUSEBUTTONDOWN):
        break
```

```
screen.fill(Background)
pygame.display.update()
```

The `screen.fill()` method will set the color to whatever we pass it. The next line, `pygame.display.update()`, actually updates the changes to our screen.

Save this off as `pygame1.py`, and we'll move on.

Now we will display some text in our bland looking window. Again, let's start with our import statements and the background

variable assignment from our last program.

```
import pygame
from pygame.locals import *
import os
Background = 208, 202, 104
```

Now, add an additional variable assignment for the foreground color of our font.

```
FontForeground = 255,255,255
# White
```

Then, we will add in the majority of the code from our last

program (shown right).

If you run this now, nothing has changed visually since all we did is add the foreground definition. Now, after the `screen.fill()` line, and before the loop portion of our code, enter the following lines:

```
font =  
pygame.font.Font(None, 27)  
text = font.render('Here is  
some text', True,  
FontForeground, Background)  
textrect = text.get_rect()  
screen.blit(text, textrect)  
pygame.display.update()
```

Go ahead, save the program as `pygame2.py`, and run the program. On the top left of our window, you should see the text "Here is some text".

Let's break down the new commands. First, we call the `Font` method and pass it two arguments. The first is the name of the font we wish to use, and the second is the font size. Right now, we'll just use 'None', and let the system pick a generic font for us, and set the font size to 27 points.

Next we have the `font.render()` method. This has four arguments. In order, they are the text we wish

to display, whether we want to use anti-aliasing (True in this case), the foreground color of the font, and, finally, the background color of the font.

The next line (`text.get_rect()`) assigns a rectangle object that we will use to put the text on the screen. This is an important thing, since almost everything else we will deal with is rectangles. (You'll understand more in a bit.) Then we blit the rectangle onto the screen. And, finally, we update the screen to show our text. What is blit, and why the heck should I want to do something that sounds so weird? Well, the term goes WAY back to the 1970s, and came from Xerox PARC (which is where we owe so much of today's technology). The term was originally called BitBLT which stands for Bit (or Bitmap) Block Transfer. That changed to Blit (possibly because it's shorter). Basically we are plopping our image or text on to the screen.

What if we want the text to be centered in the screen instead of on the top line where it takes a bit of time to see? In between the `text.get_rect()` line and the `screen.blit` line, put the following

```
# This will make our game window centered in the screen  
os.environ['SDL_VIDEO_CENTERED'] = '1'  
# Initialize pygame  
pygame.init()  
# Setup the screen  
screen = pygame.display.set_mode((800, 600))  
# Set the caption (title bar of the window)  
pygame.display.set_caption('Pygame Test #1')  
screen.fill(Background)  
pygame.display.update()  
  
# Our Loop  
doloop = 1  
while doloop:  
    if pygame.event.wait().type in (KEYDOWN,  
    MOUSEBUTTONDOWN):  
        break
```

two lines:

```
textRect.centerx =  
screen.get_rect().centerx  
textRect.centery =  
screen.get_rect().centery
```

Here we are getting the center of the screen object (remember surface) in x and y pixel positions, and setting our `textRect` object x and y center points to those values.

Run the program. Now our text is centered within our surface. You can also modify the text by using (in our sample code) `font.set_bold(True)` and/or `font.set_italic(True)` right after the `pygame.font.Font` line.

Remember we discussed very briefly the 'None' option when we set the font to a generic font. Let's say you want to use a fancier font. As I stated before, the `pygame.font.Font()` method takes two arguments. The first is the path and file name of the font we want to use, and the second is the font size. The problem is multi-fold at this point. How do we know what the actual path and filename of the font we want to use is on any given system? Thankfully, Pygame has a function that takes care of that for us. It's called `match_font`. Here's a quick program that will print the path and filename of (in this case) the Courier New font.

```
import pygame
from pygame.locals import *
import os
print
pygame.font.match_font('Courier New')
```

On my system, the returned value is “/usr/share/fonts/truetype/msttcorefonts/cour.ttf”. If, however, the font is not found, the return value is “None”. Assuming that the font IS found, then we can assign the returned value to a variable, and we can then use the following assignment.

```
courier =
pygame.font.match_font('Courier New')
font =
pygame.font.Font(courier,27)
```

Change your last version of the program to include these two lines and try it again. The bottom line is, either use a font that you KNOW will be available on the end user's machine, or include it when you distribute your program and hard code the font path and name. There are other ways around this, but I'll leave that to you to figure out so we can move on.

While text is nice, graphics are

better. I found a really nice tutorial for Pygame written by Peyton McCollugh, and thought I'd take and modify it. For this part, we need to start with a picture that will move around our surface. This picture is known as a sprite. Use GIMP or some other tool and create a stick figure. Nothing fancy, just a generic stick figure. I'll assume that you are using GIMP. Start a new image, set the size to 50 pixels in both height and width, and, under advance options, set the 'Fill With' option to Transparency. Use the pencil tool with a brush of Circle (03). Draw your little figure, and save it as stick.png into the same folder you have been using for the code this time. Here is what mine looks like. I'm sure you can do better.



I know...I'm not an artist. However, for our purposes, that will do. We saved it as a .png file, and set the background to be transparent, so that just the little black lines of our stick figure show up - and not a white or other color background will show.

Let's talk about what we want the program to do. We want to

```
import pygame
from pygame.locals import *
import os

Background = 0,255,127
os.environ['SDL_VIDEO_CENTERED'] = '1'
pygame.init()
screen = pygame.display.set_mode((800, 600))
pygame.display.set_caption('Pygame Example #4 - Sprite')
screen.fill(Background)
```

show a Pygame window that has our stick figure drawing in it. We want the figure to move when we press any of the arrow keys up, down, right and left, assuming we aren't at the edge of the screen and cannot move any further. We want the game to quit when we press the “q” key. Now, moving the sprite around might seem easy, and it is, but it is a bit harder than it initially sounds. We start by creating two rectangles. One for the sprite itself and one that is the same size but is blank. We blit the sprite onto the surface to start, then, when the user presses a key, we blit the blank rectangle over the original sprite, figure out the new position, and blit the sprite back onto the surface at its new position. Pretty much what we did with the alphabet game last time. That's about it for this program. It will give us an idea how to actually place a graphic on the screen and

move it around.

So, start a new program, and call it pygame4.py. Put in the includes we've been using during this tutorial. This time we'll use a minty green background so those values should be 0, 255, 127 (see above).

Next, we create a class that will handle our graphic or sprite (next page, shown bottom left). Put this right after the imports.

What is all this doing? Let's start with the __init__ routine. We initialize the sprite module of Pygame with the pygame.sprite.Sprite.__init__ line. We then set the surface, and call it screen. This will allow us to check to see if the sprite is going off the screen. We then create and set the position of the blank oldsprite variable, which will keep the old

position of our sprite. Now we load our stick figure sprite with the `pygame.image.load` routine, passing it the filename (and path, if it's not in the program's path). Then we get a reference (`self.rect`) to the sprite which sets up the width and height of the rectangle automatically, and set the `x,y` position of that rectangle to the position we pass into the routine.

The update routine basically makes a copy of the sprite, then checks to see if the sprite goes off the screen. If so, it's left where it was, otherwise its position is moved the amount we send into it.

Now, after the `screen.fill` statement, put the code shown on the following page (right-hand side).

Here we create an instance of our class, calling it `character`. Then we blit the sprite. We create the blank sprite rectangle, and fill it with the background color. We update the surface and start our loop.

As long as `DoLoop` is equal to 1, we loop through the code. We use `pygame.event.get()` to get a keyboard character. We then test it

against the event type. If it's `QUIT`, we exit. If it's a `pygame.KEYDOWN` event, we process it. We look at the key value returned, and compare it to constants defined by Pygame. We then call the update routine in our class. Notice here that we simply are passing a list containing the number of pixels on the X and Y axis to move the character. We bump it by 10 pixels (positive for right or down, negative for left or up. If the key value is equal to "q", we set `DoLoop` to 0, and so will break out of the loop. After all of that, we blit the blank character to the old position, blit the sprite to the new position, and finally update - but in this case, we update only the two rectangles containing the blank sprite and the active sprite. This saves a tremendous amount of time and processing.

As always, the full code is available at www.thedesignedgeek.com or at <http://fullcirclemagazine.pastebin.com/DvSpZbaj>.

There's a ton more that Pygame can do. I suggest that you hop over to their website, and look at the reference page

```
class Sprite(pygame.sprite.Sprite):
    def __init__(self, position):
        pygame.sprite.Sprite.__init__(self)
        # Save a copy of the screen's rectangle
        self.screen = pygame.display.get_surface().get_rect()
        # Create a variable to store the previous position of the sprite
        self.oldsprite = (0, 0, 0, 0)
        self.image = pygame.image.load('stick3.png')
        self.rect = self.image.get_rect()
        self.rect.x = position[0]
        self.rect.y = position[1]

    def update(self, amount):
        # Make a copy of the current rectangle for use in erasing
        self.oldsprite = self.rect
        # Move the rectangle by the specified amount
        self.rect = self.rect.move(amount)
        # Check to see if we are off the screen
        if self.rect.x < 0:
            self.rect.x = 0
        elif self.rect.x > (self.screen.width - self.rect.width):
            self.rect.x = self.screen.width - self.rect.width
        if self.rect.y < 0:
            self.rect.y = 0
        elif self.rect.y > (self.screen.height - self.rect.height):
            self.rect.y = self.screen.height - self.rect.height
```

(<http://www.pygame.org/docs/ref/index.html>). In addition, you can take a look at some of the games that others have put up.

Next time, we will be digging deeper into Pygame by creating a game that comes from my past...my very DISTANT past.

```
character = Sprite((screen.get_rect().x, screen.get_rect().y))
screen.blit(character.image, character.rect)

# Create a Surface the size of our character
blank = pygame.Surface((character.rect.width, character.rect.height))
blank.fill(Background)

pygame.display.update()
DoLoop = 1
while DoLoop:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        # Check for movement
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                character.update([-10, 0])
            elif event.key == pygame.K_UP:
                character.update([0, -10])
            elif event.key == pygame.K_RIGHT:
                character.update([10, 0])
            elif event.key == pygame.K_DOWN:
                character.update([0, 10])
            elif event.key == pygame.K_q:
                DoLoop = 0

# Erase the old position by putting our blank Surface on it
screen.blit(blank, character.oldsprite)
# Draw the new position
screen.blit(character.image, character.rect)
# Update ONLY the modified areas of the screen
pygame.display.update([character.oldsprite, character.rect])
```



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.



A while ago, I promised someone that I would discuss the differences between Python 2.x and 3.x. Last time, I said that we would continue our pygame programming, but I felt that I should keep my promise, so we'll delve into pygame more next time.

Many changes have gone into Python 3.x. There is a large amount of information about these changes on the Web, and I'll include a few links at the end of the article. There are also many concerns about making the change. I'm going to concentrate on changes that affect the things you've learned so far.

Let's get started.

PRINT

As I've said before, one of the most important issues is the way we deal with the Print command. Under 2.x we simply can use:

```
print "This is a test"
```

and be done with it. However under 3.x, if we try that we will get the error message shown above right.

Not happy. In order to use the print command, we must put what we want to print in parentheses like this:

```
print("this is a test")
```

Not a very big change, but

something we have to be aware of. You can get ready for your own migration by using this syntax under python 2.x.

Formatting and variable substitution

Formatting and variable substitution have also changed. Under 2.x, we have used things like the example shown below left, and, under 3.1, you can get the proper result. However, that is due to change since the '%s' and '%d' formatting functions are going away. The new way is to use '{x}' replacement statements is shown below.

It seems to me to be actually

```
>>> print "This is a test"
      File "<stdin>", line 1
        print "This is a test"
            ^
SyntaxError: invalid syntax
>>>
```

easier to read. You can also do things like this:

```
>>> print("Hello {0}. I'm glad you are here at {1}".format("Fred", "MySite.com"))
Hello Fred. I'm glad you are here at MySite.com
>>>
```

Remember, you can still use '%s' and so on, but they will be going away.

Numbers

Under Python 2.x, if you did:
`x = 5/2.0`
 x would contain 2.5. However if you did:
`x = 5/2`
 x would contain 2 due to truncation. Under 3.x, if you do:
`x = 5/2`

```
>>> months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
>>> print "You selected month %s" % months[3]
You selected month Apr
>>>
```

OLD WAY

```
>>> months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
>>> print("You selected month {0}".format(months[3]))
You selected month Apr
>>>
```

NEW WAY

you still get 2.5. To truncate the division you have to do:

```
x = 5//2
```

INPUT

A while back, we dealt with a menu system that used `raw_input()` to get a response from the user of our application. It went something like this:

```
response = raw_input('Enter a selection -> ')
```

That was fine under 2.x. However, under 3.x we get:

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
NameError: name 'raw_input' is not defined
```

This isn't a big issue. The `raw_input()` method has been replaced with `input()`. Simply change the line to:

```
response = input('Enter a selection -> ')
```

and it works just fine.

Not Equal

Under 2.x, we could test for 'not equal' with "`<>`". However, that's not allowed in 3.x. The test operator is now "`!=`".

Converting older programs to Python 3.x

Python 3.x comes with a utility to help convert a 2.x application to 3.x compliant code. This doesn't always work, but it will get you close in many cases. The conversion tool is named (aptly)

"2to3". Let's take a really simple program as an example. The example below is from way back in Beginning Python Part 3.

When run under 2.x, the output looks like that shown above right.

Of course, when we run it under 3.x, it doesn't work.

```
File "pprint1.py", line 18
```

```
+=====+
| Item 1           | 3.00 |
| Item 2           | 15.00 |
+-----+
| Total            | 18.00 |
+=====+
Script terminated.
```

```
print TopOrBottom('=',40)
```

```
SyntaxError: invalid syntax
```

We'll try to let the conversion app fix it for us. First, we should create a backup of our application that will be converted. I do it by

```
#pprint1.py
#Example of semi-useful functions

def TopOrBottom(character,width):
    # width is total width of returned line
    return '%s%s%s' % ('+',(character * (width-2)),'+')

def Fmt(val1,leftbit,val2,rightbit):
    # prints two values padded with spaces
    # val1 is thing to print on left, val2 is thing to print on right
    # leftbit is width of left portion, rightbit is width of right portion
    part2 = '%.2f' % val2
    return '%s%s%s%s' % ('| ',val1.ljust(leftbit-2,' '),part2.rjust(rightbit-2,' '),'| ')

# Define the prices of each item
item1 = 3.00
item2 = 15.00

# Now print everything out...
print TopOrBottom('=',40)
print Fmt('Item 1',30,item1,10)
print Fmt('Item 2',30,item2,10)
print TopOrBottom('-',40)
print Fmt('Total',30,item1+item2,10)
print TopOrBottom('=',40)
```

creating a copy of the file, and append a "v3" to the end of the filename:

```
cp pprint1.py pprintlv3.py
```

There's multiple ways to run the app. The simplest way is just to let the app check our code and tell us where the problems are, which is shown below left.

Notice that the original source

code is not changed. We have to use the "-w" flag to tell it to write the changes to the file. This is shown below right.

You'll also notice that the output is the same. This time, however, our source file (shown on the next page) is changed to a "version 3.x compatible" file.

Now the program works as it is supposed to under 3.x. And, since

it was simple, it still runs under version 2.x as well.

Do I switch to 3.x now?

Most of the issues are common to any change in a programming language. Syntax changes abound with every new version. Short cuts like += or -= sometimes come out of the blue and actually make our lives easier.

What's the downside to simply migrating to 3.x right now? Well, there's a little bit. Many of the library modules that we've used are not available for version 3.x right now. Things like Mutegen that we've used a few articles back just aren't available yet. While this is a stumbling block, it doesn't require you to completely give up on Python v3.x.

My suggestion is to start coding

```
> 2to3 pprintlv3.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored pprintlv3.py
--- pprintlv3.py (original)
+++ pprintlv3.py (refactored)
@@ -15,9 +15,9 @@
     item1 = 3.00
     item2 = 15.00
     # Now print everything out...
-print TopOrBottom('=' ,40)
-print Fmt('Item 1' ,30,item1,10)
-print Fmt('Item 2' ,30,item2,10)
-print TopOrBottom('-' ,40)
-print Fmt('Total' ,30,item1+item2,10)
-print TopOrBottom('=' ,40)
+print(TopOrBottom('=' ,40))
+print(Fmt('Item 1' ,30,item1,10))
+print(Fmt('Item 2' ,30,item2,10))
+print(TopOrBottom('-' ,40))
+print(Fmt('Total' ,30,item1+item2,10))
+print(TopOrBottom('=' ,40))
RefactoringTool: Files that need to be modified:
RefactoringTool: pprintlv3.py
```

```
> 2to3 -w pprintlv3.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored pprintlv3.py
--- pprintlv3.py (original)
+++ pprintlv3.py (refactored)
@@ -15,9 +15,9 @@
     item1 = 3.00
     item2 = 15.00
     # Now print everything out...
-print TopOrBottom('=' ,40)
-print Fmt('Item 1' ,30,item1,10)
-print Fmt('Item 2' ,30,item2,10)
-print TopOrBottom('-' ,40)
-print Fmt('Total' ,30,item1+item2,10)
-print TopOrBottom('=' ,40)
+print(TopOrBottom('=' ,40))
+print(Fmt('Item 1' ,30,item1,10))
+print(Fmt('Item 2' ,30,item2,10))
+print(TopOrBottom('-' ,40))
+print(Fmt('Total' ,30,item1+item2,10))
+print(TopOrBottom('=' ,40))
RefactoringTool: Files that were modified:
RefactoringTool: pprintlv3.py
```

using proper 3.x syntax now. Python version 2.6 supports almost everything you would need to write in the 3.x way. This way, you will be good to go once you have to change to 3.x. If you can live with the standard module library, go ahead and make the plunge. If, on the other hand, you push the envelope, you might just want to wait until the module library catches up. It will.

Below are some links that I thought might be helpful. The first is to the usage page of 2to3. The second is a 4-page cheat sheet that I have found to be a very good reference. The third is to what I consider to be just about the best book on using Python. (That is until I get around to writing mine.)

We'll see you next time.

Links

2to3 usage

<http://docs.python.org/library/2to3.html>

Moving from Python 2 to Python 3
(A 4 page cheat sheet)

http://ptgmedia.pearsoncmg.com/imprint_downloads/informit/promotions/python/python2python3.pdf

```
#pprint1.py
#Example of semi-useful functions

def TopOrBottom(character,width):
    # width is total width of returned line
    return '%s%s%s' % ('+',(character * (width-2)),'+')
def Fmt(val1,leftbit,val2,rightbit):
    # prints two values padded with spaces
    # val1 is thing to print on left, val2 is thing to print on right
    # leftbit is width of left portion, rightbit is width of right portion
    part2 = '%.2f' % val2
    return '%s%s%s%s' % ('| ',val1.ljust(leftbit-2,' '),part2.rjust(rightbit-2,' '),'| ')
# Define the prices of each item
item1 = 3.00
item2 = 15.00
# Now print everything out...
print(TopOrBottom('=' ,40))
print(Fmt('Item 1',30,item1,10))
print(Fmt('Item 2',30,item2,10))
print(TopOrBottom('-' ,40))
print(Fmt('Total',30,item1+item2,10))
print(TopOrBottom('=' ,40))
```

Dive into Python 3

<http://diveintopython3.org/>



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.

