The Python Collection
<table>
<thead>
<tr>
<th>Part</th>
<th>Page</th>
<th>Part</th>
<th>Page</th>
<th>Part</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>21</td>
<td>95</td>
<td>41</td>
<td>197</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>22</td>
<td>100</td>
<td>42</td>
<td>203</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>23</td>
<td>109</td>
<td>43</td>
<td>207</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>24</td>
<td>116</td>
<td>44</td>
<td>212</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>25</td>
<td>122</td>
<td>45</td>
<td>215</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>26</td>
<td>130</td>
<td>46</td>
<td>218</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
<td>27</td>
<td>139</td>
<td>47</td>
<td>220</td>
</tr>
<tr>
<td>8</td>
<td>35</td>
<td>28</td>
<td>146</td>
<td>48</td>
<td>223</td>
</tr>
<tr>
<td>9</td>
<td>44</td>
<td>29</td>
<td>153</td>
<td>49</td>
<td>225</td>
</tr>
<tr>
<td>10</td>
<td>48</td>
<td>30</td>
<td>160</td>
<td>50</td>
<td>227</td>
</tr>
<tr>
<td>11</td>
<td>53</td>
<td>31</td>
<td>164</td>
<td>51</td>
<td>229</td>
</tr>
<tr>
<td>12</td>
<td>58</td>
<td>32</td>
<td>169</td>
<td>52</td>
<td>234</td>
</tr>
<tr>
<td>13</td>
<td>62</td>
<td>33</td>
<td>171</td>
<td>53</td>
<td>237</td>
</tr>
<tr>
<td>14</td>
<td>67</td>
<td>34</td>
<td>174</td>
<td>54</td>
<td>240</td>
</tr>
<tr>
<td>15</td>
<td>73</td>
<td>35</td>
<td>177</td>
<td>55</td>
<td>244</td>
</tr>
<tr>
<td>16</td>
<td>78</td>
<td>36</td>
<td>181</td>
<td>56</td>
<td>252</td>
</tr>
<tr>
<td>17</td>
<td>82</td>
<td>37</td>
<td>184</td>
<td>57</td>
<td>257</td>
</tr>
<tr>
<td>18</td>
<td>85</td>
<td>38</td>
<td>188</td>
<td>57</td>
<td>259</td>
</tr>
<tr>
<td>19</td>
<td>88</td>
<td>39</td>
<td>189</td>
<td>58</td>
<td>263</td>
</tr>
<tr>
<td>20</td>
<td>90</td>
<td>40</td>
<td>193</td>
<td>59</td>
<td>269</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>272</td>
</tr>
</tbody>
</table>
Welcome to another Special Edition of Full Circle Magazine....

The Python Collection...

Here is a reprint of the Python series written by Greg Walters, Parts 1-60 from issues #27 through #102.

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!

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.fullcircl magazine. 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.

Find Us

Website: http://www.fullcircl e magazine. org/
IRC: #fullcircl e magazine on chat. freenode. net

Editorial Team

Editor: Ronnie Tucker
(aka: RonnieTucker)
ronnie@fullcircl e magazine. org

Webmaster: Lucas Westermann
admin@ fullcircl e magazine. org

Editing & Proofreading
Mike Kennedy, Gord Campbell,
Robert Orsino, Josh Hertel, Bert Jerred, Jim Dyer and Emily Gonyer

Our thanks go to Canonical and the many translation teams around the world.
Among the many programming languages currently available, Python is one of the easiest to learn. Python was created in the late 1980’s, and has matured greatly since then. It comes pre-installed with most Linux distributions, and is often one of the most overlooked when picking a language to learn. We’ll deal with command-line programming in this article. In a future one, we’ll play with GUI (Graphical User Interface) programming. Let’s jump right in, creating a simple application.

**Our First Program**

Using a text editor such as gedit, let’s type some code. Then we’ll see what each line does and go from there.

Type the following 4 lines.

```python
#!/usr/bin/env python
print "Hello. I am a python program."
name = raw_input("What is your name?~")
print "Hello there, " + name + "!"
```

That's all there is to it. Save the file as hello.py wherever you would like. I’d suggest putting it in your home directory in a folder named python_examples. This simple example shows how easy it is to code in Python. Before we can run the program, we need to set it to be executable. Do this by typing:

```bash
chmod +x hello.py
```

in the folder where you saved your python file. Now let's run the program.

```bash
greg@earth:/python_examples$ ./hello.py
```

Hello. I am a python program.
What is your name? Ferd Burphel
Hello there, Ferd Burphel!

That was simple. Now, let's look at what each line of the program does.

```bash
#!/usr/bin/env python
print 'Hello. I am a python program.'
```

This line tells the system that this is a python program, and to use the default python interpreter to run the program.

```bash
name = raw_input("What is your name?~")
```

Simply put, this prints the first line "Hello. I am a python program." on the terminal.

```bash
name = raw_input("What is your name?~")
```

This one is a bit more complex. There are two parts to this line. The first is name =, and the second is raw_input("What is your name?~"). We'll look at the second part first. The command raw_input will print out the prompt in the terminal ("What is your name?~"), and then will wait for the user (you) to type something (followed by {Enter}). Now let's look at the first part; name =. This part of the command assigns a variable named "name". What’s a variable? Think of a variable as a shoe-box. You can use a shoe-box to store things -- shoes, computer parts, papers, whatever. To the shoe-box, it doesn't really matter what's in there -- it's just stored there. In this case, it stores whatever you type. In the case of my entry, I typed Ferd Burphel. Python, in this case, is a shoe-box. It stores the variables you assign it in a shoe-box. When you write a python program, you can use this shoe-box to store various values. In this case, we are storing the name you enter.

---

Full circle magazine 4 The Compleat Python
instance, simply takes the input and stores it in the "name" shoe-box for use later in the program.

```python
print "Hello there, " + name + "!"
```

Once again, we are using the print command to display something on the screen -- in this case, "Hello there, ", plus whatever is in the variable "name", and an exclamation point at the end. Here we are concatenating or putting together three pieces of information: "Hello there", information in the variable "name", and the exclamation point.

Now, let's take a moment to discuss things a bit more deeply before we work on our next example. Open a terminal window and type:

```
python
```

You should get something like this:

```
greg@earth:/python_examples$ python
Python 2.5.2 (r252:60911, Oct 5 2008, 19:24:49)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

```python
>>> print 2+2
```

That's because the word "print" is a known command, while "Print" is not. Case is very important in Python.

Now let's play with variables a bit more. Type:

```python
var = 2+2
```

You'll see that nothing much happens except Python returns the ">>>" prompt. Nothing is wrong. What we told Python to do is create a variable (shoe-box) called var, and to stick into it the sum of "2+2". To see what var now holds, type:

```python
print var
```

```
4
```

and press enter. You'll get back

```python
>>> print 2+2
```

```
4
```

Notice that we typed the word "print" in lower case. What would happen if we typed "Print 2+2"? The response from the interpreter is this:

```python
>>> Print 2+2
File "<stdin>", line 1
  Print 2+2
```

Now we can use var over and over again as the number 4, like this:

```python
>>> print var * 2
```

```
8
```

If we type "print var" again we'll get this:

```python
>>> print var
4
```

var hasn't changed. It's still the sum of 2+2, or 4.

This is, of course, simple programming for this beginner's tutorial. Complexity will increase in subsequent tutorials. But now let's look at some more examples of variables.

In the interpreter type:

```python
>>> strng = 'The time has come for all good men to come to the aid of the party!'
```

```python
>>> print strng
```

The time has come for all good men to come to the aid of the party!

```python
>>> print var
```

You've created a variable named "strng" (short for string) containing the value 'The time has come for all good men to come to the aid of the party!'. From now on (as long as we are

```python
>>> Python 2.5.2 (r252:60911, Oct 5 2008, 19:24:49)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

```python
>>> print 2+2
```

```
4
```

```python
>>> var = 2+2
```

```python
>>> print var
```

```
4
```

```python
>>> print 2+2
```

```
4
```

```python
>>> print 2+2
```

```
4
```

```python
>>> print 2+2
```

```
4
```

```python
>>> print 2+2
```

```
4
```

```python
>>> print 2+2
```

```
4
```

```python
>>> print 2+2
```

```
4
```
in this instance of the interpreter), our strng variable will be the same unless we change it. What happens if we try to multiply this variable by 4?

>>> print strng * 4

The time has come for all good men to come to the aid of the party! The time has come for all good men to come to the aid of the party! The time has come for all good men to come to the aid of the party!

>>> 

Well, that is not exactly what you would expect, is it? It printed the value of strng 4 times. Why? Well, the interpreter knew that strng was a string of characters, not a value. You can't perform math on a string.

What if we had a variable called s that contained '4', as in the following:

>>> s = '4'
>>> print s

The string (s), which is '4', has now been converted to an

It looks as though s contains the integer 4, but it doesn't. Instead it contains a string representation of 4. So, if we type 'print s * 4' we get...

>>> print s*4
4444

Once again, the interpreter knows that s is a string, not a numerical value. It knows this because we enclosed the number 4 with single quotes, making it a string.

We can prove this by typing print type(s) to see what the system thinks that variable type is.

>>> print type(s)
<type 'str'>

Confirmation. It's a string type. If we want to use this as a numerical value, we could do the following:

>>> print int(s) * 4
16

You have now been introduced to the print command, the raw_input command, assigning variables, and the difference between strings and integers.

Let's go a bit further. In the Python Interpreter, type quit() to exit back to the command prompt.

Simple For Loop

Now, let's explore a simple programming loop. Go back to the text editor and type the following program.

```
#!/usr/bin/env python
for cntr in range(0,10):
    print cntr
```

Be sure to tab the "print cntr" line. This is important. Python doesn't use parentheses "(" or curly braces "{" as do other programming languages to show code blocks. It uses indentations instead.

Save the program as "for_loop.py". Before we try to run this, let's talk about what a for loop is.

A loop is some code that does a specified instruction, or set of instructions, a number of times. In the case of our program, we loop 10 times, printing the value of the variable cntr (short for counter). So the command in plain English is "assign the variable cntr 0, loop 10 times printing the variable cntr contents, add one to cntr and do it all over again. Seems simple enough. The part of the code "range(0,10)" says start with 0, loop until the value of cntr is 10, and quit.

Now, as before, do a

```
chmod +x for_loop.py
```

and run the program with

```
./for_loop.py
```
in a terminal.

greg@earth:~/python_examples$ ./for_loop.py
0
1
Well, that seems to have worked, but why does it count up to only 9 and then stop. Look at the output again. There are 10 numbers printed, starting with 0 and ending with 9. That's what we asked it to do -- print the value of `cntr` 10 times, adding one to the variable each time, and quit as soon as the value is 10.

Now you can see that, while programming can be simple, it can also be complex, and you have to be sure of what you ask the system to do. If you changed the range statement to be "range(1,10)" it would start counting at 1, but end at 9, since as soon as `cntr` is 10, the loop quits. So to get it to print "1,2,3,4,5,6,7,8,9,10", we should use range(1,11) - since the for loop quits as soon as the upper range number is reached.

Also notice the syntax of the statement. It is "for variable in range(start value,end value):" The ":" says, we are starting a block of code below that should be indented. It is very important that you remember the colon ":", and to indent the code until the block is finished.

If we modified our program to be like this:

```python
#!/usr/bin/env python
for cntr in range(1,11):
    print cntr
print 'All Done'
```

We would get an output of...

```
greg@earth:~/python_examples$ ./for_loop.py
1
2
3
4
5
6
7
8
9
10
All Done
greg@earth:~/python_examples$
```

Make sure your indentation is correct. Remember, indentation shows the block formatting. We will get into more block indentation thoughts in our next tutorial.

That's about all for this time. Next time we'll recap and move forward with more python programming instructions. In the meantime, you might want to consider installing a python specific editor like Dr. Python, or SPE (Stani's Python Editor), both of which are available through Synaptic.
In the last installment, we looked at a simple program using raw_input to get a response from the user, some simple variable types, and a simple loop using the "for" statement. In this installment, we will delve more into variables, and write a few more programs.

LISTS

Let's look at another type of variable called lists. In other languages, a list would be considered an array. Going back to the analogy of shoe-boxes, an array (or list) would be a number of boxes all glued side-by-side holding like items. For example, we could store forks in one box, knives in another, and spoons in another. Let's look at a simple list. An easy one to picture would be a list of month names. We would code it like this...

```
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
```

To create the list, we bracket all the values with square brackets ('[', and ']'). We have named our list 'months'. To use it, we would say something like print months[0] or months[1] (which would print 'Jan' or 'Feb'). Remember that we always count from zero. To find the length of the list, we can use:

```
print len(months)
```

which returns 12.

Another example of a list would be categories in a cookbook. For example...

```
categories = ['Main dish', 'Meat', 'Fish', 'Soup', 'Cookies']
```

Then categories[0] would be 'Main dish', and categories[4] would be 'Cookies'. Pretty simple again. I'm sure you can think of many things that you can use a list for.

Up to now, we have created a list using strings as the information. You can also create a list using integers. Looking back at our months list, we could create a list containing the number of days in each one:

```
DaysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31]
```

If we were to print DaysInMonth[1] (for February) we would get back 28, which is an integer. Notice that I made the list name DaysInMonth. Just as easily, I could have used 'daysinmonth' or just 'X'... but that is not quite so easy to read. Good programming practices suggest (and this is subject to interpretation) that the variable names are easy to understand. We'll get into the whys of this later on. We'll play with lists some more in a little while.

Before we get to our next sample program, let's look at a few other things about Python.
More on Strings

We briefly discussed strings in Part 1. Let's look at string a bit closer. A string is a series of characters. Not much more than that. In fact, you can look at a string as an array of characters. For example if we assign the string 'The time has come' to a variable named strng, and then wanted to know what the second character would be, we could type:

```python
strng = 'The time has come'
print strng[1]
```

The result would be 'h'. Remember we always count from 0, so the first character would be [0], the second would be [1], the third would be [2], and so on. If we want to find the characters starting at position 4 and going through position 8, we could say:

```python
print strng[4:8]
```

which returns 'time'. Like our for loop in part 1, the counting stops at 8, but does not return the 8th character, which would be the space after 'time'.

We can find out how long our string is by using the len() function:

```python
print len(strng)
```

which returns 17. If we want to find out where in our string the word 'time' is, we could use:

```python
pos = strng.find('time')
```

Now, the variable pos (short for position) contains 4, saying that 'time' starts at position 4 in our string. If we asked the find function to find a word or sequence that doesn't exist in the string like this:

```python
pos = strng.find('apples')
```

the returned value in pos would be -1.

We can also get each separate word in the string by using the split command. We will split (or break) the string at each space character by using:

```python
print strng.split(' ')
```

which returns a list containing ['The', 'time', 'has', 'come']. This is very powerful stuff. There are many other built-in string functions, which we'll be using later on.

```python
print len('The time has come')
print strng[4:8]
print strng.find('time')
print strng.find('apples')
print strng.split(' ')
```

Literal Substitution

Here is one other thing that I will introduce before we get to our next programming example. When we want to print something that includes literal text as well as variable text, we can use what's called Variable Substitution. To do this is rather simple. If we want to substitute a string, we use '%s' and then tell Python what to substitute. For example, to print a month from our list above, we can use:

```python
print 'Month = %s' % month[0]
```

This would print 'Month = Jan'. If we want to substitute an integer, we use '%d'. Look at the example below:

```python
DaysInMonth = [31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31]
for cntr in range(0, 12):
    print 'Month = %s has %d days.' % (Months[cntr], DaysInMonth[cntr])
```

The result from this code is:

Jan has 31 days.
Feb has 28 days.
Mar has 31 days.
Apr has 30 days.
May has 31 days.
Jun has 30 days.
Jul has 31 days.
Aug has 31 days.
Sep has 30 days.
Oct has 31 days.
Nov has 30 days.
Dec has 31 days.

Something important to understand here is the use of single quotes and double quotes. If you assign a variable to a string like this:

```python
st = 'The time has come'
```

or like this:

```python
st = "The time has come"
```

the result is the same. However, if you need to include a single quote in the string like this:
Equate

We need to learn a few more things to be able to do our next example. First is the difference between assignment and equate. We've used the assignment many times in our samples. When we want to assign a value to a variable, we use the assignment operator or the '==' (equal sign):

```python
variable = value
```

However, when we want to evaluate a variable to a value, we must use a comparison operator. Let's say we want to check to see if a variable is equal to a specific value. We would use the '==' (two equal signs):

```python
variable == value
```

So, if we have a variable named `loop` and we want to see if it is equal to, say, 12, we would use:

```python
if loop == 12:
```

Don't worry about the if and the colon shown in the example above yet. Just remember we have to use the double-equal sign to do evaluation.

Comments

The next thing we need to discuss is comments. Comments are important for many things. Not only do they give you or someone else an idea of what you are trying to do, but when you come back to your code, say 6 months from now, you can be reminded of what you were trying to do. When you start writing many programs, this will become important. Comments also allow you to make Python ignore certain lines of code. To comment a line you use the '# ' sign. For example:

```python
# This is a comment
```

You can put comments anywhere on a code line, but remember when you do, Python will ignore anything after the '#'.

If statements

Now we will return to the "if" statement we showed briefly above. When we want to make a decision based on values of things, we can use the if statement:

```python
if loop == 12:
```

This will check the variable

'loop', and, if the value is 12, then we do whatever is in the indented block below. Many times this will be sufficient, but, what if we want to say if a variable is something, then do this, otherwise do that. In pseudo code you could say:

```python
if x == y then
    do something
else
    do something else
```

and in Python we would say:

```python
if x == y:
    do something
else:
    do something else
```

The main things to remember here are:

1. End the if or else statements
with a colon.

2. INDENT your code lines.

Assuming you have more than one thing to check, you can use the if/elif/else format. For example:

```python
x = 5
if x == 1:
    print 'X is 1'
elif x < 6:
    print 'X is less than 6'
elif x < 10:
    print 'X is less than 10'
else:
    print 'X is 10 or greater'
```

Notice that we are using the '<' operator to see if x is LESS THAN certain values - in this case 6 or 10. Other common comparison operators would be greater than '>', less than or equal to '<=', greater than or equal to '>=', and not equal '!='.

### While statements

Finally, we'll look at a simple example of the while statement. The while statement allows you to create a loop doing a series of steps over and over, until a specific threshold has been reached. A simple example would be assigning a variable “loop” to 1. Then while the loop variable is less than or equal to 10, print the value of loop, add one to it and continue, until, when loop is greater than 10, quit:

```python
loop = 1
while loop <= 10:
    print loop
    loop = loop + 1
```

run in a terminal would produce the following output:

```
1
2
3
4
5
6
7
8
9
10
```

This is exactly what we wanted to see. Fig.1 (above right) is a similar example that is a bit more complicated, but still simple.

In this example, we are combining the if statement, while loop, raw_input statement, new line escape sequence, assignment operator, and comparison operator does NOT equal 'quit'.

One more quick example before we leave for this month. Let’s say you want to check to see if a user is allowed to access your program. While this example is not the best way to do this task, it’s a good way to show some things that we’ve already learned. Basically, we will ask the user for their name and a password, compare them with information that we coded inside the program, and then make a decision based on what we find. We will use two lists! one to hold the allowed users and
one to hold the passwords. Then we'll use raw_input to get the information from the user, and finally the if/elif/else statements to check and decide if the user is allowed. Remember, this is not the best way to do this. We'll examine other ways in later articles. Our code is shown in the box to the right.

Save this as 'password_test.py' and run it with various inputs.

The only thing that we haven't discussed yet is in the list checking routine starting with 'if username in users:'. What we are doing is checking to see if the user's name that was entered is in the list. If it is, we get the position of the user's name in the list users. Then we use users.index(username) to get the position in the users list so we can pull the password, stored at the same position in the passwords list. For example, John is at position 1 in the users list. His password, 'dog' is at position 1 of the passwords list. That way we can match the two. Should be pretty easy to understand at this point.

That's enough for this month. Next time, we'll be learning about functions and modules. Until then, play with what you've already learned and 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.
Program In Python - Part 3

In the last article, we learned about lists, literal substitution, comments, equate versus assignment, if statements and while statements. I promised you that in this part we would learn about modules and functions. So let’s get started.

Modules

Modules are a way to extend your Python programming. You can create your own, or use those that come with Python, or use modules that others have created. Python itself comes with hundreds of various modules that make your programming easier. A list of the global modules that come with Python can be found at http://docs.python.org/modindex.html. Some modules are operating system specific, but most are totally cross platform (can be used the same way in Linux, Mac and Microsoft Windows). To be able to use an external module, you must import it into your program. One of the modules that comes with Python is called 'random'. This module allows you to generate pseudo-random numbers. We'll use the module shown above right in our first example.

Let's examine each line of code. The first four lines are comments. We discussed them in the last article. Line five tells Python to use the random module. We have to explicitly tell Python to do this.

Line seven sets up a 'for' loop to print 14 random numbers. Line eight uses the randint() function to print a random integer between 1 and 10. Notice we must tell Python what module the function comes from. We do this by saying (in this case) random.randint. Why even create modules? Well, if every possible function were included directly into Python, not only would Python become absolutely huge and slow, but bug fixing would be a nightmare. By using modules, we can segment the code into groups that are specific to a certain need. If, for example, you have no need to use database functionality, you don't need to know that there is a module for SQLite. However, when you need it, it's already there. (In fact, we'll be using database modules later on in this series.)

Once you really get started in Python programming, you will probably make your own modules so you can use the code you've already written over and over again, without having to re-type it. If you need to change something in that group of code, you can, with very little risk of breaking the code in your main program. There are limits to this and we will delve into this later on. Now, when we used the 'import random' statement earlier, we were telling Python to give us access to every function within the random module. If, however, we only needed to use the randint() function, we
can re-work the import statement like this:

```python
from random import randint
```

Now when we call our function, we don't have to use the 'random.' identifier. So, our code changes to

```python
from random import randint
# print 14 random integers for cntr in range(1,15):
    print randint(1,10)
```

### Functions

When we imported the random module, we used the randint() function. A function is a block of code that is designed to be called, usually more than once, which makes it easier to maintain, and to keep us from typing the same code over and over and over. As a very general and gross statement, any time you have to write the same code more than once or twice, that code is a good candidate for a function. While the following two examples are silly, they make good statements about using functions. Let's say we wanted to take two numbers, add them, then multiply them, and then subtract them, displaying the numbers and results each time. To make matters worse, we have to do that three times with three sets of numbers. Our silly example would then look like the text shown right.

Not only is this a lot of typing, it lends itself to errors, either by typing or having to change something later on. Instead, we are going to create a function called 'DoTwo' that takes the two numbers and does the math, printing the output each time. We start by using the 'def' key word (which says that we are going to define the function). After 'def' we add the name we select for the function, and then a list of parameters (if any) in parentheses. This line is then closed by a colon (:). The code in the function is indented. Our improved silly example (#2) is shown below.

As you can see, there's a lot less typing involved — 8 lines instead of 12 lines. If we need to change something in our function, we can do it without causing too many issues to our main program. We call our function, in this case, by using the function name and putting the parameters after.

Here is another example of a function. Consider the following requirements.

We want to create a
program that will print out a list of purchased items in a pretty format. It must look something like the text below.

The cost of each item and for the total of all items will be formatted as dollars and cents. The width of the print out must be able to be variable. The values on the left and right must be variable as well. We will use 3 functions to do this task. One prints the top and bottom line, one prints the item detail lines including the total line and one prints the separator line. Luckily, there are a number of things that Python has that will make this very simple. If you recall, we printed a string multiplied by 4, and it returned four copies of the same string. Well we can use that to our benefit. To print our top or bottom line we can take the desired width, subtract two for the two + characters and use “=' * (width-2)”. To make things even easier, we will use variable substitution to put all these items on one line. So our string to print would be coded as 's ('+',('=' * width-2),'+')'. Now we could have the routine print this directly, but we will use the return keyword to send the generated string back to our calling line. We'll call our function ‘TopOrBottom’ and the code for this function looks like this.

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

We could leave out the comment, but it's nice to be able to tell at a glance what the parameter 'width' is. To call it, we would say 'print TopOrBottom(40)' or whatever width we wish the line to be. Now we have one function that takes care of two of the lines. We can make a new function to take care of the separator line using the same kind of code...OR we could modify the function

we just made to include a parameter for the character to use in the middle of the pluses. Let's do that. We can still call it TopOrBottom.

```
def TopOrBottom(character,width):
    # width is total width of returned line
    # character is the character to be placed between the '+' characters
    return '%s%s%s' % ('+',(character * (width-2)),'+')
```

Now, you can see where comments come in handy. Remember, we are returning the generated string, so we have to have something to receive it back when we make the call to it. Instead of assigning it to another string, we'll just print it. Here's the calling line.

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

So now, not only have we taken care of three of the lines, we've reduced the number of routines that we need from 3 down to 2. So we only have the center part of the print out to deal with. Let's call the new function 'Fmt'. We'll pass it 4 parameter values as follows:

- `val1` – the value to print on the left
- `leftbit` – the width of this “column”
- `val2` – the value to print on the right (which should be a floating value)
- `rightbit` – the width of this “column”

The first task is to format the information for the right side. Since we want to format the value to represent dollars and cents, we can use a special function of variable substitution that says, print the value as a floating point number with n number of places to the right of the decimal point. The command would be '%.2f'. We will assign this to a variable called 'part2'. So our code line would be 'part2 = %.2f % val2'. We also can use a set of functions that's built into Python strings called ljust and rjust. Ljust will left justify the string, padding the right side with whatever character you want. Rjust does
the same thing, except the padding goes on the left side.
Now for the neat bit. Using substitutions we throw together
a big string and return that to
the calling code. Here is our
next line.

```
return 'ss' % ('|
','val1.ljust(leftbit-2,' ')
',part2.rjust(rightbit-2,' ')
',' | ')
```

While this looks rather
daunting at first, let's dissect it
and see just how easy it is:
**Return** - We will send back
our created string to the
calling code.
'ss' - We are going to stick in
4 values in the string. Each %s
is a place holder.
% ( - Starts the variable list
'| ', - Print these literals
val1.ljust(leftbit-2,' ') - Take
the variable val1 that we were
passed, left justify it with
spaces for (leftbit-2)
characters. We subtract 2 to
allow the '|' on the left side.
Part2.rjust(rightbit-2,' ') -
Right justify the formatted
string of the price rightbit-2
spaces. '|' - finish the string.

That's all there is to it.

While we should really do some
error checking, you can use
that as something to play with
on your own. So...our Fmt
function is really only two lines
of code outside of the definition
line and any comments. We can
call it like this.

```
print Fmt('Item 1',30,6,10)
```

Again, we could assign the
return value to another string,
but we can just print it. Notice
that we are sending 30 for
the width of the
left bit and 10 for
the width of the
right. That
equals the 40
that we sent to our
TopOrBottom routine earlier. So,
fire up your editor and type in
the code below.

```
#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' % ('| ',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',30,6,10)
print Fmt('Item 2',30,6,10)
print TopOrBottom('-',40)
print Fmt('Total',30,6,10)
print TopOrBottom('=',40)
```

While this is a very simple
example, it should give you a
good idea of why and how to
use functions. Now, let's extend
this out a bit and learn

| Item 1   | 3.00 |
| Item 2   | 15.00 |

==+=

shown above right.
more about lists. Remember back in part 2 when we first discussed lists? Well one thing that I didn’t tell you is that a list can contain just about anything, including lists. Let’s define a new list in our program called itms and fill it like this:

```python
# define a new list in our program called itms and fill it like this:

itms = ['Soda', 1.45], ['Candy', .75], ['Bread', 1.95], ['Milk', 2.59]
```

If we were to access this as a normal list we would use `print(itms[0])`. However, what we would get back is ['Soda', 1.45], which is not really what we were looking for under normal circumstances. We want to access each item in that first list. So we would use `print(itms[0][0])` to get 'Soda' and [0][1] to get the cost or 1.45. So, now we have 4 items that have been purchased and we want to use that information in our pretty print routine. The only thing we have to change is at the bottom of the program. Save the last program as 'pprint2.py', then comment out the two itemx definitions and insert the list we had above. It should look like this now.

```python
# define a new list in our program called itms and fill it like this:

itms = ['Soda', 1.45], ['Candy', .75], ['Bread', 1.95], ['Milk', 2.59]
```

Next, remove all the lines that call Fmt(). Next add the following lines (with #NEW LINE at the end) to make your code look like the text shown right.

```python
# set up a counter variable for loop that cycles through the list for each item there. Notice that I've also added a variable called total. We set the total to 0 before we go into our for loop. Then as we print each item sold, we add the cost to our total. Finally, we print the total out right after the separator line. Save your program and run it. You should see something like the text shown below.

```
if we would get back is ['Soda', 1.45], which is not really what we were looking for under normal circumstances. We want to access each item in that first list. So we would use 'print itms[0][0]' to get 'Soda' and [0][1] to get the cost or 1.45. So, now we have 4 items that have been purchased and we want to use that information in our pretty print routine. The only thing we have to change is at the bottom of the program. Save the last program as 'pprint2.py', then comment out the two itemx definitions and insert the list we had above. It should look like this now.

```
# set up a counter variable for loop that cycles through the list for each item there. Notice that I've also added a variable called total. We set the total to 0 before we go into our for loop. Then as we print each item sold, we add the cost to our total. Finally, we print the total out right after the separator line. Save your program and run it. You should see something like the text shown below.

```
```
class Dog():
    def __init__(self,dogname,dogcolor,dogheight,dogbuild,dogmood,dogage):
        # here we setup the attributes of our dog
        self.name = dogname
        self.color = dogcolor
        self.height = dogheight
        self.build = dogbuild
        self.mood = dogmood
        self.age = dogage
        self.Hungry = False
        self.Tired = False

promised last time that we would discuss classes. So, that's what we'll concentrate on. What are classes and what good are they?

A class is a way of constructing objects. An object is simply a way of handling attributes and behaviors as a group. I know this sounds confusing, but I'll break it down for you. Think of it this way. An object is a way to model something in the real world. A class is a method we use to implement this. For example, we have three dogs at home. A Beagle, a Lab and a German Shepherd/Blue Heeler mix. All three are dogs, but are all different. There are common attributes among the three of them, but each dog has separate attributes as well. For example, the Beagle is short, chubby, brown, and grumpy. The Lab is medium-sized, black, and very laid back. The Shepherd/Heeler mix is tall, skinny, black, and more than a bit crazy. Right away, some attributes are obvious. Short/medium-sized/tall are all attributes of height. Grumpy, laid back, and crazy are all attributes of mood. On the behavior side of things, we can consider eating, sleeping, playing, and other actions.

All three are of the class 'Dog'. Going back to the attributes that we used to describe each above, we have things such as Dog.Name, Dog.Height, Dog.Build (skinny, chubby, etc.), and Dog.Color. We also have behaviors such as Dog.Bark, Dog.Eat, Dog.Sleep, and so on.

As I said before, each of the dogs is a different breed. Each breed would be a sub-class of the class Dog. In a diagram, it would look like this.

```
Dog ---|-- Lab
       \--Shepherd/Heeler
```

Each sub-class inherits all of the attributes of the Dog class. Therefore, if we create an instance of Beagle, it gets all of the attributes from its parent class, Dog.

```python
Beagle = Dog()
Beagle.Name = 'Archie'
Beagle.Height = 'Short'
Beagle.Build = 'Chubby'
Beagle.Color = 'Brown'
```

Starting to make sense? So, let's create our gross Dog class (shown above). We'll start with the keyword "class" and the name of our class.
Before we go any further in our code, notice the function that we have defined here. The function __init__ (two underscores + 'init' + two underscores) is an initialization function that works with any class. As soon as we call our class in code, this routine is run. In this case, we have set up a number of parameters to set some basic information about our class: we have a name, color, height, build, mood, age, and a couple of variables Hungry and Tired. We'll revisit these in a little bit. Now let's add some more code.

```
Beagle = 
Dog('Archie', 'Brown', 'Short',
'Chubby', 'Grumpy', 12)
print Beagle.name
print Beagle.color
print Beagle.mood
print Beagle.Hungry
```

This is UNINDENTED code that resides outside of our class, the code that uses our class. The first line creates an instance of our dog class called Beagle. This is called instantiation. When we did this, we also passed certain information to the instance of the class, such as the Beagle's name, color, and so on. The next four lines simply query the Beagle object and get back information in return. Time for more code. Add the code shown in the top right box into the class after the __init__ function.

Now we can call it with Beagle.Eat() or Beagle.Sleep(). Let's add one more method. We'll call it Bark. Its code is shown right.

```
    def Bark(self):
        if self.mood == 'Grumpy':
            print 'GRRRRR...Woof Woof'
        elif self.mood == 'Laid Back':
            print 'Yawn...Ok...Woof'
        elif self.mood == 'Crazy':
            print 'Bark Bark Bark Bark Bark Bark Bark Bark Bark'
        else:
            print 'Woof Woof'
```

def Eat(self):
    if self.Hungry:
        print 'Yum Yum...Num Num'
        self.Hungry = False
    else:
        print 'Sniff Sniff...Not Hungry'
def Sleep(self):
    print 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
    self.Tired = False

def Bark(self):
    if self.mood == 'Grumpy':
        print 'GRRRRR...Woof Woof'
    elif self.mood == 'Laid Back':
        print 'Yawn...Ok...Woof'
    elif self.mood == 'Crazy':
        print 'Bark Bark Bark Bark Bark Bark Bark Bark Bark'
    else:
        print 'Woof Woof'

create two more instances of our dog class.

```
Lab = 
Dog('Nina', 'Black', 'Medium',
'Heavy', 'Laid Back', 7)
Heeler = 
Dog('Bear', 'Black', 'Tall', 'Skinny', 'Crazy', 9)
```

print 'My name is %s' % Lab.name
print 'My color is %s' % Lab.color
print 'My Mood is %s' % Lab.mood
print 'I am hungry = %s' % Lab.Hungry
Lab.Bark()
Heeler.Bark()

Notice that I created the instances of both of the dogs before I did the print statements. That's not a problem, since I “defined” the instance before I called any of the methods. Here is the full output of our dog class program.

```
My name is Archie
My color is Brown
My mood is Grumpy
I am hungry = False
Sniff Sniff...Not Hungry
Yum Yum...Num Num
GRRRRR...Woof Woof
```
class Dog():
    def __init__(self,dogname,dogcolor,dogheight,dogbuild,dogmood,dogage):
        #here we setup the attributes of our dog
        self.name = dogname
        self.color = dogcolor
        self.height = dogheight
        self.build = dogbuild
        self.mood = dogmood
        self.age = dogage
        self.Hungry = False
        self.Tired = False

    def Eat(self):
        if self.Hungry:
            print 'Yum Yum...Num Num'
            self.Hungry = False
        else:
            print 'Sniff Sniff...Not Hungry'

    def Sleep(self):
        print 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
        self.Tired = False

    def Bark(self):
        if self.mood == 'Grumpy':
            print 'GRRRRR...Woof Woof'
        elif self.mood == 'Laid Back':
            print 'Yawn...ok...Woof'
        elif self.mood == 'Crazy':
            print 'Bark Bark Bark Bark Bark Bark Bark Bark Bark Bark Bark Bark Bark Bark Bark Bark'
        else:
            print 'Woof Woof'

Beagle = Dog('Archie','Brown','Short','Chubby','Grumpy',12)
print 'My name is %s' % Beagle.name
print 'My color is %s' % Beagle.color
print 'My mood is %s' % Beagle.mood
print 'I am hungry = %s' % Beagle.Hungry
Beagle.Eat()
Beagle.Hungry = True
Beagle.Eat()
Beagle.Bark()
I f you are like me, you will HATE the first part of this installation. I HATE it when an author tells me that I have to double read every word in their book/chapter/article, because I just KNOW it will be a snore - even when I know it's for my own good, and I will end up doing it anyway.

Consider yourself warned. PLEASE read the following boring stuff carefully. We'll get to the fun stuff soon, but we need to get some ground work covered before we can really talk about trying to program.

FIRST you need to install Boa Constructor and wxPython. Use Synaptic and select both wxPython and Boa Constructor. Once installed, you should find Boa under Applications\Programming\Boa Constructor. Go ahead and start it up. It will make things a bit easier. Once the application starts, you will see three different windows (or frames): one across the top, and two across the bottom. You might have to resize and move them a bit, but get things to a point where it looks something like this:

The top frame is called the tool frame. The bottom-left frame is the inspector frame, and the bottom-right frame is the editor frame. On the tool frame, you have various tabs (New, Containers/Layout, etc.) that will allow you to start new projects, add frames to existing projects, and add various controls to the frames for your application. The inspector frame will become very important as we start to add controls to our application. The editor frame allows us to edit our code, save our projects, and more. Moving our attention back to the tool frame, let's take a look at each tab - starting with the “New” tab. While there are many options available here, we will discuss only two of them. They are the 5th and 6th buttons from the left: wx.App and wx.Frame. Wx.App allows us to create a complete application beginning with two auto-generated files. One is a frame file and the other is an application file. This is the method I prefer to use.

The wx.Frame is used to add more frames to our application and/or create a standalone app from a single source file. We'll discuss this later.

Now look at the Containers/Layout tab. Many goodies here. The ones you'll use most are the wx.Panel (first on the left) and the sizers (2,3,4,5 and 6 from the right). Under Basic Controls, you'll find static text controls (labels), text boxes, check boxes, radio buttons, and more. Under Buttons, you'll find various forms of buttons. List Controls has data grids and other list boxes. Let's jump to Utilities where you'll find timers and menu items.

Here are a few things to remember as we are getting ready for our first app. There are a few bugs in the Linux version. One is that SOME controls won't allow you to move them in the designer. Use the <Ctrl>+Arrow keys to...
move or tweak the position of your controls. Another one you'll find when you try the tutorials that come with Boa Constructor - when placing a panel control, it's hard to see. Look for the little boxes (I'll show you this soon). You can also use the Objs tab on the Inspector frame and select it that way.

Okay, here we go. Under the 'New' tab of the tool frame, select wx.App (5th button from the left). This will create two new tabs in the editor frame: one named "*(App1)"*, the other named "*(Frame1)"*. Believe it or not, the VERY first thing we want to do is save our two new files, starting with the Frame1 file. The save button is the 5th button from the left in the Editor Frame. A “Save As” frame will pop up asking you where you want to save the file and what you want to call it. Create a folder in your home folder called GuiTests, and save the file as “Frame1.py”. Notice that the "*(Frame1)"" tab now shows as "Frame1". (The "*(" says that the file needs to be saved.) Now do the same thing with the App1 tab.

Now let's examine a few of the buttons on the Editor Tool bar. The important ones for now are the Save (5th from the left) and Run (Yellow arrow, 7th from the left). If you are in a frame tab (Frame1 for example) there will be some extra buttons you need to know about. For now it's the Designer button:

It is an important one. It allows us to design our GUI frame - which is what we'll do now. When you click on it you will be presented with a blank frame.

This is a blank canvas for you to put whatever controls you need to (within reason). The first thing we want to do is place a wx.panel control. Almost everything I have read says not to put controls (other than a wx.panel) directly on a frame. So, click on the Containers/Layout tab in the Tool Frame, then click on the wx.Panel button. Next, move over to the new frame that you are working on and click somewhere on the inside of the frame. You'll know it worked if you see something like this:

Remember when I warned you about the bugs? Well, this is one of them. Don't worry. See the 8 little black squares? That's the limits of the panel. If you wanted, you could click and drag one of them to resize the panel, but for this project what we want is to make the panel cover the entire frame. Simply resize the FRAME just a little bit at this point. Now we have a panel to put our other controls on. Move the frame you are working on until you can see the tool box for the Editor frame. Two new buttons have appeared: a check and an “X”. The “X” will cause the changes you made to be thrown away.

The Check button:

is called the “Post” button. This will cause your changes to be written into our frame file. You still have to save the frame file, but this will get the new things into the file. So, click on the Post button. There's also a post button on the Inspector frame, but we'll deal with that later. Now save your file.

Go back into the Design mode. Click the 'Buttons' tab on the Tool frame and then click the first button on the left, the wx.Button. Then add it somewhere close to the middle of your frame. You'll have something that looks close to this:
Notice that there are 8 small squares around it just like the panel. These are resize handles. It also shows us what control is currently selected. In order to move this closer to the center of the frame, hold down the Control key (Ctrl) and while that's being pressed, use the arrow keys to move it where you want it. Now, let's look at the Inspector frame. There are four tabs. Click on the 'Constr' tab. Here we can change the label, name, position, size and style. For now, let's change the name to 'btnShowDialog' and the label property to 'Click Me'.

Post (check button) and save your changes. Go back to the designer once again, and notice that (assuming you still have the 'Obj' tab in the inspector frame selected), Frame1 is now selected. This is good because it's what we want. Go back to the 'Constr' tab, and change the title from 'Frame1' to 'Our First GUI'. Post and save one more time. Now let's run our app. Click the yellow Run button on the Editor frame.

Click all you want on the button, but nothing will happen. Why? Well, we didn't tell the button to do anything. For that, we need to set up an event to happen, or fire, when the user clicks our button. Click on the X in the upper-right corner to finish running the frame. Next, go back to the designer, select the button and go into the 'Evts' tab in the inspector frame. Click on ButtonEvent and then double click on the wx.EVT_BUTTON text that shows up, and notice that in the window below we get a button event called 'OnBtnShowDialogButton'. Post and save.

Before we go any further, let's see what we've got in the way of code (page 24).

The first line is a comment that tells Boa Constructor that this is a boa file. It's ignored by the Python compiler, but not by Boa. The next line imports wxPython. Now jump down to the class definition.

At the top, there's the __init__ method. Notice the comment just under the definition line. Don't edit the code in this section. If you do, it will be sorry. Any place BELOW that routine should be safe. In this routine, you will find the definitions of each control on our frame.

Next, look at the __init__ routine. Here you can put any calls to initializing code. Finally, the OnBtnShowDialogButton routine. This is where we will put our code that will do the work when the user clicks the button. Notice that there is currently an event.Skip() line there. Simply stated, this says just exit when this event fires.

Now, what we are going to do is call a message box to pop up with some text. This is a common thing for programmers to do to allow the user to know about something - an error, or the fact that a
program in python - part 5

process has finished. In this case, we will be calling the wx.MessageBox built in routine. The routine is called with two parameters. The first is the text we wish to send in the message box and the second is the title for the message box. Comment out the line event.Skip() and put in the following line.

wx.MessageBox('You Clicked the button', 'Info')

Save and click the Run button (yellow arrow). You should see something like this:

And when you click the button you should see something like this:

Understand here that this is just about the simplest way to call the messagebox routine. You can have more parameters as well.

Here's a quick rundown on how to change the way the icons work on the message box (more next time).

wx.ICON_QUESTION - Show a question icon
wx.ICON_EXCLAMATION - Show an alert icon
wx.ICON_ERROR - Show an error icon
wx.ICON_INFORMATION - Show an info icon

The way to write this would be

wx.MessageBox('You Clicked the button', 'Info', wx.ICON_INFORMATION)

or whatever icon you wanted to use that suited the situation. There are also various button arrangement assignments which we'll talk about next time.

So, until next time, play with some of the various controls, placements, and so on. 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.
with the exception of Shell and Explorer by using the (Ctrl-W) key combination. This ensures that we will be starting totally fresh. Now create a new project by clicking on the wx.App button (see last time’s article if needed).

Before you do anything else, save Frame1 as “FrameMain.py” and then save App1 as “Gui2.py”. This is important. With the GUI2 tab selected in the Editor frame, move to the Toolbar frame, go back to the New tab, and add another frame to our project by clicking on wx.Frame (which is right next to the wx.App button). Make sure that the Application tab shows both frames under the Module column. Now go back to the new frame and save it as “FrameSecond.py”:

Next, open FrameMain in the designer. Add a wx.Panel to the frame. Resize it a bit to make the panel cover the frame. Next we are going to change some properties— we didn’t do this last time. In the inspector frame, make sure that the Constr tab is selected and set the title to “Main Frame” and the name to “FrameMain”. We’ll discuss naming conventions in a bit. Set the size to 400*340 by clicking on the Size check box. This drops down to show height and width. Height should be 400 and width should be 340:

Now click on the Props tab. Click on the Centered property and set it to wx.BOTH. Click the post check-mark and save your work. Now run your application by clicking on the button with the yellow arrow. Our application shows up in the center of the screen with the title of “Main Frame”. Now close it by clicking on the “X” in the upper right corner of the app.

Bring FrameMain back into the designer. Add two wx.Button to the frame, one above the other, and close to the center of the frame. Select the top button, name that “btnShowNew”, and set the label to “Show the other frame” in the Constr tab of the Inspector frame. Use the Shift+Arrow combination to resize the button so that all the text is visible, and then use the Ctrl+Arrow combination to move it back to the center of the frame. Select the bottom button, name that “btnExit”, and set the label to “Exit”.

I hope you’ve been playing with Boa Constructor since our last meeting. First we will have a very simple program that will show one frame, then allow you to click on a button that will pop up another frame. Last time we did a message box. This time we will do a totally separate frame. This can be helpful when doing an application with multiple frames or windows. So... here we go...
Post, save, and run to see your changes. Exit our app and go back to the designer. We are going to add button click events. Select the top button, and in the inspector frame, select the Evts tab. Click on ButtonEvent, then double click on wx.EVT_BUTTON. Notice you should have “OnBtnShowNewButton” below. Next, select the btnExit button. Do the same thing, making sure it shows “OnBtnExitButton”. Post and save. Next go to the Editor frame and scroll down to the bottom.

Make sure you have the two event methods that we just created. Here’s what the frame should look like so far:

Now it’s time to deal with our other frame. Open FrameSecond in the designer. Set the name to “FrameSecond”, and the title to “Second Frame”. Set centering to wx.BOTH. Add a wx.Button, and center it towards the lower part of the frame. Set the name to “btnFSExit”, and change the title to “Exit”. Set up a button event for it. Next add a wx.StaticText control in the upper portion of the frame close to the middle. Name it “stHiThere”, set the label to “Hi there...I’m the second form!”, and set the font to Sans, 14 point and weight to wxBOLD. Now reset the position to be centered in the form right and left. You can do this by unchecking the Position attribute and use the X position for right and left, and Y for up and down until you are happy. Post and save:

Now that we have designed our forms, we are going to create the “glue” that will tie all this together.

In the Editor frame, click on the GUI2 tab, then, below that, click on the Source tab. Under the line that says “import FrameMain”, add “import FrameSecond”. Save your changes. Next, select the “FrameMain” tab. Under the line that says “import wx”, add a line that says “import FrameSecond”. Next scroll down, and find the line that says “def __init__(self, parent)”. Add a line after the “self._init_ctrls(parent)” line that says “self.Fs = FrameSecond.FrameSecond(self)”. Now under the “def OnBtnShowNewButton(self, event):” event, comment out “event.Skip()” and add the following two lines:

```python
    self.Fs.Show()
    self.Hide()
```

Finally, under “OnBtnExitButton” method, comment out “event.Skip()”, and add a line that says “self.Close()”.

What does all this do? OK. The first thing we did was to make sure that the application knew we were going to have two forms in our app. That’s why we imported both FrameMain and FrameSecond in the GUI2 file. Next we imported a reference for FrameSecond into FrameMain so we can call it later. We initialized it in the “__init__” method. And in the “OnBtnShowNewButton” event we told it that when the button was clicked, we want to first show the second frame, and to hide the main frame. Finally we have the statement to close the application when the Exit button is clicked.

Now, switch to the code for FrameSecond. The changes here are relatively small. Under the “__init__” method, add a line that says “self.parent = parent” which adds a variable self.parent. Finally, under the click event for FSExitButton, comment out the “event.Skip()” line, and add the following two lines:

```python
    self.parent.Show()
    self.Hide()
```
PROGRAM IN PYTHON - PART 6

Remember we hid the main frame when we showed the second frame, so we have to re-show it. Finally we hide the second frame. Save your changes.

Here is all the code for you to verify everything (this page and the following page):

GUI2 code:

#!/usr/bin/env python
#Boa:App:BoaApp

import wx
import FrameMain
import FrameSecond

modules ={u'FrameMain': [1, 'Main frame of Application', u'FrameMain.py'], u'FrameSecond': [0, '', u'FrameSecond.py']}

class BoaApp(wx.App):
    def OnInit(self):
        self.main = FrameMain.create(None)
        self.main.Show()
        self.SetTopWindow(self.main)
        return True

    def main():
        application = BoaApp(0)
        application.MainLoop()

if __name__ == '__main__':
    main

Now you can run your application. If everything went right, you will be able to click on btnShownNew, and see the first frame disappear and second frame appear. Clicking on the Exit button on the second frame will cause that frame to disappear and the

FrameMain code:

#Boa:Frame:FrameMain

import wx
import FrameSecond

def create(parent):
    return FrameMain(parent)

[wxID_FRAMEMAIN, wxID_FRAMEMAINBTNEXIT, wxID_FRAMEMAINBTNSHOWNEW, wxID_FRAMEMAINPANEL1,]
    ] = [wx.NewId() for _initCtrls in range(4)]

class FrameMain(wx.Frame):
    def __initCtrls(self, prnt):
        # generated method, don't edit
        wx.Frame.__init__(self, id=wxID_FRAMEMAIN, name=u'FrameMain',
        parent=prnt, pos=wx.Point(846, 177), size=wx.Size(400, 340),
        style=wx.DEFAULT_FRAME_STYLE, title=u'Main Frame')
        self.SetClientSize(wx.Size(400, 340))
        self.Center(wx.BOTH)

        self.panell = wx.Panel(id=wxID_FRAMEMAINPANEL1, name='panell',
        parent=self, pos=wx.Point(0, 0), size=wx.Size(400, 340),
        style=wx.TAB_TRAVERSAL)

        self.btnShowNew =
        wx.Button(id=wxID_FRAMEMAINBTNSHOWNEW, label=u'Show the other frame',
        name=u'btnShowNew',
        parent=self.panell, pos=wx.Point(120, 103),
        size=wx.Size(168, 29),
        style=0)
        self.btnShowNew.SetBackgroundColour(wx.Colour(25, 175, 23))
        self.btnShowNew.Bind(wx.EVT_BUTTON, self.OnBtnShowNewButton,
        id=wxID_FRAMEMAINBTNSHOWNEW)
FrameMain Code (cont.):

```python
self.btnClose =
xw.Button(id=wxID_FRAMEMAINBTNEXIT, label=u'Exit',
          name=u'btnExit', parent=self.panell1,
          pos=wx.Point(162, 191),
          size=wx.Size(85, 29), style=0)
self.btnClose.SetBackgroundColour(wx.Colour(225, 218, 91))
self.btnClose.Bind(wx.EVT_BUTTON,
                   self.OnBtnExitButton,
                   id=wxID_FRAMEMAINBTNEXIT)

def __init__(self, parent):
    self._initCtrls(parent)
    self.Fs = FrameSecond.FrameSecond(self)

def OnBtnShowNewButton(self, event):
    #event.Skip()
    self.Fs.Show()
    self.Hide()

def OnBtnExitButton(self, event):
    #event.Skip()
    self.Close()

FrameSecond code:

#Boa:Frame:FrameSecond

import wx

def create(parent):
    return FrameSecond(parent)

[wxID_FRAMESECOND, wxID_FRAMESECONDBTNEXIT,
wxID_FRAMESECONDPANEL1,
wxID_FRAMESECONDSTATICTEXT1,]
    = [wx.NewId() for _initCtrls in range(4)]

class FrameSecond(wx.Frame):
    def __init_ctrls(self, parent, prnt):
        # generated method, don't edit
        wx.Frame.__init__(self, parent, id=wxID_FRAMESECOND,
                          name=u'FrameSecond',
                          parent=prnt, pos=wx.Point(849, 457),
                          size=wx.Size(419, 236),
                          style=wx.DEFAULT_FRAME_STYLE, title=u'Second Frame')
    self.SetClientSize(wx.Size(419, 236))
    self.Center(wx.BOTH)
    self.SetBackgroundStyle(wx.BG_STYLE_COLOUR)

    self.panell1 = wx.Panel(id=wxID_FRAMESECONDPANEL1,
                            name='panell',
                            parent=self, pos=wx.Point(0, 0),
                            size=wx.Size(419, 236),
                            style=wx.TAB_TRAVERSAL)

    self.btnExit =
xw.Button(id=wxID_FRAMESECONDBTNEXIT, label=u'Exit',
            name=u'btnFSExit', parent=self.panell1,
            pos=wx.Point(174, 180),
            size=wx.Size(85, 29), style=0)
self.btnExit.Bind(wx.EVT_BUTTON,
                   self.OnBtnFSExitButton,
                   id=wxID_FRAMESECONDBTNEXIT)

def __init__(self, parent):
    self._initCtrls(parent)
    self.parent = parent

def OnBtnFSExitButton(self, event):
    #event.Skip()
    self.parent.Show()
    self.Hide()
main frame to re-appear. Clicking on the Exit button on the main frame will close the application.

I promised you we’d discuss naming conventions. Remember way back, we discussed commenting your code? Well, by using well-formed names for GUI controls, your code is fairly self-documenting. If you just left control names as staticText1 or button1 or whatever, when you are creating a complex frame with many controls, especially if there are a lot of text boxes or buttons, then naming them something that is meaningful is very important. It might not be too important if you are the only one who will ever see the code, but to someone coming behind you later on, the good control names will help them out considerably. Therefore, use something like the following:

Control type - Name prefix
Static text - st_
Button - btn_
Text Box - txt_
Check Box - chk_
Radio Button - rb_
Frame - Frm_ or Frame_

You can come up with your own ideas for naming conventions as you grow as a programmer, and in some instances your employer might have conventions already in place.

Next time, we will leave GUI programming aside for a bit and concentrate on database programming. Meanwhile, get python-apsw and python-mysqldb loaded on your system. You will also need sqlite and sqlitebrowser for SQLite. If you want to experiment with MySql as well, that’s a good idea. All are available via Synaptic.

**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.
Program In Python - Part 7

Being a young boy, User had only a few pieces of paper. See-Quill said, “User, right now you could live with papers and file folders. However, I can get glimpses of the future, and you will someday have so many papers that they would, if placed on top of each other, be taller than you by 15 times. We should use my power.”

So, working together, User and See-Quill created a “database thingie” (a fairy technical term), and User lived happily ever after.

The End.

Of course, the story is not...
PROGRAM IN PYTHON - PART 7

completely true. However, using databases and SQL can make our lives easier. This time, we will learn about some simple SQL queries, and how to use them in a program. Some people might think that this might not be the “correct” way or the “best” way, but it is a reasonable way. So let’s begin.

Databases are like the filing cabinets in our story above. Data tables are like the file folders. The individual records in the tables are like the sheets of paper. Each piece of information is called a field. It falls together very nicely, doesn’t it? You use SQL (pronounced See-Quill) statements to do things with the data. SQL stands for Structured Query Language, and is basically designed to be an easy way to use databases. In practice, however, it can become very complicated. We will keep things pretty simple for this installment.

We need to create a plan, like starting any construction project. So, think of a recipe card, which is a good thing to think about, since we are going to create a recipe database program. Around my house, recipes come in various forms: 3x5 card, 8x10 pieces of paper, napkins with the recipe scribbled on it, pages from magazines, and even stranger forms. They can be found in books, boxes, binders, and other things. However, they all pretty much have one thing in common: the format. In almost every case, at the top you have the recipe title and maybe how many servings it makes and where it came from. The middle contains the list of ingredients, and the bottom contains the instructions - dealing with the order that things are done in, the cooking time, and so on. We will use this general format as the template of our database project. We will break this up into two parts. We’ll create the database this time, and the application to read and update the database next time.

Here’s an example. Let’s say we have the recipe shown right.

Notice the order we just discussed. Now when we design our database - we could make it very large and have one record for everything in the recipe. That, however, would be clumsy and hard to deal with. Instead, we are going to use the recipe card as a template. One table will handle the top of the card, or the gross information about the recipe; one table will handle the middle of the card, or the ingredients information; and one table will handle the bottom, or the instructions.

Make sure you have installed SQLite and APSW. SQLite is a small database engine that doesn’t require you to have a separate database server, which makes it ideal for our little application. Everything you learn here can be used with larger database systems like MySQL and others. The other good thing about SQLite is that it uses limited data types. These types are Text, Numeric, Blob, and Integer Primary Key. As you have learned already, text is pretty much anything. Our

Spanish Rice

Serves: 4

Source: Greg Walters

Ingredients:
1 cup parboiled Rice (uncooked)
1 pound Hamburger
2 cups Water
1 8 oz can Tomato Sauce
1 small Onion chopped
1 clove Garlic chopped
1 tablespoon Ground Cumin
1 teaspoon Ground Oregano
Salt and Pepper to taste
Salsa to taste

Instructions:
Brown hamburger.

Add all other ingredients.

Bring to boil.

Stir, lower to simmer and cover.

Cook for 20 minutes.

Do not look, do not touch.

Stir and serve.
ingredients, instructions, and the title of our recipe are all text types - even though they have numbers in them. Numeric datatypes store numbers. These can be integer values or floating point or real values. Blobs are binary data, and can include things like pictures and other things. Integer Primary Key values are special. The SQLite database engine automatically puts in a guaranteed unique integer value for us. This will be important later on.

APSW stands for Another Python SQLite Wrapper and is a quick way to communicate with SQLite. Now let's go over some of the ways to create our SQL statements.

To obtain records from a database, you would use the SELECT statement. The format would be:

```
SELECT [what] FROM [which table(s)] WHERE [Constraints]
```

So, if we want to get all the fields from the Recipes table we would use:

```
SELECT * FROM Recipes
```

If you wish to obtain just a record by its primary key, you have to know what that value is (pkID in this instance), and we have to include a WHERE command in the statement. We could use:

```
SELECT * FROM Recipes WHERE pkID = 2
```

Simple enough...right? Pretty much plain language. Now, suppose we want to just get the name of the recipe and the number of servings it makes - for all recipes. It's easy. All you have to do is include a list of the fields that you want in the SELECT statement:

```
SELECT name, servings FROM Recipes
```

To insert records, we use the INSERT INTO command. The syntax is:

```
INSERT INTO [table name] (field list) VALUES (values to insert)
```

So, to insert a recipe into the recipe table the command would be:

```
INSERT INTO Recipes
```

(name, servings, source) VALUES (“Tacos”, 4,”Greg”)

To delete a record we can use:

```
DELETE FROM Recipes WHERE pkID = 10
```

There's also an UPDATE statement, but we'll leave that for another time.

More on SELECT

In the case of our database, we have three tables, each can be related together by using recipeID pointing to the pkID of the recipe table. Let's say we want to get all the instructions for a given recipe. We can do it like this:

```
SELECT Recipes.name, Recipes.servings, Recipes.source, Instructions.Instructions FROM Recipes LEFT JOIN Instructions ON (Recipes.pkid = Instructions.recipeid) WHERE Recipes.pkid = 1
```

However, that is a lot of typing and very redundant. We can use a method called aliasing. We can do it like this:

```
SELECT r.name, r.servings, r.source, i.Instructions FROM Recipes r LEFT JOIN
instructions i ON (r.pkid = i.recipeid) WHERE r.pkid = 1
```

It's shorter and still readable. Now we will write a small program that will create our database, create our tables, and put some simple data into the tables to have something to work with. We could write this into our full program, but, for this example, we will make a separate program. This is a run-once program - if you try to run it a second time, it will fail at the table creation statements. Again, we could wrap it with a try...catch handler, but we'll do that another time.

We start by importing the APSW wrapper.

```
import apsw
```

The next thing we need to do is create a connection to our database. It will be located in the same directory where we
have our application. When we create this connection, SQLite automatically looks to see if the database exists. If so, it opens it. If not, it creates the database for us. Once we have a connection, we need what is called a cursor. This creates a mechanism that we can use to work with the database. So remember, we need both a connection and a cursor. These are created like this:

```python
# Opening/creating database
connection=apsw.Connection("cookbook1.db3")
cursor=connection.cursor()
```

Okay - we have our connection and our cursor. Now we need to create our tables. There will be three tables in our application. One to hold the gross recipe information, one for the instructions for each recipe, and one to hold the list of the ingredients. Couldn't we do it with just one table? Well, yes we could, but, as you will see, it will make that one table very large, and will include a bunch of duplicate information.

We can look at the table structure like this. Each column is a separate table as shown above right.

Each table has a field called pkID. This is the primary key that will be unique within the table. This is important so that the data tables never have a completely duplicated record. This is an integer data type, and is automatically assigned by the database engine. Can you do without it? Yes, but you run the risk of accidentally creating a duplicated record id. In the case of the Recipes table, we will use this number as a reference for which instruction and which set of ingredients go with that recipe.

We would first put the information into the database so that the name, source and number served goes into the recipe table. The pkID is automatically assigned. Let’s pretend that this is the very first record in our table, so the database engine would assign the value 1 to the pkID. We will use this value to relate the information in the other tables to this recipe. The instructions table is simple. It just holds the long text of the instructions, its own pkID and then a pointer to the recipe in the recipe table. The ingredients table is a bit more complicated in that we have one record for each ingredient as well as its own pkID and the pointer back to our recipe table record.

So in order to create the recipe table, we define a string variable called sql, and assign it the command to create the table:

```sql
sql = 'CREATE TABLE Recipes (pkID INTEGER PRIMARY KEY, name TEXT, servings TEXT, source TEXT)'
cursor.execute(sql)
```

Once we have the tables created, we will use the INSERT INTO command to enter each set of data into its proper table.

Remember, the pkID is
Why is this? Well, when we get data back from ASPW, it comes back as a tuple. This is something we haven't talked about yet. The quick explanation is that a tuple is (if you look at the code above) like a list, but it can't be changed. Many people use tuples rarely; others use them often; it's up to you. The bottom line is that we want to use the first value returned. We use the 'for' loop to get the value into the tuple variable x. Make sense? OK. Let's continue...

Next, we would create the insert statement for the instructions:

```python
sql = 'INSERT INTO Instructions (recipeID, instructions) VALUES( %s,"Brown hamburger. Stir in all other ingredients. Bring to a boil. Stir. Lower to simmer. Cover and cook for 20 minutes or until all liquid is absorbed.")' % lastid

cursor.execute(sql)
```

It's not too hard to understand at this point. Next time it will get a bit more complicated.

If you would like the full source code, I've placed it on my website. Go to [www.thedesignedagedgeek.com](http://www.thedesignedagedgeek.com) to download it.

Next time, we will use what we've learned over the series to create a menu-driven front end for our recipe program - it will allow viewing all recipes in a list format, viewing a single recipe, searching for a recipe, and adding and deleting recipes.

I suggest that you spend some time reading up on SQL programming. You'll be happy you did.

---

**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.
We will continue programming our recipe database that we started in Part 7. This will be a long one, with a lot of code, so grab on with all your might and don't let go. But remember, keep your hands and feet inside the car at all times. We have already created our database. Now we want to display the contents, add to it and delete from it. So how do we do that? We will start with an application that runs in a terminal, so we need to create a menu. We will also create a class that will hold our database routines. Let's start with a stub of our program shown above right.

Now we will layout our menu. We do that so we can stub our class. Our menu will be a rather big loop that will display a list of options that the user can perform. We'll use a while loop. Change the menu routine to look like the code shown below right.

Next we stub the menu with an if|elif|else structure which is shown at the top of the next page.

Let's take a quick look at our menu routine. We start off by printing the prompts that the user can perform. We set a variable (loop) to True, and then use the while function to continue looping until loop = False. We use the raw_input() command to wait for the user to select an option, and then

```python
#!/usr/bin/python
#----------------------------------------
# Cookbook.py
# Created for Beginning Programming Using Python #8
# and Full Circle Magazine
#----------------------------------------
import apsw
import string
import webbrowser

class Cookbook:
    def Menu():
        cbk = Cookbook() # Initialize the class

Menu()

def Menu():
    cbk = Cookbook() # Initialize the class
    loop = True
    while loop == True:
        print
        '=========================================
        print '
        print 'RECIPE DATABASE'
        print
        '=========================================
        print ' 1 - Show All Recipes'
        print ' 2 - Search for a recipe'
        print ' 3 - Show a Recipe'
        print ' 4 - Delete a recipe'
        print ' 5 - Add a recipe'
        print ' 6 - Print a recipe'
        print ' 0 - Exit'
        print
        '=========================================
        response = raw_input('Enter a selection -> ')
```
if response == '1':  # Show all recipes
    pass
elif response == '2':  # Search for a recipe
    pass
elif response == '3':  # Show a single recipe
    pass
elif response == '4':  # Delete Recipe
    pass
elif response == '5':  # Add a recipe
    pass
elif response == '6':  # Print a recipe
    pass
elif response == '0':  # Exit the program
    print 'Goodbye'
    loop = False
else:
    print 'Unrecognized command. Try again.'

if routine to handle whichever option the user selected. Before we can run this for a test, we need to create a stub inside our class for the __init__ routine:

```python
def __init__(self):
    pass
```

Now, save your program where you saved the database you created from the last time, and run it. You should see something like that shown above right.

It should simply print the menu over and over, until you type “0”, and then print “Goodbye” and exit.

At this point, we can now start stubs of our routines in the Cookbook class. We will need a routine that will display all the information out of the Recipes data table, one that will allow you to search for a recipe, one that will show the data for a single recipe from all three tables, one that will delete a recipe, one that will allow you to add a recipe, and one that will print the recipe to the default printer. The PrintAllRecipes routine doesn’t need a parameter other than the (self) parameter, neither does the SearchForRecipe nor the EnterNew routines. The PrintSingleRecipe, DeleteRecipe and PrintOut routines all need to know what recipe to deal with, so they will need to have a parameter that we’ll call “which”. Use the pass command to finish each stub.

Under the Cookbook class, create the routine stubs:

```python
def PrintAllRecipes(self):
    pass
def SearchForRecipe(self):
    pass
def PrintSingleRecipe(self, which):
    pass
def DeleteRecipe(self, which):
    pass
def EnterNew(self):
    pass
def PrintOut(self, which):
    pass
```

One more thing to do is to set up the __init__ routine. Replace the stub with the following lines:

```python
def __init__(self):
    global connection
    global cursor
    self.totalcount = 0
    connection=apsw.Connection("cookbook.db3")
    cursor=connection.cursor()
```

DeleteRecipe and PrintOut table – so the user can pick from that list. These will be options 1, 3, 4 and 6. So, modify the menu routine for those options, replacing the pass command with cbk.PrintAllRecipes(). Our response check routine will now look like the code at the top of the next page.
if response == '1': # Show all recipes
cbk.PrintAllRecipes()
elif response == '2': # Search for a recipe
  pass
elif response == '3': # Show a single recipe
  cbk.PrintAllRecipes()
elif response == '4': # Delete Recipe
  pass
elif response == '5': # Add a recipe
  cbk.PrintAllRecipes()
elif response == '6': # Print a recipe
  print 'Goodbye'
  loop = False
else:
  print 'Unrecognized command. Try again.'

cursor.execute(sql):
  cnt += 1
  print '%s %s %s %s '%
  %(
    str(x[0]).ljust(5),
    x[1].ljust(20),
    x[2].ljust(30),
    str(x[3]).ljust(30))
  print '-------------------'
  self.totalcount = cnt

The cnt variable will count the number of recipes we display to the user. Now our routine is done. Shown below is the full code for the routine, just in case you missed something.

Notice that we are using the tuple that is returned from the cursor.execute routine from ASPW. We are printing the pkID as the item for each recipe. This will allow us to select the correct recipe later on. When you run your program,
you should see the menu, and when you select option 1, you'll get what's shown at the top of the next page.

That's what we wanted, except if you are running the app in Dr.Python or the like, the program doesn't pause. Let's add a pause until the user presses a key so they can look at the output for a second or two. While we are at it, let's print out the total number of recipes from the variable we set up a moment ago. Add to the bottom of option 1 of the menu:

def PrintAllRecipes(self):
  print '%s %s %s %s '%
  %(
    'Item'.ljust(5),
    'Name'.ljust(30),
    'Serves'.ljust(20),
    'Source'.ljust(30))
  print '-------------------'
  sql = 'SELECT * FROM Recipes'
  cnt = 0
  for x in cursor.execute(sql):
    cnt += 1
    print '%s %s %s %s '%
    %(
      str(x[0]).rjust(5),
      x[1].ljust(30),
      x[2].ljust(20),
      x[3].ljust(30))
    print '-------------------'
    self.totalcount = cnt

First we create two global variables for our connection and cursor. We can access them from anywhere within the cookbook class. Next, we create a variable self.totalcount which we use to count the number of recipes. We’ll be using this variable later on. Finally we create the connection and the cursor.

The next step will be to flesh out the PrintAllRecipes() routine in the Cookbook class. Since we have the global variables for connection and cursor, we don’t need to re-create them in each routine. Next, we will want to do a “pretty print” to the screen for headers for our recipe list. We’ll use the “%s” formatting command, and the left justify command, to space out our screen output. We want it to look like this:

<table>
<thead>
<tr>
<th>Item</th>
<th>Serves</th>
<th>Source</th>
</tr>
</thead>
</table>

Finally, we need to create our SQL statement, query the database, and display the results. Most of this was covered in the article last time.

sql = 'SELECT * FROM Recipes'
cnt = 0
for x in cursor.execute(sql):
  cnt += 1
  print '%s %s %s %s '%
  %(
    str(x[0]).rjust(5),
    x[1].ljust(30),
    x[2].ljust(20),
    x[3].ljust(30))
  print '-------------------'
  self.totalcount = cnt
Enter a selection -> 1
Item Name | Serves | Source
--- | --- | ---
1 Spanish Rice | 4 | Greg
2 Pickled Pepper-Onion Relish | 9 half pints | Complete Guide to Home Canning

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 ->

print 'Total Recipes - %s' % cbk.totalcount
print '---------------------------------'
res = raw_input('Press A Key -> ')

We'll skip option #2 (Search for a recipe) for a moment, and deal with #3 (Show a single recipe). Let's deal with the menu portion first. We'll show the list of recipes, as for option 1, and then ask the user to select one. To make sure we don't get errors due to a bad user input, we'll use the TryExcept structure. We will print the prompt to the user (Select a recipe! ), then, if they enter a correct response, we'll call the PrintSingleRecipe() routine in our Cookbook class with the pkID from our Recipe table. If the entry is not a number, it will raise a ValueError exception, which we handle with the except ValueError: catch shown right.

Next, we'll work on our PrintSingleRecipe routine in the Cookbook class. We start with the connection and cursor again, then create our SQL statement. In this case, we use 'SELECT * FROM

Recipes WHERE pkID = %s" % str(which) where which is the value we want to find. Then we "pretty print" the output, again from the tuple returned by ASPW. In this case, we use x as the gross variable, and then each one with bracketed index into the tuple. Since the table layout is

try:
    res = int(raw_input('Select a Recipe -> '))
    if res <= cbk.totalcount:
        cbk.PrintSingleRecipe(res)
    elif res == cbk.totalcount + 1:
        print 'Back To Menu...'
    else:
        print 'Unrecognized command. Returning to menu.'
except ValueError:
    print 'Not a number... back to menu.'
out of the six finished. So, let’s deal with the search routine, again starting with the menu. Luckily this time, we just call the search routine in the class, so replace the pass command with:

cbk.SearchForRecipe()

Now to flesh out our search code. In the Cookbook class, replace our stub for the SearchForRecipe with the code shown on the next page.

There’s a lot going on there. After we create our connection and cursor, we display our search menu. We are going to give the user three ways to search, and a way to exit the routine. We can let the user search by a word in the recipe name, a word in the recipe source, or a word in the ingredient list. Because of this, we can’t just use the display routine we just created, and will need to create custom printout routines. The first two options use simple SELECT statements with an added twist. We are using the “like” qualifier. If we were using a query browser like SQLite Database Browser, our like statement uses a wildcard character of “%”. So, to look for a recipe containing “rice” in the recipe name, our query would be:

```python
def PrintSingleRecipe(self, which):
    sql = 'SELECT * FROM Recipes WHERE name like "%s"'
    str(which)
    print
    '~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
    for x in cursor.execute(sql):
        print x[0]
        print "Title: " + x[1]
        print "Serves: " + x[2]
        print "Source: " + x[3]
    print
    '~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
    sql = 'SELECT * FROM Ingredients WHERE RecipeID = %s' % recipeid
    print 'Ingredient List:'
    for x in cursor.execute(sql):
        print x[1]
    print 'Instructions:'
    sql = 'SELECT * FROM Instructions WHERE RecipeID = %s' % recipeid
    for x in cursor.execute(sql):
        print x[1]
    print
    '~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
    resp = raw_input('Press A Key -> ')
```

However, since the “%” character is also a substitution character in our strings, we have to use % in our text. To make it worse, we are using the substitution character to insert the word the user is searching for. Therefore, we must make it ‘%s%s%s’. Sorry if this is as clear as mud. The third query is called a Join statement. Let’s look at it a bit closer:

```python
sql = "SELECT r.pkid, r.name, r.servings, r.source, i.ingredian FROM Recipes r Left Join ingredients i on (r.pkid = i.recipeid) WHERE i.ingredian like ""%s%s"" GROUP BY r.pkid" %response
```

We are selecting everything from the recipe table, and the ingredients from the ingredients table, joining or relating the ingredient table ON the recipeID being equal to the pkID in the recipe table, then searching for our ingredient using the like statement, and, finally, grouping the result by the pkID in the recipe table to keep duplicates from being shown. If you remember, we have peppers twice in the second recipe (Onion and pepper relish), one green and one red. That could create confusion in our user’s mind. Our menu uses

```python
searchin = raw_input('Enter Search Type -> ')
if searchin != '4':
```

which says: if searchin (the value the user entered) is NOT equal to 4 then do the options, if it is 4, then don’t do
def SearchForRecipe(self):
    # print the search menu
    print '----------------------------------------'
    print '  Search in'
    print '----------------------------------------'
    print '  1 - Recipe Name'
    print '  2 - Recipe Source'
    print '  3 - Ingredients'
    print '  4 - Exit'

    searchin = raw_input('Enter Search Type -> ')
    if searchin != '4':
        if searchin == '1':
            search = 'Recipe Name'
        elif searchin == '2':
            search = 'Recipe Source'
        elif searchin == '3':
            search = 'Ingredients'

        parm = searchin
        response = raw_input('Search for what in %s (blank to exit) -> ' % search)

        if parm == '1':  # Recipe Name
            sql = 'SELECT pkid,name,source,servings FROM Recipes WHERE name like ' + '%s' % response
        elif parm == '2':  # Recipe Source
            sql = 'SELECT pkid,name,source,servings FROM Recipes WHERE source like ' + '%s' % response
        elif parm == '3':  # Ingredients
            sql = 'SELECT r.pkid,r.name,r.servings,r.source,i.ingredients FROM Recipes r Left Join ingredients i on (r.pkid = i.recipeid) WHERE i.ingredients like ' + '%s' % response

        try:
            if parm == '3':
                print '%s %.5s %.3s %.3s %.3s' % ('Item'.ljust(5), 'Name'.ljust(30), 'Serves'.ljust(20), 'Source'.ljust(30), 'Ingredient'.ljust(30))
            else:
                print '%s %.3s %.3s %.3s' % ('Item'.ljust(5), 'Name'.ljust(30), 'Serves'.ljust(20), 'Source'.ljust(30))
            print '----------------------------------------'

            for x in cursor.execute(sql):
                if parm == '3':
                    print '%s %.5s %.3s %.3s %.3s' % (str(x[0]).rjust(5), x[1].ljust(30), x[2].ljust(20), x[3].ljust(30), x[4].ljust(30))
                else:
                    print '%s %.3s %.3s %.3s' % (str(x[0]).rjust(5), x[1].ljust(30), x[3].ljust(20), x[2].ljust(30))

            except:
                print 'An Error Occurred'
        print '----------------------------------------'

    inkey = raw_input('Press a key')
Enter a selection -> 2
--------------------------------
Search in
--------------------------------
1 - Recipe Name
2 - Recipe Source
3 - Ingredients
4 - Exit
Enter Search Type -> 1
Search for what in Recipe Name (blank to exit) -> rice
Item Name Serves Source
--------------------------------
1 Spanish Rice 4 Greg
--------------------------------
Press a key

Easy enough. Now for the ingredient search...

Enter a selection -> 2
--------------------------------
Search in
--------------------------------
1 - Recipe Name
2 - Recipe Source
3 - Ingredients
4 - Exit
Enter Search Type -> 3
Search for what in Ingredients (blank to exit) -> onion
Item Name Serves Source Ingredient
--------------------------------
1 Spanish Rice 4 Greg 1 small
Onion chopped
2 Pickled Pepper-Onion Relish 9 half pints Complete Guide to Home Canning 6 cups
finely chopped Onions
--------------------------------
Press a key
for ingredients. We then ask the user to enter the title, source, and servings. We then enter a loop, asking for each ingredient, appending to the list. If the user enters 0, we exit the loop and continue on asking for the instructions. We then show the recipe contents and ask the user to verify before saving the data. We use INSERT INTO statements, like we did last time, and return to the menu. One thing we have to be careful of is the single quote in our entries. USUALLY, this won’t be a problem in the ingredient list or the instructions, but in our title or source fields, it could come up. We need to add an escape character to any single quotes. We do this with the string.replace routine, which is why we imported the string library. In the menu routine, put the code shown above right under option #4.

Then, in the Cookbook class, use the code shown below right for the DeleteRecipe() routine.

Quickly, we’ll go through the delete routine. We first ask the user which recipe to delete (back in the menu), and pass that pkID number into our delete routine. Next, we ask the user ‘are they SURE’ they want to delete the recipe. If the response is “Y” (string.upper(resp) == ’Y’), then we create the sql delete statements. Notice that this time we have to delete records from all three tables. We certainly could just delete the record from the recipes table, but then we’d have orphan records in the other two, and that wouldn’t be good. When we delete the record from the recipe table, we use the pkID field. In the other two tables, we use the recipeID field.

Finally, we will deal with the routine to print the recipes. We’ll be creating a VERY simple HTML file, opening the default browser and allowing them to print from there. This is why we are importing the webbrowser library. In the menu routine for option #6, insert the code shown at the top of the next page.

Again, we display a list of all the recipes, and allow them to select the one that they wish to print. We call the PrintOut routine in the Cookbook class. That code is shown at the top right of the next page.

We start with the fi.open([filename,’w’) command which creates the file. We then pull the information from the recipe table, and write it to the file with the fi.write command. We use the <H1></H1> header tag for the title, the
<H2> tag for servings and source. We then use the <li> list tags for our ingredient list, and then write the instructions. Other than that it's simple queries we've already learned. Finally, we close the file with the fi.close() command, and use webbrowser.open([filename]) with the file we just created. The user can then print from their web browser - if required.

**WHEW!** This was our biggest application to date. I've posted the full source code (and the sample database if you missed last month) on my website. If you don't want to type it all in or have any problems, then hop over to my website, [www.thedesignatedgeek.com](http://www.thedesignatedgeek.com) to get the code.

cbk.PrintAllRecipes()
print '0 - Return To Menu'
try:
    res = int(raw_input('Select a Recipe to DELETE or 0 to exit -> '))
    if res != 0:
        cbk.PrintOut(res)
    elif res == '0':
        print 'Back To Menu...' 
    else:
        print 'Unrecognized command. Returning to menu.'
except ValueError:
    print 'Not a number...back to menu.'

def PrintOut(self,which):
    fi = open('recipeprint.html','w')
    sql = "SELECT * FROM Recipes WHERE pkID = %s" % which
    for x in cursor.execute(sql):
        RecipeName = x[1]
        RecipeSource = x[3]
        RecipeServings = x[2]
    fi.write("<H1>%s</H1>" % RecipeName)
    fi.write("<H2>Source: %s</H2>" % RecipeSource)
    fi.write("<H2>Servings: %s</H2>" % RecipeServings)
    fi.write("<H3>Ingredient List: </H3>"
    sql = 'SELECT * FROM Ingredients WHERE RecipeID = %s' % which
    for x in cursor.execute(sql):
        fi.write("<li>%s</li>" % x[1])
    fi.write("<H3>Instructions: </H3>"
    sql = 'SELECT * FROM Instructions WHERE RecipeID = %s' % which
    for x in cursor.execute(sql):
        fi.write(x[1])
    fi.close()
    webbrowser.open('recipeprint.html')
print "Done"

---

**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.
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

Ahh...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 as a standalone app. Notice we use ‘\n’ to force a newline 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:

As we did last time, we

def usage():
    message = {
        '=================================================================
        "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        Usage: \n        \"t{0} <foldername>\n        \"t WHERE <foldername> is the path to your MP3 files.\n        Author: Greg Walters\n        'For Full Circle Magazine\n        '=================================================================
        ).format(sys.argv[0])
    }
    error(message)
sys.exit(1)
def main():
    global connection
    global cursor
    #----------------------------------------
    if len(sys.argv) != 2:
        usage()
    else:
        StartFolder = sys.argv[1]
        if not os.path.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!"

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

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
program in python - part 9

the cursor.execute statement. According to the ASPW website, 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 ValueErrors 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(replacement values)
```

We are using the basic syntax for the file.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**
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 <Node> 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 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:

```python
<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>
```

```python
import elementtree.ElementTree as ET
tree = ET.parse('xmlsampl e1.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
parts, and dump it out as it is in memory. Nothing fancy here.

Now, replace your code with the following:

```python
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:

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

Element: firstname - Data: Samantha
Element: lastname - Data: Pharch
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
```

Then we 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 worldwide, 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/](http://www.geocaching.com/) and the other is [http://navicache.com/](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
<?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
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.

```python
import
xml.etree.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".

```python
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.

```python
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.

```python
<name
id="N02CAC" xsi:nil="true"><! [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 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:endposition]
```

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:

```
# 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

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.

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.thedesigmenteedge.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!

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 "finished"
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/auto/wui/geo/WXCurrentObXML/index.xml?query=80013

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
currents 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.

You will remember from

""" 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 get_currents(self, debuglevel, Location):
    pass
def output(self):
    pass
def DoIt(self, Location):
    pass

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

```python
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 early code. If, when you are all happy with the way your code is working, you can remove anything related to debug level. 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

```python
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:])
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

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
        urlib.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.observe = tim.text
    # Get Current conditions
    for weather in tree.findall("//weather"):
        self.weather = weather.text
    # Get Temp
    for TempF in tree.findall("//temperature_string"):
        self.Temp = TempF.text
    # Get Humidity
    for hum in tree.findall("//relative_humidity"):
        self.humidity = 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.baro = pressure.text

gGetCurrents 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.temp
    print 'Barometric Pressure - %s' % self.bar
    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!

---

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.

---

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.

---

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://fullcirem Magazine.org/
one for the forecast. Here is the link to the forecast XML page:

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

```
<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>
    <period>2</period>
    <icon>ms</icon>
    <icons></icons>
    <title>Morning</title>
    <fcttext>
      Mostly cloudy with a mild chance of showers.
      Winds 5 to 10 mph shifting to the southwest.
      Lows in the mid 40s.
    </fcttext>
  </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:

```python
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.

```python
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

```xml
<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>
      <ampm>PM</ampm>
      <tz_short>MDT</tz_short>
      <tz_long>America/Denver</tz_long>
    </date>
    <precip>
      <fahrenheit>0</fahrenheit>
      <celsius>0</celsius>
    </precip>
    <temp>
      <fahrenheit>92</fahrenheit>
      <celsius>33</celsius>
    </temp>
    <low>
      <fahrenheit>58</fahrenheit>
      <celsius>14</celsius>
    </low>
    <high>
      <fahrenheit>92</fahrenheit>
      <celsius>33</celsius>
    </high>
    <icon>partlycloudy</icon>
    <icons>
      <skyicon>partlycloudy</skyicon>
    </icons>
    <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)
        elif 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)

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:

```python
def DoIt(self, Location, US, IncludeToday, Output):
    self.GetForecastData(Location)
    self.output(US, IncludeToday, Output)
```

Now we can call the routine as follows:

```python
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/wsSXMXQx

**Have fun until next time.**
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 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 “curs盛宴1.py” so that we can work on the file and keep the template.

```python
#!/usr/bin/env python
# CursesExample2
import curses

# MAIN LOOP
#================================================
try:
    myscreen = curses.initscr()
    myscreen.clear()
    myscreen.addstr(0, 0, "1 2 3 4 5 6 7")
    myscreen.addstr(1, 0, "1234567890123456789012345678901234567890")
    myscreen.addstr(10, 0, "10")
    myscreen.addstr(20, 0, "20")
    myscreen.addstr(23, 0, "Press Any Key to Continue")
    myscreen.refresh()
    myscreen.getch()
finally:
    myscreen.border()
```

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

def InitScreen BORDER:
    if BORDER == 1:
        myscreen.border()

# 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()
```

---

**RECIPE DATABASE**

<table>
<thead>
<tr>
<th>1 - Show All Recipes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 - Search for a recipe</td>
</tr>
<tr>
<td>3 - Show a Recipe</td>
</tr>
<tr>
<td>4 - Delete a Recipe</td>
</tr>
<tr>
<td>5 - Add a recipe</td>
</tr>
<tr>
<td>6 - Print a recipe</td>
</tr>
<tr>
<td>0 - Exit</td>
</tr>
</tbody>
</table>

Enter a selection ->
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.

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()
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.

```python
def SearchForAResultMenu():
    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'
dolooop = 1
while dolooop == 1:
    key = myscreen.getch(11,22)
    myscreen.addch(11,22,key)
    if key == ord('1'):
        ShowAllRecipesMenu()
        DoMainMen()u()
    elif key == ord('2'):
        SearchForAResultMenu()
        InKey2()
        DoMainMenu()
    elif key == ord('3'):
        ShowAResultMenu()
        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()
```
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.

---

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:

```python
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:

```python
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 colorset1.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 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 loose 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:

```python
import curses

def main(stdscr):
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)
stdscr.clear()
stdscr.addstr(3,1, " This is a test ",curses.color_pair(1))
stdscr.addstr(4,1, " This is a test ",curses.color_pair(2))
stdscr.addstr(5,1, " This is a test ",curses.color_pair(3))
stdscr.refresh()
stdscreen.getch()
curses.wrapper(main)
```

```python
import curses
import random
class Game1():
    def __init__(self):
        pass

def main(self,scr):
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_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()
```

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

---

The Compleat Python
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
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 later 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
        END
        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
        END
    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
        END
        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.
            END
            Pick a new letter and position and start everything over again.
        END
    END
ELSE
    Increment the loop counter
    Refresh the screen.

def Explode(self, scrn):
    pass

def ResetForNew(self):
    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)

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.
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

```python
def GameLoop(self, scrn):
    test = 1          # Set the loop
    while test == 1:
        curses.napms(20)
        self.MoveLetter(scrn)
        keyin = scrn.getch(scrn.ScoreLine, scrn.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()

# New Game

def NewGame(self, scrn):
    self.GunChar = curses.ACS_SSBS
    scrn.nodelay(1)      # Don't wait for a keystroke...just cache it.
    #self.ResetForNew()
    self.GameScore = 0
    self.Lives = 3
    self.PrintScore(scrn)

# New Game

def PrintScore(self, scrn):
    scrn.addstr(self.ScoreLine, self.ScorePosition, "SCORE: %d" % self.GameScore)
    scrn.addstr(self.ScoreLine, self.LivesPosition, "LIVES:
```
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

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.

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
Program In Python - Part 15

#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

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
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:

```python
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 two lines:

```python
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 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.thedesignedgedge.com or at http://fulcirclemagazine.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
```
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])
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:

```python
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.

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

```python
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.

```python
>>> 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**

```python
>>> 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**

```python
>>> 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
```
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:

```python
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:

```python
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.

```python
File "pprint1.py", line 18
print TopOrBottom('=',40)
```

```
#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
    # leftbit is thing to print on left, val2 is thing to print on right
    # leftbit is width of left portion, rightbit width of right portion
    part2 = '%.2f' % val2
    return '%s%s%s % ('+',val1.1just(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)
```

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

```
SyntaxError: invalid syntax
```
program in python - part 16

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

```
$ cp pprint.py pprrintv3.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...
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)

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.
As I was finishing up the last installment of our series, I got an email about a programming competition. While we don’t have time to deal with this one, various sites have programming competitions throughout the year. The competition information can be found at http://www.freiesmagazin.de/third_programming_contest - if you are interested. That made me realize that we haven’t talked about true Client/Server programming. So with that in mind, we’ll dig into this topic, and see where we can go with it.

So, what is a Client/Server Application? In very simple terms, anytime you use a program (or even a web interface) that accesses data from another application or computer, you are using a client/server system. Let’s look at an example that we actually used before. Remember when we made our cookbook program? That was a VERY simple example (and not a very good one) of a client/server application. The SQLite database is the server, the application we wrote is the client. A better example would be the following. There is a database on a computer in another part of your office, floors away. It holds information on the inventory of the store you work at. You use a point of sale register (one of 10) within the store. Each of those registers are a client and the database located somewhere is the server.

While we won’t try to create that kind of system here, we can learn some of the basics.

The first thing we need to think about is the location of our server. Many people have only one computer in their house. Some people might have 7 or 8.

To use a client/server system, we have to connect from the client machine to the server machine. We do this with what is called a pipe or socket. If you ever made a “tin can” telephone when you were a kid, you have an idea of what I’m going to be talking about. If not, let me paint you a picture of times gone by. First, you had to get your mother to save you two tin cans from beans or something. Then you cleaned them carefully, and took them out to the garage. You used a small nail and a hammer to poke a small hole in the bottom of each. Then you got about 15 feet of string (again from your loving mother), ran the end of the string through each can, and tied a large knot in each end of the string to hold it inside the can. You then got your best buddy, and stretched the string tightly and yelled into the can while your friend held his can up to his ear. The vibrations from the bottom of the can went through the taut string, and caused the other can bottom to vibrate. Of course, you could hear without the can, but that was beside the point. It was cool. The socket is about the same thing. The client has a direct connection (think of the string) to the server. If many clients are connecting to the server, each client would have a tin can of their own, and the poor server has to have the same number of tin cans all held tightly to each client’s string phone. The bottom line here is each client has its own direct line to the server.

Let’s make a simple server and client. We’ll start with the server first. In pseudo code, here’s what happens.

Create a socket
Get name of server machine
Select a port
Bind socket to address and port
Listen for a connection
If connected...
   Accept the connection
   Print we got a connection
   Close the connection

The actual code to our server is shown on the next page, bottom left.

So, we create the socket, get the hostname of the machine we are running the server on, bind the socket to the port, and start to listen. When we get a connection request, we accept it, we print the fact we are connected, send “Hello and Goodbye”, and close the socket.
Now we need to have a client to make the whole thing work (shown bottom right).

The code is almost like the server, but, in this case, we connect, print what we receive, and close the socket.

The output from the programs are very predictable. On the server side of things we get...

My hostname is earth

I'm now connected to ('127.0.1.1', 45879)

and on the client side we get...

Hello and Goodbye

So, it's pretty simple. Now let's do something a bit more realistic.

We'll create a server that actually will do something. The code for server version 2 can be found at: http://fullcirlcemagazine.pastebin.com/Az8vNUv7

Let's break it down. After our imports, we set up some variables. BUFSIZE holds the size of the buffer that we will use to hold the information that we receive from the client. We also set up the port we will listen on, and a list holding the host and port number.

We next create a class called ServCmd. In the __init__ routine, we create a socket, and bind the interface to that socket. In the run routine, we start listening, and wait for a command from the client.

When we do get a command from the client, we use the os.popen() routine. This basically creates a command shell and runs the command.

Next the client (above right), which is a good deal easier.
We'll skip everything here except the send command, since you now have enough information to figure it out on your own. The `conn.sendCmd()` line (line 31) sends a simple `ls -al` request. Here's what my responses look like. Yours will be somewhat different.

### Server:

```python
python server2.py
...listening
...connected: ('127.0.0.1', 42198)
Command received - ls -al
Command received - BYE
...listening
```

```python
python_client2a.py total 72 drwxr-xr-x 2 greg greg 4096 2010-11-08 05:49 .
drwxr-xr-x 5 greg greg 4096 2010-11-04 06:29 ..
-rw-r--r-- 1 greg greg 751 2010-11-08 05:31 client2a.py
-rw-r--r-- 1 greg greg 760 2010-11-08 05:28 client2a.py~
-rw-r--r-- 1 greg greg 737 2010-11-08 05:25 client2.py
-rw-r--r-- 1 greg greg 733 2010-11-08 04:37 client2.py~
-rw-r--r-- 1 greg greg 1595 2010-11-08 05:30 client2.pyc
-rw-r--r-- 1 greg greg 449 2010-11-08 05:30 server2.pyc
```

We can also connect from another machine without changes anywhere - with the single exception of the `conn = CmdLine('localhost')` (line 29) in the client program. In this case, change the `localhost` portion to the IP address of the machine that the server is running on. For my home setup, I use the following line:

```python
conn = CmdLine('192.168.2.12')
```

So, now we are able to send information back and forth from one machine (or terminal) to another.

Next time, we'll make our client/server applications much more robust.

---

**Ideas & Writers Wanted**

We’ve created Full Circle project and team pages on LaunchPad. The idea being that non-writers can go to the project page, click ‘Answers’ at the top of the page, and leave your article ideas, but **please be specific with your idea!** Don’t just put ‘server article’, please specify what the server should do!

Readers who fancy writing an article, but aren’t sure what to write about, can register on the Full Circle team page, then assign article ideas to themselves, and get writing! We do ask that **if you can’t get the article written within several weeks (a month at most) that you reopen the question** to let someone else grab the idea.

**Project page, for ideas:** https://launchpad.net/fullcircle
**Team page for writers:** https://launchpad.net/~fullcircle
last time, we created a very simple client/server system. This time, we are going to extend it a bit. The server is a tic-tac-toe (or naughts and crosses) board and checker. The client portion acts as the input/output.

We'll start by using the same server code as last time, and modifying it as we go. If you didn't save the code from then, go to http://fullcirclmagazine.pastebin.com/UhquVK4N, get the source code for this time, and follow along. The first change comes in the __init__ routine where we initialize two new variables, self.player and self.gameboard. The gameboard is a simple list of lists or a basic array. We can access it as follows (more visual than just the flat list). This list will hold our data. There are three possible entries per cell. "-" means the cell is empty. "X" means the cell is occupied by player 1 and "O" means the cell is occupied by player 2. The grid looks like this when put in two dimensions:

```
[0][0] | [0][1] | [0][2]  
[1][0] | [1][1] | [1][2]  
[2][0] | [2][1] | [2][2]  
```

So starting with the server code from last month, in the routine __init__ routine, add the following lines:

```python
# The next three lines are new...
self.player = 1
self.gameboard = [['-', '-', '-', '-', 'N', 'N'], ['-','-', '-', '-', 'N', 'N'], ['-','-', '-', '-', 'N', 'N']]
self.run()
```

The run, listen, and servCmd routines have no changes, so we'll concentrate on the changes to the procCmd routine next.

In last time's article, the server waited for a command from the client, then sent it to the os.popen routine. This time, we will parse the command sent in. In this case, we have three separate commands we will listen for. They are 'Start', 'Move', and 'GOODBYE'. When we receive the 'Start' command, the server should initialize the game board to all “-” and then send a "print out" of the board to the client.

The 'Move' command is a compound command, in that it contains the command, and the position that the player wants to move to. For example, 'Move A3'. We parse the command to get three parts, the 'move' command itself, and the the row and column. Finally the 'GOODBYE' command simply resets the game board for another game.

So, we receive the command from the client in the procCmd routine. We then check the command to see what we are supposed to do. Within the procCmd routine, find the 5th line down, and, after the line that says "if self.processingloop":, remove the rest of that set of code. Now we'll set up the commands as we laid the out. Here's the code for the Start command:

```python
if cmd[:4] == 'Move':
    print "MOVE COMMAND"
position = cmd[5:]  
if position[0] == 'A':
    row = 0
elif position[0] == 'B':
    row = 1
elif position[0] == 'C':
    row = 2
else:
    self.cli.send('Invalid position')
    return

col = int(position[1])-1
```

Next, let's look at the Move portion of the routine (shown below). We first check the first four characters of the passed-in command to see if they match 'Move'. If they match, we then pull the rest of the string starting at position 5 (since things are 0 based), and assign that to a variable named position. We then check to see if the first character is either an 'A', 'B', or 'C'. These represent the row that the client has sent. We then take the integer value of the next character and that's our column:
Next, we make a quick check to verify that the row position is within the allowable positions:

```python
if row < 0 or row > 2:
    self.cli.send('Invalid position')
    return
```

Finally, we verify that the position is empty ('-'), and if the current player is number 1, we put an "X" otherwise we put a "O". We then call the PrintGameBoard routine with a "0" parameter:

```python
if self.gameboard[row][col] == '-':
    if self.player == 1:
        self.gameboard[row][col] = 'X'
    else:
        self.gameboard[row][col] = 'O'
```

That finishes the changes to the procCmd routine. Next we have the "initialize the game board" routine. All it does is to set each position to a ".", which the move logic uses to verify that a space is empty:

```python
def InitGameBoard(self):
    self.gameboard = [['-', '-'], ['-', '-'], ['-', '-']]}
```

The PrintGameBoard routine (below) prints the game board, calls the checkwin routine, and sets the player number. We build a large string to send to the client so it only has to enter the listen routine once per move. The firsttime parameter is included to send the pretty print of the gameboard when the client first connects or resets the game:

```python
def PrintGameBoard(self, firsttime):
    #Print the header row
    outp = (' 1 2 3') + chr(13) + chr(10)
    outp += ('  A {0} | {1} | {2}' .format(self.gameboard[0][0], self.gameboard[0][1], self.gameboard[0][2])) + chr(13) + chr(10)
    outp += ('  -------------') + chr(13) + chr(10)
    outp += ('  B {0} | {1} | {2}' .format(self.gameboard[1][0], self.gameboard[1][1], self.gameboard[1][2])) + chr(13) + chr(10)
    outp += ('  -------------') + chr(13) + chr(10)
    outp += ('  C {0} | {1} | {2}' .format(self.gameboard[2][0], self.gameboard[2][1], self.gameboard[2][2])) + chr(13) + chr(10)
    outp += ('  -------------') + chr(13) + chr(10)

    if firsttime == 0:
        if self.player == 1:
            ret = self.checkwin("X")
        else:
            ret = self.checkwin("O")
        if ret == True:
            if self.player == 1:
                outp += "Player 1 WINS!"
            else:
                outp += "Player 2 WINS!"
        else:
            if self.player == 1:
                self.player = 2
            else:
                self.player = 1
            outp += ('Enter move for player %s' % self.player)
            self.cli.send(outp)
```

Next, we check to see if the firsttime parameter is set to 0 or 1 (below). Only if firsttime is set to 0, we check to see if the current player has won, and, if so, add the 'Player X WINS!' text to the output string. If the current player did not win, we then add the "Enter move..." text to the output string. Finally we send the string out to the client with the cli.send routine.

Finally, on the next page, we have the server check for a win routine. We have already set the player to either an "X" or "O", so we start by using a simple for loop. If we find a win, we return True from the routine. Our for variable 'C' represents each row in our list of lists. First, we will check each Row for a horizontal win:

```python
if firsttime == 0:
    if self.player == 1:
        ret = self.checkwin("X")
    else:
        ret = self.checkwin("O")
    if ret == True:
        if self.player == 1:
            outp += "Player 1 WINS!"
        else:
            outp += "Player 2 WINS!"
    else:
        if self.player == 1:
            self.player = 2
        else:
            self.player = 1
        outp += ('Enter move for player %s' % self.player)
        self.cli.send(outp)
```
First, we will check each Row for a horizontal win:

def checkwin(self, player):
    #loop through rows and columns
    for c in range(0,3):
        #check for horizontal line
        if self.gameboard[c][0] == player and
           self.gameboard[c][1] == player and
           self.gameboard[c][2] == player:
            print "**********\n\n%s wins\n\n**********" % player
            playerwin = True
            return playerwin

Next, we check each Column for a win:
    #check for vertical line
    elif self.gameboard[0][c] == player and
       self.gameboard[1][c] == player and
       self.gameboard[2][c] == player:
       print "** %s wins **" % player
       playerwin = True
       return playerwin

Now we check for the diagonal win from left to right...
    #check for diagonal win (left to right)
    elif self.gameboard[0][0] == player and
       self.gameboard[1][1] == player and
       self.gameboard[2][2] == player:
       print "** %s wins **" % player
       playerwin = True
       return playerwin

Then from right to left...
    #check for diagonal win (right to left)
    elif self.gameboard[0][2] == player and
       self.gameboard[1][1] == player and
       self.gameboard[2][0] == player:
       print "** %s wins **" % player
       playerwin = True
       return playerwin

Finally, if there is no win, we return False:
else:
    playerwin = False
    return playerwin

The Client

Once again, we start with the simple routine that we had last time. The changes start right after the call to conn.makeConnection. We send a Start, various Moves, and finally a Goodbye command. The biggest thing to remember here is that you must send a command, then get a response before sending another command. Think of it as a polite conversation. Make your statement, listen for a response, then make another statement, listen for a response, and so on. In this sample we use raw_input simply so you can see what is going on:

if __name__ == '__main__':
    conn = CmdLine('localhost')
    conn.makeConnection()
    conn.sendCmd('Start')
    conn.getResults()
    conn.sendCmd('Move A3')
    conn.getResults()
    r = raw_input("Press Enter")
    conn.sendCmd('Move B2')
    conn.getResults()
    r = raw_input("Press Enter")

    Continue the sendCmd, getResults, raw_input routine set with the following commands

    (you already have the code for the A3 and B2 moves), C1, A1, C3, B3, C2, then end with a GOODBYE command.

Moving Forward

So, here is your “homework” assignment. In the client app, remove the hard coded move commands, and use raw_input() to prompt for and get moves from the player(s) in the form of “A3” or “B2”, then prepend the command “Move” before sending it to the server.

Next time, we’ll modify our server to actually play the other player.

Server and Client Full Source Code can be found at http://fullcirclemagazine.pastebin.com/UhuVK4N or at http://thedesignatedgeek.com

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.
Program In Python - Part 19

This time, we are going to work on finishing our Tic-Tac-Toe program. However, unlike most of my other articles, I won't be providing the code. You will! I will however, be giving you the rules. After 18 months, you have the tools and knowledge to finish this project. I'm sure of it.

First, let's look at the logic of playing Tic-Tac-Toe. We'll look at it in pseudo-code. Let's look first at the game board. It's laid out like this...

```
<table>
<thead>
<tr>
<th>Corner</th>
<th>Side</th>
<th>Corner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Side</td>
<td>Center</td>
<td>Side</td>
</tr>
<tr>
<td>Corner</td>
<td>Side</td>
<td>Corner</td>
</tr>
</tbody>
</table>
```

Now, whoever is “X”, goes first. Their first best move is to take a corner square. Any corner square, it doesn't matter. We'll deal with the permutations of playing “X” first, these are shown right.

The standpoint of the “O” player is shown below right.

```
IF “O” takes a CORNER square THEN

# Scenario 1

“X” should take one of the remaining corner squares. Doesn't matter which.
IF “O” blocks the win THEN

“X” takes remaining corner square.
Finish for win.
ELSE

Finish for win.

ELIF “O” takes a SIDE square THEN

# Scenario 2

“X” takes CENTER square
IF “O” blocks win THEN

“X” takes corner square that is not bordered by any “O”
Finish for win.
ELSE

Finish for win.

ELSE

# “O” has played in the CENTER square - Scenario 3
“X” takes corner square diagonally to original move
IF “O” plays on corner square

“X” plays remaining open corner square
Finish for win.
ELSE

# Game will be a draw - Scenario 4
Block “O” win.
Block any other possible wins
```

Some possible play outs are shown on the next page.

As you can see, the logic is somewhat complex, but can easily be broken down in a series of IF statements (notice I used “Then”, but in Python, we don't, we use the “:”). You should be able to modify the code from last month to deal with this, or at least write one from scratch to simply be a desktop tic-tac-toe program.

```
IF “X” plays to non-center square THEN

“O” takes Center Square
IF “X” has corner square AND side square THEN

#Scenario 5
“O” takes corner diagonally from corner “X”
Block possible wins to a draw.
ELSE

# “X” has two Edge squares
- Scenario 6
“O” moves to corner bordered by both “X”s
IF “X” blocks win THEN

“O” takes any square.
Block and force draw
ELSE

Finish for win.
```

```
```
Scenario 1
X | - | - | - | - | - | - | X | - | - | - | - | - | - | - | X | - | - | - | - | - | - | - | X | X | X | X | X
- | - | - | - | - | - | - | O | - | - | - | - | - | - | - | O | - | - | - | - | - | - | - | O | O | O | O | -
Scenario 2
X | - | - | - | - | - | - | X | - | - | - | - | - | - | - | X | - | - | - | - | - | - | - | X | X | X | X | X
- | - | - | - | - | - | - | O | - | - | - | - | - | - | - | O | - | - | - | - | - | - | - | O | O | O | O | -
Scenario 3
X | - | - | - | - | - | - | X | - | - | - | - | - | - | - | X | - | - | - | - | - | - | - | X | X | O | X | X
- | - | - | - | - | - | - | O | - | - | - | - | - | - | - | O | - | - | - | - | - | - | - | O | O | O | O | -
Scenario 4
X | - | - | - | - | - | - | X | - | - | - | - | - | - | - | X | - | - | - | - | - | - | - | X | X | O | X | O
- | - | - | - | - | - | - | O | - | - | - | - | - | - | - | O | - | - | - | - | - | - | - | O | O | O | O | -
Scenario 5
X | - | - | - | - | - | - | X | - | - | - | - | - | - | - | X | - | - | - | - | - | - | - | X | X | X | X | X
- | - | - | - | - | - | - | O | - | - | - | - | - | - | - | O | - | - | - | - | - | - | - | O | O | O | O | -
Scenario 6
- | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | X | O | - | X | X
- | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | X | O | - | X | X
- | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | X | O | - | X | X
- | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | X | O | - | X | X

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.
Welcome back. This time we will re-address GUI programming, but this time we will be using the pyGTK library. We won't be working with a GUI designer right now, we'll just be working with the library.

Use Synaptic to install python-gtk2, python-gtk2-tutorial, and python-gtk2-doc.

Let's jump right in and make our first program using pyGTK, it's shown above right.

For awhile, we will be building on this simple code set. On line #3 is a new command. The line “pygtk.require('2.0')” means that the application will not run unless the pygtk module is at least version 2.0. In the __init__ routine, we assign a window to the self.window variable (line 8), and then show it (line 9). Remember that the __init__ routine is run as soon as we instantiate the class (line 13). Save this code as “simple1.py”.

Run it in a terminal. You'll see a simple window show up somewhere on your desktop. On mine, it shows up in the upper left corner of my desktop. In order to end the program, you have to hit Ctrl-C in the terminal. Why? We haven't added any code to destroy and actually end the app. That's what we'll do next. Add the following line before the self.window.show() line...

self.window.connect("delete_event", self.delete_event)

Then after the gtk.main() call, add the following routine...

def delete_event(self, widget, event, data=None):
    gtk.main_quit()
    return False

Now save your app as “simple2.py”, and, once again, run it from a terminal. Now, when you click the “X” on the title bar, the application will exit. What is actually happening here? The first line we added (self.window.connect...) connects the delete event to a callback routine, in this case self.delete_event. By returning “False” to the system, it also destroys the actual window from system memory.

Now, I don't know about you, but I prefer my applications to open in the center of the screen, not someplace random, or in a corner - where it might be obscured by something else. Let's modify the code to do this. All we need to do is add the following line before the self.window.connect line in the __init__ function:

self.window.set_position(gtk.WIN_POS_CENTER)

As you might guess, this sets the position of the window in the center of the screen. Save the app as “simple3.py” and run it.

That's much nicer, but there's not much there. So, let's try to add a widget. If you remember WAY back when we worked with Boa Constructor, widgets are simply predefined controls that we can add to our window to do things. One of the simplest controls to add is a button. We will add the following code right after the self.window.connect line in our previous code in the __init__ routine:
As you can see, the routine doesn’t do much. It prints in the terminal “Button 1 clicked”, and then calls the `gtk.main_quit()` routine. This will close the window and terminate the application - just as if you had clicked the “X” on the title bar. Again, save this as “simple4.py”, and run it in a terminal. You’ll see our centered window with a button that says “Close me”. Click on it, and the application closes, as designed. Notice, however, that the window is much smaller than it was in the simple3.py application. You can resize the application, but the button resizes with it. Why is this? Well, we simply shoved a button into the window and the window resized to fit the control.

We sort of broke the rules of GUI programming by putting the button directly on the form, without using a container. Remember back when we did our first series on GUI programming using Boa Constructor - we used sizer boxes (containers) to hold our controls. We should do this, even if we only have just one control. For our next example, we’ll add a HBox (Horizontal box) to hold our button, and add another button. If we wanted a vertical container, we would use a VBox.

To start, use “simple4.py” as our base code. Delete everything between the lines
self.window.connect(...) and self.window.show(). This is where we will add our new lines. The code for the HBox and our first button are...

```python
self.box1 = gtk.HBox(False, 0)
self.window.add(self.box1)
self.button =
gtk.Button("Button 1")
self.button.connect("clicked",
, self.btn1Clicked, None)
self.box1.pack_start(self.button,True,True,0)
self.button.show()
```

Breaking down this code, we add a HBox, naming it `self.box1`. The parameters we pass to the HBox are homogeneous (True or False), and a spacing value:

```
HBox =
gtk.HBox(homogeneous=False,
spacing=0)
```

---

**Ideas & Writers Wanted**

We’ve created Full Circle project and team pages on LaunchPad. The idea being that non-writers can go to the project page, click ‘Answers’ at the top of the page, and leave your article ideas, but please be specific with your idea! Don’t just put ‘server article’, please specify what the server should do!

Readers who fancy writing an article, but aren’t sure what to write about, can register on the Full Circle team page, then assign article ideas to themselves, and get writing! We do ask that if you can’t get the article written within several weeks (a month at most) that you reopen the question to let someone else grab the idea.

Project page, for ideas: https://launchpad.net/fullcircl
The homogeneous parameter controls whether each widget in the box has the same size (width in the case of an HBox and height in the case of a VBox.) In this case, we pass it false, and a spacing value of 0. Next, we add the box to the window. Now, we create the button as before, and connect the clicked event to our routine.

Now, we come to a new command. The self.box1.pack_start command is used to add the button to the container (HBox). We use this command instead of the self.window.add command for the widgets we want to be in the container. The command (as above) is...

```python
box.pack_start(widget, expand=True, fill=True, padding=0)
```

The pack_start command has the following parameters. First is the widget, next is expand (True or False), then fill (True or False), and a padding value. Spacing for the containers is the amount of space in between the widgets, and padding is for the right/left side of the widgets. The expand argument allows you to choose whether the widgets in the box will fill all the extra space in the box (True), or if the box shrinks to fit the widgets (False). The fill argument has an effect only if the expand argument is True. Finally we show the button. Next is the code for the second button:

```python
self.button2 =
gtk.Button("Button 2")
self.button2.connect("clicked ",self.btn2Clicked,None)
self.box1.pack_start(self.button2,True,True,0)
self.button2.show()
self.box1.show()  
```

Notice that this code is pretty much the same thing as the first button widget. The last line of this new code shows the box.

Now, we have to add the self.btn2Clicked routine. After the self.btn1Clicked routine, add the following code...

```python
def btn2Clicked(self,widget,data=None):
    print "Button 2 clicked"
    
and in the btn1Clicked routine, comment out the line:
    gtk.main_quit()  
```

We want both buttons to print their “Button X clicked” response without closing the window.

Save this as “simple4a.py”. Run it in a terminal. What you will see is a centered window with two buttons (right up to the edges of the window) marked “Button 1” and “Button 2”. Click on them and notice that they properly respond to the click event as we have discussed. Now, before closing the window, resize it (drag at the bottom right of the window), and notice that the buttons grow and shrink equally as you resize the window. To understand the expand parameter, change the code for the self.box1.pack_start from True to False in both lines. Re-run your program and see what happens. This time, the window starts out looking the same, but when you resize the window, the buttons stay the same width, and there is empty space to the right as you expand the window. Next, change the expand parameter back to True and set the fill parameter to False. Re-run and notice that the buttons stay the same width, but there is empty space to the left and right of the buttons as you resize the window. Remember the fill parameter doesn't do anything if the expand parameter is set to False.

Another way to pack widgets is by using a table. Many times, if everything you have can fit into a grid-like structure easily, then a table is your best (and easiest) bet. You can think of a table like a spreadsheet grid with rows and columns holding widgets. Each widget can take up one or more cells - as your application requires. Maybe the following diagram will help visualize the possibility. Here is a 2x2 grid:

```
0+-------------------+ 1+-------------------+
|                    |                    |
| 0+-------------------+ 1+-------------------+
|                    |                    |
| 2+-------------------+                    |
```

The Compleat Python
Into the first row, we will place two buttons. One in column 1 and one in column 2. Into the second row, we will place one button spanning both columns. Like this...

```
0+-------------------+ 2
| Button 1 | Button 2 |
1+-------------------+ 2
| Button 3 |
```

The only required parameters are the first 5. So, to attach a button to the table in row 0 column 0, we would use the following command...

```
table.attach(button,0,1,0,1)
```

If it were to be placed into row 0 column 1 (remember this is zero based) as button 2 is above, the call would be...

```
table.attach(buttonx,1,0,1,0)
```

...table.attach(buttonx,1,2,0,1)

Hopefully, this is as clear as mud for you now. Let's get started with our code, and you'll understand better. First the common part...

```python
# table1.py
import pygtk
pygtk.require('2.0')
import gtk
class Table:
    def __init__(self):
        self.window =
            gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.set_position(gtk.
WIN_POS_CENTER)
    self.window.set_title("Table Test 1")
    self.window.set_border_width(20)
    self.window.set_size_request(250, 100)
    self.window.connect("delete_event", self.delete_event)

    There are a couple of new things here that we need to discuss before we move on. Line 9 sets the title of the window to “Table Test 1”. We use the “set_border_width” call to give a border of 20 pixels around the entire window before any widgets are placed. Finally, we are forcing the window to 250 x 100 pixels using the “set_size_request” function. Makes sense so far? Now, we create the table and add it to the window...
```

```
table = gtk.Table(2, 2, True)
# Create a 2x2 grid
self.window.add(table)
```

Next, we create our first button, set up the event connection, attach it to the table grid point, and show it...

```
button1 = gtk.Button("Button 1")
button1.connect("clicked", self.callback,"button 1")
table.attach(button1,0,1,0,1)
button1.show()
```

Now button number 2...

```
button2 = gtk.Button("Button 2")
button2.connect("clicked", self.callback,"button 2")
table.attach(button2,1,2,0,1)
button2.show()
```

Almost exactly the same as button number 1, but notice the change in the table.attach call. Also notice that the routine we will be using for the event handling is called “self.callback”, and is the same for both buttons. That's good for now. You'll understand what we're doing in a moment.

Now for the third button. This will be our “Quit” button:

```
button3 = gtk.Button("Quit")
button3.connect("clicked", self.ExitApp,"button 3")
table.attach(button3,0,2,1,2)
button3.show()
```

Finally, show the table and the window. Also here is the main routine and the delete
routine we have used before:

```python
    def main(self):
        gtk.main()
    def delete_event(self, widget, event, data=None):
        gtk.main_quit()
        return False
```

Now for the fun part. For both button 1 and button 2, we set the event handler routine to “self.callback”. Here’s the code for that.

```python
    def callback(self, widget, data=None):
        print "%s was pressed" % data
```

What happens is that when the user clicks on the button, the click event is triggered, and the data that was provided when we set the event connection is sent in. For button 1, the data that will be sent is “button 1”, and for button 2 it is “button 2”. All we are doing here is printing “button x was pressed” into the terminal. I’m sure you can see that this could be a very useful tool when combined with a nicely structured IF | ELIF | ELSE routine.

Now to finish up, we have to define the “ExitApp” routine for when the “Quit” button is clicked...

```python
    def ExitApp(self, widget, event, data=None):
        print "Quit button was pressed"
        gtk.main_quit()
```

And now the final main code...

```python
    if __name__ == '__main__':
        table = Table()
        table.main()
```

Combine all this code into a single app called “table1.py”. Run it in a terminal.

So to recap, when we want to use pyGTK to create a GUI program, the steps are...

- Create the window.
- Create HBox(s), VBox(s) or Table(s) to hold your widgets.
- Pack or attach the widgets (depending on box or table).
- Show the widgets.
- Show the box or table.
- Show the window.

Now we have many of the tools and knowledge to go forward. All code is up on Pastebin at http://fullcirelemagazine.pastebin.com/wnzRsXn9. See you next time.
If you’ve been with me for a long while, you might remember back to parts 5 and 6. We talked about using Boa Constructor to design our GUI application. Well, this time, we are going to deal with Glade Designer. Different, but similar. You can install it from the Ubuntu Software Center: search for glade, and install GTK+ 2 User Interface Builder.

Just to let you know, this will be an application that we’ll need multiple parts of these tutorials to cover. The ultimate goal is to build a playlist maker for our MP3, and other media files. This portion of the tutorial will be focusing on the design portion. Next time, we’ll deal with the code that glues all the parts of the GUI together.

Now to start designing our application. When you first start the Glade designer, you will have a preferences window open (above). Select Libglade, and “inside toplevels”, then click close. This will give us our designer main window.

Let’s take a look at the main window (right). On the left is our toolkit, in the middle is the designer area, and on the right is our attribute and hierarchy areas.

In the toolkit area, find the group marked “Toplevels”, and click on the first tool there (if you hover over it, it should show “Window”). This will give us our blank window “canvas” that we will be working with.

Notice that, in the hierarchy area, you see window1 under the Widgets section. Now move down to the attributes section, change the name from window1 to MainWindow, and set the Window Title to “Playlist Maker v1.0”. Save what you have as “PlaylistMaker.glade”. Before we can move on, in the attributes section of the General tab, find the Window Position pulldown and set it to Center. Click the check box for Default Width, and set this to 650. Do the same for Default Height, but set it to 350. Next, click on the Common tab, and scroll down to the entry marked “Visible”. BE SURE TO SET THIS TO “YES” - otherwise your window won’t show. Finally, select the Signals tab, scroll down to the GtkObject section, and click the arrow pointing to the right. Under destroy, click the pulldown in the Handler column, and select “on_MainWindow_destroy” setting. This gives us an event that gets raised when the user closes our window by clicking on the “X” in the titlebar. One word of warning... After setting the destroy event, click somewhere above or below
to make the change take. This seems to be a bug in Glade Designer. Again, save your project.

Just as before when we were doing GUI design, we need to put our widgets in vboxes and hboxs. This is the hardest thing to remember when doing GUI programming. We will be adding a vertical box to hold our widgets in the window, so, on the toolbox under Containers, select Vertical Box (second icon from the left on the top row), and click in our blank window in the designer section. You will be presented with a pop up window that asks how many slots or items you want. The default is three, but we need five. The layout, from top to bottom, will be a toolbar, an area for a tree list control, two horizontal areas for labels, buttons and text entry boxes, and a status bar.

Now we can start adding our widgets. First, add a toolbar from the toolbox. It's the (in my setup) fourth icon on the second line under containers. Click in the topmost slot of the vbox. That slot will shrink and almost disappear. Don't worry, we'll get it back in a few minutes.

Next, we need to add a scrolled Window to the next slot down to hold our tree list. This will allow us to scroll within the tree list. So, find the Scrolled Window icon under the Containers section of the toolbox (second icon from the left on the fifth row on my setup), and click that into the second slot of the vbox. Next, we will add two horizontal boxes, one to each of the next slots. Each needs three slots. Finally, add a Status Bar to the bottom slot. This is under the Control and Display section of the toolbox near the bottom. Now your designer should look something like the image below.

Last, but not least, add a Tree View widget from the Control and Display section of the toolbox into the scrolled window widget. You'll get a pop-up asking which TreeView model you wish to use. Just click the "OK" button for now. We'll set that up later.

Now we need to concentrate on the Scroll Window for a second. Click on it in the hierarchy area. Scroll down in the General tab to the entry marked "Horizontal Scrollbar Policy". Change that to 'Always', and then do the same for the Vertical Scrollbar Policy. Save again.

OK, now let's concentrate on our toolbar. This area will be at the top of our application right under the title bar. It will hold various buttons for us that will do the majority of the work. We will use eleven buttons in the toolbar, and, from left to right, they are...

Add, Delete, Clear List, a Separator, Move To Top, Move Up, Move Down, Move to bottom, another Separator, About, and Exit.

Over on the hierarchy area, click on "toolbar1". That should highlight it. At the top of the Glade Designer is something that looks like a pencil. Click that. That brings up the tool bar editor. Click on the Hierarchy tab. You'll see something like this:

We will be adding all of our toolbar buttons from here. The steps will be:
• Click the Add Button.
• Change the name of the button.
• Modify the label of the button.
• Select the image.
This will be repeated for all eleven of our widgets. So, Click Add, then in the name box, type “tbtnAdd”. Scroll down to the Edit Label portion and type “Add” in the Label box, then a little further down under Edit Image, in the text box for Stock ID, use the pulldown to select “Add”. That takes care of our Add button. We named it “tbtnAdd” so we can reference it in our code later. The “tbtn” is shorthand for ‘Toolbar Button’.

This way, in our code, it’s easy to find and is fairly self documenting.

Now, we need to add the rest of the widgets to our tool bar. Add another button for Delete. This one will be named (as you might guess) “tbtnDelete”. Again, set the label and the icon. Next, add another button naming it “tbtnClearAll” and use the Clear icon. Now we want a Separator. So, click Add, under name type “Sep1” and in the pulldown for type, select Separator.

Add the rest of the widgets naming them “tbtnMoveToTop”, “tbtnMoveUp”, “tbtnMoveDown”, “tbtnMoveToBottom”, “Sep2”, “tbtnAbout” and “tbtnQuit”. I’m sure you can find the correct icons. Once you are finished, you can quit the hierarchy window and save your work. You should have something that looks like the image below.

Now, we need to set the event handlers for all the buttons we created. In the hierarchy area, select the tbtnAdd widget. This should highlight both the entry in the hierarchy and the button itself. Go back to the attributes section, select the Signals tab, and expand the GtkToolButton to reveal the clicked event. Under handler in the clicked event, as before, select “on_tbtnAdd_clicked”, then click above or below to force the change. Do this for all the other buttons we created - selecting the “on_tbtnDelete_clicked” event and so on. Remember to click off of it to force the change, and save your project. Our separators don’t need events, so just pass over them.

Next, we need to fill in our hboxes. The top hbox will contain (from left to right) a label, a text widget, and a button. In the toolbox, select the label widget (not the blue one), and put it in the left slot. Now put a Text Entry widget in the center slot and a button in the right slot. Do the same for the second hbox.

It’s now time to set our attributes for the widgets we just added. In the hierarchy area, select label1 under hbox1. In the attributes section, select the General tab, scroll down to “Edit label appearance” area, and set the label to read “Path to save file:”. Next, go to the Packing tab and set Expand to “No”. You might remember the discussion on packing from last month. Set the padding to 4, which gives a little bit of room on the left and right side of our label. Now select button1 and set the Expand under the Packing tab to “No” also. Go back to the General tab and set the name of our button to “tbtnGetFolder”. Notice that since this isn’t a toolbar button, we didn’t preface it with a ‘t’. Scroll down to the Label entry and enter “Folder...”. Then click on the Signals tab and set the button event of GtkButton_clicked to “on_tbtnGetFolder_clicked”. Before we set the attributes of the next set of widgets in the next hbox, we need to do one more thing. Select the hbox1 in the hierarchy area and under the Packing tab, set expand to “No”. This makes the hbox take up less space. Finally, set the name of the Text Entry widget to “txtPath”.

Now, do the same thing for hbox2, setting its Expand to “No”, then set the label text to “Filename:”, expand to “No”, padding to 4. Set the name of the button to “tbtnSavePlaylist”, its text to “Save Playlist File...”, its Expand attribute to “No”, set up its clicked event, and set the name of the Text Entry widget here to “txtFilename”. Once again, save everything.
So now our window should look something like the image below left.

All that is wonderful, but what did we really do? We can't run this as a program, since we don't have any code. What we have done is create an XML file called "playlistmaker.glade". Don't let the extension fool you. It's really an XML file. If you are very careful, you can open it with your favorite editor (gedit in my case) and look at it.

You'll see plain text describing our window and each widget with their attributes. For example, let's look at the code (above) for the main widget, the actual window itself.

You can see that the name of the widget is “MainWindow”, its title is “Playlist Maker v1.0”, the event handler, and so on.

Let's take a look the code (shown below) for one of our toolbar buttons.

```
<child>
  <widget class="GtkToolButton" id="tbtnAdd">
    <property name="visible">True</property>
    <property name="label" translatable="yes">Add</property>
    <property name="use_underline">True</property>
    <property name="stock_id">gtk-add</property>
    <signal name="clicked" handler="on_tbbtnAdd_clicked"/>
  </widget>
  <packing>
    <property name="expand">False</property>
    <property name="homogeneous">True</property>
  </packing>
</child>
```

Hopefully this is starting to make sense to you. Now, we need to write some code to allow us to see our hard work actually do something. Bring up your code editor and start with this...

```
Let's take a look the code (shown below) for one of our toolbar buttons.

article number 9, so if you don't have that on your system, refer back to that one. We'll need the mutagen import for next time, and the sys import is set so the system can exit properly on the last exception.

Next, we need to create our class that will define our window. This is shown above right.

Pretty much the same kind of thing we've done before. Notice the last two lines here. We are defining the glade file (self.gladefile) to be the name of the file we created in the Glade designer. Notice also that we didn't include a path, just a file name. If your glade file is going to reside somewhere away from your actual code, you must put a path as well. However, it's always smart to keep them together. Next, we define our window as self.wTree. We'll be referring to that every time we need to refer to the window. We are also saying that the file is an XML file, and the window we will be using is the one named “MainWindow”. You can

```
<article number 9, so if you don't have that on your system, refer back to that one. We'll need the mutagen import for next time, and the sys import is set so the system can exit properly on the last exception.

Next, we need to create our class that will define our window. This is shown above right.

Pretty much the same kind of thing we've done before. Notice the last two lines here. We are defining the glade file (self.gladefile) to be the name of the file we created in the Glade designer. Notice also that we didn't include a path, just a file name. If your glade file is going to reside somewhere away from your actual code, you must put a path as well. However, it's always smart to keep them together. Next, we define our window as self.wTree. We'll be referring to that every time we need to refer to the window. We are also saying that the file is an XML file, and the window we will be using is the one named “MainWindow”. You can

```

```
understand. I'm only going to give you two events for now (shown below)...

So we have two events: “on_MainWindow_destroy” and “on_tbtnQuit_clicked” are the keys in our dictionary. The data for our dictionary is “gtk.main_quit” for both entries. Whenever an event is triggered by our GUI, the system uses the event to find the key of our dictionary, then knows what routine to call - from the data segment. Next we need to connect the dictionary to the signal handler of our window. We do it with the following line of code.

```
self.wTree.signal_autoconnect(dict)
```

We're almost ready. We still need our main routine code:

```
if __name__ == "__main__":
    plm = PlayListMaker()
    gtk.main()
```

Save this file as "playlistmaker.py". Now you can run it (shown above right).

It doesn't do much right now, other than open and close properly. The rest is for next time. Just to whet your appetite, we'll be discussing the use of the TreeView, Dialog boxes, and adding a bunch more code. So tune in next time.

**Glade file:**

http://fullcirclemagazine.pastebin.com/YM6U0Ee3

**Python source:**

http://fullcirclemagazine.pastebin.com/wbfDmmBh

---

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,
Correction

Last month, in part 21, you were told to save what you have as "PlaylistMaker.glade", but, in the code, it was referred to as: "playlistmaker.glade". I'm sure you noticed that one has capitals and the other does not. The code will run only if you use both the call and file name with, or want to play - including the full path. There's also an extension that can be added before each entry that includes the length of the song, the album name the song comes from, the track number, and the song name. We'll bypass the extension for now and just concentrate on the basic version.

Here is an example of a M3U playlist file..

```
#EXTM3U
Adult Contemporary/Chris Rea/Collection/02 - On The Beach.mp3
Adult Contemporary/Chris Rea/Collection/07 - Fool (If You Think It's Over).mp3
Adult Contemporary/Chris Rea/Collection/11 - Looking For The Summer.mp3
```

All path names are relative to the location of the playlist file.

OK...now let's get to coding. Shown right is the opening of the source code from last month.

Now, we need to create an event handler routine for each

```
#!/usr/bin/env python
import sys
from mutagen.mp3 import MP3
try:
    import pyt
    pyt.require(“2.0”)
except:
    pass
try:
    import gtk
    import gtk.glade
except:
    sys.exit(1)

next the class definition
class PlayListCreator:
def __init__(self):
    self.gladefile = “playlistmaker.glade”
    self.wTree = gtk.glade.XML(self.gladefile,”MainWindow”)

and the main routine
if __name__ == “__main__”:
    plc = PlayListCreator()
    gtk.main()

Next, we have our dictionary which should go after the __init__ routine.
def SetEventDictionary(self):
    dict = {“on_MainWindow_destroy”: gtk.main_quit,
            “on_tbtnQuit_clicked”: gtk.main_quit,
            “on_tbtnAdd_clicked”: self.on_tbtnAdd_clicked,
            “on_tbtnDelete_clicked”: self.on_tbtnDelete_clicked,
            “on_tbtnClearAll_clicked”: self.on_tbtnClearAll_clicked,
            “on_tbtnMoveToTop_clicked”: self.on_tbtnMoveToTop_clicked,
            “on_tbtnMoveUp_clicked”: self.on_tbtnMoveUp_clicked,
            “on_tbtnMoveDown_clicked”: self.on_tbtnMoveDown_clicked,
            “on_tbtnAbout_clicked”: self.on_tbtnAbout_clicked,
            “on_btnGetFolder_clicked”: self.on_btnGetFolder_clicked,
            “on_btnSavePlaylist_clicked”: self.on_btnSavePlaylist_clicked}
of our events that we have set up. Notice that
on_MainWindow_destroy and
on_tbtnQuit_clicked are already
done for us, so we need to
have only 10 more (shown top
right). Just make stubs for now.

We'll modify these stubbed
routines in a few minutes. For
now, this should get us up and
running with an application,
and we can test things as we go.
But, we need to add one
more line to the __init__ routine
before we can run the app.
After the self.wTree line, add...

        self.SetEventDictionary()

Now, you can run
the application, see the window,
and click the Quit toolbar
button to exit the application
properly. Save the code as
"playlistmaker-1a.py" and give
it a try. Remember to save it in
the same folder as the glade
file we created last time, or
copy the glade file into
the folder you saved this code in.

We also need to define a few
variables for future use. Add
these after the
SetEventDictionary call in the

        self.CurrentPath = ""
        self.CurrentRow = 0
        self.RowCount = 0

        __init__ function.

Now, we will create a function
that allows us to display a
popup dialog box whenever we
need to give some information
to our user. There is a built-in
set of routines that we will use,
but we'll make a routine of our
own to make it easier for us. It
is the gtk.MessageDialog
routine, and the syntax is as
follows...

        gtk.MessageDialog(parent,flags,MessageType,Buttons,message)

        Some discussion is needed
before we go too much further.
The message type can be one
of the following...

        GTK_MESSAGE_INFO - Informational message
        GTK_MESSAGE_WARNING - Nonfatal warning message
        GTK_MESSAGE_QUESTION - Question requiring a choice
        GTK_MESSAGE_ERROR - Fatal error message

        And the button types are...

        GTK_BUTTONS_NONE - no buttons at all
        GTK_BUTTONS_OK - an OK button
        GTK_BUTTONS_CLOSE - a Close button
        GTK_BUTTONS_CANCEL - a Cancel button
        GTK_BUTTONS_YES_NO - Yes and No buttons
        GTK_BUTTONS_OKCANCEL - OK and Cancel Buttons

        Normally, you would use the
following code, or similar, to
create the dialog, display it,
wait for a response, and then
destroy it.

        dlg =

        gtk.MessageDialog(None,0,gtk.MESSAGE_INFO,gtk.BUTTONS_OK,"This is a test message...")
        response = dlg.run()
        dlg.destroy()

        However, if you want to display
a message box to the user
more than once or twice, that's
a LOT of typing. The general
rule of thumb is that if you
write a series of lines-of-code
more than once or twice, it's
usually better to create a
function and then call that.
Think of it this way: If we want
to display a message dialog to
the user, say ten times in your

full circle magazine 101 The Compleat Python
application, that’s 10 X 3 (or 30) lines of code. By making a function to do this for us (using the example I just presented), we would have 10 + 3 (or 13) lines of code to write. The more we call a dialog, the less code we actually have to type, and the more readable our code is. Our function (top right) will allow us to call any of the four message dialog types with just one routine using different parameters.

This is a very simple function that we would then call like this...

```python
def MessageBox(self, level, text):
    if level == "info":
        dlg = gtk.MessageDialog(None, 0, gtk.MESSAGE_INFO, gtk.BUTTONS_OK, text)
    elif level == "warning":
        dlg = gtk.MessageDialog(None, 0, gtk.MESSAGE_WARNING, gtk.BUTTONS_OK, text)
    elif level == "error":
        dlg = gtk.MessageDialog(None, 0, gtk.MESSAGE_ERROR, gtk.BUTTONS_OK, text)
    elif level == "question":
        dlg = gtk.MessageDialog(None, 0, gtk.MESSAGE_QUESTION, gtk.BUTTONS_YES_NO, text)
    if level == "question":
        resp = dlg.run()
        dlg.destroy()
        return resp
    else:
        resp = dlg.run()
        dlg.destroy()
```

You can see how you can check the value of the button returned. So now, replace the "pass" call in each of our event handler routines with something like that shown below right.

```python
def on_btnAdd_clicked(self, widget):
    self.MessageBox("info","Button Add was clicked...")
def on_btnDelete_clicked(self, widget):
    self.MessageBox("info","Button Delete was clicked...")
def on_btnClearAll_clicked(self, widget):
    self.MessageBox("info","Button ClearAll was clicked...")
def on_btnMoveToTop_clicked(self, widget):
    self.MessageBox("info","Button MoveToTop was clicked...")
def on_btnMoveUp_clicked(self, widget):
    self.MessageBox("info","Button MoveUp was clicked...")
def on_btnMoveDown_clicked(self, widget):
    self.MessageBox("info","Button MoveDown was clicked...")
def on_btnMoveToBottom_clicked(self, widget):
    self.MessageBox("info","Button MoveToBottom was clicked...")
def on_btnAbout_clicked(self, widget):
    self.MessageBox("info","Button About was clicked...")
def on_btnGetFolder_clicked(self, widget):
    self.MessageBox("info","Button GetFolder was clicked...")
def on_btnSavePlaylist_clicked(self, widget):
    self.MessageBox("info","Button SavePlaylist was clicked...")
```

We won’t keep it like this, but this gives you a visual indication that the buttons work the way we want. Save...
the code now as "playlistmaker-1b.py", and test your program.
Now we are going to create a function to set our widget
references. This routine is going to be called only once, but it
will make our code much more manageable and readable.
Basically, we want to create
local variables that reference
the widgets in our glade
window - so we can make calls
to them whenever (if ever) we
need to. Put this function
(above right) below the
SetEventDictionary function.

Please notice that there is one
thing that isn't referenced in
our routine. That would be the
treeview widget. We'll make
that reference when we set up
the treeview itself. Also of note
is the last line of our routine. In
order to use the status bar, we
need to refer to it by its context
id. We'll be using this later on.

Next, let's set up the function
that displays the “about” dialog
when we click the About
toolbar button. Again, there is a
built-in routine to do this
provided by the GTK library. Put
this after the MessageBox

function. Here's the code,
below right.

Save your code and then
give it a try. You should
see a pop-up box,
centered in our
application, that displays
everything we have set.
There are more attributes
that you can set for the
about box (which can be
found at
http://www.pygtk.org/docs
/pygtk/class-
gtkaboutdialog.html), but
these are what I would
consider a minimum set.

```python
def SetWidgetReferences(self):
    self.txtFilename = self.wTree.get_widget("txtFilename")
    self.txtPath = self.wTree.get_widget("txtPath")
    self.btnAdd = self.wTree.get_widget("btnAdd")
    self.btnDelete = self.wTree.get_widget("btnDelete")
    self.btnClearAll = self.wTree.get_widget("btnClearAll")
    self.btnExit = self.wTree.get_widget("btnQuit")
    self.btnExit = self.wTree.get_widget("btnAbout")
    self.btnClose = self.wTree.get_widget("btnClose")
    self.btnMoveToTop = self.wTree.get_widget("btnMoveToTop")
    self.btnMoveUp = self.wTree.get_widget("btnMoveUp")
    self.btnMoveDown = self.wTree.get_widget("btnMoveDown")
    self.btnMoveToBottom = self.wTree.get_widget("btnMoveToBottom")
    self.btnGetSize = self.wTree.get_widget("btnGetSize")
    self.btnSavePlaylist = self.wTree.get_widget("btnSavePlaylist")
    self.btnCancel = self.wTree.get_widget("btnCancel")
    self.context_id = self.sbar.get_context_id("Statusbar")

    self.SetWidgetReferences()

def ShowAbout(self):
    about = gtk.AboutDialog()
    about.set_program_name("Playlist Maker")
    about.set_version("1.0")
    about.set_copyright("(c) 2011 by Greg Walters")
    about.set_comments("Written for Full Circle Magazine")
    about.set_website("http://thedesignatedgeek.com")
    about.run()
    about.destroy()

Now, comment out (or simply remove) the messagebox call in
the on_btnAbout_clicked routine, and replace it with a call to the
ShowAbout function. Make it look like this.

```python
def on_btnAbout_clicked(self,widget):
    self.MessageBox("info","Button About was clicked...")
    self.ShowAbout()
```
Before we go on, we need to discuss exactly what will happen from here. The general idea is that the user will click on the "Add" toolbar button, we'll pop up a file dialog box to allow them to add files to the playlist, and then display the file information into our treeview widget. From there, they can add more files, delete single file entries, delete all file entries, move a file entry up, down, or to the top or down to the bottom of the treeview. Eventually, they'll set the path that the file will be saved to, provide a filename with a "m3u" extension, and click the save file button. While this seems simple enough, there's a lot that happens behind the scenes. The magic all happens in the treeview widget, so let's discuss that. This will get pretty deep, so you might want to read carefully, since an understanding of this will keep you from making mistakes later on.

A treeview can be something as simple as a columnar list of data like a spreadsheet or database representation, or it could be more complex like a file-folder listing with parents and children, where the folder would be the parent and the files in that folder would be the children, or something even more complex. For this project, we'll use the first example, a columnar list. In the list, there will be three columns. One is for the name of the music file, one is for the extension of the file (mp3, ogg, wav, etc) and the final column is for the path. Combining this into a string (path, filename, extension) gives us the entry into the playlist we will be writing. You could, of course, add more columns as you wish, but for now, we'll deal with just three.

A treeview is simply a visual storage container that holds and displays a model. The model is the actual "device" that holds and manipulates our data. There are two different pre-defined models that are used with a treeview, but you can certainly create your own. That having been said, for 98% of your work, one of the two pre-defined models will do what you need. The two types are GTKListStore and GTKTreeStore. As their names suggest, the ListStore model is usually used for lists, the TreeStore is used for Trees. For our application, we will be using a GTKListStore.

The basic steps are:

- Create a reference to the TreeView widget.
- Add the columns.
- Set the type of renderer to use.
- Create the ListStore.
- Set the model attribute in the Treeview to our model.
- Fill in the data.

The third step is to set up the type of renderer the column will use to display the data. This is simply a routine that is used to draw the data into the tree model. There are many different cell renderers that come with GTK, but most of the ones that you would normally use include GtkCellRendererText and GtkCellRendererToggle.

So, let's create a function (shown above) that sets up our TreeView widget. We'll call it SetupupTreeview. First we'll define some variables for our columns, set the variable reference of the Treeview itself, add the columns, set up the ListStore, and set the model. Here's the code for the function. Put it after the

```python
def SetupTreeview(self):
    self.cFName = 0
    self.cFType = 1
    self.cFPath = 2
    self.sFName = "Filename"
    self.sFType = "Type"
    self.sFPath = "Folder"
    self.treeview = self.wTree.get_widget("treeview1")
    self.AddPlaylistColumn(self.sFName,self.cFName)
    self.AddPlaylistColumn(self.sFType,self.cFType)
    self.AddPlaylistColumn(self.sFPath,self.cFPath)
    self.playList = gtk.ListStore(str,str,str)
    self.treeview.set_model(self.playList)
    self.treeview.set_grid_lines(gtk.TREE_VIEW_GRID_LINES_BOTH)
```
SetWidgetReferences function.

The variables cFName, cFType and cFPath define the column
titles. The variables sFName, sFType and sFPath will hold the
column names in our displayed view. The seventh line sets the
variable reference of the treview widget as named in our glade file.

Next we call a routine (next page, top right), which we’ll create in
just a moment, for each column we want. Then we define our
GTKListStore with three text fields, and finally set the model attribute
of our TreeView widget to our GTKListStore. Let’s create the
AddPlaylistColumn function next. Put it after the SetupTreeview
function.

Each column is created with this function. We pass in the title of the
column (what’s displayed on the top line of each column) and a
columnID. In this case, the variables we set up earlier
(sFName and cFname) will be passed here. We then create a
column in our TreeView widget giving the title, what kind of cell
renderer it will be using, and, finally, the id of the column. We
then set the column to be

```python
def AddPlayListColumn(self,title,columnId):
    column = gtk.TreeViewColumn(title,gtk.CellRendererText(),text=columnId)
    column.set_resizable(True)
    column.set_sort_column_id(columnId)
    self.treeview.append_column(column)
```

Add these two functions to your code. I choose to put them right
after the SetWidgetReferences function, but you can put it
anywhere within the PlayListCreator class. Add the
following line after the call to SetWidgetReferences() in the
__init__ function to call the function.

```python
self.SetupTreeview()
```

Save and run your program, and you will see that we now have
two columns with headers in our TreeView widget.

There are so many things left to
do. We have to have a way to get
the music filenames from the user
and put them into the TreeView as
rows of data. We have to create
our Delete, ClearAll, movement
functions, save routine, and file
path routines, plus a few
"pretty" things that will make
our application look more
professional. Let’s start with the

```python
addRoutine. After all, that’s the
first button on our toolbar.
When the user clicks the Add
button, we want to pop up a
"standard" open-file dialog that
allows for multiple selections.
Once the user has made their
selection, we then want to take
this data and add it into the
treeview, as I stated above. So the
first logical thing to do is work on
the File Dialog. Again, GTK
does us a way to call a
"standard" file dialog in code. We
could hard code this as just lines
in the on_tBtnAdd_clicked event
handler, but let’s make a separate
class to handle this. While we are
at it, we can make this class
handle not only a file OPEN dialog,
but a folder SELECT dialog as well.
As before with the MessageBox
function, you can pull this into a
snippet file that has all kinds of
reuseable routines for later use.

```python
def AddFileDialog(self, which, CurrentPath):
    dialog = gtk.FileChooserDialog("Select files to add...",None,
gtk.FILE_CHOOSER_ACTION_OPEN,
    (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
gtk.STOCK_OPEN, gtk.RESPONSE_OK))
```

We’ll start by defining a new class
called FileDialog which will have
only one function called
ShowDialog. That function will
take two parameters, one called
‘which’ (a ’0’ or a ’1’), that
designates whether we are
creating an open-file or select-
folder dialog, and the other is the
path that should be used for the
default view of the dialog called
CurrentPath. Create this class just
before our main code at the
bottom of the source file.

```python
class FileDialog:
    def ShowDialog(self, which, CurrentPath):
        dialog = gtk.FileChooserDialog("Select files to add...",None,
gtk.FILE_CHOOSER_ACTION_OPEN,
    (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
gtk.STOCK_OPEN, gtk.RESPONSE_OK))
```

The first part of our code should be an
IF statement

```python
if which == 0: # file chooser
    ...”
```
These set the default response to be the OK button, and then to turn on the multiple select feature so the user can select (you guessed it) multiple files to add. If we didn't set this, the dialog would only allow one file to be selected at a time, since set_select_multiple is set to False by default. Our next lines are setting the current path, and then displaying the dialog itself. Before we type in the code, let me explain why we want to deal with the current path. Every time you pop up a file dialog box, and you DON'T set a path, the default is to the folder where our application resides. So, let's say that the music files that the user would be looking for are in /media/music_files/, and are then broken down by genre, and further by artist, and further by album. Let's further assume that the user has installed our application in /home/user2/playlistmaker. Each

```python
dialog = gtk.FileChooserDialog("Select files to add...",None,
gtk.FILE_CHOOSER_ACTION_OPEN,
(gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
gtk.STOCK_OPEN, gtk.RESPONSE_OK))
```

```python
class FileDialog:
    def ShowDialog(self,which,CurrentPath):
        if which == 0: #file chooser
            dialog = gtk.FileChooserDialog("Select files to add...",None,
                gtk.FILE_CHOOSER_ACTION OpenFileDialog,
                (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
                gtk.STOCK_OPEN, gtk.RESPONSE_OK))
        else: #folder chooser
dialog = gtk.FileChooserDialog("Select Save Folder...",None,
                gtk.FILE_CHOOSER_ACTION_SELECT_FOLDER,
                (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
                gtk.STOCK_OPEN, gtk.RESPONSE_OK))

dialog.set_default_response(gtk.RESPONSE_OK)
```

```python
if CurrentPath != ":
    dialog.set_current_folder(CurrentPath)
response = dialog.run()
```

Next, we need to handle the response from the dialog.

```python
if response == gtk.RESPONSE_OK:
    fileselection = dialog.get_filenames()
    CurrentPath = dialog.get_current_folder()
    dialog.destroy()
    return (fileselection,CurrentPath)
elif response == gtk.RESPONSE_CANCEL:
    print 'Closed, no files selected'
    dialog.destroy()
```
time we pop up the dialog, the
starting folder would be
/home/user2/playlistmaker.
Quickly, the user would become
frustrated by this, wanting the last
folder he was in to be the starting
folder next time. Make sense? OK.
So, bottom right are our next lines
of code.

Here we check the responses sent
back. If the user clicked the 'Open'
button which sends back a
gtk.RESPONSE_OK, we get the
name or names of the files the
user selected, set the current path
to the folder we are in, destroy the
dialog, and then return the data
back to the calling routine. If, on
the other hand, the user clicked
on the 'Cancel' button, we simply
destroy the dialog. I put the print
statement in there just to show
you that the button press worked.
You can leave it or take it out.
Notice that when we return from
the Open button part of the
routine, we are returning two sets
of values. 'fileselection' is a list of
the files selected by the user, as
well as the CurrentPath.

In order to get the routine to do
something, add the following line
under the on_tbtnAdd_clicked
routine...

```python
fd = FileDialog()
```

Here we retrieve the two return
values that are sent from our
return call. For now, add the
following code to see what the
information returned will look like.

```python
for f in selectedfiles:
    print "User selected %s" % f
print "Current path is %s" % self.CurrentPath
```

When you run the program,
click on the 'Add' button. You'll see
the file dialog. Now move to
somewhere where you have some
files and select them. You can hold
down the [ctrl] key and click on
multiple files to select them
individually, or the [shift] key to
select multiple contiguous files.
Click on the 'Open' button, and
look at the response in your
terminal window. Please note that
if you click on the 'Cancel' button
right now, you'll get an error
message. That's because the
above code assumes that there
are no files selected. Don't worry

about that right now - we'll handle
that in a little bit. I just wanted to
let you see what comes back if the
'Open' button is pressed. One
thing we should do is add a filter
to our file-open dialog. Since we
expect the user to normally select
music files, we should (1) give the
option to display only music files,
and (2) give the option to show all
files just-in-case. We do this by
using the filefilter attributes of the
dialog. Here's the code for that
which should go in the which == 0
section right after the dialog set
line.

```python
filter = gtk.FileFilter()
filter.set_name("Music Files")
filter.add_pattern("*.mp3")
filter.add_pattern("*.ogg")
filter.add_pattern("*.wav")
dialog.add_filter(filter)
filter = gtk.FileFilter()
filter.set_name("All files")
filter.add_pattern("*")
dialog.add_filter(filter)
```

We are setting up two "groups",
one for music files
(filter.set_name("Music Files")),
and the other for all files. We use
a pattern to define the types of
files we want. I have defined three
patterns, but you can add or
delete any that you wish. I put the
music filter first, since that's what
we will assume the user is going
to be mainly concerned with. So
the steps are...

- Define a filter variable.
- Set the name.
- Add a pattern.
- Add the filter to the dialog.

You can have as many or as few
filters as you wish. Also notice that
once you have added the filter to
the dialog, you can re-use the
variable for the filter.

Back in the on_tbtnAdd_clicked
routine, comment out the last
lines we added and replace them
with this one line.

```python
self.AddFilesToTreeview(selectedfiles)
```

so our routine now looks like the
code shown on the next page.

So, when we get the response
back from file dialog, we will send the
list containing the selected files to
this routine. Once here, we set up a
counter variable (how many files we
are adding), then parse the list.
Remember that each entry contains
the fully qualified filename with path
and extension. We'll want to split
the filename into path, filename,
def on_btnAdd_clicked(self,widget):
    fd = FileDialog()
    self.AddFilesToTreeview(selectedfiles)

We now have to create the function that we just put the call to. Put this function after the on_btnSavePlaylist_clicked routine.

    def AddFilesToTreeview(self,FileList):
        counter = 0
        for f in FileList:
            extStart = f.rfind(".")
            fnameStart = f.rfind("/")
            extension = f[extStart+1:]
            fname = f[fnameStart+1:extStart]
            fpath = f[:fnameStart]
            data = [fname,extension,fpath]
            self.playList.append(data)
            counter += 1
        self.RowCount += counter
        self.sbar.push(self.context_id,"%s files added

and extension. First we get the very last 'period' from the filename and assume that is the beginning of the extension and assign its position in the string to extStart. Next we find the very last '/' in the filename to determine the beginning of the filename. Then we break up the string into extension, filename and file path. We then stuff these values into a list named 'data' and append this into our playlist ListStore. We increment the counter since we have done all the work. Finally we increment the variable RowCount which holds the total number of rows in our ListStore, and then we print a message to the status bar.

Now you can run the application and see the data in the TreeView.

As always, the full code can be found at http://pastebin.com/JtrHuE7I.

Next time, we'll finalize our application, filling in the missing routines, etc.

---

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,
Program In Python - Part 23

**T**his time, we are going to finish our playlistmaker program. Last time, we got a good bit done, but we left some things incomplete. We can't save the playlist, we don't have the movement functions done, we can't select the file path to store the file in, and so on. However, there are a few things that we need to do before we start coding. First, we need to find an image for the logo for our application in the about box, and for when the application is minimized. You can dig around in the /usr/share/icons folder for an icon you like, or you can go on the web and get one, or create one yourself. Whatever you get, put it into your code folder with the glade file and the source code from last month. Name it logo.png. Next, we need to open the glade file from last month and make a few changes.

First, using the MainWindow, go to the General tab, and scroll down until you find Icon. Using the browse tool, find your icon and select that. Now the text box should contain “logo.png”. Next, in the hierarchy box, select treeview1, go to the signal tab, and, under GtkTreeView | cursor-changed, add a handler for on_treeview1_cursor_changed. Remember, as I told you last month, to click off that to make the change stick. Finally, again in the hierarchy box, select txtFilename, and go to the signal tab. Scroll down until you find 'GtkWidget', and scroll down further until you get to 'key-press-event'. Add a handler for 'on_txtFilename_key_press_event'. Save your glade project and close glade.

Now it's time to complete our project. We'll start from where we left off using last month's code.

```python
elif response == gtk.RESPONSE_CANCEL:
    print 'Closed, no files selected'
    dialog.destroy()

Notice that we aren't returning anything. This is what caused the error. So to fix this, we want to add the following line of code after the dialog.destroy() line.

```python
Return ([],"")
```

This will keep the error from happening. Next, let's add the text box event handler we created in glade. To our dictionary, add the following line.

```python
"on_txtFilename_key_press_event": self.txtFilenameKeyPress,
```

As you remember, this creates a function to handle the keypress event. We'll next create the function.

```python
def txtFilenameKeyPress(self,widget,data):
    if data.keyval == 65293: # The value of the return key
```

The first thing I want to do is modify the code in class FileDialog. If you remember from last time, if the user clicked the 'Cancel' button, there was an error raised. We will fix that first. At the end of the routine, you have the code shown above.

You might imagine, this simply looks at the value of each key that is pressed when the user is in the txtFilename text box, and compares it to the value 65293, which is the code that is assigned to the return key (enter key). If it matches, then it calls the SavePlaylist function. The user doesn't have to even click the button.
def AddFilesToTreeview(self, FileList):
    counter = 0
    for f in FileList:
        extStart = f.rfind(".")
        fnameStart = f.rfind("/")
        extension = f[extStart+1:]  
        fname = f[fnameStart+1:extStart]
        fpath = f[:fnameStart]
        data = [fname,extension,fpath]
        self.playList.append(data)
    counter += 1

def on_tbtnDelete_clicked(self, widget):
    sel = self.treeview.get_selection()
    (model,rows) = sel.get_selected_rows()
    iters=[]
    for row in rows:
        iters.append(self.playList.get_iter(row))
    for i in iters:
        if i is not None:
            self.playList.remove(i)
            self.RowCount -= 1
    self.sbar.push(self.context_id, "%d files in list." % (self.RowCount))

def on_tbtnGetFolder_clicked(self, widget):
    fd = FileDialog()
    self.txtPath.set_text(filepath[0])

    then display the number of files in the status bar.

    Now, before we get to the move functions, let's deal with the save-file-path function.
    We'll use our FileDialog class as before. We'll do all the code (bottom right) for this in the
    on_tbtnGetFolder_clicked routine.

    The only thing really different from before is the last line of this code. We are putting
    the name of the path returned by the FileDialog into the textbox that we set up
def SavePlaylist(self):
    fp = self.txtPath.get_text()  # Get the filepath from the text box
    fn = self.txtFilename.get_text()  # Get the filename from the filename text box

    Now check the values...

    if fp == "":  # IF the path is blank...
        self.MessageBox("error","Please provide a filepath for the playlist.")
    elif fn == "":  # IF the filename is blank...
        self.MessageBox("error","Please provide a filename for the playlist file.")
    else:
        # Otherwise we are good to go.

plfile = open(fp + "/" + fn,"w")  # Open the file
plfile.writelines(:'#EXTM3U\n')  # Print the M3U Header
for row in self.playList:
    plfile.writelines("%s/%s.%s\n % (row[2],row[0],row[1])))  #Write the line data
plfile.close  # Finally close the file

Lastly, we pop up a message box informing the user that the file has been saved.

self.MessageBox("info","Playlist file saved!")

We now need to put in a call to this routine in our on_btnSavePlaylist_clicked event handler routine.

def on_btnSavePlaylist_clicked(self,widget):
    self.SavePlaylist()

Save your code and test it. Your play list should save properly and look something like the

We can now start work on the move functions. Let's start with the Move To Top routine.
Like we did when we wrote the delete function, we get the selection and then the selected row.
Next we have to step through the rows to get two
variables. We will call them path1 and path2. Path2, in this case will be set to 0, which is the “target” row. Path1 is the row the user has selected. We finally use the model.move_before() method to move the selected row up to row 0, effectively pushing everything down. We’ll put the code (below right) directly in the on_btnMoveToTop_clicked routine.

For the MoveToBottom function, we will use almost exactly the same code as the MoveToTop routine, but, in place of the model.move_before() method, we will use the model.move_after() method, and, instead of setting path2 to 0, we set it to self.RowCount-1. Now you understand why we have a RowCount variable. Remember the counts are zero based, so we have to use RowCount-1 (above right).

Now let’s take a look at what it will take to do the MoveUp routine. Once again, it is fairly similar to the last two functions we created. This time, we get path1 which is the selected row and then assign that row number-1 to path2. Then if path2 (the target row) is greater than or equal to 0, we use the model.swap() method (second down, right).

The same thing applies for the MoveDown function. This time however, we check to see if path2 is LESS than or equal to the value of self.RowCount-1 (third down, right).

Now let’s make some changes to the abilities of our play list. In last month’s article, I showed you the basic format of the play list file (bottom).

However, I did say that there was an extended format as well. In the extended format, there is an extra line that can be added to the file before each song file entry that contains extra information about the song. The format of this line is as follows...

def on_btnMoveToBottom_clicked(self, widget):
    sel = self.treeview.get_selection()
    (model, rows) = sel.get_selected_rows()
    for path1 in rows:
        path2 = self.RowCount-1
        iter1=model.get_iter(path1)
        iter2 = model.get_iter(path2)
        model.move_after(iter1, iter2)

def on_btnMoveUp_clicked(self, widget):
    sel = self.treeview.get_selection()
    (model, rows) = sel.get_selected_rows()
    for path1 in rows:
        path2 = (path1[0]-1,)
        if path2[0] >= 0:
            iter1=model.get_iter(path1)
            iter2 = model.get_iter(path2)
            model.swap(iter1, iter2)

def on_btnMoveDown_clicked(self, widget):
    sel = self.treeview.get_selection()
    (model, rows) = sel.get_selected_rows()
    for path1 in rows:
        path2 = (path1[0]+1,)
        iter1=model.get_iter(path1)
        if path2[0] <= self.RowCount-1:
            iter2 = self.RowCount-1:
            model.swap(iter1, iter2)

#EXTINF:[Length of song in seconds],[Artist Name] –
[Song Title]

You might have wondered why we included the mutagen library from the beginning since we never used it. Well, we will now. To refresh your memory, the mutagen library is for accessing ID3 tag information from inside of MP3 files. To get

**#EXTM3U**

**Adult Contemporary/Chris Rea/Collection/02 - On The Beach.mp3**

**Adult Contemporary/Chris Rea/Collection/07 - Fool (If You Think It's Over).mp3**

**Adult Contemporary/Chris Rea/Collection/11 - Looking For The Summer.mp3**
the full discussion about this, please refer to issue 35 of Full Circle which has my part 9 of this series. We'll create a function to deal with the reading of the MP3 file and return the Artist name, the Song Title, and the length of the song in seconds, which are the three things we need for the extended information line. Put the function after the ShowAbout function within the PlaylistCreator class (next page, top right).

Again, to refresh your memory, I'll walk through the code. First we clear the three return variables so that if anything happens they are blank upon return. We then pass in the filename of the MP3 file we are going to look at. Next we pull the keys into (yes, you guessed it) an iterator, and walk through that iterator looking for two specific tags. They are 'TPE1' which is the artist name, and 'TIT2' which is the song title. Now, if the key doesn't exist, we would get an error, so we wrap each get call with a 'try|except' statement. We then pull the song length from the audio.info.length attribute, and return the whole shebang.

Now, we will want to modify the SavePlaylist function to support the extended information line. While we are there, let's check to see if the filename exists, and, if so, flag the user and exit the routine. Also, to make things a bit easier for the user, since we don't support any other filetype, let's

```python
import os.path

Then, go ahead and comment out your existing SavePlaylist function and we'll replace it.

def SavePlaylist(self):
    fp = self.txtPath.get_text()  # Get the file path from the text box
    fn = self.texFilename.get_text() # Get the filename from the text box
    if fp == "":  # IF file path is blank...
        self.MessageBox("error","Please provide a file path for the playlist.")
    elif fn == "":  # IF filename is blank...
        self.MessageBox("error","Please provide a filename for the playlist file.")
    else:  # Otherwise
        extStart = fn.rfind(".")  # Find the extension start position
        if extStart == -1:
            fn += ".m3u"  #append the extension if there isn't one.
        self.texFilename.set_text(fn)  #replace the filename in the text box
```

```python
def GetMP3Info(self,filename):
    artist = ''
title = ''
songlength = 0
audio = MP3(filename)
keys = audio.keys()
for key in keys:
    try:
        if key == "TPE1":  # Artist
            artist = audio.get(key)
        except:
            artist = ''
    try:
        if key == "TIT2":  # Song Title
            title = audio.get(key)
        except:
            title = ''
    songlength = audio.info.length  # Audio Length
return (artist,title,songlength)
```
automatically append the extension '.m3u' to the path and filename if it doesn't exist. First add an import line at the top of the code importing os.path between the sys import and the mutagen import (bottom right).

Just like in the AddFilesToTreeview function, we will use the 'rfind' method to find the position of the last period ('.') in the filename fn. If there isn't one, the return value is set to -1. So we check to see if the return value is -1, and, if so, we append the extension and then put the filename back in the text box just to be nice.

```python
if os.path.exists(fp + "/" + fn):
    # Open the file
    pfile.write('EXTM3U
    for row in self.playList:
        fn = '%s\%s' % (row[2],row[0],row[1])
        artist,title,songlength = self.GetMP3Info(fn)
        if songlength > 0 and (artist != '' and title != 'autoComplete'): pfile.write('EXTINF:%d,%s
    pfile.write('%s\n' % fn)
    pfile.close # Finally Close the file
    self.MessageBox("info","Playlist file saved!")
```

The rest of the code is mostly the same as before, but let's look at it anyway.

Next, we want to wrap the rest of the function with an IF|ELSE clause (top right) so if the file already exists, we simply fall out of the routine. We use os.path.exists(filename) to do this check.

Line 7 then checks to see if we have values in all three variables. If so, we write the extended information line in line 8, otherwise we don't try. Line 9 writes the filename line as before. Line 10 closes the file gracefully, and line 11 pops up the message box letting the user know the process is all done.

```python
def SetupToolTips(self):
    self.btnAdd.setToolTip("Add a file or files to the playlist.")
    self.btnAbout.setToolTip("Display the About Information.")
    self.btnDelete.setToolTip("Delete selected entry from the list.")
    self.btnClearAll.setToolTip("Remove all entries from the list.")
    self.btnExit.setToolTip("Quit this program.")
    self.btnMoveToTop.setToolTip("Move the selected entry to the top of the list.")
    self.btnMoveUp.setToolTip("Move the selected entry up in the list.")
    self.btnMoveDown.setToolTip("Move the selected entry down in the list.")
    self.btnDelete.setToolTip("Move the selected entry to the bottom of the list.")
    self.btnGetFolder.setToolTip("Select the folder that the playlist will be saved to.")
    self.btnSavePlaylist.setToolTip("Save the playlist.")
    self.txtFilename.setToolTip("Enter the filename to be saved here. The extension ".m3u" will be added for you if you don't include it.")
```
Go ahead and save your code and give it a test drive.

At this point about the only thing that should be added would be some tool tips for our controls when the user hovers the mouse pointer over them. It adds that professional flair (below). Let's create a function to do that now.

We are using the widget references we set up earlier, and then setting the text for the tooltip via the (you guessed it) set_tooltip_text attribute. Next we need to add the call to the routine. Back in the _init_ routine, after the self.SetWidgetReferences line, add:

```python
self.SetupToolTips()
```

Last, but certainly not least, we want to put our logo into our About box. Just like everything else there, there's an attribute for that. Add the following line to the ShowAbout routine.

```python
about.set_logo(gtk.gdk.pixbuf._new_from_file("logo.png"))
```

That's about it. You now have a fully functioning program that looks good, and does a wonderful job of creating a playlist for your music files.

The full source code, including the glade file we created last month, can be found at pastebin: http://pastebin.com/tQjizcwT

Until next time, enjoy your new found skills.

---

**EXTRA! EXTRA!**

READ ALL ABOUT IT!

**Full Circle Special Editions Released On Unsuspecting World**

Full Circle Special Editions Released On Unsuspecting World*  

**THE PERFECT SERVER SPECIAL EDITION**

This is a special edition of Full Circle that is a direct reprint of the Perfect Server articles that were first published in FCM#31-#34.

http://fullcirclemagazine.org/special-edition-1-the-

**PYTHON SPECIAL EDITION #01**

This is a reprint of Beginning Python Parts 01 - 08 by Greg Walters.

http://fullcirclemagazine.org/python-special-edition-

---

* Neither Full Circle magazine, nor its makers, apologize for any hysteria caused in the release of its publications.
Program In Python - Part 24

Wow! It's hard to believe that this is the 24th issue already. Two years we've been learning Python! You've come a very long way.

This time we are going to cover two topics. The first is printing to a printer, the second is creation of RTF (Rich Text Format) files for output.

Generic Printing under Linux

So let's start with printing to a printer. The idea to cover this came from an email sent by Gord Campbell. It's actually easy to do most printing from Linux, and easier than that other operating system that starts with "WIN" - and I won't deal with that OS.

As long as all you want to print is straight text, no bold, italics, font changes, etc, it's fairly easy. Here's a simple app that will print directly to your printer...

```python
import os
pr = os.popen('lpr', 'w')
pr.write('print test from linux via python\n')
pr.write('Print finished\n')
pr.close()
```

This is fairly easy to understand as long as you expand your mind just a bit. In the above code, 'lpr' is the print spooler. The only requirement is that we have already configured 'lpd' and that it's running. More than likely, when you use a printer under Ubuntu, it's already done for you. 'Lpd' is usually referred to as a "magic-filter" that can automatically convert different types of documents to something the printer can understand. We are going to print to the 'lpr' device/object. Think of it simply as a file. We open the file. We have to import 'os'. Then in line 2, we open the 'lpr' with write access - assigning it to the object variable 'pr'. We then do a 'pr.write' with anything we want to print. Finally (line 5) we close the file, which will send the data out to the printer.

We can also create a text file, then send it out to the printer like this...

```python
import os
filename = 'dummy.file'
os.system('lpr %s' % filename)
```

In this case, we are still using the lpr object, but we are using the 'os.system' command to basically create a command that looks to linux like we sent it from a terminal.

I'll leave you to play with this for now.

PyRTF

Now let's deal with RTF files. RTF format (that's kind of like saying PIN number since PIN stands for Personal Identification Number, so that translates to Personal-Identification-Number Number. Something from the department of redundancy department, huh?) was originally created by the Microsoft Corporation in 1987, and its syntax was influenced by the TeX typesetting language. PyRTF is a wonderful library that makes it easy to write RTF files. You have to do some planning up front on how you want your files to look, but the results will be well worth it.

First, you need to download and install the PyRTF package. Go to http://pyrtf.sourceforge.net and
get the PyRTF-0.45.tar.gz package. Save it someplace and use archive manager to unpack it. Then using terminal, go to where you unpacked it. First we need to install the package, so type “sudo python setup.py install” and it will be installed for you. Notice there is an examples folder there. There’s some good information there on how to do some advanced things.

Here we go. Let’s start as we usually do, creating the stub of our program which is shown on the next page, top right.

Before going any further, we’ll discuss what’s going on. Line 2 imports the PyRTF library. Note that we are using a different import format than normal. This one imports everything from the library.

Our main working routine is MakeExample. We’ve stubbed for now. The OpenFile routine creates the file for us with the name we pass into it, appends the extension “.rtf”, puts it into the “write” mode, and returns a file handle.

We’ve already discussed the if __name__ routine before, but to refresh your memory, if we are running the program in a standalone mode, the internal variable __name__ is set to “__main__”. If we call it as an import from another program, then it will just ignore that portion of the code.

Here, we create an instance of the Renderer object, call the MakeExample routine, getting the returned object doc. We then write the file (in doc) using the OpenFile routine.

Now for the meat of our worker routine MakeExample. Replace the pass statement with the code shown below.

Let’s look at what we have done. In the first line we create an instance of Document. Then we create an instance of the style sheet. Then we create an instance of the section object and append it to the document. Think of a section as a chapter in a book. Next we create a paragraph using the Normal style. The author of PyRTF has preset this to be 11-point Arial font. We then put whatever text we want into the paragraph, append that to the section, and return our doc document.

That is very easy. Again, you need to plan your output fairly carefully, but nothing too onerous.

Save the program as “rtftesta.py” and run it. When it’s completed, use openoffice (or LibreOffice) to open the file.

```python
#!/usr/bin/env python
from PyRTF import *

def MakeExample():
    pass

def OpenFile(name):
    return file('%s.rtf' % name, 'w')

if __name__ == '__main__':
    DR = Renderer()
    doc = MakeExample()
    DR.Write(doc, OpenFile('rtftesta'))
    print "Finished"

# Document
ss = doc.StyleSheet
section = Section()
doc.Sections.append(section)

p = Paragraph(ss.ParagraphStyles.Normal)
p.append('This is our first test writing to a RTF file. ' +
     'This first paragraph is in the preset style called normal ' +
     'and any following paragraphs will use this style until we change it."
section.append(p)
return doc
```
Now let's do some neat things. First, we'll add a header. Once again, the author of PyRTF has given us a predefined style called Header1. We'll use that for our header. In between the doc.Sections.append line and the p = Paragraph line, add the following.

```
p = Paragraph(ss.ParagraphStyles.Normal)
p.append('It is also possible to provide overrides for elements of a style. ',
    'For example you can change just the font ',
    TEXT(' size to 24 point', size=48),
    ' or ',
    TEXT(' typeface to Impact', font=ss.Fonts.Impact),
    ' or even more Attributes like',
    TEXT(' BOLD', bold=True),
    TEXT(' or Italic', italic=True),
    TEXT(' or BOTH', bold=True, italic=True),
    ' .')
section.append(p)
```

Save this as rtftestb.py and run it. So now we have a header. I'm sure your mind is going down many roads thinking about what more can we do. Here's a list of what the author has given us as the predefined styles.

Normal, Normal Short, Heading 1, Heading 2, Normal Numbered, Normal Numbered 2. There's also a List style, which I will let you play with on your own. If you want to see more, on this and other things, the styles are defined in the file Elements.py in the distribution you installed.

While these styles are good for many things, we might want to use something other than the provided styles. Let's look at how to change fonts, font sizes and attributes (bold, italic, etc) on the fly. After our paragraph and before we return the document object, insert the code shown top right, and change the output filename to rtftestc. Save the file as rtftestc.py. And run it. The new portion of our document should look like this...

It is also possible to provide overrides for elements of a style. For example you can change just the font size to 24 point or typeface to Impact or even more Attributes like BOLD or Italic or BOTH.

Now what have we done? Line 1 creates a new paragraph. We then start, as we did before, putting in our text. Look at the fourth line (TEXT(' size to 24 point', size = 48)),. By using the TEXT qualifier, we are telling PyRTF to do something different in the middle of the sentence, which in this case is to change the size of the font (Arial at this point) to 24-point, followed by the 'size = ' command. But, wait a moment. The 'size =' says 48, and what we are printing says 24 point, and the output is actually in 24-point text. What's going on here? Well the size command is in half points. So if we want an 8-point font we have to use size = 16. Make sense?

Next, we continue the text and then change the font with the 'font =' command. Again, everything within the inline
PROGRAM IN PYTHON - PART 24

TEXT command between the single quotes is going to be affected and nothing else.

Ok. If that all makes sense, what else can we do?

We can also set the color of the text within the TEXT inline command. Like this.

```
p = Paragraph()
p.append('This is a new paragraph with the word ', TEXT('RED', colour=ss.Colours.Red), ' in Red text. ')
section.append(p)
```

Notice that we didn't have to restate the paragraph style as Normal, since it sticks until we change it. Also notice that if you live in the U.S., you have to use the “proper” spelling of colour.

Here are the colors that are (again) predefined: Black, Blue, Turquoise, Green, Pink, Red, Yellow, White, BlueDark, Teal, GreenDark, Violet, RedDark, YellowDark, GreyDark and Grey.

And here is a list of all the predefined fonts (in the notation you must use to set them):

Arial, ArialBlack, ArialNarrow, BitstreamVeraSans, BitstreamVeraSerif, BookAntiqua, BookmanOldStyle, BookmanOldStyle, Castellar, CenturyGothic, ComicSansMS, CourierNew, FranklinGothicMedium, Garamond, Georgia, Haettenschweiler, Impact, LucidaConsole, LucidaSansUnicode, MicrosoftSansSerif, PalatinoLinotype, MonotypeCorsiva, Papyrus, Sylfaen, Symbol, Tahoma, TimesNewRoman, TrebuchetMS and Verdana.

So now you must be thinking that this is all well and good, but how do we make our own styles? That's pretty easy. Move to the top of our file, and before our header line, add the following code.

```
result = doc.StyleSheet
NormalText = TextStyle(TextPropertySet(result.Fonts.CourierNew, 16))
ps2 = ParagraphStyle('Courier', NormalText.Copy())
result.ParagraphStyles.append(ps2)
```

Before we write the code to actually use it, let's see what we have done. We are creating a new stylesheet instance called result. In the second line, we are setting the font to 8-point Courier New, and then “registering” the style as Courier. Remember, we have to use 16 as the size since the font size is in half-point values.

Now, before the return line at the bottom of the routine, let's include a new paragraph using the Courier style.

So now you have a new style you can use anytime you want. You can use any font in the list above and create your own styles. Simply copy the style code and replace the font and size information as you wish. We can also do this...

```
NormalText = TextStyle(TextPropertySet(result.Fonts.Arial, 22, bold=True, colour=ss.Colours.Red))
ps2 = ParagraphStyle('Arial', NormalText.Copy())
result.ParagraphStyles.append(ps2)
```

<table>
<thead>
<tr>
<th>Column Header 1</th>
<th>Column Header 2</th>
<th>Column Header 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1 data 1</td>
<td>Row 1 data 2</td>
<td>Row 1 data 3</td>
</tr>
<tr>
<td>Row 2 data 1</td>
<td>Row 2 data 2</td>
<td>Row 2 data 3</td>
</tr>
</tbody>
</table>
columns go down. Easy concept.

Let's start a new application and call it rtfTable-a.py. Start with our standard code (shown on the next page) and build from there.

We don't need to discuss this since it's basically the same code that we used before. Now, we'll flesh out the TableExample routine. I'm basically using part of the example file provided by the author of PyRTF. Replace the pass statement in the routine with the following code...

def TableExample():
    pass

def OpenFile(name):
    return file('%s.rtf' % name, 'w')

if __name__ == '__main__':
    DR = Renderer()
    doc = TableExample()
    DR.Write(doc, OpenFile('rtftable-a'))
    print "Finished"

This line (yes, it's really one line, but is broken up for easy viewing) creates our basic table. We are creating a table with 3 columns, the first is 7 tabs wide, the second and third are three tabs wide. We don't have to deal with tabs alone, you can enter the widths in twips. More on that in a moment.

c1 = Cell(Paragraph(ss.ParagraphStyles.Heading2, 'Heading2 Style'))
c2 = Cell(Paragraph(ss.ParagraphStyles.Normal, 'Back to Normal Style'))
c3 = Cell(Paragraph('More Normal Style'))
table.AddRow(c1,c2,c3)

c1 = Cell(Paragraph('Row One, Cell One'))
c2 = Cell(Paragraph('Row One, Cell Two'))
c3 = Cell(Paragraph('Row One, Cell Three'))
table.AddRow(c1,c2,c3)

Here we are setting the data that goes into each cell in the first row.

This group of code sets the data for row number two. Notice we can set a different style for a single or multiple cells.
c1 = Cell(Paragraph(ss.ParagraphStyles.Heading2,'Heading2 Style'))

c2 = Cell(Paragraph(ss.ParagraphStyles.Normal,'Back to Normal Style'))
c3 = Cell(Paragraph('More Normal Style'))
table.AddRow(c1,c2,c3)

This sets the final row.

section.append(table)

return doc

This appends the table into the section and returns the document for printing.

Save and run the app. You’ll notice that everything is about what you would expect, but there is no border for the table. That can make things difficult. Let’s fix that. Again, I’ll mainly use code from the example file provided by the PyRTF author.

Save your file as rtftable-b.py. Now, delete everything between 'doc.Sections.append(section)' and 'return doc' in the

### TableExample routine, and replace it with the following...

```
thin_edge = BorderPS( width=20, style=BorderPS.SINGLE )

thick_edge = BorderPS( width=80, style=BorderPS.SINGLE )

thin_frame = FramePS( thin_edge, thin_edge, thin_edge, thin_edge )

thick_frame = FramePS( thick_edge, thick_edge, thick_edge, thick_edge )

mixed_frame = FramePS( thin_edge, thick_edge, thin_edge, thick_edge )

Here we are setting up the edge and frame definitions for borders and frames.
```

table = Table( TabPS.DEFAULT_WIDTH * 3, TabPS.DEFAULT_WIDTH * 3, TabPS.DEFAULT_WIDTH * 3 )
c1 = Cell( Paragraph( 'R1C1' ), thin_frame )
c2 = Cell( Paragraph( 'R1C2' ) )
c3 = Cell( Paragraph( 'R1C3' ), thick_frame )
table.AddRow( c1, c2, c3 )

In row one, the cells in column one (thin frame) and column 3 (thick frame) will have a border around them.

c1 = Cell( Paragraph( 'R2C1' ) )
c2 = Cell( Paragraph( 'R2C2' ) )
c3 = Cell( Paragraph( 'R2C3' ) )
table.AddRow( c1, c2, c3 )

None of the cells will have a border in the second row.

c1 = Cell( Paragraph( 'R3C1' ), mixed_frame )
c2 = Cell( Paragraph( 'R3C2' ) )
c3 = Cell( Paragraph( 'R3C3' ), mixed_frame )
table.AddRow( c1, c2, c3 )

Once again, cells in column 1 and three have a mixed frame in row three.

section.append( table )

So. You have just about everything you need to create, through code, RTF documents.

---

**See you next time!**

Source code can be found at pastebin as usual. The first part can be found at [http://pastebin.com/3Rs7T3D7](http://pastebin.com/3Rs7T3D7) which is the sum of rtftest.py (a-e), and the second rtftable.py (a-b) is at [http://pastebin.com/XbaE2uP7](http://pastebin.com/XbaE2uP7).

---

**Greg Walters** is owner of RainyDay Solutions, LLC, a consulting company in Colorado and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family. His website is
A number of you have commented about the GUI programming articles and how much you’ve enjoyed them. In response to that, we will start taking a look at a different GUI toolkit called Tkinter. This is the “official” way to do GUI programming in Python. Tkinter has been around for a long time, and has gotten a pretty bad rap for looking “old fashioned”. This has changed recently, so I thought we’d fight that bad thought process.

**PLEASE NOTE** - All of the code presented here is for Python 2.x only. In an upcoming article, we’ll discuss how to use tkinter in Python 3.x. If you MUST use Python 3.x, change the import statements to “from tkinter import *”.

**A Little History And A Bit Of Background**

Tkinter stands for “Tk interface”. Tk is a programming language all on its own, and the Tinter module allows us to use the GUI functions there. There are a number of widgets that come natively with the Tkinter module. Some of them are Toplevel (main window) container, Buttons, Labels, Frames, Text Entry, Checkbuttons, Radiobuttons, Canvas, Multiline Text entry, and much more. There are also many modules that add functionality on top of Tkinter. This month, we’ll focus on four widgets. Toplevel (from here I’ll basically refer to it as the root window), Frame, Labels, and Buttons. In the next article, we’ll look at more widgets in more depth.

Basically, we have the Toplevel container widget which contains (holds) other widgets. This is the root or master window. Within this root window, we place the widgets we want to use within our program. Each widget, other than the Toplevel root window container, has a parent. The parent doesn’t have to be the root window. It can be a different widget. We’ll explore that next month. For this month, everything will have a parent of the root window.

In order to place and display the child widgets, we have to use what’s called “geometry management”. It’s how things get put into the main root window. Most programmers use one of three types of geometry management, either Packer, Grid, or Place management. In my humble opinion, the Packer method is very clumsy. I’ll let you dig into that on your own. The Place management method allows for extremely accurate placement of the widgets, but can be complicated. We’ll discuss the Place method in a future article set. For this time, we’ll concentrate on the Grid method.

Think of a spreadsheet.

<table>
<thead>
<tr>
<th>COLUMNS</th>
<th>- &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWS</td>
<td>0,0</td>
</tr>
<tr>
<td></td>
<td>0,1</td>
</tr>
<tr>
<td></td>
<td>0,2</td>
</tr>
<tr>
<td></td>
<td>0,3</td>
</tr>
</tbody>
</table>

There are rows and columns. Columns are vertical, rows are horizontal. Here’s a simple text representation of the cell addresses of a simple 5-column by 4-row grid (above right).

So parent has the grid, the widgets go into the grid positions. At first glance, you might think that this is very limiting. However, widgets can span multiple grid positions in either the column direction, the row direction, or both.

**Our First Example**

Our first example is SUPER simple (only four lines), but shows a good bit.

```
from Tkinter import *
root = Tk()
button = Button(root, text =
```
PROGRAM IN PYTHON - PART 25

```
“Hello FullCircle”).grid()
root.mainloop()

Now, what’s going on here? Line one imports the Tkinter library. Next, we instantiate the Tk object using root. (Tk is part of Tkinter). Here’s line three.

button = Button(root, text = “Hello FullCircle”).grid()

We create a button called button, set its parent to the root window, set its text to “Hello FullCircle,” and set it into the grid. Finally, we call the window’s main loop. Very simple from our perspective, but there’s a lot that goes on behind the scenes. Thankfully, we don’t need to understand what that is at this time.

Run the program and let’s see what happens. On my machine the main window shows up at the lower left of the screen. It might show up somewhere else on yours. Clicking the button doesn’t do anything. Let’s fix that in our next example.

Our Second Example

```
class App:
    def __init__(self, master):
        frame = Frame(master)
        self.lblText = Label(frame, text = “This is a label widget”)
        self.btnExit = Button(frame, text=“Quit”, fg=“red”, command=frame.quit)
        self.btnExit.grid(column = 0, row = 0)
        self.btnExit.grid(column = 0, row = 0, columnspan = 2)
        self.btnExit.grid(column = 1, row = 1)

This time, we’ll create a class called App. This will be the class that actually holds our window. Let’s get started.

from Tkinter import *

This is the import statement for the Tkinter library.

We define our class, and, in the __init__ routine, we set up our widgets and place them into the grid.

The first line in the __init__ routine creates a frame that will be the parent of all of our other widgets. The parent of the frame is the root window (Toplevel widget). Next we define a label, and two buttons. Let’s look at the label creation line.

```
self.lblText = Label(frame, text = “This is a label widget”)  

We create the label widget and call it self.btnExit. That’s inherited from the Label widget object. We set its parent (frame), and set the text that we want it to display (text = “this is a label widget”). It’s that simple. Of course we can do much more than that, but for now that’s all we need. Next we set up the two Buttons we will use:

```
self.btnExit = Button(frame, text=“Quit”, fg=“red”, command=frame.quit)
self.btnExit = Button(frame, text=“Hello”, command=self.saySomething)

We name the widgets, set their parent (frame), and set the text we want them to show. Now btnQuit has an attribute marked fg which we set to “red”. You might have guessed this sets the foreground color or text color to the color red. The last attribute is to set the callback command we want to use when the user clicks the button. In the case of btnQuit, it’s frame.quit, which ends the program. This is a built-in function, so we don’t need to actually create it. In the case of btnHello, it’s a routine called self.saySomething. This we have to create, but we have a bit more to go through first.

We need to put our widgets into the grid. Here’s the lines again:

```
frame.grid(column = 0, row = 0)
self.btnExit.grid(column = 0, row = 0, columnspan = 2)

Now btnQuit has an attribute.
self.btnHello.grid(column = 0, row = 1)

self.btnExit.grid(column = 1, row = 1)

First, we assign a grid to the frame. Next, we set the grid attribute of each widget to where we want the widget to go. Notice the colspan line for the label (self.lblText). This says that we want the label to span across two grid columns. Since we have only two columns, that's the entire width of the application. Now we can create our callback function:

def SaySomething(self):
    print "Hello to FullCircle Magazine Readers!!"

This simply prints in the terminal window the message “Hello to FullCircle Magazine Readers!!”

Finally, we instantiate the Tk class - our App class - and run the main loop.

root = Tk()
app = App(root)
root.mainloop()

Give it a try. Now things actually do something. But again, the window position is very inconvenient. Let's fix that in our next example.

Our Third Example

Save the last example as example3.py. Everything is exactly the same except for one line. It's at the bottom in our main routine calls. I'll show you those lines with our new one:

root = Tk()
root.geometry ('150x75+550+150')

app = App(root)
root.mainloop()

What this does is force our initial window to be 150 pixels wide and 75 pixels high. We also want the upper left corner of the window to be placed at X-pixel position 550 (right and left) and the Y-pixel position at 150 (top to bottom). How did I come up with these numbers? I started with some reasonable values and tweaked them from there. It's a bit of a pain in the neck to do it this way, but the results are better than not doing it at all.

Our Fourth Example – A Simple Calculator

Now, let's look at something a bit more complicated. This time, we'll create a simple “4 banger” calculator. If you don't know, the phrase “4 banger” means four functions: Add, Subtract, Multiply, and Divide. Right is what it looks like in simple text form.

We'll dive right into it and I'll explain the code (middle right) as we go.

Outside of the geometry statement, this (left) should be pretty easy for you to understand by now. Remember, pick some reasonable values, tweak them, and then move on.
We begin our class definition and set up our __init__ function. We set up three variables as follows:

- CurrentValue - Holds the current value that has been input into the calculator.
- HolderValue - Holds the value that existed before the user clicks a function key.
- CurrentFunction - This is simply a “bookmark” to note what function is being dealt with.

Next, we define the CurrentDisplay variable and assign it to the StringVar object. This is a special object that is part of the Tkinter toolkit. Whatever widget you assign this to automatically updates the value within the widget. In this case, we will be using this to hold whatever we want the display label widget to... er... well... display. We have to instantiate it before we can assign it to the widget. Then we use the built in 'set' function. We then define a boolean variable called DecimalNext, and a variable DecimalCount, and then call the DefineWidgets function, which creates all the widgets, and then call the PlaceWidget function, which actually places them in the root window.

```python
def DefineWidgets(self, master):
    self.lb1Display = Label(master, anchor=E, relief =
                             SUNKEN, bg="white", height=2, textvariable=self.CurrentDisplay)
```

Now, we have already defined a label earlier. However, this time we are adding a number of other attributes. Notice that we aren't using the 'text' attribute. Here, we assign the label to the parent (master), then set the anchor (or, for our purposes, justification) for the text, when it gets written. In this case, we are telling the label to justify all text to the east or on the right side of the widget. There is a justify attribute, but that's for multiple lines of text. The anchor attribute has the following options... N, NE, E, SE, S, SW, W, NW and CENTER. The default is CENTER. You should think compass points for these. Under normal circumstances, the only really usable values are E (right), W (left), and Center.

Next, we set the relief or visual style of the label. The "legal" options here are FLAT, SUNKEN, RAISED, GROOVE, and RIDGE. The default is FLAT if you don't specify anything. Feel free to try the other combinations on your own after we're done. Next, we set the background (bg) to white in order to set it off from the rest of the window a bit. We set the height to 2 (which is two text lines high, not in pixels), and finally assign the variable we just defined a moment ago (self.CurrentDisplay) to the textvariable attribute. Whenever the value of self.CurrentDisplay changes, the label will change its text to match automatically.

Shown above, we'll create some of the buttons.

I've shown only 4 buttons here. That's because, as you can see, the code is almost exactly the same. Again, we've created buttons earlier in this tutor, but let's take a closer look at what we are doing here.

We start by defining the parent (master), the text that we want on the button, and the width and height. Notice that the width is in characters and the height is in text lines. If you were doing a graphic in the button, you would use pixels to define the height and width. This can become a bit confusing until you get your head firmly wrapped around it.
Next, we are setting the bind attribute. When we did the buttons in the previous examples, we used the 'command' attribute to define what function should be called when the user clicks the button. This time, we are using the 'bind' attribute. It's almost the same thing, but this is an easier way to do it, and to pass information to the callback routine that is static. Notice that here we are using '<ButtonRelease-1>' as the trigger for the bind. In this case, we want to make sure that it's only after the user clicks AND releases the left mouse button that we make our callback. Lastly, we define the callback we want to call, and what we are going to pass to it. Now, those of you who are astute (which is each and every one of you) will notice something new. The 'lambda e:' call.

In Python, we use Lambda to define anonymous functions that will appear to interpreter as a valid statement. This allows us to put multiple segments into a single line of code. Think of it as a mini

```python
self.btnDash = Button(master, text = '-',width = 4,height=3)
sel.btnDash.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('ABS'))
sel.btnDot = Button(master, text = '.',width = 4,height=3)
sel.btnDot.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Dec'))
```

The btnDash sets the value to the absolute value of the value entered. 523 remains 523 and -523 becomes 523. The btnDot button enters a decimal point. These examples, and the ones below, use the callback funcFuncButton.

```python
self.btnPlus = Button(master,text = '+', width = 4, height=3)
sel.btnPlus.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Add'))
sel.btnMinus = Button(master,text = '-', width = 4, height=3)
sel.btnMinus.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Subtract'))
sel.btnStar = Button(master,text = '*', width = 4, height=3)
sel.btnStar.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Multiply'))
sel.btnDiv = Button(master,text = '/', width = 4, height=3)
sel.btnDiv.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Divide'))
sel.btnEqual = Button(master, text = '=')
sel.btnEqual.bind('<ButtonRelease-1>', lambda e: self.funcFuncButton('Eq'))
```

Here are the four buttons that do our math functions.

```python
self.btnClear = Button(master, text = 'CLEAR')
sel.btnClear.bind('<ButtonRelease-1>', lambda e: self.funcClear())
```

Finally, here is the clear button. It, of course, clears the holder variables and the display. Now we place the widgets in the PlaceWidget routine. First, we initialize the grid, then start putting the widgets into the grid. Here's the first part of the routine.

```python
def PlaceWidgets(self,master):
master.grid(column=0,row=0)
sel.lblDisplay.grid(column=0,row=0,columnspan = 4,sticky=EW)
sel.btn1.grid(column = 0, row = 1)
sel.btn2.grid(column = 1, row = 1)
sel.btn3.grid(column = 2, row = 1)
sel.btn4.grid(column = 0, row = 2)
sel.btn5.grid(column = 1, row = 2)
sel.btn6.grid(column = 2, row = 2)
sel.btn7.grid(column = 0, row = 3)
sel.btn8.grid(column = 1, row = 3)
sel.btn9.grid(column = 2, row = 3)
sel.btn0.grid(column = 1, row = 4)
```
function. In this case, we are setting up the name of the callback function and the value we want to send as well as the event tag (e:). We’ll talk more about Lambda in a later article. For now, just follow the example.

I've given you the first four buttons. Copy and paste the above code for buttons 5 through 9 and button 0. They are all the same with the exception of the button name and the value we send the callback. Next steps are shown right.

The only thing that hasn't been covered before are the colspan and sticky attributes. As I mentioned before, a widget can span more than one column or row. In this case, we are “stretching” the label widget across all four columns. That’s what the “colspan” attribute does. There's a “rowspan” attribute as well. The “sticky” attribute tells the widget where to align its edges. Think of it as how the widget fills itself within the grid. Above left is the rest of our buttons.

Before we go any further let's take a look at how things will work when the user presses buttons.

Let's say the user wants to enter 563 + 127 and get the answer. They will press or click (logically) 5, then 6, then 3, then the “+,” then 1, then 2, then 7, then the “=” buttons. How do we handle this in code? We have already set the callbacks for the number buttons to the funcNumButton function. There's two ways to handle this. We can keep it as a number the entire time. We will use the latter method. To do this, we will keep the value that is already there (0 when we start) in a variable called “self.CurrentValue”, When a number comes in, we take the value, multiply it by 10 and add the new value. So, when the user enters 5, 6 and 3, we do the following...

```python
def funcNumButton(self,val):
    if self.DecimalNext == True:
        self.DecimalCount += 1
    else:
        self.CurrentValue = (self.CurrentValue * 10) + val
    self.DisplayIt()
```

Of course we then display the “self.CurrentValue” variable in the label.

Next, the user clicks the “+” key. We take the value in “self.CurrentValue” and place it into the variable “self HOLDER VALUE,” and reset the “self.CurrentValue” to 0. We then repeat the process for the clicks on 1, 2 and 7. When the user clicks the “=” key, we then add the values in “self.CurrentValue” and “self HOLDER VALUE,” display them, then clear both variables to continue.

User clicks 5 - 0 * 10 + 5 (5)
User clicks 6 - 5 * 10 + 6 (56)
User clicks 3 - 56 * 10 + 3 (563)
Above is the code to start defining our callbacks.

The “funcNumButton routine receives the value we passed from the button press. The only thing that is different from the example above is what if the user pressed the decimal button (‘.’). Below, you’ll see that we use a boolean variable to hold the fact they pressed the decimal button, and, on the next click, we deal with it. That’s what the “if
self.DecimalNext == True:” line is all about. Let’s walk through it.

The user clicks 3, then 2, then the decimal, then 4, to create “32.4”. We handle the 3 and 2 clicks through the “funcNumButton” routine. We check to see if self.DecimalNext is True (which in this case it isn’t until the user clicks the “.” button). If not, we simply multiply the held value (self.CurrentValue) by 10 and add the incoming value. When the user clicks the “.”, the callback “funcFuncButton” is called with the “Dec” value. All we do is set the boolean variable “self.DecimalNext” to

```python
def funcFuncButton(self, function):
    if function == 'Dec':
        self.DecimalNext = True
    else:
        self.DecimalNext = False
        self.DecimalCount = 0
        if function == 'ABS':
            self.CurrentValue *= -1
            self.DisplayIt()

    The ABS function simply takes the current value and multiplies it by -1.

    elif function == 'Add':
        self.HolderValue = self.CurrentValue
        self.CurrentValue = 0
        self.CurrentFunction = 'Add'


    elif function == 'Subtract':
        self.HolderValue = self.CurrentValue
        self.CurrentValue = 0
        self.CurrentFunction = 'Subtract'

    elif function == 'Multiply':
        self.HolderValue = self.CurrentValue
        self.CurrentValue = 0
        self.CurrentFunction = 'Multiply'

    elif function == 'Divide':
        self.HolderValue = self.CurrentValue
        self.CurrentValue = 0
        self.CurrentFunction = 'Divide'

    The “Eq” function (Equals) is where the “magic” happens. It will be easy for you to understand the following code by now.

    elif function == 'Eq':
        if self.CurrentFunction == 'Add':
            self.CurrentValue += self.HolderValue
        elif self.CurrentFunction == 'Subtract':

        elif self.CurrentFunction == 'Multiply':
            self.CurrentValue *= self.HolderValue

        elif self.CurrentFunction == 'Divide':

        self.DisplayIt()
        self.CurrentValue = 0
        self.HolderValue = 0
```
True. When the user clicks the 4, we will test the “self.DecimalNext” value and, since it's true, we play some magic. First, we increment the self.DecimalCount variable. This tells us how many decimal places we are working with. We then take the incoming value, multiply it by (10**self.DecimalCount). Using this magic operator, we get a simple “raised to the power of” function. For example 10**2 returns 100. 10**-2 returns 0.01. Eventually, using this routine will result in a rounding issue, but for our simple calculator, it will work for most reasonable decimal numbers. I'll leave it to you to work out a better function. Think of this as your homework for this month.

The “funcClear” routine simply clears the two holding variables, then sets the display.

def funcClear(self):
    self.CurrentValue = 0
    self.HolderValue = 0
    self.DisplayIt()

Now the functions. We've already discussed what happens with the function 'Dec'. We set this one up first with the “if” statement. We go to the “else,” and if the function is anything else, we clear the “self.DecimalNext” and “self.DecimalCount” variables.

The next set of steps are shown on the previous page (right hand box).

The DisplayIt routine simply sets the value in the display label. Remember we told the label to “monitor” the variable “self.CurrentDisplay”. Whenever it changes, the label automatically changes the display to match. We use the “.set” method to change the value.

def DisplayIt(self):
    print('CurrentValue = {0} - HolderValue = {1}'.format(self.CurrentValue ,self.HolderValue))

Finally we have our startup lines.

if __name__ == '__main__':
    StartUp()

    Now you can run the program and give it a test.

    As always, the code for this article can be found at PasteBin. Examples 1, 2 and 3 are at:
    http://pastebin.com/mBAS1Um and the Calc.py example is at:
    http://pastebin.com/LbMibF0u

    Next month, we will continue looking at Tkinter and its
    wealth of widgets. In a future article, we'll look at a GUI
    designer for tkinter called PAGE. In the meantime, have
    fun playing. I think you'll enjoy Tkinter.

    "Greg Walters is owner of RainyDay Solutions, LLC, a
    consulting company in Colorado and has been programming since
    1972. He enjoys cooking, hiking, music, and spending time with
    his family. His website is www.thedesigantedgeek.com."
Radiobuttons are considered a one of many type selection widget. It also has two options, on and off. However, they are grouped together to provide a set of options that logically can have only one selection. You can have multiple groups of Radiobuttons that, if properly programmed, won't interact with each other.

A Listbox provides a list of items for the user to select from. Most times, you want the user to select only one of the items at a time, but there can be occasions that you will allow the user to select multiple items. A scroll bar can be placed either horizontally or vertically to allow the user to easily look through all the items available.

Our project will consist of a main window and seven main frames that visually group our widget sets:

• The first frame will be very basic. It simply consists of various labels, showing the different relief options.
  • The second will contain buttons, again pretty simple, that use the different relief options.
  • In this frame, we'll have two checkboxes and a button that can programatically toggle them, and they will send their state (1 or 0) back to the terminal window when clicked or toggled.
  • Next, we'll have two groups of three radio buttons, each sending a message to the terminal window when clicked. Each group is separate.
    • This has some text or entry boxes, which aren't new to you, but there's also a button to enable and disable one of them. When disabled, no entry can be made to that textbox.
    • This is a list box with a vertical scroll bar that sends a message to the terminal whenever an item is selected, and will have two buttons. One button will clear the list box and the other will fill it with some dummy values.

So now, we'll start our project. Let's name it "widgetdemo1.py". Be sure to save it because we will be writing our project in little pieces, and build on them to make our full app. Each piece revolves around one of the frames. You'll notice that I'm including a number of comments as we go, so you can refer back to what's happening. Above are first few lines.
def BuildWidgets(self, master):
    # Define our widgets
    frame = Frame(master)
    # Labels
    self.lblframe = Frame(frame, relief = SUNKEN, padx = 3, pady = 3,
                          borderwidth = 2, width = 500)
    self.lbl1 = Label(self.lblframe, text="Flat Label", relief = FLAT,
                      width = 13, borderwidth = 2)
    self.lbl2 = Label(self.lblframe, text="Sunken Label", relief = SUNKEN,
                      width = 13, borderwidth = 2)
    self.lbl3 = Label(self.lblframe, text="Ridge Label", relief = RIDGE, width = 13,
                      borderwidth = 2)
    self.lbl4 = Label(self.lblframe, text="Raised Label", relief = RAISED,
                      width = 13, borderwidth = 2)
    self.lbl5 = Label(self.lblframe, text="Groove Label", relief = GROOVE,
                      width = 13, borderwidth = 2)
    return frame

The first two lines (comments) are the name of the application and what we are concentrating on in this part. Line three is our import statement. Then we define our class. The next line starts our __init__ routine, which you all should be familiar with by now, but, if you are just joining us, it's the code that gets run when we instantiate the routine in the main portion of the program. We are passing it the Toplevel or root window, which comes in as master here. The last three lines (so far), call three different routines. The first (DefineVars) will set up various variables we'll need as we go. The next (BuildWidgets) will be where we define our widgets, and the last (PlaceWidgets) is where we actually place the widgets into the root window. As we did last time, we'll be using the grid geometry manager. Notice that BuildWidgets will return the object "f" (which is our root window), and we'll pass that along to the PlaceWidgets routine.

Above right is our BuildWidgets routine. Each of the lines that start with "self." have been split for two reasons. First, it's good practice to keep the line length to 80 characters or less. Secondly, it makes it easier on our wonderful editor. You can do two things. One, just make each line long, or keep it as is. Python lets us split lines as long as they are within parentheses or brackets. As I said earlier, we are defining the widgets before we place them in the grid. You'll notice when we do the next routine, that we can also define a widget at the time we place it in the grid, but defining it before we put it in the grid in a routine like this makes it easier to keep track of everything, since we are doing (most of) the definitions in this routine.

So, first we define our master frame. This is where we will be putting the rest of our widgets. Next, we define a child (of the master frame) that will hold five labels, and call it lblframe. We set the various attributes of the frame here. We set the relief to 'SUNKEN', a padding of 3 pixels on left and right (padx), and 3 pixels on the top and bottom (pady). We also set the borderwidth to 2 pixels so that its sunken relief is noticeable.

By default, the borderwidth is set to 0, and the effect of being sunken won't be noticed. Finally, we set the total width of the frame to 500 pixels.

Next, we define each label widget that we will use. We set the parent as self.lblframe, and not to frame. This way all the labels are children of lblframe, and lblframe is a child of frame. Notice that each definition is pretty much the same for all five of the labels except the name of the widget (lbl1, lbl2, etc), the text, and the relief or visual effect. Finally, we return the frame back to the calling routine (__init__).
The following page (top right) shows our PlaceWidgets routine.

We get the frame object in as a parameter called master. We assign that to 'frame' to simply be consistent with what we did in the BuildWidgets routine. Next, we set our main grid up (frame.grid(column = 0, row = 0)). If we don't do this, nothing works correctly. Then we start putting our widgets into the grid locations. First we put the frame (lblframe) that holds all our labels, and set its attributes. We put it in column 0, row 1, set the padding to 5 pixels on all sides, tell it to span 5 columns (left and right), and finally use the "sticky" attribute to force the frame to expand fully to the left and right ("WE", or West and East). Now comes the part that sort of breaks the rule that I told you about. We are placing a label as the first widget in the frame, but we didn't define it ahead of time. We define it now. We set the parent to lblframe, just like the other labels. We set the text to "Labels |", the width to 15, and the anchor to east ('e'). If you remember from last time, using the anchor attribute, we can set where in the widget the text will display. In this case, it's along the right border. Now the fun part. Here we define the grid location (and any other grid attributes we need to), simply by appending ".grid" at the end of the label definition.

Next, we lay out all of our other labels in the grid - starting at column 1, row 0.

Here is our DefineVars routine. Notice that we simply use the pass statement for now. We'll be filling it in later on, and we don't need it for this part:

```python
def DefineVars(self):
    # Define our resources
    pass
```

And lastly we put in our main routine code:

```python
def PlaceWidgets(self, master):
    frame = master
    # Place the widgets
    frame.grid(column = 0, row = 0)
    # Place the labels
    self.lblframe.grid(column = 0, row = 1, padx = 5, pady = 5, columnspan = 5, sticky='WE')
    l = Label(self.lblframe, text='Labels |', width=15, anchor='e').grid(column=0,row=0)
    self.lbl1.grid(column = 1, row = 0, padx = 3, pady = 5)
    self.lbl2.grid(column = 2, row = 0, padx = 3, pady = 5)
    self.lbl3.grid(column = 3, row = 0, padx = 3, pady = 5)
    self.lbl4.grid(column = 4, row = 0, padx = 3, pady = 5)
    self.lbl5.grid(column = 5, row = 0, padx = 3, pady = 5)
```

First, we instantiate an instance of Tk. Then we set the size of the main window to 750 pixels wide by 40 pixels high, and locate it at 150 pixels from the left and top of the screen. Then we set the title of the window and instantiate our Demo object, and finally call the Tk mainloop.

Give it a try. You should see the five labels plus the “last minute” label in various glorious effects.

**Buttons**

Now save what you have as widgetdemo1a.py, and let's add some buttons. Since we built our base program to be added to, we'll simply add the parts that apply. Let's start with:

```python
# Place the buttons
self.btnframe.grid(column=0, row = 2, padx = 5, pady = 5, columnspan = 5, sticky = 'WE')
l = Label(self.btnframe, text='Buttons |', width=15, anchor='e').grid(column=0,row=0)
self.btn1.grid(column = 1, row = 0, padx = 3, pady = 3)
self.btn2.grid(column = 2, row = 0, padx = 3, pady = 3)
self.btn3.grid(column = 3, row = 0, padx = 3, pady = 3)
self.btn4.grid(column = 4, row = 0, padx = 3, pady = 3)
self.btn5.grid(column = 5, row = 0, padx = 3, pady = 3)
```
the BuildWidgets routine. After
the labels definitions, and
before the “return frame” line, 
add what is shown on the next
page, top right.

Nothing really new here.
We’ve defined the buttons, with
their attributes, and set their
callbacks via the .bind
configuration. Notice that we
are using lambda to send the
values 1 through 5 based on
which button is clicked. In the
callback, we’ll use that so we
know which button we are
dealing with. Now we’ll work in
the PlaceWidgets routine. Put
the code below after the last
label placement.

Once again, nothing really
new here, so we’ll move on.
Bottom right is our callback
routine. Put it after the
DefineVars routine.

Again, nothing really fancy
here. We just use a series of
IF/ELIF routines to print what
button was clicked. The main
ting to look at here (when we
run the program) is that the
sunken button doesn’t “move”
when you click on it. You would
not usually use the sunken

relief unless you
were making a
button that stays
“down” when you
click it. Finally, we
need to tweak the
geometry
statement to
support the extra
widgets we put in:

root.geometry(’750
x110+150+150’)

Ok. All done with
this one. Save it
and run it.

Now save this as
widgetdemo1b.py, and we’ll
move on to checkboxes.

Checkboxes

As I said earlier, this part of
the demo has a normal button
and two checkboxes. The first
checkbox is what you would
normally expect a checkbox to
look like. The second is more
like a “sticky” button - when it’s
not selected (or checked), it
looks like a normal button.
When you select it, it looks like
a button that is stuck down. We
can do this by simply setting
the indicator attribute to

# Buttons
self.btnframe = Frame(frame,relief = SUNKEN,padx = 3, pady = 3,
borderwidth = 2, width = 500)
self.btn1 = Button(self.btnframe,text="Flat Button",
relief = FLAT, borderwidth = 2)
self.btn2 = Button(self.btnframe,text="Sunken Button",
relief = SUNKEN, borderwidth = 2)
self.btn3 = Button(self.btnframe,text="Ridge Button",
relief = RIDGE, borderwidth = 2)
self.btn4 = Button(self.btnframe,text="Raised Button",
relief = RAISED, borderwidth = 2)
self.btn5 = Button(self.btnframe,text="Groove Button",
relief = GROOVE, borderwidth = 2)
self.btn1.bind(’<ButtonRelease-1>’,lambda e: self.BtnCallback(1))
self.btn2.bind(’<ButtonRelease-1>’,lambda e: self.BtnCallback(2))
self.btn5.bind(’<ButtonRelease-1>’,lambda e: self.BtnCallback(5))

def BtnCallback(self,val):
    if val == 1:
        print(”Flat Button Clicked...”)
    elif val == 2:
        print(”Sunken Button Clicked...”)
    elif val == 3:
        print(”Ridge Button Clicked...”)
    elif val == 4:
        print(”Raised Button Clicked...”)
    elif val == 5:
        print(”Groove Button Clicked...”)

False. The “normal” button will
toggle the checkboxes from
checked to unchecked, and vice
versa, each time you click the
button. We get to do this
programmatically by calling the
.toggle method attached to the
checkbox. We bind the left
mouse button click event
(button release) to a function
so we can send a message (in
this case) to the terminal. In
addition to all of this, we are
setting two variables (one for
each of the checkboxes) that
we can query at any time. In
this case, each time the checkbox is clicked we query this value and print it. Pay attention to the variable portion of the code. It is used in many widgets.

Under the BuildWidget routine, after the button code we just put in and before the return statement, put the code shown on the next page, top right.

Again, you have seen all of this before. We create the frame to hold our widgets. We set up a button and two check boxes. Let’s place them now using the code on the next page, middle right.

Now we define the two variables that we will use to monitor the value of each check box. Under DefineVars, comment out the pass statement, and add this...

```python
self.Chk1Val = IntVar()
self.Chk2Val = IntVar()
```

After the button callback return, put the text shown bottom right.

And finally replace the geometry statement with this:

```python
root.geometry('750x170+150+150')
```

# Check Boxes
```python
self.cbframe = Frame(frame, relief = SUNKEN, padx = 3, pady = 3, borderwidth = 2, width = 500)
self.chk1 = Checkbutton(self.cbframe, text = "Normal Checkbox", variable=self.Chk1Val)
self.chk2 = Checkbutton(self.cbframe, text = "Checkbox", variable=self.Chk2Val, indicatoron = False)
self.chk1.bind('<ButtonRelease-1>',lambda e: self.ChkBoxClick(1))
self.chk2.bind('<ButtonRelease-1>',lambda e: self.ChkBoxClick(2))
self.btnToggleCB = Button(self.cbframe,text="Toggle Cbs")
self.btnToggleCB.bind('<ButtonRelease-1>',self.btnToggle)
```

# Place the Checkboxes and toggle button
```python
self.cbframe.grid(column = 0, row = 3, padx = 5, pady = 5, columnspan = 5,sticky = 'WE')
1 = Label(self.cbframe,text='Check Boxes |',width=15, anchor='e').grid(column=0,row=0)
self.btnToggleCB.grid(column = 1, row = 0, padx = 3, pady = 3)
self.chk1.grid(column = 2, row = 0, padx = 3, pady = 3)
self.chk2.grid(column = 3, row = 0, padx = 3, pady = 3)
```

```python
def btnToggle(self,p1):
    self.chk1.toggle()
    self.chk2.toggle()
    print("Check box 1 value is {0}".format(self.Chk1Val.get()))
    print("Check box 2 value is {0}".format(self.Chk2Val.get()))

def ChkBoxClick(self,val):
    if val == 1:
        print("Check box 1 value is {0}".format(self.Chk1Val.get()))
    elif val == 2:
        print("Check box 2 value is {0}".format(self.Chk2Val.get()))
```
In this demo, the first group of buttons is grouped by the variable named self.RBVal. The second is grouped by the variable self.RBValue2. We also need to set the value attribute at design time. This ensures that the buttons will return a value that makes sense whenever they are clicked.

Back to BuildWidgets, and, just before the return statement, add the code shown on the following page.

```python
# Radio Buttons
self.rbframe = Frame(frame, relief = SUNKEN, padx = 3, pady = 3, borderwidth = 2, width = 500)
self.rb1 = Radiobutton(self.rbframe, text = "Radio 1", variable = self.RBVal, value = 1)
self.rb2 = Radiobutton(self.rbframe, text = "Radio 2", variable = self.RBVal, value = 2)
self.rb3 = Radiobutton(self.rbframe, text = "Radio 3", variable = self.RBVal, value = 3)
self.rb1.bind('<ButtonRelease-1>',lambda e: self.RBClick())
self.rb2.bind('<ButtonRelease-1>',lambda e: self.RBClick())
self.rb3.bind('<ButtonRelease-1>',lambda e: self.RBClick())
self.rb4 = Radiobutton(self.rbframe, text = "Radio 4", variable = self.RBVal2, value = "1-1")
self.rb5 = Radiobutton(self.rbframe, text = "Radio 5", variable = self.RBVal2, value = "1-2")
self.rb6 = Radiobutton(self.rbframe, text = "Radio 6", variable = self.RBVal2, value = "1-3")
self.rb4.bind('<ButtonRelease-1>',lambda e: self.RBClick2())
self.rb5.bind('<ButtonRelease-1>',lambda e: self.RBClick2())
self.rb6.bind('<ButtonRelease-1>',lambda e: self.RBClick2())

In PlaceWidgets, add this:

```python
# Place the Radio Buttons and select the first one
l = Label(self.rbframe, text='Radio Buttons |',
          width=15, anchor='e').grid(column=0, row=0)
self.rb1.grid(column = 2, row = 0, padx = 3, pady = 3, sticky = 'EW')
self.rb2.grid(column = 3, row = 0, padx = 3, pady = 3, sticky = 'WE')
self.rb3.grid(column = 4, row = 0, padx = 3, pady = 3, sticky = 'WE')
self.RBVal.set("1")
l = Label(self.rbframe, text='| Another Set |',
          width=15,
          anchor = 'e').grid(column = 5, row = 0)
self.rb4.grid(column = 6, row = 0)
self.rb5.grid(column = 7, row = 0)
self.rb6.grid(column = 8, row = 0)
self.RBVal2.set("1-1")
```
print("Radio Button clicked - Value is \{0\}".format(self.RBVal.get()))
}
def RBClick2(self):
    print("Radio Button clicked - Value is \{0\}".format(self.RBVal2.get()))

and, finally, rework the geometry statement as follows.
	next, add this code to the PlaceWidget routine:
    # Place the Textboxes
    self.tbframe.grid(column = 0, row = 5, padx = 5, pady = 5,
        columnspan = 5, sticky = 'WE')
    l = Label(self.tbframe,text='Textboxes |',width=15,
        anchor='e').grid(column=0,row=0)
    self.txt1.grid(column = 2, row = 0, padx = 3, pady = 3)
    self.txt2.grid(column = 3, row = 0, padx = 3, pady = 3)
    self.btnDisable.grid(column = 1, row = 0, padx = 3, pady = 3)

Add this line to the bottom of the DefineVars routine:
    self.Disabled = False

Now, add the function that responds to the button click event:
    def btnDisableClick(self,pl):
        if self.Disabled == False:
            self.Disabled = True
            self.txt2.configure(state='disabled')
        else:
            self.Disabled = False
            self.txt2.configure(state='normal')

And finally, rework the geometry statement:
    root.geometry('750x270+150+150')

Save it as widgetdemo1d.py, and run it.
HOWTO - PROGRAM IN PYTHON - PART 26

# Place the Listbox and support buttons
self.lstframe.grid(column = 0, row = 6, padx = 5,
pady = 5, columnspan = 5,sticky = 'WE')
1 = Label(self.lstframe,text='List Box |',width=15,
anchor='e').grid(column=0,row=0,rowspan=2)
selself.lobx.grid(column = 2, row = 0,rowspan=2)
selself.VScroll.grid(column = 3, row = 0,rowspan = 2,
sticky = 'NSW')
self.btnClearLBox.grid(column = 1, row = 0, padx =
5)
selself.btnFillLBox.grid(column = 1, row = 1, padx =
5)

In DefineVars add this...
# List for List box items
self.exmaples = ['Item One','Item Two','Item
Three','Item Four']

And add the following support routines:
def ClearList(self):
    self.lbox.delete(0,END)

def FillList(self):
    # Note, clear the listbox first...no check is done
    for ex in self.examples:
        self.lbox.insert(END,ex)
    # insert([0,ACTIVE,END],item)

def LBoxSelect(self,p):
    print("Listbox Item clicked")
    items = self.lbox.cuselction()
    selitem = items[0]
    print("Index of selected item =
{0}".format(selitem))
    print("Text of selected item =
{0}".format(self.lbox.get(selitem)))

Finally, update the geometry line.
root.geometry('750x370+150+150')

Save this as widgetdemo1.py, and run it. Now we will do our last
modifications to our application.

# List Box Stuff
self.lstframe = Frame(frame,
    relief = SUNKEN,
padx = 3,
pady = 3,
borderwidth = 2,
width = 500
)
# Scrollbar for list box
self.VScroll = Scrollbar(self.lstframe)
selself.lbox = Listbox(self.lstframe,
    height = 5,
yscrollcommand = self.VScroll.set)
# default height is 10
self.lbox.bind('<<ListboxSelect>>',self.ListboxSelect)
selself.VScroll.config(command =
selself.lbox.yview)
self.btnClearLBox = Button(  
    self.lstframe,
    text = "Clear List",
    command = self.ClearList,
    width = 11
)
self.btnFillLBox = Button(  
    self.lstframe,
    text = "Fill List",
    command = self.FillList,
    width = 11
)
<<ListboxSelect>> is virtual event
# Fill the list box
self.FillList()

Now, we'll deal with the additional code for the
PlaceWidgets routine, and that's shown on the following
page, left side.

Message Dialogs

This section is simply a series of “normal” buttons that
will call various types of Message Dialogs. We've done
them before in a different GUI
toolkit. We will explore only 5 different types, but there are more. In this section, we'll look at Info, Warning, Error, Question, and Yes/No dialogs. These are very useful when you need to pass some information to your user in a rather big way. In the BuildWidgets routine add the code shown below.

Here is the support routine. For the first three (Info, Warning, and Error), you simply call `tkMessageBox.showinfo`, `tkMessageBox.showwarning`, or `tkMessageBox.showerror`, or whichever you need, with two parameters. First is the title for the message dialog, and second is the actual message you want to show. The icon is handled for you by tkinter. For the dialogs that provide a response (question, yes/no), we provide a variable that receives the value of which button was clicked. In the case of the question dialog, the response is either “yes” or “no”, and, in the case of the yes/no dialog, the response is either “True” or “False”.

Finally, modify the geometry line:

```
root.geometry('750x490+550+150')
```

Save this as `widgetdemo01f.py`, and play away.

I've put the code for `widgetdemo01f.py` on pastebin at

```
def ShowMessageBox(self, which):
    if which == 1:
        tkMessageBox.showinfo('Demo','This is an INFO messagebox')
    elif which == 2:
        tkMessageBox.showwarning('Demo','This is a WARNING messagebox')
    elif which == 3:
        tkMessageBox.showerror('Demo','This is an ERROR messagebox')
    elif which == 4:
        resp = tkMessageBox.askquestion('Demo','This is a QUESTION messagebox?')
        print('{} was pressed...'.format(resp))
    elif which == 5:
        resp = tkMessageBox.askyesno('Demo','This is a YES/NO messagebox')
        print('{} was pressed...'.format(resp))

# Buttons to show message boxes and dialogs
self.mbframe = Frame(frame,relief = SUNKEN,padx = 3, pady = 3, borderwidth = 2)
self.btnMBInfo = Button(self.mbframe,text = "Info")
self.btnMBWarning = Button(self.mbframe,text = "Warning")
self.btnMBError = Button(self.mbframe,text = "Error")
self.btnMBQuestion = Button(self.mbframe,text = "Question")
self.btnMBYesNo = Button(self.mbframe,text = "Yes/No")
self.btnMBInfo.bind('<ButtonRelease-1>', lambda e: self.ShowMessageBox(1))
self.btnMBWarning.bind('<ButtonRelease-1>', lambda e: self.ShowMessageBox(2))
self.btnMBError.bind('<ButtonRelease-1>', lambda e: self.ShowMessageBox(3))
self.btnMBQuestion.bind('<ButtonRelease-1>', lambda e: self.ShowMessageBox(4))
self.btnMBYesNo.bind('<ButtonRelease-1>', lambda e: self.ShowMessageBox(5))
```

Now, add the code for the PlaceWidgets routine:

```
# Messagebox buttons and frame
self.mbframe.grid(column = 0,row = 7, columnspan = 5, padx = 5, sticky = 'WE')
l = Label(self.mbframe,text='Message Boxes [ ]',width=15, anchor='e').grid(column=0,row=0)
self.btnMBInfo.grid(column = 1, row = 0, padx= 3)
self.btnMBWarning.grid(column = 2, row = 0, padx= 3)
self.btnMBError.grid(column = 3, row = 0, padx= 3)
self.btnMBQuestion.grid(column = 4, row = 0, padx= 3)
self.btnMBYesNo.grid(column = 5, row = 0, padx= 3)
```
If you’ve ever waited in line to buy a movie ticket, you’ve been in a queue. If you’ve ever had to wait in traffic at rush hour, you’ve been in a queue. If you’ve ever waited in a government office with one of those little tickets that says you’re number 98, and the sign says "Now serving number 42," you’ve been in a queue.

In the world of computers, queues are common. As a user, most times, you don’t have to think about them. They are invisible to the user. But if you ever have to deal with realtime events, you will eventually have to deal with them. It’s just data of one type or another, waiting in line for its turn to be processed. Once it's in the queue, it's there until it gets accessed, and then it's gone. You can’t get the value of the next data item unless you pull it out of the queue. You can’t, for example, get the value of the 15th item in the queue. You have to access the other 14 items first. Once it's accessed, it's out of the queue. It's gone, and unless you save it to a long-term variable, there's no way to get the data back.

There are multiple types of queues. The most common ones are FIFO (First In, First Out), LIFO (Last In, First Out), Priority, and Ring. We’ll talk about ring queues another time.

FIFO queues are what we see in everyday life. All of the examples I listed above are FIFO queues. The first person in the line gets handled first, moves on, then everyone moves up one spot in the line. In a FIFO buffer, there is (within reason) no limit to the number of items it can hold. They just stack up in order. As an item is handled, it is pulled out (or dequeued) of the queue, and everything moves closer to the front of the queue by one position.

LIFO Queues are less common in life, but there are still real-world examples. The one that comes to mind most quickly is a stack of dishes in your kitchen cabinet. When the dishes are washed and dried, they get stacked in the cabinet. The last one in on the stack is the first one that comes out to be used. All the rest have to wait, maybe for days, to be used. It’s a good thing that the movie ticket queue is FIFO, isn’t it? Like the FIFO queue, within reason, there is no limit to the size of a LIFO queue. The first item in the queue has to wait as newer items are pulled out of the buffer (plates pulled off the stack) until it's the only one left.

Priority queues are a bit harder for many people to imagine right off the bat. Think of a company that has one printer. Everyone uses that one printer. The print jobs are handled by department priority. Payroll has a higher priority (and thankfully so) than say, you, a programmer. You have a higher priority (and thankfully so) than the receptionist. So in short, the data that has a higher priority gets handled, and gets out of the queue, before data that has a lower priority.

FIFO

FIFO queues are easy to visualize in terms of data. A python list is an easy mental representation. Consider this list...

\[1, 2, 3, 4, 5, 6, 7, 8, 9, 10\]

There are 10 items in the list. As a list, you access them by index. However, in a queue, you can’t access the items by index. You have to deal with the next one in the line and the list isn’t static. It’s VERY dynamic. As we request the next item in the queue, it gets removed. So using the example above, you request one item from the
import Queue
first = Queue.Queue()
for i in range(5):
    first.put(i)
while not first.empty():
    print first.get()

queue. It returns the first item (1) and the queue then looks like this.

\[2, 3, 4, 5, 6, 7, 8, 9, 10\]

Request two more and you get 2, then 3, returned, and then the queue looks like this.

\[4, 5, 6, 7, 8, 9, 10\]

I'm sure you get the idea. Python provides a simple library, surprisingly enough, called Queue, that works well for small-to-medium sized queues, up to about 500 items. Above is a simple example to show it.

In this example, we initialize the queue (first = Queue.Queue()) then put the numbers 0 through 4 into our queue (first.put(i)). We then use the internal method .get() to pull items off the queue until the queue is empty, .empty(). What is returned is 0, 1, 2, 3, 4. You can also set the maximum number of items that the queue can handle by initializing it with the size of the queue like this.

first = Queue.Queue(300)

Once the maximum number of items have been loaded, the Queue blocks any additional entries going into the queue. This has a side effect of making the program look like it's "locked" up, though. The easiest way to get around this is to use the Queue.full() check (above right).

In this case, the queue is set for a maximum of 12 items. As we put items into the queue, we start with '0' and get up to '11'. When we hit number 12, though, the buffer is already full. Since we check to see if the buffer is full before we try to put the item in, the last item is simply discarded.

There are other options, but they can cause other side-effects, and we will address this in a future article. So, for the majority of the time, the bottom line is either use a queue with no limit or make sure you have more space in your queue than you will need.

**LIFO**

import Queue
lifo = Queue.LifoQueue()
for i in range(5):
    lifo.put(i)
while not lifo.empty():
    print lifo.get()

The Queue library also supports LIFO queues. We'll use the above list as a visual example. Setting up our queue, it looks like this:

\[1, 2, 3, 4, 5, 6, 7, 8, 9, 10\]

Pulling three items from the queue, it then looks like this:

\[1, 2, 3, 4, 5, 6, 7\]

Remember that in a LIFO queue, items are removed in a LAST-in FIRST-out order. Here's the simple example modified for a LIFO queue:

When we run it, we get "4,3,2,1,0".

As with the FIFO queue, you have the ability to set the size of the queue, and you can use the .full() check.

pq = Queue.PriorityQueue()
pq.put((3, 'Medium 1'))
pq.put((4, 'Medium 2'))
pq.put((10, 'Low'))
pq.put((1, 'High'))

while not pq.empty():
    next = pq.get()
    print next
    print next[1]

**PRIORITY**

While it's not often used, a Priority queue can sometimes be helpful. It's pretty much the same as the other queue structures, but we need to pass a tuple that holds both the priority and the data. Here's an example using the Queue

(1, 'High')
high
(3, 'Medium')
Medium
(4, 'Medium')
Medium
(10, 'Low')
Low

\[1, 'High'
high
\[3, 'Medium'
Medium
\[4, 'Medium'
Medium
\[10, 'Low'
Low
library:

First, we initialize the queue. Then we put four items into the queue. Notice we use the format (priority, data) to put our data. The library sorts our data in an ascending order based on the priority value. When we pull the data, it comes back as a tuple, just like we put it in. You can address by index the data. What we get back is...

In our first two examples, we simply printed the data that comes out of our queue. That’s fine for these examples, but in real-world programming, you probably need to do something with that information as soon as it comes out of the queue, otherwise it’s lost. When we use the ‘print fifo.get’, we send the data to the terminal and then it’s destroyed. Just something to keep in mind.

Now let’s use some of what we’ve already learned about tkinter to create a queue demo program. This demo will have two frames. The first will contain (to the user) three buttons. One for a FIFO queue, one for a LIFO queue, and one for a PRIORITY queue. The second frame will contain an entry widget, two buttons, one for adding to the queue, and one for pulling from the queue, and three labels, one showing when the queue is empty, one showing when the queue is full, and one to display what has been pulled from the queue. We’ll also be writing some code to automatically center the window within the screen. Above left is the beginning of the code.

Here we have our imports and the beginning of our class. As before, we create the \_init\_ routine with the DefineVars, BuildWidgets, and PlaceWidgets routines. We also have a routine called ShowStatus (above right) which

```python
import sys
from Tkinter import *
import ttk
import tkMessageBox
import Queue

class QueueTest:
    def __init__(self, master = None):
        self.DefineVars()
        f = self.BuildWidgets(master)
        self.PlaceWidgets(f)
        self.ShowStatus()

    def DefineVars(self):
        self.QueueType = ''
        self.FullStatus = StringVar()
        self.EmptyStatus = StringVar()
        self.Item = StringVar()
        self.Output = StringVar()
        # Define the queues
        self.fifo = Queue.Queue(10)
        self.lifo = Queue.LifoQueue(10)
        self.pq = Queue.PriorityQueue(10)
        self.obj = self.fifo

    def BuildWidgets(self, master):
        # Define our widgets
        frame = Frame(master)
        self.f1 = Frame(frame,
                        relief = SUNKEN,
                        borderwidth=2,
                        width = 300,
                        padx = 3,
                        pady = 3)
        
        self.btnFifo = Button(self.f1,
                              text = "FIFO")
        
        self.btnFifo.bind('<ButtonRelease-1>',
                           lambda e: self.btnMain(1))
        
        self.btnLifo = Button(self.f1,
                              text = "LIFO")
        
        self.btnLifo.bind('<ButtonRelease-1>',
                           lambda e: self.btnMain(2))
        
        self.btnPriority = Button(self.f1,
                                   text = "PRIORITY")
        
        self.btnPriority.bind('<ButtonRelease-1>',
                               lambda e: self.btnMain(3))
```

```
```
will... well, show the status of our queue.

We now create our DefineVars routine. We have four StringVar() objects, an empty variable called QueueType, and three queue objects - one for each of the types of queues that we are going to play with. We have set the maximum size of the queues at 10 for the purposes of the demo. We also have created an object called obj, and assigned it to the FIFO queue. When we select a queue type from the buttons, we will set this object to the queue that we want. This way, the queue is maintained when we switch to another queue type (code is on previous page, bottom right).

Here we start the widget definitions. We create our first frame, the three buttons, and their bindings. Notice we are using the same routine to handle the binding callback. Each button sends a value to the callback routine to denote which button was clicked. We could just as easily have created a dedicated routine for each button. However, since all three buttons are dealing with a common task, I thought it would be good to work them as a group (code shown right).

Next (below right), we set up the second frame, the entry widget, and the two buttons. The only thing here that is out of the ordinary is the binding for the entry widget. Here we bind the self.AddToQueue routine to the <Return> key. This way, the user doesn't have to use the mouse to add the data. They can just enter the data into the entry widget, and press <Return> if they want to.

Here (next page, bottom) is the last three widget definitions. All three are labels. We set the textvariable attribute to the variables we defined earlier. If you remember, when that variable changes, so does the text in the label. We also do something a bit different on the lblData label. We will use a different font to make it stand out when we display the data pulled from the queue. Remember that we have to return the frame object so it can be used in the

```python
self.f2 = Frame(frame,
    relief=SUNKEN,
    borderwidth=2,
    width=300,
    padx=3,
    pady=3
)
self.txtAdd = Entry(self.f2,
    width=5,
    textvar=self.Item
)
self.txtAdd.bind('<Return>',self.AddToQueue)
self.btnAdd = Button(self.f2,
    text='Add to Queue',
    padx=3,
    pady=3
)
self.btnAdd.bind('<ButtonRelease-1>',self.AddToQueue)
self.btnGet = Button(self.f2,
    text='Get Next Item',
    padx=3,
    pady=3
)
self.btnGet.bind('<ButtonRelease-1>',self.GetFromQueue)

self.lblEmpty = Label(self.f2,
    textvariable=self.EmptyStatus,
    relief=FLAT
)
self.lblFull = Label(self.f2,
    textvariable=self.FullStatus,
    relief=FLAT
)
self.lblData = Label(self.f2,
    textvariable=self.Output,
    relief=FLAT,
    font=('Helvetica',16),
    padx=5
)

return frame
```
PlaceWidget routine.

This (next page, middle) is the beginning of the PlaceWidgets routine. Notice here that we put five empty labels at the very top of the root window. I'm doing this to set spacing. This is an easy way to “cheat” and make your window placement much easier. We then set the first frame, then another “cheater” label, then the three buttons.

Here we place the second frame, another “cheater” label, and the rest of our widgets.

def Quit(self):
    sys.exit()

Next we have our “standard” quit routine which simply calls sys.exit() (above right).

def PlaceWidgets(self, master):
    frame = master
    # Place the widgets
    frame.grid(column = 0, row = 0)
    l = Label(frame,text=' ',relief=FLAT,width = 15, anchor = 'e').grid(column = 0, row = 0)
    l = Label(frame,text=' ',relief=FLAT,width = 15, anchor = 'e').grid(column = 1, row = 0)
    l = Label(frame,text=' ',relief=FLAT,width = 15, anchor = 'e').grid(column = 2, row = 0)
    l = Label(frame,text=' ',relief=FLAT,width = 15, anchor = 'e').grid(column = 3, row = 0)
    l = Label(frame,text=' ',relief=FLAT,width = 15, anchor = 'e').grid(column = 4, row = 0)

    self.f1.grid(column = 0,row = 1,sticky='nsew',columnspan=5,padx = 5, pady = 5)
    l = Label(self.f1,text=' ',width = 25,anchor = 'e').grid(column = 0, row = 0)
    self.btnClose.grid(column = 1,row = 0,padx = 4)
    self.btnLife.grid(column = 2,row = 0,padx = 4)
    self.btnClose.grid(column = 3, row = 0, padx = 4)

    self.f2.grid(column = 0,row = 2,sticky='nsew',columnspan=5,padx = 5, pady = 5)
    l = Label(self.f2,text=' ',width = 15,anchor = 'e').grid(column = 0, row = 0)
    self.txtAdd.grid(column=1,row=0)
    self.btnExit.grid(column=2,row=0)
    self.btnClose.grid(column=3,row=0)
    self.lblEmpty.grid(column=2,row=1)
    self.lblFull.grid(column=3,row = 1)
    self.lblData.grid(column = 4,row = 0)

    def btnMain(self,p1):
        if p1 == 1:
            self.QueueType = 'FIFO'
            self.obj = self.fifo
            root.title('Queue Tests - FIFO')
        elif p1 == 2:
            self.QueueType = 'LIFO'
            self.obj = self.lifo
            root.title('Queue Tests - LIFO')
        elif p1 == 3:
            self.QueueType = 'PRIORITY'
            self.obj = self.pq
            root.title('Queue Tests - Priority')
        print self.QueueType
        self.ShowStatus()
window (you don't really have to do that), and call the ShowStatus routine. Next (following page, top right) we'll make the ShowStatus routine.

As you can see, it's pretty simple. We set the label variables to their proper state so they display if the queue we are using is either full, empty, or somewhere in between.

The AddToQueue routine (next page, bottom right) is also fairly straight-forward. We get the data from the entry box using the .get() function. We then check to see if the current queue type is a priority queue. If so, we need to make sure it's in the correct format. We do that by checking for the presence of a comma. If it isn't, we complain to the user via an error message box. If everything seems correct, we then check to see if the queue that we are currently using is full. Remember, if the queue is full, the put routine is blocked and the program will hang. If everything is fine, we add the item to the queue and update the status.

The GetFromQueue routine (middle right) is even easier. We check to see if the queue is empty so as not to run into a blocking issue, and, if not, we pull the data from the queue, show the data, and update the status.

We are getting to the end of our application. Here is the center window routine (above left). We first get the screen width and screen height of the screen we are on. We then get the width and height of the root window by using the wininfo_screenwidth() and wininfo_screenheight() routines built into tkinter. These routines, when called at the

---

```
if __name__ == '__main__':
    def Center(window):
        # Get the width and height of the screen
        sw = window.wininfo_screenwidth()
        sh = window.wininfo_screenheight()
        # Get the width and height of the window
        rw = window.wininfo_reqwidth()
        rh = window.wininfo_reqheight()
        xc = (sw-rw)/2
        yc = (sh-rh)/2
        window.geometry("%dx%d+%d+%d"%(rw,rh,xc,yc))
        window.deiconify()

def ShowStatus(self):
    # Check for Empty
    if self.obj.empty() == True:
        self.EmptyStatus.set('Empty')
    else:
        self.EmptyStatus.set('')
    # Check for Full
    if self.obj.full() == True:
        self.FullStatus.set('FULL')
    else:
        self.FullStatus.set('')

def GetFromQueue(self,pl):
    self.Output.set('')
    if not self.obj.empty():
        temp = self.obj.get()
        self.Output.set("Pulled {0}".format(temp))
        self.ShowStatus()

def AddToQueue(self,pl):
    temp = self.Item.get()
    if self.QueueType == 'PRIORITY':
        comnapos = temp.find(',',')
        if comnapos == -1:
            print "ERROR"
            tkinter.messagebox.showerror('Queue Demo',
                'Priority entry must be in format \r\n(priority,data)')
        else:
            self.obj.put(self.Item.get())
    elif not self.obj.full():
        self.obj.put(self.Item.get())
        self.Item.set('')
        self.ShowStatus()
```
right time, will return the width and height of the root window based on the widget placement. If you call it too early, you'll get data, but it won't be what you really need. We then subtract the required window width from the screen width, and divide it by two, and do the same thing for the height information. We then use that information to set the geometry call. In MOST instances, this works wonderfully. However, there might be times that you need to set the required width and height by hand.

Finally, we instantiate the root window, set the base title, instantiate the QueueTest class. We then call root.after, which waits x number of milliseconds (in this case 3) after the root window is instantiated, and then calls the Center routine. This way, the root window has been completely set up and is ready to go, so we can get the root window width and height. You might have to tweak the delay time a bit. Some machines are much faster than others. 3 works fine on my

```python
root = Tk()
root.title('Queue Tests - FIFO')
demo = QueueTest(root)
root.after(3,Center,root)
```

machine, your mileage may vary. Last but not least, we call the root window mainloop to get the application to run.

As you play with the queues, notice that if you put some data in one queue (let's say the FIFO queue) then switch to another queue (let's say the LIFO queue), the data that was put into the FIFO queue is still there and waiting for you. You can completely or partially fill all three queues, then start playing with them.

Well, that's it for this time. Have fun with your queues. The QueueTest code can be found at http://pastebin.com/5BBUiDce.

Below Zero is a Co-located Server Hosting specialist in the UK.

Uniquely we only provide rack space and bandwidth. This makes our service more reliable, more flexible, more focused and more competitively priced. We concentrate solely on the hosting of Co-located Servers and their associated systems, within Scotland's Data Centres.

At the heart of our networking infrastructure is state-of-the-art BGP4 routing that offers optimal data delivery and automatic multihomed failover between our outstanding providers. Customers may rest assured that we only use the highest quality of bandwidth; our policy is to pay more for the best of breed providers and because we buy in bulk this doesn't impact our extremely competitive pricing.

At Below Zero we help you to achieve Zero Downtime.

www.zeroTdowntime.co.uk
We are going to explore even more widgets provided by tkinter. This time we will look at menus, combo boxes, spin boxes, separator bar, progress bars and notebooks. Let's talk about them one at a time.

You've seen menus in almost every application that you have ever used. Tkinter makes it VERY easy for us to make menus. Combo Boxes are similar to the list box that we explored in the last widget demo article, except the list "pops down" instead of being visible at all times. Spin box controls are great for giving a fixed range of values that can "scroll" up or down. For example, if we want the user to be able to choose from integers between 1 and 100, we can easily use a spin box. Progress bars are a wonderful way to show that your application hasn't locked up when something takes a lot of time, like reading records from a database. It can show the percentage of completion of a task. There are two types of progress bars, Determinate and Indeterminate. You use a determinate progress bar when you know just how many items you are dealing with. If you don't know the number of items or the percentage of how done your task is at any point, you would use the Indeterminate version. We will work with both. Finally a notebook widget (or tabbed widget) is used many times for things like configuration screens. You can logically group a series of widgets on each tab.

So, let's get started. As usual, we will create a base application and build on to it with each extra widget we add. Shown right is the first part of our application. You've seen most of this before.

Save all of this as widgetdemo2a.py. Remember we will use this as the base to build the full demo. Now we will start the process of creating the

class WidgetDemo2:

    def __init__(self, master = None):
        self.DefineVars()
        f = self.BuildWidgets(master)
        self.PlaceWidgets(f)

    def DefineVars(self):
        pass

And here is the bottom of our program. Again, you have seen this before. Nothing new here.

if __name__ == '__main__':
    def Center(window):
        # Get the width and height of the screen
        sw = window.winfo_screenwidth()
        sh = window.winfo_screenheight()
        # Get the width and height of the window
        rw = window.winfo_reqwidth()
        rh = window.winfo_reqheight()
        xc = (sw-rw)/2
        yc = (sh-rh)/2
        print "{0}x{1}".format(rw,rh)
        window.geometry("%dx%d+%d+%d"%(rw,rh,xc,yc))
        window.deiconify()

    root = Tk()
    root.title('More Widgets Demo')
    demo = WidgetDemo2(root)
    root.after(13,Center,root)
    root.mainloop()}
def BuildWidgets(self, master):
    frame = Frame(master)
    #===================
    # MENU STUFF
    #===================
    # Create the menu bar
    self.menubar = Menu(master)

# Create the File Pull Down, and add it to the menu bar
filemenu = Menu(self.menubar, tearoff = 0)
filemenu.add_command(label = "New", command = self.FileNew)
filemenu.add_command(label = "Open", command = self.FileOpen)
filemenu.add_command(label = "Save", command = self.FileSave)
filemenu.add_separator()
filemenu.add_command(label = "Exit", command = root.quit)
self.menubar.add_cascade(label = "File", menu = filemenu)

# Create the Edit Pull Down
editmenu = Menu(self.menubar, tearoff = 0)
editmenu.add_command(label = "Cut", command = self.EditCut)
editmenu.add_command(label = "Copy", command = self.EditCopy)
editmenu.add_command(label = "Paste", command = self.EditPaste)
self.menubar.add_cascade(label = "Edit", menu = editmenu)

# Create the Help Pull Down
helpmenu = Menu(self.menubar, tearoff=0)
helpmenu.add_command(label = "About", command = self.HelpAbout)
self.menubar.add_cascade(label = "Help", menu = helpmenu)

# Now, display the menu
master.config(menu = self.menubar)

#===================
# End of Menu Stuff
#===================

menu. Here are the steps we need to do. First, we define a variable to hold the menu instance. Like most any widget we use, the format is...

OurVariable = Widget(parent, options).

In this case, we are using the Menu widget and we will assign it to master as the parent. We do this under the BuildWidgets routine. Next we create another menu item, this time calling it filemenu. We add commands and separators as needed. Finally we add it to the menu bar and do it all over again until we are done. In our example, we'll have the menubar, a File pulldown, an Edit pulldown and a Help pulldown (top right). Let's get started.

Next (middle right) we concentrate on the File Menu. There will be five elements. New, Open, Save, a separator and Exit. We'll use the .add_command method to add the command. All we really need to do is call the method with the text (label = ) and then provide a callback function to handle when the user clicks the item. Finally we use the menubar.add_cascade function to attach the menu to the bar.

Notice that the Exit command uses “root.quit” to end the program. No call back needed for that. Next we'll do the same thing for the Edit and Help menus.

Notice the part in each of the menu group definitions that says “tearoff=0”. If you were to change the “=0” to “=1”, the menu would start with what looks like a dashed line and if you drag it, it “tears off” and creates its own window. While this might be helpful sometime in the future, we don't want that here.

Last but not least, we need to place the menu. We don't do a normal placement with the .grid() function. We simply add it by using the parent.config function (bottom right).

All of this has gone in the BuildWidgets routine. Now (next page, top right) we need...
HOWTO - PROGRAM IN PYTHON - PART 28

to add a generic frame and set the return statement before we move on to the PlaceWidgets routine.

Finally (next page, bottom right) we need to create all the callbacks we defined earlier. For the demo, all we'll do is print something in the terminal used to launch the program.

That's it. Save and run the program. Click on each of the menu options (saving File|Exit for last).

Now (below) we'll deal with the combo box. Save your file as widgetdemo2b.py and we'll get started. The imports, class definition and the def __init__ routines are all the same, as is the bottom part of the program. We'll add two lines to the DefineVars routine. Either comment out the "pass" statement or erase it and put in the following code. (I included the definition line just for clarity.)

First we define a label, which we've done before. Next we define the combo box. We use "ttk.Comboobox", define the parent and set the height to 19,

def DefineVars(self):
    self.cmbo1Val = StringVar()
    self.c1Vals = ['None','Option 1','Option 2','Option 3']

    # Combo Box
    self.lb1cb = Label(self.f1, text = "Combo Box: ")
    self.cmbo1 = ttk.Comboobox(self.f1,
                                 height = "19",
                                 width = 20,
                                 textvariable = self.cmbo1Val)
    self.cmbo1["values"] = self.c1Vals
    # Bind the virtual event to the callback
    self.cmbo1.bind("<<ComboboxSelected>>",self.cmbo1test)

    return frame

Next we (as we have done multiple times) deal with placing our other widgets.

def PlaceWidgets(self,master):
    # Get the master.
    frame = master
    frame.grid(column = 0, row = 0)

    self.f1.grid(column = 0,
                 row = 0,
                 sticky = 'nsew'
    )

    return frame

self.f1 = Frame(frame,
                relief = SUNKEN,
                borderwidth = 2,
                width = 500,
                height = 100
)

def FileNew(self):
    print "Menu - File New"

def FileOpen(self):
    print "Menu - File Open"

def FileSave(self):
    print "Menu - File Save"

def EditCut(self):
    print "Menu - Edit Cut"

def EditCopy(self):
    print "Menu - Edit Copy"

def EditPaste(self):
    print "Menu - Edit Paste"

def HelpAbout(self):
    print "Menu - Help About"
the width to 20 and the textvariable to “self.cmb01Val”. Remember that we set textvariables in the last widget demo, but just in case you forgot...this is changed anytime the value in the combo box is changed. We defined it in DefineVars as a StringVar object. Next we load the values that we want the user to choose from, again we defined that in DefineVars. Finally we bind the virtual event <<ComboboxSelected>> to the cmbotest routine that we will flesh out in a minute.

Next let's place the combo box and the label into our form (top right).

Save everything and test it out.

Now save as widgetdemo2c.py and we'll start with the separator bar. This is SO super easy. While the updated tkinter provides a separator bar widget, I've never been able to get it to work. Here's an easy work around. We use a frame with a height of 2. The only changes to our program will be the

definition of the frame in BuildWidgets after the combo box bind statement and placing the frame in the Place Widgets routine. So, in BuildWidgets put in the following lines (shown middle right)...

Once again, you've seen all this before. Save and test it. You'll probably have to expand the topmost window to see the separator, but it will become much more evident in the next demo. Save as widgetdemo2d.py and we'll add the spin control.

Under DefineVars, add the following line...

```python
self.spin1val = StringVar()
```

By now, you know that this is so we can get the value at any time we want. Next, we'll add some code to the BuildWidgets routine...just before the “return frame” line (bottom right).

Here we define a label and the spin control. The spin control definition is as follows:

```python
ourwidget = Spinbox(parent,low value, high value, width, textvariable, wrap)
```

And finally we put in the callback which simply prints what the user selected into the terminal window.

```python
def cmbotest(self,p1):
    print self.cmb01Val.get()
```

```python
self.fsep = Frame(self.f1, 
width = 140, 
height = 2, 
relief = RIDGE, 
borderwidth = 2 
)
```

And in PlaceWidgets put in this ...

```python
self.fsep.grid(column = 0, 
row = 3, 
columnspan = 8, 
sticky = 'we', 
padx = 3, 
pady = 3
)
```

```python
self.lblsc = Label(self.f1, text = "Spin Control:"
self.spin1 = Spinbox(self.f1, 
from_ = 1.0, 
to = 10.0, 
width = 3, 
textvariable = self.spin1val, 
wrap=True
)```
The low value must be called as "from_" since the word "from" is a keyword and using that would simply confuse everything. The values "from_" and "to" must be defined as float values. In this case we want it to have a low value of 1 and a high value of 10. Finally the wrap option says that if the value is (in our case) 10 and the user clicks on the up arrow, we want it to wrap around to the low value and keep going. The same works for the low value. If the user clicks the down arrow of the control and the value is 1, it wraps to 10 and keeps going. If you set "wrap=False", the control simply stops at whichever direction the user is going.

Now we'll place the widgets in PlaceWidgets (below).

Again, that's it. Save and play. You'll really notice the separator now.

Save as widgetdemo2e.py

```python
self.lblsc.grid(column = 0, row = 4)
self.spin1.grid(column = 1,
row = 4,
pady = 2)
```
we are looking at percentage. Mode in this case is 'indeterminate'. Remember, we use this mode when we don't know how far we've gotten in a task so we just want to let the user know that something is happening.

Next we add a button (you've done this before), another label another progress bar and another spin control. The mode for the second progress bar is "determinate".

We will use the spin control to set the “percentage” of completion. Next add the following lines (next page, top left) into the PlaceWidgets routine.

Lastly, we add two routines to control our progress bars (bottom right).

The TestPBar routine controls the indeterminate progress bar. Basically, we are starting and stopping an internal timer that is built into the progress bar.

```python
# Progress Bar
self.frmPBar.grid(column = 0,
    row = 5,
    columnspan = 8,
    sticky = 'nsew',
    padx = 3,
    pady = 3
)
sself.lb10.grid(column = 0, row = 0)
self.lb11.grid(column = 0,
    row = 1,
    pady = 3
)
sself.pbar.grid(column = 1, row = 1)
sself.btnptest.grid(column = 3, row = 1)
sself.lb12.grid(column = 0,
    row = 2,
    pady = 3
)
sself.pbar2.grid(column = 1, row = 2)
sself.spin2.grid(column = 3, row = 2)
```

def TestPBar(self):
    if self.btnStatus == False:
        self.btnptest.config(text="Stop")
        self.btnStatus = True
        self.pbar.start(10)
    else:
        self.btnptest.config(text="Start")
        self.btnStatus = False
        self.pbar.stop()

def Spin2Do(self):
    v = self.spinval2.get()
    print v
    self.pbar2val.set(v)
The line “self.pbar.start(10)” sets the timer to 10 milliseconds. This makes the bar move fairly quickly. Feel free to play with this value up and down on your own. The Spin2Do routine simply sets the progress bar to whatever value the spin control has. We print it as well to the terminal.

That’s all the changes for this. Save and play.

Now save as widgetdemo2f.py and we’ll deal with the tabbed notebook widgets. In BuildWidgets put the following code (below)

```python
self.nframe.grid(column = 0,
                 row = 6,
                 columnspan = 8,
                 rowspan = 7,
                 sticky = 'nsew' )
self.notebook.grid(column = 0,
                  row = 0,
                  columnspan = 11,
                  sticky = 'nsew' )
self.lsp1.grid(column = 0,row = 0)
self.lsp2 = Label(self.p2,
               text = 'This is a label on PAGE 2',
               padx = 3,
               pady = 3
        ).grid(
               column = 0,
               row = 1
        )
```

...before the “return frame” line...

Let’s look at what we did. First, we define a frame for our notebook widget. Now we define the widget. All the options are ones we’ve seen before. Next we define two frames named self.p1 and self.p2. These act as our pages. The next two lines (self.notebook.add) attach the frames to the notebook widget and they get a tag attached to them. We also set the text for the tabs. Finally, we put a label on page number one. We’ll put one on page number two when we place the controls just for fun.

In the PlaceWidgets routine put the following code (below).

The only thing that might possibly be strange is the label on page two. We combine the definition and placement in the grid with the same command. We did that when we did our first widget demo app.

That’s it. Save and play.

As always the full code for the full application is up on pastebin at http://pastebin.com/qSPkSNU1.

Enjoy. Next time we’ll deal with some more database stuff.
A little while ago, I was asked to convert a MySQL database to SQLite. Looking around the web for a quick and easy (and free) solution, I found nothing that worked with the current version of MySQL for me. So I decided to go ahead and “roll my own”.

The MySQL Administrator program allows you to backup a database into a flat text file. Many SQLite browsers allow you to read a flat sql definition file and create the database from there. However, there are many things that MySQL supports that SQLite doesn’t. So this month, we’ll write a conversion program that reads a MySQL dump file and creates a SQLite version.

Let’s start by looking at the MySQL dump file. It consists of a section that creates the database, and then sections that create each table within the database followed by the data for that table, if it’s included in the dump file. (There’s an option to export the table schema(s) only). Shown above right is an example of one of the create table sections.

The first thing that we would need to get rid of is in the last line. Everything after the ending parenthesis needs to go away. (SQLite does not support an InnoDB database). In addition to that, SQLite doesn’t support the “PRIMARY KEY” line. In SQLite, we set a primary key by using “INTEGER PRIMARY KEY AUTOINCREMENT” when we define the field. The other thing that SQLite doesn’t support is the “unsigned” keyword.

When it comes to the data, the “INSERT INTO” statements are also non-compatible. The problem here is that SQLite doesn’t allow multiple inserts within the same statement. Here’s a short example from the MySQL dump file. Notice (right) that the end-of-line

```
DROP TABLE IF EXISTS `categoriesmain`;
CREATE TABLE `categoriesmain` (
  `idCategoriesMain` int(10) unsigned NOT NULL auto_increment,
  `CatText` char(100) NOT NULL default '',
  PRIMARY KEY (`idCategoriesMain`)
) ENGINE=InnoDB AUTO_INCREMENT=40 DEFAULT CHARSET=latin1;
```

```
INSERT INTO `categoriesmain`(`idCategoriesMain`,`CatText`) VALUES
(1,'Appetizer'),
(2,'Snack'),
(3,'Barbecue'),
(4,'Cake'),
(5,'Candy'),
(6,'Beverages');
```

To make this compatible, we need to change this from a single statement format to a series of single statements like this:

```
INSERT INTO `categoriesmain`(`idCategoriesMain`,`CatText`) VALUES (1,'Appetizer');
INSERT INTO `categoriesmain`(`idCategoriesMain`,`CatText`) VALUES (2,'Snack');
INSERT INTO `categoriesmain`(`idCategoriesMain`,`CatText`) VALUES (3,'Barbecue');
INSERT INTO `categoriesmain`(`idCategoriesMain`,`CatText`) VALUES (4,'Cake');
INSERT INTO `categoriesmain`(`idCategoriesMain`,`CatText`) VALUES (5,'Candy');
INSERT INTO `categoriesmain`(`idCategoriesMain`,`CatText`) VALUES (6,'Beverages');
```
marker is a semicolon.

We will also ignore any comment lines, and the CREATE DATABASE and USE statements. Once we have the converted SQL file, we'll use a program similar to the public domain program SQLite Database Browser to actually deal with the process of creating the database, tables, and data.

Let's get started. Start a new project folder and a new python file. Name it MySQL2SQLite.py.

Shown above right is the import statement, the class definition, and the __init__ routine.

This will be a commandline driven program, so we'll need to create the “if __name__” statement, a command line argument handler, and a usage routine (if the user doesn't know how to use the program). This goes at the very end of the program. All other code we create will go above this:

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

Below is the handler that does the printing of the usage statement.

The DoIt() routine is called if our program is being run standalone from the command line, which is the design. However, if we want to keep this as a library to be included in another program at another time, we
```

```python
def DoIt():
    # Setup Variables
    SourceFile = ''
    OutputFile = ''
    Debug = False
    Help = False
    SchemaOnly = False

    #
!
```
can just use the class. Here we set up a number of variables to make sure that everything works correctly. The code shown bottom right then parses the command line arguments passed to our program, and gets things ready for the main routines.

When we start the program, we need to provide at least two variables on the command line. These are the Input file, and the Output file. We also will provide support for the user to see what is happening as the program is running, an option to just create the tables and not stuff the data, and for the user to call for help. Our “normal” command line to start the program looks like this:

MySQL2SQLite Infile=Foo 
Outfile=Bar

where “Foo” is the name of the MySQL dump file, and “Bar” is the name of the SQLite sql file we want the program to create.

You can also call it like this:

MySQL2SQLite Infile=Foo 
Outfile=Bar Debug SchemaOnly

Which will add the option to show the debug messages and to ONLY create the tables and not import the data.

Finally if the user asks for help, we just go to the usage portion of the program.

Before we continue, let’s take another look at how the command line argument support works.

When a user enters the program name from the command line (terminal), the operating system keeps track of the information entered and passes it to the program just in case there are any options entered. If no options (also called arguments) are entered, the number of arguments is one, which is the name of the application - in our case MySQL2SQLite.py. We can access these arguments by calling the sys.argv command. If the count is greater than one, we will access them in a for loop. We will step through the list of arguments and check each one. Some programs require you to enter the arguments in a specific order. By using the for loop approach, the arguments can be entered in any order. If the user doesn't supply any arguments, or uses the help arguments, we show the usage screen. Shown above is the routine for that.

Moving on, once we have...
parsed the argument set, we instantiate the class, call the setup routine, which fills certain variables and then call the DoWork routine. We'll start our class now (which is shown on the next page, bottom right).

This (next page, top right) is the definition and the __init__ routine. Here we setup the variables that we will need as we go through the code. Remember that right before we call the DoWork routine, we call the Setup routine. We take our empty variables and assign the correct values to them here.

Notice that there is the ability to not write to a file, useful for debugging purposes. We also have the ability to simply write the schema, or database structure, without writing the data. This is helpful if you are taking a database and starting a new project without wanting to use any existing data.

We start off by opening the SQL Dump file, then setting some internal scope variables.

```python
while 1:
    line = f.readline()
    cntr += 1
    if not line:
        break
    # Ignore blank lines, lines that start with "--" or comments (/*)
    if line.startswith("--") or comments (/\*)
        pass
    elif len(line) == 1:
        # Blank Lines
        pass
    elif line.startswith("/\*!"): # Comments
        pass
    elif line.startswith("USE"):
        #Ignore USE lines
        pass
    elif line.startswith("CREATE DATABASE "):
        pass
```

Now, we'll deal with the DoWork routine, which is where the actual "magic" happens.

```python
def DoWork(self):
    f = open(self.InputFile)
    print "Starting Process"
    cntr = 0
    insertmode = 0
    CreateTableMode = 0
    InsertStart = "INSERT INTO "
    AI = "auto_increment"
    PK = "PRIMARY KEY"
    IPK = " INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL"
    CT = "CREATE TABLE"
    # Begin
    if self.WriteFile == 1:
        OutFile = open(self.OutputFile,'w')
```

```python
#===============================================
# BEGIN CLASS MySQL2SQLite
#===============================================
class MySQL2SQLite:
    def __init__(self):
        self.InputFile = ""
        self.OutputFile = ""
        self.WriteFile = 0
        self.DebugMode = 0
        self.SchemaOnly = 0
```

We also define some strings to save us typing later on. Then, if we are to write to an output file, we open it and then we start the entire process. We will read each line of the input file, process it, and potentially write it to the output file. We use a forced while loop to assist reading each line, with a break command when there is nothing left in the input file. We use f.readline() to get the line to work, and assign it to the variable “line”. Some lines, we can safely ignore. We'll simply use an if/elif statement followed by a pass statement to accomplish this (below).

Next we can stop ignoring things and actually do something. If we have a CreateTable statement, we'll start that process. Remember we defined CT to be equal to “Create Table”. Here (above right), we set a variable “CreateTableMode” to be equal to 1, so we know that's what we are doing, since each field definition is on a separate line. We then take our line, remove the carriage return, and get that ready to write to our out file, and, if required, write it.

Now (middle right) we need to start dealing with each line within the create table statements - manipulating each line to keep SQLite happy. There are many things that SQLite won't deal with. Let's look at a Create Table statement from MySQL again.

One thing that SQLite will absolutely have an issue with is the entire last line after the closing parenthesis. Another is the line just above that, the Primary Key line. Yet another thing is the unsigned keyword in the second line. It will take a bit of code (below) to work around these issues, but we can make it happen.

```python
elif pl != -1:
    # Line is primary key line
    l = line.strip()
    fnpos = l.find(" int")
    if fnpos != -1:
        fn = l[:fnpos]
        newline = fn + IPK="#",
    if self.writeFile == 1:
        OutFile.write(newline)
    if self.DebugMode == 1:
        print "Writing Line {0}".format(newline)
```

```sql
CREATE TABLE `categoriesmain` (  
  `idCategoriesMain` int(10) unsigned NOT NULL auto_increment,  
  `CatText` char(100) NOT NULL default ",",  
  PRIMARY KEY (`idCategoriesMain`)  
) ENGINE=InnoDB AUTO_INCREMENT=40 DEFAULT CHARSET=latin1;
```

```python
elif line.startswith("CT"):  
    CreateTableMode = 1  
    l1 = len(line)  
    line = line[:l1-1]  
    if self.DebugMode == 1:
        print "Starting Create Table"  
        print line  
    if self.writeFile == 1:
        OutFile.write(line)
```
this will be the primary key line. While this might be true 98.6% of the time, it won’t always be.
However, we'll keep it simple. Next we check to see if the line starts with “)”. This will signify
this is the last line of the create table section. If so, we simply set a string to close the
statement properly in the variable “newline”, turn off the CreateTableMode variable, and, if we are writing to file, write it out.

Now (bottom right) we use the information we found about the auto increment key word.
First, we strip the line of any spurious spaces, then check to see where (we are assuming it is there) the phrase “ int(“ is within the line. We will be replacing this with the phrase “ INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL”.
The length of the integer doesn’t matter to SQLite.

Again, we write it out if we should.

Now we look for the phrase “PRIMARY KEY “ within the line. Notice the extra space at the end - that’s on purpose. If it arises, we ignore the line.

```python
elif line.strip().startswith(PK):
    pass
```

Now (top right) we look for the phrase “ unsigned “ (again keep the extra spaces) and replace it with “ “.

That's the end of the create table routine. Now (below) we move on to the insert statements for the data. The InsertStart variable is the phrase “INSERT INTO “. We check for that because MySQL allows for multiple insert statements in a single

```python
elif line.startswith(InsertStart):
    if insertmode == 0:
        insertmode = 1
        # Get tablename and field list here
        istatement = line
        # Strip CR/LF from istatement line
        l = len(istatement)
        istatement = istatement[:l-2]
```

```python
elif line.find(" unsigned ") != -1:
    line = line.replace(" unsigned ","")
    line = line.strip()
    ll = len(line)
    line = line[:ll-1]
    if self.WriteFile == 1:
        OutFile.write(""," + line)
    if self.DebugMode == 1:
        print "Writing Line {0}".format(line)
```

Otherwise, we can deal with the line.

```python
else:
    ll = len(line)
    line = line.strip()
    line = line[:ll-4]
    if self.DebugMode == 1:
        print "," + line
    if self.WriteFile == 1:
        OutFile.write(""," + line)
```

```python
if posx != -1:
    ll = line[:posx+3]
    insertmode = 0
    if self.DebugMode == 1:
        print istatement + ll
        print "---------------------------------------"
    if self.WriteFile == 1:
        OutFile.write(istatement + ll+"\n")
```

Otherwise, we join the prelude to the value statement and end it with a semicolon.

```python
e elif pos1 != -1:
    ll = line[:pos1+2]
    if self.DebugMode == 1:
        print istatement + ll + ";"
    if self.WriteFile == 1:
        OutFile.write(istatement + ll + ";\n")
```
The line “self.pbar.start(10)” sets the timer to 10 milliseconds. This makes the bar move fairly quickly. Feel free to play with this value up and down on your own. The Spin2Do routine simply sets the progress bar to whatever value the spin control has. We print it as well to the terminal.

That's all the changes for this. Save and play.

Now save as widgetdemo2f.py and we'll deal with the tabbed notebook widgets. In BuildWidgets put the following code (below) before the “return frame” line...

Let's look at what we did. First, we define a frame for our notebook widget. Now we define the widget. All the options are ones we've seen before. Next we define two frames named self.p1 and self.p2. These act as our pages. The next two lines (self.notebook.add) attach the frames to the notebook widget and they get a tab attached to them. We also set the text for the tabs. Finally, we put a label on page number one. We'll put one on page number two when we place the controls just for fun.

In the PlaceWidgets routine put the following code (below).

The only thing that might possibly be strange is the label on page two. We combine the definition and placement in the grid with the same command. We did that when we did our first widget demo app.

That's it. Save and play.

As always the full code for the full application is up on pastebin at http://pastebin.com/gSPkSNU1.

Enjoy. Next time we'll deal with some more database stuff.

```python
else:
    if self.DebugMode == 1:
        print "Testing line {0}".format(line)
        pos1 = line.find("",")
        posx = line.find("",")
        if self.DebugMode == 1:
            print "pos1 = {0}, posx = {1}".format(pos1,posx)
        if pos1 != -1:
            ll = line[:pos1+1]
            if self.DebugMode == 1:
                print istatement + ll + ";"
            if self.WriteFile == 1:
                OutFile.write(istatement + ll + ";\n")
    else:
        insertmode = 0
        ll = line[:posx+1]
        if self.DebugMode == 1:
            print istatement + ll + ";"
        if self.WriteFile == 1:
            OutFile.write(istatement + ll + ";\n")
```
This month, we'll explore yet another GUI designer, this time for Tkinter. Many people have an issue with Tkinter because it doesn't offer a built-in designer. While I've shown you how to easily design your applications without a designer, we will examine one now. It's called Page. Basically it's a version of Visual TCL with Python support on top. The current version is 3.2 and can be found at http://sourceforge.net/projects/page/files/latest/download.

Prerequisites

You need TCK/TK 8.5.4 or later, Python 2.6 or later, and pytk - which you can get (if you don't already have it) from http://pypi.python.org/pypi/pyttk. You probably have all of these with the possible exception of pyttk.

Installation

You can't really ask for an easier installation routine. Simply unpack the distribution file into a folder of your choice. Run the script called "configure" from the folder where you just unpacked everything. This will create your launch script called "page" which you use to get everything going. That's it.

Learning Page

When you start Page, you'll get three windows (forms). One is a "launch pad", one is a toolbox, and one shows the Attribute Editor.

To start a new project, click on the Toplevel button in the toolbox.

This creates your main form. You can move it wherever you wish on your screen. Next, and from now on, click on a widget in the tool box and then click where you want it on the main form.

For now, let's do a button. Click on the Button button on the toolbox, and then click somewhere on the main form.

Next, in the launch pad form, click on Window and select Attribute Editor (if it's not already showing). Your single button should be highlighted already, so move it around the form and when you release the mouse button you should see the position change in the attribute editor form under 'x position' and 'y position'.

Here we can set other attributes such as the text on the button (or most any other widget), the alias for the widget (the name we will refer to in our code), color, the name we will call it and more. Near the bottom of the attribute editor is the text field. This is the text that appears to the user for, in this case, the button widget.
Let's change this from “button” to “Exit”. Notice that now the button says “Exit”. Now resize the form to just show the button and center the button in the form.

Next click in the main form someplace where the button isn't. The attribute editor form now shows the attributes for the main form. Find the “title” field and change this from “New Toplevel 1” to “Test Form”.

Now, before we save our project, we need to create a folder to hold our project files. Create a folder somewhere on your drive called “PageProjects”. Now, in the launch pad window, select File then Save As. Navigate to your PageProjects folder, and, in the dialog box, type TestForm.tcl and click the Save button. Notice this is saved as a TCL file, not a Python file. We'll create the python file next.

In the launch pad, find the Gen_Python menu item and click it. Select Generate Python and a new form appears.

Page has generated (as the name suggests) our python code for us and placed it in a window for us to view. At the bottom of this form, are three buttons...Save, Run, and Close.

PageProjects folder, you will see the python file (TestForm.py). Now click on the Run button. In a few seconds, you'll see the project start up. The button is not connected to anything yet, so it won't do anything if you click on it. Simply close the form with the “X” in the corner of the window. Now close the Python Console window with the close button at the bottom right.

Back at our main form, highlight the Exit button and right click on it. Select “Bindings...”. Under the menu is a set of buttons.

Save and generate the python code again. Scroll down in the Python Console to the bottom of the file. Above the “class Test_Form” code is the function we just asked to be created. Notice that at this point, it simply is passed. Look further down and you'll see the code that creates and controls our button. Everything is done for us already. However, we still have to tell the button what to do. Close the Python Console and we'll continue.

On the launch pad, click
Click on the checkmark and we are done with this.

Next we have to bind this routine to the button. Select the button in the form, right click it, and select “Bindings...”. As before, click on the far left button on the toolbar and select Button-1. This is the event for the left mouse button click. In the right text box, enter “Button1Click”. Make sure you use the same case that you did for the Function we just created. Click the checkmark on the right side.

Now save and generate your python code.

You should see the following code near the bottom, but OUTSIDE of the Test_Form class...

```python
def Button1Click(p1):
    sys.exit()
```

And the last line of the class should be...

```python
self.Button1.bind('<Button-1>', Button1Click)
```

Now, if you run your code and click on the Exit button, the form should close properly.

**Moving Forward**

Now let’s do something more complicated. We’ll create a demo showing some of the widgets that are available. First close Page and restart it. Next, create a new Toplevel form. Add two frames, one above the other and expand them to pretty much take up the entire width of the form. In the top frame, place a label, and, using the attributes editor, change the text to “Buttons:”. Next, add two buttons along the horizontal plane. Change the text of the left one to “Normal”, and the right one to “Sunken”. While the sunken button is selected, change the relief to “sunken” and name it btnSunken. Name the “Normal” button “btnNormal”. Save this project as “Demos.tcl”.

Next, place in the lower frame a label saying “Radio Buttons” and four radio buttons like in the image below. Finally, place an Exit button below the bottom frame.

Before we work on the bindings, let’s create our click functions. Open the Function List and create two functions. The first should be called btnNormalClicked and the other btnSunkenClicked. Make sure you set the arguments box to include p1. Here’s the code you should have for them...

```python
def btnNormalClicked(p1):
    print "Normal Button Clicked"
def btnSunkenClicked(p1):
    print "Sunken Button Clicked"
```

Let’s add our button bindings. For each button, right click it, select “Bindings...”, and add, as before, a binding to the functions we created. For the normal button, it would be “btnNormalClicked”, and for the sunken button it would be
btnSunkenClicked. Save and
generate your code. Now, if you
were to test the program under
the “Run” option of the Python
Console, and click any of the
buttons, you won't see
anything happen. However,
when you close the application,
you should see the print
responses. This is normal for
Page and if you simply run it
from the command line as you
normally do, things should work
as expected.

Now for our radio buttons.
We have grouped them in two
“clusters”. The first two (Radio
1 and Radio 2) will be cluster 1
and the other two will be
cluster 2. Click on Radio1 and
in the Attribute Editor, set the
value to 0 and the variable to
“rbc1”. Set the variable for
Radio 2 to “rbc1” and the value
to 1. Do the same thing for
Radio 3 and Radio 4 but for
both of these set the variable
to “rbc2”. If you want, you can
deal with the click of the
radiobuttons and print
something to the terminal, but
for now, the important thing is
that the clusters work. Clicking
Radio1 will deselect Radio2 and
not influence Radio3 or Radio4,
and the same for Radio2 and so
on.

Finally, you should create a
function for the Exit button, and
bind it to the button like we did
in the first example.

If you've been following
along as we have done our
other Tkinter applications, you
should be able to understand
the code shown above right. If
not, please go back a few
issues for a full discussion of
this code.

You can see that using Page
makes the basic design process
much easier than doing it
yourself. We've only scratched
the surface of what Page can
do, and we'll start doing
something much more realistic
next time.

The python code can be
found on pastebin at
http://pastebin.com/qqOYVqTb.

One note before we go for
this month. You might have
noticed that I've missed a
couple of issues. This is due to
my wife being diagnosed with
cancer last year. As hard as I
have tried to keep things from
falling through the cracks, a
number of things have. One of
these things is my old
domain/web site at
blew it and missed the renewal.
Due to this, the domain was
sold out from under me. I have
set up
www.thedesigantedgeek.net
with all the old stuff. I will be
working hard the next month to
bring it all up to date.

See you next time.
After our last meeting you should have a fairly good idea of how to use Page. If not, please read last month's article. We'll continue this time by creating a file list application with a GUI. The goal here is to create a GUI application that will recursively walk through a directory, looking for files with a defined set of extensions, and display the output in a treeview. For this example we will look for media files with the extensions of “.avi”, “.mkv”, “.mov”, “.mp3” and “.ogg”.

This time, the text might seem a bit terse in the design portion. All I'm going to do is give you directions for placement of widgets and the required attributes and values like this...

**Widget**

**Attribute:** Value

I will only quote text string when it is needed. For example for one of the buttons, the text should be set to “...”.

Here's what the GUI of our application will look like...

![Image of GUI](image)

As you can see, we have our main form, an exit button, a text entry box with a button that will call up an ask for directory dialog box, 5 check boxes for extension selecting extension types, a “GO!” button to actually start the processing and a treeview to display our output.

So, let's get started. Fire up Page and create a new top level widget. Using the Attribute Editor set the following attributes.

**Alias:** Searcher  
**Title:** Searcher

Be sure to save often. When you save the file, save it as “Searcher”. Remember, Page puts the .tcl extension for you and when you finally generate the python code, it will be saved in the same folder.

Next add a frame. It should go at the very top of the main frame. Set the attributes as follows.

**Width:** 595  
**Height:** 55  
**x position:** 0  
**y position:** 0

In this frame, add a button. This will be our Exit button.

**Alias:** btnExit  
**Text:** Exit

Move this close to the center of the frame or close to the frame's right side. I set mine to X 530 and Y 10.

Create another frame.

**Width:** 325  
**Height:** 185  
**y position:** 60

Here is what this frame will look like, to give you a guide going forward through this section.

In this frame, add a label. Set the text attribute to “Path:”. Move it close to the top left of the frame.

In the same frame, add an entry widget.

**Alias:** txtPath  
**Text:** FilePath  
**Width:** 266  
**Height:** 21

Add a button to the right of the entry widget.

**Alias:** btnSearchPath  
**Text:** “...” (no quotes)

Add five (5) check buttons. Put them in the following order...
The three check buttons on the left are for video files and the two on the right are for audio files. We will deal with the three on the left first, then the two on the right.

**Alias:** chkAVI  
**Text:** “.avi”  
**Variable:** VchkAVI

**Alias:** chkMKV  
**Text:** “.mkv”  
**Variable:** VchkMKV

**Alias:** chkMV4  
**Text:** “.mv4”  
**Variable:** VchkMV4

**Alias:** chkMP3  
**Text:** “.mp3”  
**Variable:** VchkMP3

**Alias:** chkOGG  
**Text:** “.ogg”  
**Variable:** VchkOGG

Finally, in this frame add a button somewhere below the five check boxes and somewhat centered within the frame.

**Alias:** btnGo  
**Text:** Go!

Now add one more frame below our last frame.

Next create another function called btnGoClick. Remember to add a passed parameter of “p1”. Leave the “pass” statement. We’ll change that later.

Finally, add another function called “btnSearchPath”. Again, leave the pass statement.

Lastly, we need to bind the buttons to the functions we just created.

Right-click on the exit button we created, select Bind. A large box will pop up. Click on the New binding button, Click on Button-1 and change the word “TODO” in the right text entry box to “btnExitClick”. Do NOT include the parens () here.

Bind the GO button to btnGoClick and the “…” button to btnSearchPathClick.

Save your GUI and generate the python code.

Now all we have left is to create the code that “glues” the GUI together.

Open up the code we just generated in your favorite editor. Let’s start off by examining what Page created for us.

At the top of the file is our standard python header and a single import statement to import the sys library. Next is some rather confusing (at first glance) code. This basically looks at the version of python you are trying to run the application in and then to import the correct versions of the tkinter libraries. Unless you are using python 3.x, you can basically ignore the last two.

We’ll be modifying the 2.x code portion to import other tkinter modules in a few moments.

Next is the “vp_start_gui()” routine. This is the program’s main routine. This sets up our gui, sets the variables we need, and then calls the tkinter main loop. You might notice the line “w = None” below this. It is not indented and it isn't supposed to be.

Next are two routines (create_Searcher and destroy_Searcher) that are used to replace the main loop routine if we are calling this
application as a library. We don't need to worry about these.

Next is the “set_Tk_var” routine. We define the tkinter variables used that need to be set up before we create the widgets. You might recognize these as the text variable for the FilePath entry widget and the variables for our check boxes. The next three routines here are the functions we created using the function editor and an “init()” function.

Run the program now. Notice that the check buttons have grayed out checks in them. We don't want that in our “release” app, so we'll create some code to clear them before the form is displayed to the user. The only functioning thing other than the check boxes is the Exit button.

Go ahead and end the program.

Now, we'll take a look at the class that actually holds the GUI definition. That would be “class Searcher”. Here is where all the widgets are defined and placed in our form. You should be familiar with this by now.

Two more classes are created for us that hold the code to support the scrolled tree view. We don't have to change any of this. It was all created by Page for us.

Now let's go back to the top of the code and start modifying.

We need to import a few more library modules, so under the “import sys” statement, add...

```
import os
from os.path import join, getsize, exists
```

Now find the section that has the line “py2 = True”. As we said before, this is the section that deals with the tkinter imports for Python version 2.x. Below the “import ttk”, we need to add the following to support the FileDialog library. We also need to import the tkFont module.

```
import tkFileDialog
import tkFont
```

Next we need to add some variables to the “set_Tk_var()” routine. At the bottom of the routine, add the following lines...

```
global exts, FileList
exts = []
FileList=[]
```

Here we create two global variables (exts and FileList) that will be accessed later on in our code. Both are lists. “exts” is a list of the extensions that the user selects from the GUI. “FileList” holds a list of lists of the matching files found when we do our search. We'll use that to populate the treeview widget.

Since our “btnExitClick” is already done for us by Page, we'll deal with the “btnGoClick” routine. Comment out the pass statement and add the code so it looks like this...

```
def btnGoClick(pl) :
    #pass
    BuildExts()
    fp = FilePath.get()
    el = tuple(exts)
    Walkit(fp,el)
```

Finally we call a routine called “LoadDataGrid”.

Next we need to flesh out the “btnSearchPathClick” routine. Comment out the pass statement and change the code to look like this...

```
def btnSearchPathClick(pl) :
    #pass
    path = tkFileDialog.askdirectory() #**self.file_opt)
    FilePath.set(path)
```
The init routine is next. Again, make the code look like this...

def init():
    #pass

    # Fires AFTER Widgets and Window are created...

global treeview

BlankChecks()

treeview = w.Scrolledtreeview1

SetupTreeview()  

Here we create a global called “treeview”. We then call a routine that will clear the gray checks from the check boxes, assign the “treeview” variable to point to the Scrolled treeview in our form and call “SetupTreeview” to set the headers for the columns. Here's the code for the BlankChecks routine which needs to be next.

def BlankChecks():
    VchkAVI.set('0')
    VchkMKV.set('0')
    VchkOGG.set('0')

    Here, all we are doing is setting the variables (which automatically sets the check state in our check boxes) to “0”. If you remember, whenever the check box is clicked, this variable is automatically updated. If the variable is changed by our code, the check box responds as well. Now (above right) we'll deal with the routine that builds the list of extensions from what the user has clicked.

    Cast your memory back to my ninth article in FCM#35. We wrote some code to create a catalog of MP3 files. We'll use a shortened version of that routine (middle right). Refer back to FCM#35 if you have questions about this routine.

    Next (bottom right) we call the SetupTreeview routine. It's fairly straightforward. We define a variable “ColHeads” with the headings we want in each column of the treeview. We do this as a list. We then set the heading attribute for each column. We also set the column width to the size of this header.

    Finally we have to create the “LoadDataGrid” routine (next page, top right) which is where we load our data into the treeview. Each row of the treeview is one entry in the FileList list variable. We also adjust the width of each column (again) to match the size of the column data.

    That's it for the first blush of the application. Give it a run

    def BuildExts():
        if VchkAVI.get() == '1':
            exts.append(".avi")
        if VchkMKV.get() == '1':
            exts.append(".mkv")
        if VchkMP3.get() == '1':
            exts.append(".mp3")
        if VchkMV4.get() == '1':
            exts.append(".mv4")
        if VchkOGG.get() == '1':
            exts.append(".ogg")

    def Walkit(musicpath,extensions):
        rcntr = 0
        fl = []
        for root, dirs, files in os.walk(musicpath):
            rcntr += 1 # This is the number of folders we have walked for file in [f for f in files if f.endswith(extensions)]:
                fl.append(file)
                fl.append(root)
                FileList.append(fl)
        fl=[]

    def SetupTreeview():
        global ColHeads
        ColHeads = ['Filename', 'Path']
        treeview.configure(columns=ColHeads, show="headings")
        for col in ColHeads:
            treeview.heading(col, text = col.title(),
                command = lambda c = col: sortby(treeview, c, 0))
        # adjust the column’s width to the header string
        treeview.column(col, width = tkFont.Font().measure(col.title()))
and see how we did. Notice that if you have a large number of files to go through, the program looks like it's not responding. This is something that needs to be fixed. We'll create routines to change our cursor from the default to a "watch" style cursor and back so when we do something that takes a long time, the user will notice.

In the "set_Tk_var" routine, add the following code at the bottom.

global busyCursor, preBusyCursors, busyWidgets

busyCursor = 'watch'
preBusyCursors = None
busyWidgets = (root, )

What we do here is set up global variables, assign them and then set the widget(s) (in busyWidgets) we wish to respond to the cursor change. In this case we set it to root which is our full window. Notice that this is a tuple.

Next we create two routines to set and unset the cursor. First the set routine, which we will call "busyStart". After our "LoadDataGrid" routine, insert the code shown middle right.

    def busyStart(newcursor=None):
        global preBusyCursors
        if not newcursor:
            newcursor = busyCursor
            preBusyCursors = {}
        for component in busyWidgets:
            newPreBusyCursors[component] = component['cursor']
            component.configure(cursor=newcursor)
            component.update_idletasks()
        preBusyCursors = (newPreBusyCursors, preBusyCursors)

    def busyEnd():
        global preBusyCursors
        if not preBusyCursors:
            return
        oldPreBusyCursors = preBusyCursors[0]
        preBusyCursors = preBusyCursors[1]
        for component in busyWidgets:
            try:
                component.configure(cursor=oldPreBusyCursors[component])
            except KeyError:
                pass
            component.update_idletasks()

development. From today's article, you can see how having a good design of your GUI ahead of time can make the development process easy and fairly painless.

    The tcl file is saved in

    pastebin at http://pastebin.com/AA1kE4Dy
and the python code is saved at http://pastebin.com/VZm5un3e.

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

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

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

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

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

Many web pages show how to load SL4A into the Android Emulator for Desksots. We'll look at doing that another time,
but for now we'll deal with the Android device itself. To install SL4A on your Android device, go to http://code.google.com/p/android-scripting/; you'll find the installation file for SL4A. Don't be absolutely confused here. There's a square High Density barcode that you tap to download the APK. Be sure that you have the “Unknown Sources” option enabled in the Application settings. It's a quick download. Once you have it downloaded and installed, go ahead and find the icon, and tap it. What you will see is a rather disappointing black screen saying “Scripts...No matches found”. That's OK. Hit the menu button and select View. You'll see a menu. Select Interpreters. Then select menu again, and select Add. From the next menu, select Python 2.6.2. This should ask you to start a browser session to download Python for Android. Once this is installed, select Open. You'll get a screen menu with the options to Install, Import Modules, Browse Modules, and Uninstall modules. Select Install. Now Python will download and install along with other extra modules. In addition, you'll get some sample scripts. Finally, tap the back button and you'll see Python 2.6.2 installed in the interpreters screen. Tap again on the back button and you'll see a list of some sample python scripts.

That's all we are going to do this time. All I wanted to do is whet your appetite. Explore Python on Android. You might also want to visit http://developer.android.com/sdk/index.html to get the Android SDK (Software Development Kit) for your desktop. It includes an Android Emulator so you can play along. Setting up the SDK is really pretty easy on Linux, so you shouldn't have too much trouble.

How to Include Accents from the Keyboard
by Barry Smith

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

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

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

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

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

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

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

To get ç and ï (inverted exclamation mark) which I use all the time in Spanish before questions, and exclamations, press Alt Gr + Shift, keeping both keys pressed, then hit _ (underscore) for ç or hit ! (exclamation mark) for ï.
This time, we’ll set up the Android SDK on our Linux desktop. We’ll also create a virtual Android device, install SL4A and python on it, and do a quick test.

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

Here’s a quick list of what we’ll do. We’ll go step-by-step in a minute.

- Install the Java JDK6.
- Install the Android SDK starter pack.
- Create and setup AVDs.
- Test AVD, and install SL4A and Python.

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

**STEP 1 – Java JDK 6**

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

```
sudo add-apt-repository ppa:ferramroberto/java
sudo apt-get update
sudo apt-get install sun-java6-jdk
```

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

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

Save the file and move on to step 2.

**STEP 2 – Android SDK Starter Pack**

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

**STEP 3 – Create and set up your first AVD**

Back in the Android SDK Manager, select Tools from the...
main menu, then select Manage AVDs. This will open a new window. Since this is the first time, there won’t be any virtual devices set up. Click on the “New” button. This opens yet another window where we define the properties of the virtual Android device. Here’s the steps that you should use to set up a simple Android emulator device:

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

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

### STEP 4 - Testing the AVD and installing SL4A and Python

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

Once the AVD starts up and you have the “home” screen up, you will install SL4A. Using the browser or the google web search box on the home screen, search for “sl4a“ Go to the downloads page, and you’ll eventually find the web page for the downloads at

http://code.google.com/p/androi
d-scripting/downloads/list.

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

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

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

- Back - Escape
- Home - Home
- Menu - F2

Now we will want to download and install python into SL4A. To do this, first tap Menu (press F2). Select “View” from the menu. Now select “Interpreters”. It looks like nothing happened, but tap Menu again (F2), then select “Add” from the popup. Now scroll down and select “Python 2.6.2”. This will download the base package for Python for Android. Install the package, then open it. You will be presented with four options. Install, Import Modules, Browse Modules, and Uninstall Module. Tap on Install. This will start
Now let’s create a shortcut on the Android home screen. Tap the Home key (Home button). If you chose the 2.1 platform, you should see a slider bar on the far right of the screen. If you chose another platform, it might be a square or rectangle consisting of small squares. Either way, this gets you to the Apps screen. Tap that, and find the SL4A icon. Now perform a “long tap” (long click), which will create a shortcut on the Home screen. Move the shortcut wherever you want it.

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

```python
import android

droid = android.Android()

droid.makeToast(“Hello from Python on Android”)
```

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

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

That’s all for this time. For now, there’s a TON of documentation about SL4A for free on the web. You can play a bit on your own until next time. I’d suggest that you start by going to 

Greg is the owner of RainyDay Solutions, LLC, a consulting company in Aurora, Colorado, and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family. His website is www.thedesigantedgeek.net.
This time, we’ll finish up using SL4A. We’ll make a larger program and then send it to the virtual machine via ADB.

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

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

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

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

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

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

List of devices attached
emulator-5554 device

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

`adb push source_filename destination_path_and_filename`

So, in my case it would be...

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

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

11 KB/s (570 bytes in 0.046s)

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

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

I’m sure you can figure out that this set of code simply checks the response, and, if it’s ‘None’ due to a timeout, we simply print “Timed out.” And, if it’s actually something we want, then we assign the data to the variable rdial. Now add the next bit of code (below)...

This part of the code will look at the data passed back by the button-press event. We check to see if the response has a “which” key, and, if so, it’s a legitimate button press for us. We then check to see if the result is a “positive” (‘Ok’ button) response. If so, we’ll create another alert dialog, but this time, we will add a list of items for the user to choose from. In this case, we offer the
user to select from a list including Checkers, Chess, Hangman, and Thermal Nuclear War, and we assign the values 0 to 3 to each item. (Is this starting to seem familiar? Yes, it’s from a movie.) We then display the dialog and wait for a response. The part of the response we are interested in is in the form of a dictionary. Assuming the user tapped (clicked) on Chess, the resulting response comes back like this...

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

In this case, we are really interested in the result portion of the returned data. The selection is #1 and is held in the ‘item’ key. Here’s the next part of the code (above right)...

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

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

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

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

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

See you next time.

---

**Greg** is the owner of RainyDay Solutions, LLC, a consulting company in Aurora, Colorado, and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family. His website is...
This time, we are going to take a short detour from our exploration of Android programming, and look at a new framework for GUI programming called **Kivy**. You’ll want to head over to [http://kivy.org](http://kivy.org) and download and install the package – before getting too far into this month’s installment. The Ubuntu installation instructions can be found at [http://kivy.org/docs/installation/installation-ubuntu.html](http://kivy.org/docs/installation/installation-ubuntu.html).

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

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

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

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

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

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

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

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

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

The next eight lines are the main application class. The class is defined, then a routine called build is created. You will almost always have a build routine somewhere in your Kivy programs. Next we set a root object from the Accordion widget. Next we create five AccordionItems and set their title. We then add ten labels with the text “Very big
Before we look at the code, run the program.

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

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

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

```python
class BoundedLabel(Label):
    pass

Next a class called “Selector” (below) is created:
class Selector(GridLayout):
    app = ObjectProperty(None)

Now the Application class is created.

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

```python
for valign in ('bottom', 'middle', 'top'):
    for halign in ('left', 'center', 'right'):
        ```

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

```python
label = BoundedLabel(text='V: %s\nH: %s' % (valign, halign),
        size_hint=(None, None),
        halign=halign, valign=valign)

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

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

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

```python
if self.grid:
    self.root.remove_widget(self.grid)
```

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

```python
grid.bind(minimum_size=grid.setter('size'))
self.grid = grid
self.root.add_widget(grid)
```

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

Finally the application is allowed to run.

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

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

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

```python
#:kivy 1.0
```

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

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

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

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

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

Let’s talk for a moment about two widgets that I have passed over in the discussion of

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

```python
def build(self):
    self.root = FloatLayout()
    self.selector = Selector(app=self)
    self.root.add_widget(self.selector)
    self.grid = None
    self.select(0)
    return self.root
<BoundingLabel>:
    canvas.before:
        Color:
            rgb: 1, 0, 0
    Rectangle:
        pos: self.pos
        size: self.size
```
the application code, The GridLayout and the FloatLayout.

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

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

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

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

Until then, explore more of the examples in Kivy, and be sure to go to the documentation page for Kivy at http://kivy.org/docs/.
Before I begin, I want to note that this article marks three years of the Beginning Programming using Python series. I want to thank Ronnie and the entire staff of Full Circle Magazine for all their support and especially, you, the readers. I NEVER thought that this would continue this long.

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

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

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

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

It will look SOMETHING like the text below.

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

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

Here is the beginning of our class called “Transposer”.

```
Transposer Ver 0.1

<table>
<thead>
<tr>
<th>C</th>
<th>C#/Db</th>
<th>D</th>
<th>D#/Eb</th>
<th>E</th>
<th>F</th>
<th>F#/Gb</th>
<th>G</th>
<th>G#/Ab</th>
<th>A</th>
<th>A#/Bb</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C#/Db</td>
<td>D</td>
<td>D#/Eb</td>
<td>E</td>
<td>F</td>
<td>F#/Gb</td>
<td>G</td>
<td>G#/Ab</td>
<td>A</td>
<td>A#/Bb</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>D#/Eb</td>
<td>E</td>
<td>F</td>
<td>F#/Gb</td>
<td>G</td>
<td>G#/Ab</td>
<td>A</td>
<td>A#/Bb</td>
<td>B</td>
<td>C</td>
<td>C#/Db</td>
</tr>
</tbody>
</table>
```
class Transpose(App):
    
def exit(instance):
        sys.exit()

    Now we work on our build routine (middle right). This is needed for every Kivy app.

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

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

    The scrollview contains

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

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

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

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

The properties that are set should be fairly self-explanatory. The only ones that might give you some trouble would be the padding and size_hint properties. The padding is the number of pixels around the item in an x,y reference. Taken directly from the Kivy documentation size_hint (for X which is same as Y) is defined as:

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

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

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

Again, most of this should be fairly clear.

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

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

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

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

## Button Creation and Alignment

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

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

sv =
ScrollView(size_hint=(None, None),
    size=(600,400))
sv.center = Window.center
root.add_widget(sv)

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

sv.add_widget(s)

return root

## Button Placement

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

if __name__ in ('__main__', '__android__'):
    Transpose().run()

#### Code Example

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

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

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

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

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

Greg is the owner of RainyDay Solutions, LLC, a consulting company in Aurora, Colorado, and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family. His website is
This month, we’ll finish up the transposer program that we wrote in Kivy. Hopefully, you saved the code from last time, because we’ll be building upon it. If not, grab the code from FCM#64.

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

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

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

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

Let’s get started by creating a .kv file (above right). This is what will give us the colored labels. It’s a very simple file.
def LoadLabels(w):
    if w == 0:
        tex0 = self.text1
        tex1 = self.text2
    else:
        tex0 = self.text3
        tex1 = self.text4
    for i in range(0,22):
        if i <= 12:
            if i < 10:
                t1 = " " + str(i) + " | 
            else:
                t1 = str(i) + " | "
            t = tex1
        else:
            t = '
        1 = BoundedLabel(text=t1+t[(i*6):(i*6)+78], size=(780, 35),
                          size_hint=(None,None),halign='left',
                          font_name='data/fonts/DroidSansMono.ttf')
        s.add_widget(1)

class BoundedLabel(Label):
    pass

    def Swap(instance):
        if self.whichway == 0:
            self.whichway = 1
            btnWhich.text = "Guitar --> Piano"
            btn1.text = " " + self.text3
            s.clear_widgets()
            LoadLabels(1)
        else:
            self.whichway = 0
            btnWhich.text = "Piano --> Guitar"
            btn1.text = " " + self.text1
            s.clear_widgets()
            LoadLabels(0)
Now we will tweak the root line definition. Change it from...

```python
root = GridLayout(orientation='vertical', spacing=10, cols=1, rows=3)
```

to:

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

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

Now change the button definition line from...

```python
btn1 = Button(text = " " + text1,size=(680,40),
size_hint=(None,None),
halign='left',
```

to:

```python
font_name='data/fonts/DroidSansMono.ttf',
padding=(20,20)
```

Also add:

```python
btn1 = Button(text = " " + self.text1,size=(780,20),
size_hint=(None,None),
halign='left',
font_name='data/fonts/DroidSansMono.ttf',
padding=(20,2),
background_color=[0.39,0.07,0.92,1])
```

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

Next, we will define three AnchorLayout widgets for the three buttons that we will add in later. I named them a0 (AnchorLayout0), a1 and a2. We also add the code for the About Popup, and define our buttons along with the bind statements. This is shown on the next page, top left.

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

```
LoadLabels(0)
```

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

Next, comment out the entire for loop code. This starts with “for i in range(0,19):” and ends with “s.add_widget(btn)”. We don’t need this since the LoadLabels routine does this for us.
Now, save your code and try to run it. You should see a deep purple button at the top, and our pretty blue BoundLabels. Plus, you will notice that the BoundLabels in the scroll window are closer together, which makes it much easier to read.

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

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

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

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

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

```
def ShowAbout(instance):
    popup.open()
```

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

Now our code is written. You can find the full code at [http://pastebin.com/GftmjEnS](http://pastebin.com/GftmjEnS)

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

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

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

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

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

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

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

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

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

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

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

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

```
./build.py - this is the application
--dir ~/transposer - this is the directory where our application code lives.
--package org.RainyDay.transposer - This is the name of the package
```

--name “RainyDay Transposer” - this is the name of the application that will show up in the apps drawer.

--version 1.0.0 - the version of our application

debug - this is the level of release (debug or release)

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

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

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

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

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

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

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

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

- Search for ShowID based on a show name - http://services.tvrage.com/feeds/search.php?show={SomeShow}
- Pull the show level data based on the Show ID (sid) - http://services.tvrage.com/feeds/showinfo.php?sid={SomeShowID}
- Pull the episode list for Show ID (sid) - http://services.tvrage.com/feeds/episode_list.php?sid={SomeShowID}

<?xml version="1.0" encoding="UTF-8" ?>
<ROOT_TAG>
  <PARENT_TAG>
    <CHILD_TAG>DATA</CLOSING CHILD_TAG>
    <CHILD_TAG>DATA</CLOSING CHILD_TAG>
    <CHILD_TAG>DATA</CLOSING CHILD_TAG>
  </CLOSING PARENT_TAG>
</CLOSING ROOT_TAG>
What gets returned is a stream of data in XML format. Let’s take a moment to review what XML looks like. The first line should always be similar to the one shown below to be considered a proper XML data stream (below).

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

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

```
<TAG INFORMATION = VALUE>
<CHILD TAG>DATA</CLOSING CHILD TAG>
</CLOSING TAG>
```

Sometimes, you might see a tag with no data associated with it.

```
<prodnum/>
```

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

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

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

The returned XML file would look like this:

```
http://pastebin.com/Eh6Ztj9N.
```

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

```
<Results> - This is the ROOT of the XML data. The last line of
```

the stream we get back should be the closing tag </Results>. Basically, this marks the beginning and end of the XML stream. There could be zero results or fifty results.

```
<show> This is the parent node that says “What follows (until the end show tag) is the information about a single tv show”. Again, it’s ended by its end tag </show>. Anything within these two tags should be considered one show’s worth of information.
```

```
<showid>2930</showid> This is the showid tag. This holds the sid that we have to use to get the show information, in this case 2930.
```

```
<name>Buffy the Vampire Slayer</name> This is the name of the show
```

```
<link>...</link> This would be the link to the show itself (or, in the case of an episode, the episode information) on the TVRage website.
```

```
</country>...</country> The country of origin for the show.
...
</show>
</Results>
```

In the case of our program, we would be really interested in only the two fields <showid> and <name>. We might also consider paying attention to the <started> field as well. This is because we rarely get back just one set of data, especially if we didn’t give the absolutely complete show name. For example, if we were interested in the show “The Big Bang Theory,” and searched using only the string “Big Bang”, we would get twenty or so data sets back because anything that even remotely matched “big” or “bang” would be returned. However, if we were interested in the show “NCIS,” and we searched for that, we would get back many
take the case of one of my favorite TV shows, Doctor Who. Doctor Who is a British TV show that, in its original form, started in 1963 and ran for 26 seasons (‘series’ for our friends in the UK) until 1989. Its first season alone had 42 episodes, while other seasons/series have around 24 episodes. You can see where you might have a HUGE stream to parse through.

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

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

In the Show
Information stream we would (normally) want...
<seasone>
<started>
<start_date>
<origin_country>
<status>
<genres>
<runtime>
<network>
<airtime>
<airday>
<timezone>

and from the episode list stream...
<Season>
<episode_number>
<season_number>
<production_number>
<airdate>
<link>
<title>

A word of “warning” here. Season number and Episode number data are not what you might think right away. In the case of the data from TVRage, the season number is the number of the episode within the season. The episode number is the number for that episode within the total life span of the series. The production number is a number that was used internally to the series, that, for many people, means little if anything.

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

Until then, have a good holiday season.

---

Greg Walters is owner of RainyDay Solutions, LLC, a consulting company in Aurora, Colorado, and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family. His website is
Last time, we had a gross discussion about the TVRAGE web API. Now we will start to look at writing code to work with it.

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

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

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

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

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

```python
def FindIdByName(self, showname, debug = 0):

def GetShowInfo(self, showid, debug = 0):

def GetEpisodeList(self, showid, debug = 0):

def DisplaySearchResult(self, showListDict):

def DisplayShowInfo(self, dict):

def DisplayEpisodeList(self, self, SeriesName, SeasonCount, EpisodeList):

def main()
```

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

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

```python
self.ApiKey = "Itn18IyY1hS9n0IP6zI"


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

```python
http://services.tvrage.com/myfeeds/search.php?key=Itn18IyY1hS9n0IP6zI&show={ShowName}
```

We combine the string like this...

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

For the purposes of testing, I will be using a show named “Continuum” which, if you’ve never seen it, is a wonderful science fiction show on the Showcase network out of Canada. I’m using this show for a few reasons. First, there are...
You’ll create a file with the name of “tvrage.py”. We’ll be using this for the next issue or two.

We’ll start with our imports shown above right.

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

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

As I said earlier, the first four lines are our partial strings to build the URL for the function that we want to use.

First (middle right), we set up the string that will be used as the URL. Next, we set up the

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

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

Now we start our class. The name of the class is “TvRage”. We’ll also make our __init__ routine now.

class TvRage:
    def __init__(self):
        self.ApiKey = "Itnl8lyY1hsR9n0IP6zI"
        self.ShowList = []
socket with an 8 second default timeout. Then we call
urllib.urlopen with our
generated URL and (hopefully)
receive our xml file in the usock
object. We call ElementTree
setup so we can parse the xml
information. (If you are lost
here, please re-read my articles
on XML (parts 10, 11 and 12
appearing in FCM #36, 37 and
38)). Next, we close the socket,
and initialize the counter for
the number of matches found,
and reset the list ‘showlist’ to
an empty list.

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

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

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

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

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

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

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

Finally, we increment the foundcounter variable, and append this dictionary into the list ‘showlist’. Then we start the entire thing over until there is no more XML data. Once everything is done, we return the list of dictionaries (bottom right).

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

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

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

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

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

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

See you soon.

```python
foundcounter += 1
self.showList.append(dict)
```
Programming In Python: Pt 41

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

    for child in tree:
        if child.tag == 'showid' :
            dict['ID'] = child.text
        elif child.tag == 'showname' :
            dict['Name'] = child.text
        elif child.tag == 'showlink' :
            dict['Link'] = child.text
        elif child.tag == 'origin_country' :
            dict['Country'] = child.text
        elif child.tag == 'seasons' :
            dict['Seasons'] = child.text
        elif child.tag == 'image' :
            dict['Image'] = child.text
        elif child.tag == 'started' :
            dict['Start'] = child.text
        elif child.tag == 'startdate' :
            dict['StartDate'] = child.text

    Show Link
    Origin Country of network
    Number of seasons
    Series image
    Year Started
    Date Started
    Date Ended
    Status
    (canceled, returning, current, etc)
    Classification
    (scripted, reality, etc)
    Series Summary

    Genre(s)
    Runtime in minutes
    Name of the network that originally aired the show
    Network country
    (pretty much the same thing as Origin Country)
    Air time
    Air Day (of week)
    Time Zone

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

• Show ID
• Show Name

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

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

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

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

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

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

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

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

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

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

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

* Season
* Episode Number

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

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

Kiera’s first desire is to get "home." But until she figures out a way to do that, she must survive in our time period and use all the resources available to her to track and capture the terrorists before they alter

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

    def GetEpisodeList(self, showid, debug=0):
        showidstr = str(showid)
        strng = self.GetEpisodeListString + self.ApiKey
        showidstr
        urllib.socket.setdefaulttimeout(8)
        usock = urllib.urlopen(strng)
        tree = ET.parse(usock).getroot()
        usock.close()
        for child in tree:
```

... Screen Capture Image of Episode (if available)

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

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

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

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

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

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

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

Next, we need to create our display routine. Again, it’s fairly straightforward. The only thing that you might not recognize is the “if e.has_key(‘keynamehere’)” lines. This is a check to make sure that there is actually data in the “Rating” and “Summary” variables. Some shows don’t have this information, so we include the check to make our print-to-screen data a little prettier (shown above right).

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

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

That’s it for this month. As always, you can find the full source code on pastebin at http://pastebin.com/kWSEfs2E.

I hope you enjoy playing with the library. There is additional data available from the API that you can include. Please remember, TVRage provides
def main():
    tr = TvRage()
    # Find Series by name
    nam = raw_input("Enter Series Name -> ")
    if nam != None:
        sl = tr.FindIdByName(nam)
        which = tr.DisplayShowResult(sl)
        if which == 0:
            sys.exit()
    else:
        option = int(which)-1
        id = sl[option]['ID']
        print "ShowID selected was %s" % id
    # Get Show Info
    showinfo = tr.GetShowInfo(id)
    # Display Show Info
    tr.DisplayShowInfo(showinfo)
    # Get Episode List
    SeriesName,TotalSeasons,episodelist = tr.GetEpisodeList(id)
    # Display Episode List

this information for free, so consider donating to them to help their efforts at updating the API and for all their hard work.

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

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

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

```
name here if you
care.extension
```

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

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

and the very latest episode would be:

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

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

```
Power of Three.mp4
```

The directory structure should be as follows:

```
TvShows
   2 Broke Girls
     Season 1
       Episode 1
       Episode 2
```

... Season 2
... Doctor Who 2005
   Season 1
   ... Season 2
   ...

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

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

```
 avi
 mkv
 m4v
 mp4
```

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

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

Let's start with our imports:

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

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

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

This (next page, bottom right) is our main worker
routine. In it, we create a connection to the SQLite database provided by apsw. Next we create a cursor to interact with it. Then we call the MakeDatabase routine which will create the database if it doesn't exist.

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

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

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

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

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

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

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

```python
def WalkThePath(filepath):
    # Open the error log file
    efile = open('errors.log', 'w')
    for root, dirs, files in os.walk(filepath, topdown=True):
        # Is a string which is the path to the directory, directory names is a list of the names of subdirectories in the path, and the filenames is a list of non-directory names. We then parse through the list of filenames, checking to see if the filename ends with one of our target extensions.
```

```
```
for file in [f for f in files if f.endswith(('avi','mkv','mp4','m4v'))]:
    if isok:
        shoname = data[0]
        season = data[1]
        episode = data[2]
        print("Season {0} Episode {1}".format(season,episode))
    else:
        print("No Season/Episode")
        efile.writelines('----------
')
        efile.writelines('{} has no series/episode information
'.format(file))

sqlquery = 'SELECT count(pkid) as rowcount from TvShows where Filename = "%s"; % f
try:
    for x in cursor.execute(sqlquery):
        rcntr = x[0]
        if rcntr == 0:  # It's not there, so add it
            try:
                sql = 'INSERT INTO TvShows (Series,RootPath,Filename,Season,Episode)
VALUES (?,?,?,,?)'
                cursor.execute(sql,(shoname,root,fl,season,episode))
            except:
                print("Error")
                efile.writelines('----------
')
                efile.writelines('Error writing to database...
')
                efile.writelines('Filename = {0}
'.format(file))
                efile.writelines('----------
')
            except:
                print("Error")

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

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

Now, let's look at the
GetSeasonEpisode routine.

#================================

OriginalFilename,ext = os.path.splitext(file)
fl = file
isok,data = GetSeasonEpisode(fl)

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

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

    shonname = shonname.replace(" ", " ")

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

    ret = [shonname, season, episode]
    return True, ret

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

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

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

    Season 02 Episode 04
    SELECT count(pkid) as rowcount from TvShows where Filename = "InSecurity.S02E04.avi";
    Series - INSECURITY File -

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

<table>
<thead>
<tr>
<th>pkID</th>
<th>Series</th>
<th>Root Path</th>
<th>Filename</th>
<th>Season</th>
<th>Episode</th>
</tr>
</thead>
<tbody>
<tr>
<td>2526</td>
<td>NCIS</td>
<td>/extramедia/tv_files/NCIS/Season 7</td>
<td>NCIS.S07E04.Good.Cop.Bad.Cop.avi</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

anything from the first character up to the ".s", then two numbers, skip the "e", then two numbers, and repeat. So the filename "Monk.S01E05.S01E06.avi" returns the following groups...

```python
S[1] = Monk
S[2] = 01
S[3] = 05
S[4] = 01
S[5] = 06
```

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

Our first pattern tries to catch multi-episode files. There are various naming schemes, but the one we will support is similar to ’S01E03.S01E04’. We use the pattern string "(.*).\s+(\d{1,2})\s+\((\d{1,2})\s+(\d{1,2})\s+(\d{1,2})\). This returns (hopefully) five groups which consist of: the series name (S[1]), season(S[2]), episode number 1 (S[3]), season (S[4]), and episode number 2 (S[5]). Remember that the parens create each group for returns. In the case above, we group named “GoOn” to true. This allows us to know what we should do after we’ve fallen through the various If lines.

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

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

Next, in our MakeDatabase routine, we will change the create SQL statement to add the two new fields (next page, top).

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

In our WalkThePath routine, the only changes are the lines that actually insert into the database. This is to support the new structure. If you remember

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

This pattern checks for Sdddd.
```python
# check for Sdddd
resp = re.search(r'(\d\d)\d\d\d', filename, re.I)
if resp:
    showname = resp.group(1)
    GoOn = True
else:
```

And finally we try for ddd
```python
# Should catch xxx
resp = re.search(r'(\d)((\d\d)\d)', filename, re.I)
if resp:
    showname = resp.group(1)
```
def MakeDataBase():
    # If the table does not exist, this will create the table.
    # Otherwise, this will be ignored due to the 'IF NOT EXISTS' clause
    sql = 'CREATE TABLE IF NOT EXISTS TvsShows (pkID INTEGER PRIMARY KEY, Series TEXT, RootPath TEXT, Filename TEXT, Season TEXT, Episode TEXT, tvrageid TEXT, status TEXT);
    cursor.execute(sql)

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

def WalkTheDatabase():
    tr = Tvrage()
    SeriesCursor = connection.cursor()
    sqlstring = "SELECT DISTINCT series FROM TvsShows WHERE tvrageid = -1"
    print sqlstring
    SeriesCursor.execute(sqlstring)
    Series = SeriesCursor.fetchall()
    print Series

    sqlstring = "SELECT DISTINCT series FROM TvsShows WHERE tvrageid = -1"
    SeriesCursor = connection.cursor()
    SeriesCursor.execute(sqlstring)
    Series = SeriesCursor.fetchall()
    print Series

WalkThePath routine, and runs through the database, getting the series name and querying the Tvrage server for the id number. Once we have that, we update the database, then use that id number to once again query Tvrage to get the current status of the series. This status can be "New Series", "Returning Series", "Canceled", "Ended" and "On Haltus". The reason we want this information is that, when we go to check for new episodes, we don't want to bother with series that won't have any new episodes because they are cancelled. So, now we have the status and can write that to the database (above).

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

SELECT DISTINCT series FROM TvsShows WHERE tvrageid = -1

Which says, give me just one instance of the series name, no matter how many of them I have, where the field tvrageid is equal to"-1". If, for example, we have 103 episodes of Doctor Who 2005. By using the Distinct, I will get back only one record, assuming that we haven't gotten a TvrageID yet.
def UpdateDatabase(seriesname, id):
    idcursor = connection.cursor()
    sqlstring = 'UPDATE tvshows SET tvrageid = ' + id + ' WHERE series = ' + seriesname + ''
    try:
        idcursor.execute(sqlstring)
    except:
        print "error"

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

for x in SeriesCursor.execute(sqlstring):
    seriesname = x[0]
    searchname = string.capwords(x[0], "")

    We are using the capwords routine from the string library to change the series name (x[0]) to a "proper case" from the all-uppercase we currently store the show name in. We do this because TvRage expects something other than all-uppercase entries, and we won't get the results we are looking for. So the series name "THE MAN FROM UNCLE" will be converted to "The Man From Uncle". We use that in the call to our TvRage Library FindIdByName. This gets the list of matching shows, and displays them for us to pick the best one. Once we pick one, we update the database with the id number and then call the GetShowStatus routine to get the current show status from TvRage (bottom right).

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

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

    startfolder = ["/extramedia/tv_files", "/media/freeagnt/tv_files_2"]
    #for cntr in range(0,2):
    #    WalkThePath(startfolder[tntr])
    WalkTheDatabase()
    # Close the cursor and the database
cursor.close()
connection.close()
HOWTO - PROGRAMMING PYTHON Pt43

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

We are almost done with our code. We finally add one line to our main routine from last month (in black, below) to call the “WalkTheDatabase” routine after we are done getting all our filenames. Again, I’m going to give you only part of the Main routine, just so you can find the correct place to put the new line.

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

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

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

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

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

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

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

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

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

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

Next time we will continue with the integration with rRage. Until then have a great onth!

---

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

---

full circle magazine 211 The Compleat Python
Make sure that 'Main Window' is selected, and click the 'Create' button. Now you will have a blank form that you can drag and drop controls onto.

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

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

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

Just above that is the objectName property—which, by default, is set to 'pushButton'. Change that to 'btnExit'. Now scroll down on the Property Editor list to the 'QAbstractButton' section, and set the 'text' property to 'Exit'. You can see on our form that the text on the button has changed.
Now, add another button and position it at 200,200. Change its objectName property to 'btnClickMe,' and set the text to 'Click Me!'.

Next add a label. You will find it in the toolbox on the left under 'DisplayWidgets'. Put it close to the center of the form (I put mine at 210,130), and set its objectName property to lblDisplay. We will want to make it bigger than what it is by default, so set its size to somewhere around 221 x 20. In the property editor, scroll down to the 'QLabel' section, and set the Horizontal alignment to 'AlignHCenter'. Change the text to blank. We will set the text in code—when the btnClickMe is clicked. Now save the project again.

**Slots & Signals**

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

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

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

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

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

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

```
pyuic4 -x pytest1.ui -opystest1.py
```

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

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

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

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

```python
QtCore.QObject.connect(self.btnExit,
QtCore.SIGNAL(_fromUtf8("clicked()")), MainWindow.close)
```

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

```python
QtCore.QObject.connect(self.btnCloseMe,
QtCore.SIGNAL(_fromUtf8("clicked()")), self.SetLabelText)
```

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

```python
def SetLabelText(self):
    self.lblDisplay.setText(_fromUtf8("That Tickles!!!"))
```

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

Now run your code. Everything should work as expected.

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

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

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

---

**MY STORY QUICKIE**

By Anthony Venable

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

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

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

This experience just goes to show once again how I manage to find the coolest stuff by accident.
This time, we are going to rework our database program from the previous few articles (parts 41, 42 and 43). Then, over the next few articles, we will use QT to create the user interface.

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

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

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

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

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

Replace the line that says

```
sqlquery = ‘SELECT count(pkid) as rowcount from TvShows where Filename = %s’ % fl
```

with

```
sqlquery = ‘SELECT count(pkid) as rowcount from series where seriesName = %s’ % showname
```

This (to refresh your memory) will check to see if we have already put the series into the table. Now find the two lines that say:
HOWTO - PYTHON PT45

sql = ‘INSERT INTO Series(RootPath,Filename,Season,Episode,tvrageid) VALUES (?, ?, ?, ?, ?)’

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

sql = ‘SELECT pkid, SeriesName FROM Series WHERE SeriesID = -1’

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

seriesname = x[0]

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

with
pkid = x[0]

seriesname = x[1]

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

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

UpdateDatabase(seriesname, id)

to

UpdateDatabase(seriesname, id, pkid)

Next, we need to change the query string from

sqlstring = ‘UPDATE tvshows SET tvrageid = ‘ + id + ‘ WHERE series = ‘ + seriesname + ‘’

to

sqlstring = ‘UPDATE Series SET SeriesID = ‘ + id + ‘ WHERE pkID = ‘ + pkid

Now we need to create the GetShowData routine (top). We’ll grab the information from TvRage and insert it into the Series table.
Finally, we create the query string to do the update (bottom). Again, this should all be on one line, but I’ve broken it up here to make it easy to understand.

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

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

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

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

```
sqlstring = "Update Series SET Seasons = ", StartDate = ", Ended = ",
OriginCountry = ", Status = ", Classification = ",
Summary = ", Genres = ", Runtime = ", Network = ",
AirDay = ", AirTime = " WHERE pkID =".format(seasons,startdate,ended,
origincountry,status,classification,summary,
genres,runtime, network, airday,airtime,pkID)```

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

```
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
```

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

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

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

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

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

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

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

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

```python
>>> Basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)
set(['orange', 'pear', 'apple', 'banana'])
```

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

```python
>>> 'orange' in fruit
True
>>> 'kiwi' in fruit
False
```

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

```python
>>> shoppinglist = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'grapes']
>>> basket = ['apple', 'kiwi', 'banana']
>>> sl = set(shoppinglist)
>>> b = set(basket)
```

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

```python
>>> sl-b
set(['orange', 'pear', 'grapes'])
```

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

The routine, when you run it, prints out [5, 8, 15], which is correct. Now let's look at the code. The first line creates a set...
operator on the two sets. We do this so we can sort it with the list.sort() method. You can
certainly return the list if you wish, but in this iteration of the
routine, we’ll just print it out.

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

Have a good month and see
you soon.

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

Now we can create a list
using the set difference

```python
def FindMissing(expected, have):
    # ==================================
    # ‘expected’ is the number of episodes we should have
    # ‘have’ is a list of episodes that we do have
    # returns a sorted list of missing episode numbers
    # ==================================
    EpisodesNeeded = set(range(1, expected+1))
    EpisodesHave = set(have)
    StillNeed = list(EpisodesNeeded - EpisodesHave)
    StillNeed.sort()
    print StillNeed
```
Last month, we discussed using sets to show us missing episode numbers. Now's the time to put the rough code we presented into practice.

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

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

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

```python
currentseason = ''
```

By now, you should recognize that all we're doing here is initializing variables. There are three string variables and one list. We will use the list to hold the episode numbers (hence the `elist` name).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    We define the function and set up four parameters. We will be passing the episode list (elist), the number of episodes we should expect (shouldhave) which is the highest episode
```

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

```python
```
Next, we create a set that contains a list of numbers using the range built-in function, starting with 1 and going to the value in shouldhave + 1. We then call the difference function - on this set and a converted set from the episode list (temp-set(episode)) - and convert it back to a list. We then check to see if there is anything in the list - so we don't print a line with an empty list, and if there's anything there, we print it out.

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

I've put the two routines up on pastebin for you if you just want to do a quick replace into your working code. You can find it at http://pastebin.com/XHTrv2dQ.

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

podcast.ubuntu-uk.org

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

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

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

http://fullcirelmagazine.org/issue-

http://fullcirelmagazine.org/python-

special-edition-issue-three/

http://fullcirelmagazine.org/python-

special-edition-volume-four/

http://fullcirelmagazine.org/python-

special-edition-volume-five/

http://fullcirelmagazine.org/python-

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

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

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

```python
>>> a = "Hello Python"
>>> print("String a is %s" % a)
and the output will be in bold, like this:
String a is Hello Python

**PYTHON 2.x**

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

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

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

```python
>>> pi = 3.14159
>>> print('PI = %5.3f.' % pi)
PI = 3.142.
```

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

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

```python
>>> info = {
  "FName":"Fred","LName":"Farkel","City":"Denver"
} 
>>> print('Greetings %(FName)s %(LName)s of %(City)s!' % info)
Greetings Fred Farkel of Denver!
```

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

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'d'</td>
<td>Signed integer decimal</td>
</tr>
<tr>
<td>'D'</td>
<td>Signed integer decimal</td>
</tr>
<tr>
<td>'u'</td>
<td>Obsolete - identical to 'd'</td>
</tr>
<tr>
<td>'U'</td>
<td>Signed octal value</td>
</tr>
<tr>
<td>'x'</td>
<td>Signed hexadecimal - lowercase</td>
</tr>
<tr>
<td>'X'</td>
<td>Signed hexadecimal - uppercase</td>
</tr>
<tr>
<td>'e'</td>
<td>Floating point decimal</td>
</tr>
<tr>
<td>'E'</td>
<td>Floating point decimal</td>
</tr>
<tr>
<td>'f'</td>
<td>Floating point exponential - lowercase</td>
</tr>
<tr>
<td>'F'</td>
<td>Floating point exponential - uppercase</td>
</tr>
<tr>
<td>'g'</td>
<td>Floating point format - uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise</td>
</tr>
<tr>
<td>'G'</td>
<td>Floating point format - uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise</td>
</tr>
<tr>
<td>'r'</td>
<td>Single character</td>
</tr>
<tr>
<td>'y'</td>
<td>String (converts valid Python object using repr())</td>
</tr>
<tr>
<td>'s'</td>
<td>String (converts valid Python object using str())</td>
</tr>
<tr>
<td>'A'</td>
<td>No argument is converted, results in a '%A' character</td>
</tr>
</tbody>
</table>
Python 3.x

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

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

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

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

```python
>>> print("This version of Python is {0}. format("Python","3.3.2")
```

This version of Python is 3.3.2

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

```python
>>> print('|{:<20}|{>:20}|{:^20}|
|Left  | Center |
|Right |       |
```

Here is an example:

```python
>>> print("This is a big number {0}. format(7219219281)"
This is a big number 7,219,219,281
```

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

```python
>>> d =
datetime.datetime(2013,10,9,10,45,1)
>>> print("{:m/%d/%y}. format(d)
10/09/13
```

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

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

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

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

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

podcast.ubuntu-uk.org
W

H O W-TO

Written by Greg Walters

This is called Representation
Error, and exists in almost
every modern programming
language (Python, C, C++,
Java, and even Fortran and
more), and on almost every
>>>2.2+3.3
modern computer. This is
>>>1/10.0
5.5
because these machines use
0.1
IEEE-754 floating-point
Now you are even more
Ok.
Reality
is
back.
No,
not
arithmetic which (on most
confused and you think to
really.
Python
is
simply
showing
machines and OS platforms)
yourself “Ok. This is either a
you
a
rounded
version
of
the
bug or some kind of sick Easter answer. So, how do we see the maps to an IEEE-754 doubleegg.” No, it’s neither a bug nor “real” answer? We can use the precision number. This doubleprecision number has a
an Easter egg. It’s real. While I decimal library to see what’s
precision of 53 bits. So, our 0.1,
knew about this a very long
really
happening.
when represented in this 53-bit
time ago, it had slipped into the
double-precision, turns into:
>>> from decimal import *
cobwebs hidden in the dark
>>> Decimal(1/10.0)
recesses of my old mind, so I
(1.1+2.2) == 3.3,
Decimal('0.100000000000000005 0.000110011001100110011001100
had to bring it up here. What
55111512312578270211815834045 11001100110011001100110011010
you might be surprised that the we are seeing is the joy of
41015625')
binary floating-point numbers.
shell responds “False”.
That’s close to .1, but not
WOW. So let’s try our original
close enough to avoid issues.
We all know that ⅓ equates formula and see what that
WHAT?! ! ?! ?
to .33333333333333333… for would show:
So what do we do about it?
Now, confused, you type at ever and a day, but take, for
>>> Decimal(1.1+2.2)
Well, the quick answer is that
example, the fraction 1/10.
Decimal('3.300000000000000266
the prompt:
you probably can live with it for
Everyone knows that 1/10 is
45352591003756970167160034179
90% of the things we have to
>>>1.1+2.2
equal to .1, right? If you use the 6875')
do out there in the real world –
shell you can see
And the shell responds back: interactive
by using the round() method.
It seems to just be getting
that:
While you have to decide on
worse and worse. So what is
3.3000000000000003
>>>1/10
the number of decimal points
really happening?
0
that you must have in your
full circle magazine
225 The Compleat Python
contents ^
hile I was working
this week, a very
wise person by the
name of Michael
W. suggested that I should
consider what happens with
floating-point numbers and
equality.
Take for example a simple
calculation: 1.1 + 2.2
The answer, you say, is 3.3!
Any school-kid who has dealt
with fractions knows that. Well,
tell your computer. If you start
up the Python Interactive Shell
and at the prompt type

You stare at the screen in
disbelief and first think “I must
have typed something wrong”.
Then you realize that you
didn’t. So you type:

Programming In Python - Part 49
Oh, right. We have to have
at least one of the values a
floating-point value to show any
decimal points since an
integer/integer returns an
integer. So we try again.


world to carry the precision that you need, for the most part, this will be an acceptable workaround.

I honestly don’t remember if we have gone over the round method, so I’ll briefly go over it. The syntax is very simple:

```python
round(v, d)
```

where `v` is the value you want to round and `d` is the number of decimals (maximum) you want after the decimal point. According to the Python documentation, “Values are rounded to the closest multiple of 10 to the power of minus `n` digits; if two multiples are equally close, rounding is done away from 0”. All that being said, if the number is 1.4144, and we round it to 3 decimal places, the returned value will be 1.414. If the number is 1.4145 it would be returned as 1.415.

For example, let’s use the value of pi that comes from the math library. (You must import the math library before you can do this, by the way.)

```python
>>> math.pi
3.141592653589793
```

Now, if we wanted to round that value down to 5 decimal places, we would use:

```python
>>> round(math.pi, 5)
3.14159
```

That is the “standard” value of pi that most everyone knows off the top of their head. That’s great. However, if we set the number of decimal places to be returned to 4, look what happens.

```python
>>> round(math.pi, 4)
3.1416
```

All that sounds good until you run into a value like 2.675 and try to round it to 2 decimal places. The assumption (since it is exactly halfway between 2.67 and 2.68) is that the returned value will be 2.68. Try it.

```python
>>> round(2.675, 2)
2.67
```

That might cause a problem. It goes back to the initial issue we have been talking about. The actual conversion to a binary floating-point number that is 53 bits long, the number becomes:

```
2.674999999999982365316059974953221893310546875
```

which then rounds down to 2.67.

The bottom line here is when trying to compare floating-point numbers, be aware that some things just don’t translate well.

**See you next time!**
Notice that we included None as the translate table. While this part is cool, it gets better. There is a function called maketrans. It takes an input string and an output string as parameters and returns a table that is used as the first parameter into the translate method. Here (top right) is a very simple example.

It returns:
Th2 t3m2 hls c4m2

Let’s look at what this does. We assign intable to a string of vowels as before. outtable is assigned the numbers 1, 2, 3, 4, 5 as a string. When we make the call to maketrans, our actual tratable is as follows (shown below). The “\x” means that it is hexadecimal char):

If you look at it carefully, you’ll see that the lowercase vowel letters are replaced with the numbers we specified:

1bcd2fgh3jkln4pqrs5vxwyz

If you look even closer, you’ll see that there actually 256 entries starting with “\x00” and ending with “\xff”. So the table contains the entire 256 possible ascii character set. So, when the translate method gets the table, it iterates (or walks through) each character, getting that characters value in Hex, and then finds that value in the translate table and substitutes it in the output string. The Hex representation of our original astr string (‘The time has come’) is shown below.

So now it should be making sense.

Now the purpose of this whole thing. Think back to your schooling where you learned about Julius Caesar. Whenever he wanted to send a message of a confidential matter, he would use a cipher that would shift all the letters of the

\x54\x68\x74\x61\x73\x74\x65\x73\x74\x20\x79\x73\x74\x65\x73\x74\x20\x61\x72\x74\x69\x6f\x6e\x20\x63\x6f\x6d\x70\x65\x74\x20\x74\x61\x69\x6d\x69\x73\x6f\x6e\x65\x72\x20\x74\x68\x75\x72\x65
The time has come

'\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f '!"#$&\'()\*+/0123456789:;<=?>@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcd2fgh3jkln4pqrs5vxwyz{|}~\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb
full circle magazine 227 The Compleat Python contents
alphabet three characters to the right. So, using todays
english alphabet:
ABCDEFGIJKLMNOPQRSTUVWXYZabc
defghijklmnopqrstuvwxyz

becomes:
DEFGHIJKLMNOPQRSTUVWXYZabcdef
gijklmnopqrstuvwxyzABC

While this seems very simple
by today’s standards, when I
was a school kid, we used this
all the time to send messages
each other. We used a
different index into the string to
start the encryption string, the
logic behind it was the same.

No one knows how effective
this actually was for good old
Julius. One would think that if
someone intercepted the
message, they would have
thought that it was in some
foreign language. We can only
speculate.

We can easily use the
translate method and the
maketrans helper function to
allow us to have fun with this.
Let’s say we want to make a
simple program that allows us
to enter a string of “plain text”
and get back an encrypted
string using the same side right
method that Caesar used. For
simplicity sake, let’s only use

full circle magazine

from string import maketrans
#-----------------------------
intab = "ABCDEFGIJKLMNOPQRSTUVWXYZ"
outtab = "DEFGHIJKLMNOPQRSTUVWXYZABC"
EncTrantab = maketrans(intab,outtab) #Encode
DecTrantab = maketrans(outtab,intab) #Decode

 instructed = raw_input("Enter the plaintext string -> ")
EncString = instructed.translate(EncTrantab)
DecString = EncString.translate(DecTrantab)

intab string and in between the
"Z" and the "A" in the outbound
string. This helps keep the
actual words from being too
obvious in the encrypted string.
The next change is where we
ask if the user wants to encode
or decode the string. Finally we
added an if statement to
control what we print (shown
bottom right).

The output from the
program is:

Encode or Decode (E or D) -> E
Enter the string -> THE TIME HAS

#Be sure to include the space character in the strings
intab = "ABCDEFGIJKLMNOPQRSTUVWXYZ"
outtab = "DEFGHIJKLMNOPQRSTUVWXYZABC"
EncTrantab = maketrans(intab,outtab) #Encode
DecTrantab = maketrans(outtab,intab) #Decode

which = raw_input("Encode or Decode (E or D) -> ")
instructed = raw_input("Enter the string -> ")
EncString = instructed.translate(EncTrantab)
DecString = instructed.translate(DecTrantab)

if which == "E":
print("Encoded string is - %s" % EncString)
else:
print("Decoded string is - %s" % DecString)
The month, I’m going to discuss a product that is new to me, but has apparently been around for a number of years. It’s called NextReports from Advantage Software Factory, and you can get it free from http://www.next-reports.com/. Not only that, but it’s open source and it runs under Windows and Linux!

Before I start telling you about the product, let me get on my soapbox and vent for a moment or two. For a long time, I’ve been working with databases and reports. One of the things that I’ve had issues with is that while there are free database solutions out there, like SQLite and MySql, there was precious little available that was free for report designer tools. More times than not, any reports either had to be done with very expensive software tools, or the developer had to roll his own. Some tools were available, but were lacking. When it came to charting, well, you pretty much had no choice but to use the expensive stuff. Believe me, I’ve looked for years for really good free reporting tools, and I’m not sure how I have missed this package for so many years (version 2.1 was released in March of 2009 and they are currently up to version 6.3). But now that I’ve found it, I’m absolutely pumped about it.

Now that I’ve stepped down from my soap box, I can begin to sing its praises. It is a suite of three parts, a reports designer, a report engine and a report server. All I have had a chance to play with is the reports designer, but if the designer is any indication of the power, ease and flexibility of the rest of the suite, this thing is a winner.

This month, we are going to concentrate on the designer. Because of some constraints on my time, I’m working on a Windows machine, but everything that I show can be done in Linux (so please forgive me in advance).

One of the first things you should know is that it supports databases like Oracle, MySql, SQLite, MSSQL and more. Everything is based on queries and a really good thing is that only SELECT type queries are allowed. This means that nothing in the source database can be changed by accident. You can enter your own queries or use a visual designer.

The screenshot shows how nice a UI it is. Things are pretty intuitive and it won’t take you long to be productive at this. Let’s take a look at the steps to get going.

Start with File | New | Data Source. Next, name your source whatever you want to call it.

Now tell NextReports what kind of database it is in the dropdown called “Type:”. You can skip over the Driver section and go to the URL: section. This
is where you put the path to the database. If you are using, for example, a SQLite database, this will be filled in for you: “jdbc:sqlite:<dbfile-path>”. Replace the <dbfile-path> with the path to your database. Other types of databases have similar types of information already populated to help you. Next, click the “Test” button to make sure you can connect. If everything goes correctly, then click “Save” and you’ll see it added to the Connections tree. The next thing you need to do is make a connection to your database that you have just added. Now, right click on the database and then click on Connect. Once you are connected, you’ll see that you have four possible things to choose from. The “%” is the database tables. The next three are so you can create new queries, reports and charts. Simple enough. Now click on the “+” sign to the left of “%” which will open up your database table display. Now you will have Tables, Views and Procedures in the tree. Once again, click on the “+” sign next to “Tables”. This will show all your tables. Now if you want to use the visual query designer, just drag the table(s) you want to deal with onto the designer canvas to the right.

Once you have all your tables there, you can start making connections between the tables.

In the example here, I have two tables, one with information about kids in a confirmation class and the other with entries for worship notes taken. The worship note table doesn’t have the kid’s name in it, just an id that points to the kid information table. I did a drag and drop to make
that link between the kidId field and the pkID of the kid table. Then I selected each field I wanted to have in the result set. In this case, the kid's first and last name and an active (or not-deleted) flag in the kids table and multiple fields from

```sql
SELECT
  kl.fname,
  kl.lname,
  w1.WorshipNote,
  w1.Usher,
  w1.Other,
  w1.Lector,
  w1.Description,
  w1.Counts,
  w1.Acolyte
FROM
  kids kl,
  worshipNotes w1
WHERE
  w1.kidID = kl.pkID AND
  kl.Active = 1
ORDER BY
  kl.lname,
  kl.fname
```

the notes table. The grid below shows each of the fields, which table it comes from, and other information.

As you can see, we can set criteria like "Active = 1", choose to display a field or not, and set sort type and sort order. Once you are satisfied with this, you can click on the tab below and see your actual SQL query.

To test your query, simply click on the “running man” and you will (hopefully if you did it correctly!) get the query results in a grid below the editor. If you want to add manual lines you can. For example, I want to combine the kids first and last names (fname and lname) into a full name. We can do that by putting a line after the “kl.lname,” line like this:

```sql
kl.fname || " " || kl.lname as FullName,
```

The “||” characters are concatenation characters so we will have the two fields with a space between in a field named “FullName”. Don’t forget the comma at the end. Once you have your query the way you want it, click on the save button to save the query. You will be asked what you want to call it.

Next, click on the Query item in the tree and right click on the query you just created.

Select “New Report from Query”. The query designer canvas goes away and is replaced by the report designer.

On the left is the properties window for any given field or the entire report. On the right is the report designer itself. Notice that it looks like a spreadsheet. Each row is considered a “band” and holds information for that report row. In the case of this example, we
have four rows, two header rows, one detail row and a footer row. You can add or delete rows as needed. This method is not quite as freeform as some other report designers, but makes for a very nice and clean report.

The two header rows hold our report title and column headers. The detail row has each field we will be reporting on and the footer row is the report footer. Let’s take a look at how the report looks as a default. Click on the button at the top of the bar marked “To Html” to see the report. (I blurred the kids last names, that’s not an issue in the generator.)

For a report with almost no work, that’s really nice. But let’s pretty it up a bit. Let’s create a group that puts all of the data for any given kid under the kid’s name.

Right click on the first column of the data row. Select Group and then Add.

You will be presented with a new window asking which of the fields you want to create the group upon. In this case, I select FullName and then click the Ok button. Now we have a grouping break. We can also get rid of the three fields (name, Iname and FullName) in the detail section, since we’ll be displaying the name in the group band. Simply right-click on them and select “Delete Cell”. Now you can resize the three empty cells on the left to make the gap less obvious.

Taking a quick peek at what the report looks like now will show you that the information for each kid is all nicely grouped together.

That’s nicer, but now let’s do something kind of fun. All the 1s and 0s obviously stand for yes and no. That’s rather boring for a report, so let’s add an advanced conditional statement for each of those fields that will show a box with a check for Yes (or 1) and an empty box for No (or 0). It’s really easy to do, but makes your report look like you spent days on it. By using the Wingdings font from Windows, the two characters we want are 0x6F(0168) for an empty box and 0xFE (0254) for a checked box.

Before I go on, the one thing that Windows does better than Linux (that I have found) is the use of the Alt+NumPad entry of special characters. Linux doesn’t allow that. There was a work around that used Ctrl+Shift+U then the unicode value for the character you wanted. However, that doesn’t work on all machines. The easiest way I’ve found to do this on Linux, is to open Character Map, use the search function to find the unicode character you want, double-click the character to copy it to the “Text to copy:” box, then click the “Copy” and then paste it into your document. The unicode characters for them are 2610 (empty box) and 2611 (checked box) using the WingDings 2 font. I’m sure there are many other easier ways to deal with this, but I’m shy on time. (Be sure you have

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Garrett</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Trevor</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Zachary</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Common selected in the Script list.)

We’ll start with the WorshipNotes field in the parentheses of the editor. Simply click in between the two parentheses to place the cursor and then double-click on the field you want to go in there. BAM! It’s filled in for you. Now click after the field name in the editor and then double-click on the “== (eq)” operator. Then add a “1”, so the editor line reads

```python
if ( $C_WorshipNote == 1 ) {
; } else { ; }
```

We are almost finished with our expression. The first set of curly brackets define what to do if the expression is True and the second is what to do if it’s false. In this case, we’ll use the CharMap (in windows, Linux has one as well, for example gucharrmap if you are using Gnome) to copy the characters into our editor string. Or, under windows, you can hold the {Alt} key and press 0168 for the empty box and 0254 for the checked box. So now our expression is (at least in Windows):

```python
if ( $C_WorshipNote == 1 ) ( 
"1"; ) else { "o"; }
```

Name the expression (I used WNotes) and save it.

Under properties for that field, select the font (WingDings is what I used here) and this is what it will look like.

There’s our pretty little boxes. Doing this to the other fields is just as simple.

It only took me about 3 hours of playing with the package to get to this point and a whole lot further. I can truly say that I have a great amount more to learn but that’s for another day. You can use templates to color your report, you can add images, and much more.

Next time, I’ll talk about how we might go about embedding these reports into a Python program. Until then, have fun playing with this wonder FREE software.
Before we get started on this month's actual python subject, let me toot my own horn for just a minute. In late December and early January, my first book on Python was published by Apress. It is named "The Python Quick Syntax Reference", and is available from a number of places. You can find it on the Apress site (http://www.apress.com/9781430264781), Springer.com (http://www.springer.com/computer/book/978-1-4302-6478-1) and Amazon (http://www.amazon.com/The-Python-Quick-Syntax-Reference/dp/1430264780) as well as others. It is, as the title suggests, a syntax reference that will help those of us who program in other languages as well as Python, to remember how a certain command works and the requirements for that command. Please help a poor old programmer make a living by buying the book, if you can.

Now on to bigger and better things.

While I was working on my latest book for Apress, I rediscovered a SQL command that I didn't discuss when we were working with SQL databases a long time ago, so I thought I'd share the information with you. It is the CREATE TABLE AS SELECT command, which allows us to pull a query from one table (or joined tables) and create another table on the fly. The general syntax is:

```
CREATE TABLE [IF NOT EXISTS] 
{New Table Name} AS SELECT 
```

The part in square brackets (IF NOT EXISTS) is totally optional, which will create the table only if it doesn't exist already. The part in curly brackets, however, is not. The first is the new table name and the second is the query that you want to use to pull data and create the new table.

Assume we have a database that has multiple tables in it. One of the tables is named "study" that holds data from a receiving operation. There are six fields which are shown below.

One of the datasets that we will need to produce from this raw data is a grouping of package count and the number of days within the study that quantity of packages came in on, assuming that the days are weekdays (Monday thru Friday) and that the day is not a holiday, since holidays have less than normal number of

```sql
SELECT pkgs, Count(DOW) as CountOfDOW FROM study
WHERE (Holiday <> 1)
    AND DayName in ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
GROUP BY pkgs
```
packages. Our query is shown above.

This then provides us with data that would look something like this:

<table>
<thead>
<tr>
<th>pkgs</th>
<th>CountOfDow</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>3</td>
</tr>
</tbody>
</table>

So the data is showing that during the study of 65 days, only one weekday had 31 packages but 3 weekdays had 48 packages and so on. Similar queries could be created that would cover holidays and weekends.

While having the data simply as a returned dataset from the query, we might want to do further analysis on the data, so we want to put the resulting data from the query into a table. That's why we would create a table from the query. So in the following example, shown above right, we create a table named "weekdays" using the same query we just showed above.

Now anytime we need the data for that weekday result set, we can just run a query on the weekdays table.

Once we know what we need, and have tested the query, then we can begin our code. Assuming we already have the study table created and populated, we can use Python to then create our new table in the main database. Just as an FYI, I am using the APSW SQLite library to do the database work.

We, of course, have to open a connection (right) and create a cursor to the SQLite database. We have covered this in a number of past articles.

```
def OpenDB():
    global connection
    global cursor
    connection = apsw.Connection("labpackagestudy.db3")
    cursor = connection.cursor()
```

Now we need to create the routine that will actually create the table with the returned dataset from the query, shown below, then alter it and run some calculations.

As you can see, we want to create a second cursor, so that we don't run any risk of the first cursor having data we need to maintain. We will be using it in the final part of the code. We then drop the table if it exists and run our query on the “study” table.

Now we create three more columns (shown below) within the weekdays table named probability.

```
def DoWeekDays():
    # Create a second cursor for updating the new table
    cursor2 = connection.cursor()
    q1 = "DROP TABLE IF EXISTS weekdays"
    cursor.execute(q1)
    query = "'CREATE TABLE IF NOT EXISTS weekdays AS SELECT pkgs, Count(DOW) as CountOfDOW FROM study WHERE (Holiday <> 1) AND DayName in ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday') GROUP BY pkgs'

    addcolquery = 'ALTER TABLE weekdays ADD COLUMN probability REAL'
    cursor2.execute(addcolquery)
    addcolquery = 'ALTER TABLE weekdays ADD COLUMN lower REAL'
    cursor2.execute(addcolquery)
    addcolquery = 'ALTER TABLE weekdays ADD COLUMN upper REAL'
    cursor2.execute(addcolquery)
```

```
“probability”, “lower” and “upper”. We do this by using the “ALTER TABLE” SQL command.

The next step (top right) will be to sum the data in the CountOfDow field.

There is only one record returned, but we do the for loop thing anyway. Remember from the above discussion that the “CountOfDow” field holds the number of days during the study that a particular number of packages came in. This gives us a value that contains the sum of all of the “CountOfDow” entries. Just so you have a reference as we go forward, the number I got from all my dummy data is 44.

upquery = "SELECT * FROM weekdays"
c1 = cursor.execute(upquery)

Here we have done a 'SELECT all' query so every record in the datatable is in the ‘c1’ cursor. We’ll walk through each row of the dataset, pulling the pkgs (row[0]) and CountOfDow (row[1]) data into variables.

LastUpper = .0
for row in c1:
    cod = row[1]
pkg = row[0]

Now we will create a probability of each daily package count in the database and calculate an upper and lower value that will be used in another process later on. Notice that we check to see if the LastUpper variable contains ‘.0’. If it does, we set it to the probability value, otherwise we set it to the lower plus the probability value.

Finally we use an update SQL statement to put the new computed values into the database.

What we end up with is a package count (pkgs), a count of the number of days that package count came in, a probability of that occurring within the whole of the study (31 packages on 1 day out of a total of 44 (weekdays in that 60+ day study), will have a probability of 0.02.).

If we add up all the probability values in the table it should add up to 1.0.

sumquery = "SELECT Sum(CountOfDOW) as Num FROM weekdays"
tmp = cursor.execute(sumquery)
for t in tmp:
    DaySum = t[0]
    prob = cod / float(DaySum)
    if LastUpper != .0:
        lower = LastUpper
        LastUpper = (lower + prob)
    else:
        lower = .0
        LastUpper = prob

query = 'UPDATE weekdays SET probability = %f, \
        lower = %f, upper = %f WHERE pkgs = %d' \
        % (prob,lower,LastUpper,pkg)
u = cursor2.execute(nquery)
#=================================================================================
# End of DoWeekDays
#=================================================================================

The upper and lower values then reflect a number between floating point number 0 and 1 that will mirror the possibility of any random number within that range that will give us a randomized number of packages. This number can then be used for a statistics analysis of this data. A “normal real-world” example would be to predict the number of cars that arrive at a carwash based on observational data done in the field. If you want to understand more, you could look at

http://www.algebra.com/algebra/homework/Probability-and-statistics/Probability-and-statistics.faq.question.309110.html to see an example of this. All we did is generate (the hard part) easily with Python.

The code for the two routines that we presented this time is at:
http://pastebin.com/kMc9EXes

Until next time.
This month, I thought I would create a routine that makes a license key from an email. We all know the reason for having a license key, and if you ever need to have a quick and dirty set of routines to do it, you can use this. Remember, Python is a scripting language, so the source is always readable. There are ways around this; we'll discuss them in another article. Let's take a look at the "gross" logic behind the code, before we actually dive into the code.

First, we will ask for an email address and then break it into two parts, the local part (the part before the "@" character) and the domain part (the part after the "@" character). There are very specific rules for email address validity, and it can get very complicated. For our purposes, we will only use some of the rules and only on the local part. You can do a web search on the actual rule set. In our code, we will only look at:

- lowercase characters
- upper case characters
- numbers between 0 and 9
- special characters (!#$%&'()*+,-/=?^`{|}~.)
- period characters are allowed, but may not be repeated next to each other (..., etc)

Once we have validated the email, we then will create a "checksum character" which is based on the ascii value of each character in the entire email address, and then divide it by the number of characters in the email address. For example, let's use a mythical email address of fredjones@someplace.com. If we walk through the email address, we can get the ascii value of each character by using the ord() function. When we add up each of the ascii values, we get a sum of 1670, then we divide that by the length of the email address (23); we get 72. Remember we are using integer division here, so our result will be an integer.

Now that we have our checksum value, we subtract 68 from that (ascii ‘D’) to create an offset. We use this offset when we encode each character in the email. Just to make things a bit harder to decode, we put the length (with offset) as character position 2 and the checksum as character position 4.

So for the email fredjones@someplace.com we get a license key of:

\[
\text{jvHihnsriwDwsqitpegi2gsq}
\]

Let's get started with the code. Since this is the 53rd article in the series, I won't be quite as verbose from here on out.

First our imports.

```python
import sys
```

Now (as shown above right) we will create a string that will include all of our "legal" characters for the IsValidEmail function. I've split it into 3 strings so it fits nicely for the magazine. We combine them in the IsValidEmail routine. We also set a global variable 'Offset' to 0. This will be the value that we add (later on) to each character when we create the encoded string.

```python
def IsValidEmail(s,debug=0):
    email = s
    pos = email.rfind("@")
    local = email[:pos]
    domain = email[pos+1:]
    if debug == 1:
        print local
        print domain
    isgood = False
    localvalid = localvalid1 + localvalid2 + localvalid3
```
Now for our first function. This (below) is the IsValidEmail routine. Basically we pass the email in the variable s, and an optional debug flag. We use the debug flag, as we have done in the past, to provide some print statements to see how things are going. Usually we would simply pass a 1 as the second parameter if we want to see the progress verbosely.

First we assign the passed in email address to the variable ‘email’ and find the ‘@’ character that separates the local from the domain portions of the email. We then assign the local portion of the email to (I think it’s appropriate) ‘local’, and the domain portion to ‘domain’. We then set the boolean isgood flag to False and finally create the ‘localvalid’ string from the 3 shorter strings we set up earlier.

Next (top right) we simply walk through each character in the local portion of the email against the list of valid characters using the in keyword. If any character in the local portion of the email fails the test, we break out of the for loop, setting the ‘isgood’ flag to False.

Finally, we look for any set of period characters that are contiguous. We use the string.find routine that will match anything that is like ‘..’ or ‘...’ and so on. Being a lazy programmer, I used only a single “double dot” check that works for anything more.

```python
r = email.find("..")
if r > -1:
    isgood = False
```

The last thing we do in the routine is return the value of the ‘isgood’ flag.

```
return isgood
```

Now for the EncodeKey routine. While it looks simple, it requires some concentration so pay attention! We assign the Offset variable to global status so we can change it within the function and so it can be used in other functions. We then set the Offset variable to the checksum minus 68. As in the example presented at the beginning of the article, it would be 72-68 which equals 4. We then step through each character of the email address adding the offset to the ascii value of that character. For the ‘f’ in ‘fredjones’, it would be 102 + 4 or 106 which equates to ‘i’. Using the counter variable ‘cntr’, we then determine what we add to the ‘NewEmail’ string we build up character by character. Notice in the code that we go from 0 to the length of the email, so character 0 is ‘f’, character 1 is ‘r’ and so on. Now comes the
The Dolt function (below) asks for an email address using ‘raw_input’, then calls the functions in order to create the license key.

Lastly, we call the Dolt routine.
if __name__ == "__main__":
  DoIt()

Now, obviously the output is not super-encrypted, and if someone were to put in a fair amount of time, they could figure out what we used to create the key fairly easily. However, it should give you enough of a starting point that you could simply modify the code to make it much harder to break. You could, for example, use a random number rather than the ‘D’ (68). If you do that, set a seed in the code so that it will always generate the same random number. You could also go a bit deeper and put the offset value somewhere into

The DecodeKey routine (bottom right) basically reverses the process we used in the EncodeKey routine. One thing you might notice here is that in the first ‘if debug’ statement of this function, I used ‘!= 0’ rather than ‘== 1’, simply to remind you that the two can be interchangeable.

The decodeKey function below takes the license key, or whatever you get from the server or from the email, and converts it back to the email address. You can use it with the MAC address to create the license key, maybe the last character so you could use that as the decryption offset.

As always, the full source is available at http://pastebin.com/MH9nVTKN. Until next time, enjoy.
Many years ago, I was dealing with high blood pressure issues. My doctor suggested that I do something that allowed me to concentrate on something fairly useful, but rather trivial. I dealt with it by trying to do counted cross stitch. It’s creative, focused, and keeps your mind occupied on what you are doing, not what is bothering you. I find myself in that position again, so I broke out the hoop and needles and started again.

In case you aren’t familiar with counted cross stitch, I’ll give you a gross overview of what it is. Cross stitch is a type of needlework that uses tiny ‘x’ patterns of thread that eventually make up a picture. The thread is called “floss” and the fabric that you use is called “aida”. According to Wikipedia, aida is a special fabric that has tiny squares that have small holes at regular intervals that form the squares. This facilitates the placement of the “x” patterns that make the image. There are two types of cross stitch. One has an image printed on the aida (sort of like paint by numbers), and the other uses totally blank aida that you count stitches from the pattern. The second is much harder than the first. Go to your favorite fabric store or craft section of your local mega-mart and you’ll get the idea.

Also a while back, I started playing with creating a program that would take an image and convert it into a cross stitch pattern. One thing lead to another, and I had to shelve the program for other things. I’ve now dusted off the idea and started anew.

We will spend the next few articles dealing with this project. It will take a while, since some things are fairly complex and have many parts to them. Here is the “game plan”:

- Create a database for the pixel colors to floss colors.
- Create a GUI using Tkinter for the application.
- Flesh out the application to do the manipulation of the image files.
- Create a PDF file that will be the ultimate pattern for the project.

What you will learn
- Revisitation of database and XML manipulation.
- Revisitation of Tkinter GUI programming. If you missed the previous articles on this, please refer to FCM issues 51 thru 54.
- PDF creation using pyFPDF (https://code.google.com/p/pyfpdf/).

GETTING STARTED

The first thing in our list of tasks is to create the database that will hold the DMC(™) floss colors and reference them to the closest approximation to the RGB (Red, Green, Blue) values that are used in images on the computer. At the same time, the database will hold the hex value and the HSV (Hue, Saturation, Value) representations for each floss color. It seems that HSV is the easiest way to find the “closest” representation of a color that will match the floss colors. Of course, the human eye is the ultimate decision maker. If you are not familiar with HSV color representations, there is a rather complex writeup on Wikipedia at http://en.wikipedia.org/wiki/HSL_and_HSV. It might help, but it might make things less clear.

The first thing we need is an XML file that has the DMC floss
colors with a RGB conversion. The best one I found is at
http://sourceforge.net/p/kxstitch/feature-requests/9/. The file
you want is dmc.xml. Download it and put it in a folder that you
will use to hold the Python
code.

Now we will be using apsw (below) to do our database
manipulation, which you should already have and ElementTree
do the XML parsing (which is
included in Python version
2.7+).

As always, we start with our
imports. In this program, we
have only the two. We also set
the name of the table.

The next portion should be
familiar if you have been
reading the articles for a while.
We create a function that will
read the XML file, and parse it
for us. We then can use the
information to load the
database. A snippet of the XML
file is shown top right.

We are looking for the
<floss> tag for each line of
information. To do this, we use
the .findall(‘floss’) command.
Once we have the information
line, we break each tag (name, description, etc.) into separate
variables to place into the
database. When it comes to the
<fcolor> tag, we use the
.floss.findall(‘color’) command
to get each value of Red, Green
and Blue.

We start by telling the
function that we will be using
the global variables connection
and cursor. We then set the
filename of the XML file, parse
the XML file, and get started.
We also use a counter variable
to show that something is
happening while the parsing
and database inserts are going
on.

Now that we have all our
data, we need to create the
SQL insert statement and
execute it. Notice the “\” after
the word VALUES in the SQL
statement. That is a line-
continuation character to make
it easier for printing here in the
magazine. We will be creating
the database and table in a few
moments.

SQL = "INSERT INTO DMC
(DMC, Description, Red, Green, Blue)
VALUES \
<floss>
  <name>150</name>
  <description>Dusty Rose Ultra VDK</description>
  <color>
    <red>171</red>
    <green>2</green>
    <blue>73</blue>
  </color>
</floss>
def ReadXML():
global connection
global cursor
fn = 'dmc.xml'
tree = ET.parse(fn)
root = tree.getroot()
cntr = 0
for floss in root.findall(‘floss’):
  name = floss.find(‘name’).text
desc = floss.find(‘description’).text
for colour in floss.findall(‘color’):
  red = colour.find(‘red’).text
green = colour.find(‘green’).text
def OpenDB():
global connection
global cursor
global ucursor
global dbname
connection = apsw.Connection("floss.db3")
cursor = connection.cursor()

('%s', '%s', '%s', '%s', '%s') %
(name, desc, red, green, blue)
cursor.execute(SQL)

print "Working record
{0}".format(cntr)
cntr += 1

Now we print to the
terminal window that
something is going on:

Now we create and/or open
the database in the OpenDB
routine (bottom right). If you’ve

full circle magazine 241  The Compleat Python
been with us when we have done database work before, you will notice that we are using two cursors this time. The cursor variable is used for the “normal” inserts, and later on in the select statement for the update to set the hex and HSV values. We have to use two cursors, since if you modify a cursor in the middle of a logic statement, you lose everything with the new command. By using ‘ucursor’, we can use that for the update statements. Other than that, it is our normal OpenDB routine.

Now that the database is created and/or opened, we can set up our table (top right). Notice that the SQL statement below uses the triple quote to allow for the line to break neatly for viewing.

The EmptyTables routine (middle right) is there just to make sure that if we want to or need to run the application more than once, we start with a clean and empty table if it exists.

If we were to stop here, we would have a reasonable working database with the DMC color, color name and the RGB values associated with each. However, as I alluded to before, it is easier to pick the closest floss color by using the HSV data.

We next create the hex value from the RGB values (middle left).

The next function creates the HSV values from the RGB values. I found the algorithm on the internet. You can research it there.

Finally, we create the UpdateDB function (next page, top left). We use the SELECT * FROM DMC command and use the “standard” cursor variable to hold the data. We then step through the returned data, and read the RGB values, and pass them to the rgb2hex function as a tuple and to the rgb2hsv function as three separate values. Once we get the return values, we use the update SQL command to match the proper record by using the primary key (pkID). As I stated before, we have to use a separate cursor for the update statement.

The last thing we do is call each of the functions in order to create the database, and, at the end, we print “Finished” so the user knows everything is done.
print "Finished"

I named this program “MakeDB”. The database should be created in the same folder where the code and XML file are located. As always, the full code can be found at http://pastebin.com/Zegqw3pi.

Next time, we will work on the GUI. We use Tkinter for the GUI, so, in the meantime, you might want to refresh your memory by looking at FCM issues 51 thru 54 where I take you through Tkinter.

Until next time, have a good month.
This is the second in a multi-part tutorial on creating a Cross Stitch pattern generator. In the first part (FCM85), we created a database containing the DMC™ floss colors with their closest RGB values. In this part, we will create the GUI using Tkinter. We will also use PIL (Python Imaging Library) and PMW (Python Mega Widgets). You’ll need to download those libraries and install them before we go too far. For PIL, go to the Pillow fork at https://github.com/python-imaging/Pillow and download the latest version. For PMW, go to http://pmw.sourceforge.net/ and download from there.

You will also need two image files. One is a simple grey rectangle 500x400 pixels. You can use GIMP or some other image manipulating program to create it. Name it default.jpg, and place it into your source code directory along with the database. The other is an image of a folder for the open image button. I got one from open clipart and searched for the word “folder”. I found a reasonable one at https://openclipart.org/detail/177890/file-folder-by-thebyteman-177890. Open it in GIMP, resize it to 30x30 and save it in the same directory as the other two files as “open.gif”.

Above is a screenshot of what the finished GUI will look like. There are four main frames in the GUI. Three on the left side and one on the right. When we go through the build widget process, I refer to them as Top Frame, Middle Frame, Bottom Frame and Side Frame. The top frame deals with the original image. The middle frame deals with the processing of the image. The bottom frame shows the original image on the left and the processed image on the right, and the side frame displays the colors and floss required. It seems from first glance there is a lot of wasted space here, but when you see the program run, it doesn’t really have that much empty space, once we get through the processing portion.

Now we are ready to start working on the code. Here is our long list of imports...

```python
from Tkinter import *
import tkFileDialog
import tkCommonDialog
import tkMessageBox
import ttk
from PIL import Image, ImageTk, ImageOps
import Pmw
import apsw # Database Access
import math # Math library
import sys
```

From the sheer number of imports, you can tell this is going to be a long program. In fact, the UI portion of the code will be over 300 lines, including comments. The “good” news is that about 200 of the lines of code deal with the Tkinter
portion of the program, the actual GUI itself. The majority of the remaining lines of code in this portion are stubs for functions needed for the next article.

We’ll create a class to hold all of our UI processing code (next page, top right).

First, we have the class definition and next we have the __init__ function which we pass the TopLevel “root” window into. We create the TopLevel window in the last four lines of the program. Within the __init__ function we are defining all the global variables and doing some initial assignments before we start the other functions. The first thing we do is create a list of Tuples that hold the picture file formats that we need when we call the OpenFile dialog. The next two lines below, define and ready the two image files we just created (open folder GIF file, and the grey rectangle – which will be used as placeholders for our images used to create the pattern.

As you can see, we are setting a variable called OriginalFilename, which holds the image that we want to create the pattern from, OriginalColorCount which holds the number of colors in the original image file, and OriginalSize which holds the size in pixels of the original image. As they say on tv... “BUT WAIT! THERE’S MORE!” (bottom right):

The ComboStitch variable is set by a combobox, and handles the stitch size of the aida that you wish to use for your project. The ComboSize variable is also set by a combobox and holds the size of the
aida fabric. FabricHeight and FabricWidth are the breakdowns from the aida size. MaxColors is a value from an entry box to set the number of colors, and BorderSize is a floating point value that specifies the amount of unused aida for framing.

global ProcessedColors
ProcessedColors = StringVar()
global ProcessedImage
ProcessedImage = ""
global GridImage
GridImage = ""
global backgroundColor1
backgroundColor1 = (120,)*3
global backgroundColor2
backgroundColor2 = (0,)*3
global ReadyToProcess

Finally we have assigned all our globals, and now have the code that actually creates the GUI. We open the database, create the menu, set up the widgets, and finally place the widgets into the proper places. Just to give you a heads-up, we will be using the Grid geometry placement manager. More on that later.

global ShowGrid
ShowGrid = True
global ProcessedImage
ProcessedImage = ""
global GridImage
GridImage = ""
global backgroundColor1
backgroundColor1 = (120,)*3
global backgroundColor2
backgroundColor2 = (0,)*3
global ReadyToProcess
#----------------------------------------------
# BEGIN UI DEFINITION
#----------------------------------------------
def MakeMenu(self,master):
    menu = Menu(master)
    root.config(menu=menu)
    filemenu = Menu(menu, tearoff=0)
    process = Menu(menu,tearoff=0)
    #----------------------------------------------
    # File Menu
    #----------------------------------------------
    filemenu.add_command(label="New")
    filemenu.add_command(label="Open", command=self.GetFileName)
    filemenu.add_command(label="Save", command=self.FileSave)
    filemenu.add_separator()
    filemenu.add_command(label="Exit", command=self.DoExit)
    #----------------------------------------------
    #----------------------------------------------
    self.OpenDB()
self.MakeMenu(master)
frm = self.BuildWidgets(master)
self.PlaceWidgets(frm)

The next portion of our code (middle right) will set up the menu bar. I’ve tried to lay it out logically so it will be easy to understand.

We define a function called MakeMenu, and pass in the TopLevel window. We then define the three menu sets we will be creating. One for File, one for Process, and one for Help.

menu.add_cascade(label="File", menu=filemenu)
menu.add_cascade(label="Process",menu=process)
menu.add
Now we set up the File menu options (bottom right). Open will open our image and uses a function called “GetFileName”. Save will create the output PDF file and uses the FileSave function. We add a separator and finally an Exit function.

Now we have the Process option and the Help functions (next page, top right).

All of the options in the menu bar are also available from various buttons within the program.

Now we will make our BuildWidgets function. This is where we create all the widgets that will be used on the GUI.

def BuildWidgets(self, master):

    self.frame = Frame(master, width=900, height=850)

    We start with the function (bottom right) definition, passing in the TopLevel window (master) and placing a frame that holds all of our other widgets. I've added comments to help realize which part of code deals with which frame. We'll deal with the top frame first.

    Assuming you remember or refreshed your memory on Tkinter, it should be fairly straight-forward. Let's look at the first label as a discussion item.

    self.label1 = Label(self.frm1, text = "Original Filename: ")

    First, we define the name of the widget (self.label1 =). Next we set that variable to which widget type we want to use; in this case Label. Finally we set the parameters we want to apply to that widget starting with the parent widget (self.frm1), and in this case, the text that will show up in the label. Now let's take a moment to look at the button

    self.btnGetFN = Button(self.frm1, width=28, image=self.openimage,

    # --------------- MIDDLE FRAME ---------------
    self.frm2 = Frame(self.frame, width=900, height=160, bd=4, relief=GROOVE)
    self.lb14 = Label(self.frm2, text="Aida Stitch Size: ")
    self.lb15 = Label(self.frm2, text="Aida Fabric Size: ")
    self.TCombobox1 = ttk.Combobox(self.frm2, textvariable=CombiStitch, width=8)
    self.TCombobox1.bind('<<ComboboxSelected>>', self.StitchSizeSelect)
    self.TCombobox1['values'] = (7,10,11,12,14,16,18,22)
    self.TCombobox2 = ttk.Combobox(self.frm2,textvariable=ComboSize,width = 8)
    self.TCombobox2.bind('<<ComboboxSelected>>',self.AidaSizeSelect)
    self.TCombobox2['values'] = ("12x18","15x18","30")
First thing to notice is that this is broken into two lines. You can safely place everything on one line...it is just too long to fit into a 72-character line. We'll really pay attention to the parameters we use here. First the parent (frm1), next the width which is set at 28. When we use a widget that has the option of text or an image, we have to be careful setting the width. If it will contain text, the width parameter is the number of characters it will hold. If it is to display an image, it will be set at the number of pixels. Finally we set the command parameter, which tells the system what function to call when the button is clicked.

One more thing to look at is the textvariable parameter. This tells us what variable will hold the information that will be displayed in the widget. We set these in the __init__ function earlier. One other thing to mention is that the frame itself has two parameters you might not remember. The Relief parameter sets the border type of the frame, which in this case is GROOVE, and the bd parameter sets the border width. Border width defaults at 0 so if you want to see the effect, you have to set the border width (bd is a shortcut).

Now we'll deal with the middle frame widgets.

The last 6 lines of this section (previous page, middle right) deal with the two combo boxes in the UI. Each combo box uses three lines (the way I programmed it to make it easy to understand). In the first line, we set the basic parameters. The next line, we bind the combobox selection-changed event to the function.
StitchSizeSelect, and the last line has a list of the values that
will be available for the pulldown.

Everything else above is pretty “normal” stuff. Now we
set our defaults for the widgets that need them. Again, we are
using the global variables that we set up in the _init_
function and wrapped to the widget variable class.

Combostitch.set(14)
ComboSize.set("15x18")
FabricWidth.set(15)
FabricHeight.set(18)
MaxColors.set(50)
Bordersize.set(1.0)

Now we deal with the bottom frame. This is really
simple, since we have to set up only the frame and two labels
which we will use to hold our images.

Finally we deal with the side frame. The side frame will hold
a ScrolledFrame from the PMW library. It’s really easy to use
and provides a nice interface to

<table>
<thead>
<tr>
<th>ROW</th>
<th>Col 0</th>
<th>Col 1 - Col 6</th>
<th>Col 7</th>
<th>Col 9</th>
<th>Col 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Label1</td>
<td>entFileName</td>
<td>btnGenFN</td>
<td>Label2</td>
<td>lblOriginalColorCount</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>Label3</td>
<td>lblOriginalSize</td>
<td></td>
</tr>
</tbody>
</table>

def PlaceWidgets(self,frame):
def PlaceWidgets(self,frame):
    frame.grid(column = 0, row = 0)
    # --------------- TOP FRAME -----------------
    self.frm1.grid(column=0,row=0,rowspan=2,sticky="new")
    self.label1.grid(column=0,row=0,sticky='w')
    self.entFileName.grid(column=1,row=0,sticky='w',columnspan = 5)
    self.btnGetFN.grid(grid(column=7,row = 0,sticky='w')
    self.label2.grid(column=9,row=0,sticky='w',padx=10)
    self.lblOriginalColorCount.grid(column=10,row=0,sticky='w')
    self.label3.grid(column=9,row=1,sticky='w',padx=10,pady=5)

    # --------------- MIDDLE FRAME -----------------
    self.frm1.grid(column=0,row=2,rowspan=2,sticky="new")
    self.lb14.grid(column=0,row=0,sticky="new",pady=5)
    self.lb15.grid(column=0,row=1,sticky="new")
    self.TCombox1.grid(column=1,row=0,sticky="new",pady=5)
    self.TCombox2.grid(column=1,row=1,sticky="new")
    self.lb16.grid(column=2,row = 0,sticky="new",padx=5,pady=5)
    self.entMaxColors.grid(column=3,row=0,sticky="new",pady=5)
    self.lb17.grid(column=2,row=1,sticky='new',padx=5)
    self.entBordersize.grid(column=3,row=1,sticky='new')
    self.frmLine.grid(column=4,row=0,rowspan=2,sticky='new',padx=15
    self.lb18.grid(column=5,row=0,sticky='new',pady=5)
    self.lb19.grid(column=5,row=1,sticky='new')
    self.lbProcessedColors.grid(column=6,row=0,sticky='w')
    self.lbProcessedSize.grid(column=6,row=1,sticky='w')
    self.btnDoIt.grid(column=7,row=0,sticky='e',padx=5,pady = 5)
    self.btnShowGrid.grid(column=7,row=1,sticky='e',padx=5,pady = 5)

    # --------------- BOTTOM FRAME -----------------
    self.frm3.grid(column=0,row=4,sticky="nsw")
    self.lb1ImageL.grid(column=0,row=0,sticky="w")
    self.lb1ImageR.grid(column=1,row=0,sticky="e")
That’s all for the widgets. Now we have to place them. As I said earlier, we will be using the Grid geometry manager, rather than the absolute or pack managers.

The Grid method places the widgets in (you guessed it) a grid, referenced by row and column designations. I’ll use the top frame as an example (shown top right).

First we place the frame.

You can see that we place the widget by using the {widgetname}.grid command, then the row and column positions. Notice that we are telling the entry widget to span 5 columns. Padx and pady values will place some extra space on both the right and left sides (padx) or the top and bottom (pady). The sticky parameter is similar to a justify command for text.

The middle frame is a bit more complicated, but basically the same as the top frame. You might notice an extra frame in the middle of the code (self.frmLine). This gives us a nice divider between the options section and the display section. Since there is no horizontal or vertical line widget, I cheated and used a frame with a width of 6 pixels and border width of 3, making it just look like a fat line.

The bottom frame is simple since we have only the frame and the two labels to hold the images.

The side frame is pretty much the same thing, except the ScrolledFrame allows for a frame to be set to the interior of the scrolled frame widget. We then create three widgets here and place them in their grids as column headers. We do this since we assigned the interior frame for the scroll frame here and we have to assign the parent (self.sfFrame) after we have created it.

That’s all the hard work for now. At this point, we will create all of the functions that we need to get the GUI to run, stubbing most of them until next month. There are a few we will go ahead and complete, but they are fairly short.

The first function will be the Exit option from the menu bar. It’s under the File menu option.

def Thumbnail(self, file, hsize, wsize):
    size = hsize, wsize
    extpos = file.rfind(".")
    outfile = file[:extpos] + ".thumbnail"
    im = Image.open(file)
    im.thumbnail(size)
    im.save(outfile, "JPEG")

# ----------------- SIDE FRAME -----------------
self.frm4.grid(column=2, row=0, rowspan=12, sticky="new")
self.sf.grid(column=0, row=1)
self.sfFrame = self.sf.interior()
self.lbich1 = Label(self.sfFrame, text="Original")
self.lbich2 = Label(self.sfFrame, text="DMC")
self.lbich3 = Label(self.sfFrame, text="Name/Number")
self.lbich1.grid(column=0, row=0, sticky="w")
self.lbich2.grid(column=1, row=0, sticky="w")
self.lbich3.grid(column=2, row=0, sticky="w")

def DoExit(self):
    sys.exit()

The only other one is the Thumbnail function. We need this to fill the grey rectangles into the labels in the bottom frame. We pass the filename and the width and height that we want the thumbnail image to be.

def ShowHelp(self):, def ShowAbout(self):, def OpenDB(self):, def ShowHideGrid(self):
    def StitchSizeSelect(self, p):
        def AidaSizeSelect(self, p):
        def Process(self):
            def CreatePDF(self):
                def OriginalInfo(self, file):
                    def GetColorCount(self, file):
                        def GetHW(self, file):
                            def GetHW2(self, file):
                                def GetColors(self, image):
                                    def Pixelate(self, im, pixelSize):
                                        def ReduceColours(self, im, imageName):
                                            def MakeLines(self, im, pixelSize):
                                                def MakeLines2(self, im, pixelSize):
                                                    def Rgb2Hex(self, rgb):
                                                        def FillScrolledList(self, filename):
                                                            def GetBestDistance(self, rl, gl, bl):
                                                            # ----------------- SIDE FRAME -----------------
                                                            self.frm4.grid(column=2, row=0, rowspan=12, sticky="new")
                                                            self.sf.grid(column=0, row=1)
                                                            self.sfFrame = self.sf.interior()
                                                            self.lbich1 = Label(self.sfFrame, text="Original")
                                                            self.lbich2 = Label(self.sfFrame, text="DMC")
                                                            self.lbich3 = Label(self.sfFrame, text="Name/Number")
                                                            self.lbich1.grid(column=0, row=0, sticky="w")
                                                            self.lbich2.grid(column=1, row=0, sticky="w")
                                                            self.lbich3.grid(column=2, row=0, sticky="w")
                                                            def Thumbnail(self, file, hsize, wsize):
                                                                size = hsize, wsize
                                                                extpos = file.rfind(".")
                                                                outfile = file[:extpos] + ".thumbnail"
                                                                im = Image.open(file)
                                                                im.thumbnail(size)
                                                                im.save(outfile, "JPEG")
                                                            def DoExit(self):
                                                                sys.exit()
Since this article is so long, I’m going to give you a list of function names and all you have to do is stub it out by using the pass command. We’ll fill them in next month. I’ll give you the first one as an example, but you should already know how to do it.

```python
def GetFileName(self):
    pass
```

For the rest of the functions, I’ll just give you the def lines. Be sure to include them all in your code.

You can see, we have a large amount of work to do next month. We still have four more lines to write to finish up for this month. This is out of our class code.

```python
root = Tk()
root.title("Cross Stitch Pattern Creator")
test = XStitch(root)
root.mainloop()
```

The first line sets up the root TopLevel window. The next line sets the title on the top line. The third line instantiates our XStitch class, and the last line starts the main loop that shows the UI and gives control over to it.

Well that’s a lot for this month, but we are finally done. You can actually run the program to see the GUI.

As always, the code is available on Pastebin at http://pastebin.com/XtBawJps.

Next month we will flesh out the code. See you then.
We've been working on a Cross Stitch pattern generator. Last month we did the UI portion, and now it's time to do the code that does the most of the work. Next month we will start working on the PDF file output portion.

We'll work on the menu items first. The code is shown below.

The global ReadyToProcess variable is used to make sure that if the user presses the Process button, the system doesn't try to process things without anything to process. We use the tkFileDialog askopenfilename built-in dialog routine to get the filename of the original image. We then get the number of colors in the original image as well as the width and height. We save those values and display them in the GUI. We then open the image and create a thumbnail image to display in the bottom frame. See the text box to the right.

Next we do the ShowHideGrid function. This simply exchanges two images in the right image label based on the global variable ShowGrid. If False, we change the text on the show/hide button, then set the ShowGrid variable to True and set the image to the one with the grid. Otherwise we change the text on the show/hide button to “Show Grid”, set the ShowGrid variable to False and put up the ungridded image. Code is on the next page, top left.

The StitchSizeSelect function is fired whenever the stitch size combobox is changed. We get

```python
def GetFileName(self):
    global ReadyToProcess
    #---------------------------
    fileName = tkFileDialog.askopenfilename(parent=root,filetypes=self.picFormats ,title="Select File to open...")
```
def ShowHideGrid(self):
global ShowGrid
#------------------------------------
if ShowGrid == False:
    self.btnShowGrid['text'] = 'Hide Grid'
    ShowGrid = True
    self.im2 = Image.open(self.GridImage)
    self.im2.thumbnail((400, 400))
    self.img3 = ImageTk.PhotoImage(self.im2)
    self.lblImageR['image'] = self.img3
else:
    self.btnShowGrid['text'] = 'Show Grid'
    ShowGrid = False
    self.im2 = Image.open(self.ProcessedImage)
    self.im2.thumbnail((400, 400))
    self.img3 = ImageTk.PhotoImage(self.im2)

the value from the combo box and assign it to a local variable.

def StitchSizeSelect(self,p):
    selection = ComboStitch.get()

The AidaSizeSelect function (top right) is very similar to the StitchSizeSelect function. We set the FabricWidth and FabricHeight globals based on

the selection on the combo box. We also default to 30x30 if they select 30.

We have a variable called ReadyToProcess (below) just in case the user tries to run the process function before the image is loaded.

We pixelate the original file to a 5x5 pixel matrix. This

allows us to group that 5x5 matrix to a single color. We then reduce the colors, get the width and height of the processed image and set the size so the user can see how big the resulting image will be.

# Place image
self.im2 = Image.open(Reduced)
self.im2.thumbnail((400, 400))
self.img3 = ImageTk.PhotoImage(self.im2)

The above set of code places the processed image into the image that will hold the processed image. The next set of code will create a grid so that the user will have the grid to do the cross stitching.

self.MakeLines(Reduced, 5)
self.MakeLines2('output.png', 50)
**HOWTO - PYTHON PART 56**

```
def Pixelate(self, im, pixelSize):
    image = Image.open(im)
    self.GetColors(image)
    image = image.resize((image.size[0]/pixelSize, image.size[1]/pixelSize), Image.NEAREST)
    image = image.resize((image.size[0]*pixelSize, image.size[1]*pixelSize), Image.NEAREST)
    self.GetColors(image)
    #image.show()
    image.save('newimage.png')
```

The `GetColorCount` function uses the `.getCount` method to get the number of colors in the image file. We have to use 1600000 as the maxcolors parameter because if the image contains more than 256 colors (or whatever is in the parameter, the method returns 'None'. This function is similar to the `GetColors` function except the `GetColors` works with an already opened image file. If you use `GetColorCount`, you have to pass an unopened file.

```
def GetColorCount(self, file):
    im = Image.open(file)
    numColors = im.getcolors(1600000)
    self.colors = len(numColors)
    return self.colors
```

The next two functions return the height and width of the image file in pixels. The difference between the two is that `GetHW` returns a string like 1024x768 and `GetHW2` returns two integers.

```
def GetHW(self, file):
    im = Image.open(file)
    tmp = '{0}x{1}'.format(im.size[0], im.size[1])
    return tmp
```

```
def GetHW2(self, file):
    im = Image.open(file)
    return im.size[0], im.size[1]
```

`GetColors` will get the number of colors in the passed image file. We use 1.6 million colors as the parameter, because the `image.getcolors()` routine defaults to 0 over color count over 256.

```
def GetColors(self, image):
    numColors = image.getcolors(1600000)
    colors = len(numColors)
```

```
def ReduceColors(self, ImageName):
    #Reduce colors
    numcolors=MaxColors.get()
    image = Image.open(ImageName)
    output = image.convert('P', palette=Image.ADAPTIVE, colors=numcolors)
    x = output.convert('RGB')
    self.GetColors(x)
    numcolors = x.getcolors()
    ProcessedColors.set(len(numcolors))
    x.save('im1.png')
```
The Pixelate function (above) takes two parameters, image filename (im) and the size of pixels you want. The work is done by the image.resize method. I found this routine on the web in a number of places. In this instance we will be passing a pixel size of 5, which works well for Cross Stitch projects. We also tell the method to take the color of the nearest neighbor. This returns a new image, which we save as a file and return the filename.

The ReduceColors routine (below) basically uses the Image.ADAPTIVE pallet so we can get a much smaller number of colors.

There are two MakeLines (top right) routines. They create the grid we spoke of earlier.

Rgb2Hex() returns a hex value of the RGB value that is passed in. We will use this to try to compare the colors in the database with the colors in the image.

```python
def FillScrolledList(self, filename):
    im = Image.open(filename)
    numColors = im.getcolors()
    colors = len(numColors)
    cntr = 1
    for c in numColors:
        hexcolor = self.Rgb2Hex(c[1])
        lblColor=Label(self.sfFrame,text='"',bg=hexcolor,relief=GROOVE)
        lblColor.grid(row = cntr, column = 0, sticky = 'nsew',padx=10,pady=5)
        pkID = self.GetBestDistance(c[1][0],c[1][1],c[1][2])
        sql = "SELECT * FROM DMC WHERE pkID = {0}".format(pkID)
        rset = cursor.execute(sql)
        for r in rset:
            hexcolor2 = r[6]
            dmcnum = r[1]
            colorname = r[2]
            lblColor2=Label(self.sfFrame,text='"',bg="#" + hexcolor2,relief=GROOVE)
            lblColor2.grid(row = cntr,column = 1,sticky = 'w',padx=5,pady=5)
            lblColor3=Label(self.sfFrame,text = str(dmcnum) + "-" + colorname,justify=LEFT)
            DmcColor.set(dmcnum)
```

```python
def MakeLines(self,im,pixelSize):
    global bgColor1
    #--------------------------
    image = Image.open(im)
pixel = image.load()
for i in range(0,image.size[0],pixelSize):
    for j in range(0,image.size[1],pixelSize):
        for r in range(pixelSize):
            pixel[i+r,j] = bgColor1
            pixel[i,j+r] = bgColor1
    image.save('output.png')
```

```python
def MakeLines2(self,im,pixelSize):
    global bgColor2
    #--------------------------
    image = Image.open(im)
pixel = image.load()
for i in range(0,image.size[0],pixelSize):
    for j in range(0,image.size[1],pixelSize):
        for r in range(pixelSize):
            try:
                pixel[i+r,j] = bgColor2
                pixel[i,j+r] = bgColor2
            except:
```

```python
```
def GetBestDistance(self,r1,g1,b1):
    # dist = math.sqrt(((r1-r2)**2) + ((g1-g2)**2) + ((b1-b2)**2))
    sql = "SELECT * FROM DMC"
    rset = cursor.execute(sql)
    BestDist = 100000.0
    for r in rset:
        pkID = r[0]
        r2 = r[3]
        g2 = r[4]
        b2 = r[5]
        dist = math.sqrt(((r1-r2)**2) + ((g1-g2)**2) + ((b1-b2)**2))
        if dist < BestDist:
            BestDist = dist
            BestpKID = pkID

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

As always, the code is available on PasteBin at http://pastebin.com/DmQ1GeUX.

We will continue in the next month or so. I’m facing some surgery soon so I’m not sure how soon I will be able to sit for any long periods of time. Until then, enjoy.

The ScrollList (below) on the right side holds the colors that will be used to get the proper floss colors. We simply create labels to hold the colors (visual) and text.

This (next page) is the routine that we use to try to find the closest match between the color in the image and the color in the database. There are many different algorithms on the web that you can look at and try to understand the logic behind it. It gets rather complicated.

Ok. That’s all for this month. Next time, we will start creating the PDF output file so the cross stitcher has something to work with.

The ScrollList (below) on the right side holds the colors that will be used to get the proper floss colors. We simply create labels to hold the colors (visual) and text.

This (next page) is the routine that we use to try to find the closest match between the color in the image and the color in the database. There are many different algorithms on the web that you can look at and try to understand the logic behind it. It gets rather complicated.

Ok. That’s all for this month. Next time, we will start creating the PDF output file so the cross stitcher has something to work with.

As always, the code is available on PasteBin at http://pastebin.com/DmQ1GeUX.

We will continue in the next month or so. I’m facing some surgery soon so I’m not sure how soon I will be able to sit for any long periods of time. Until then, enjoy.
Program In Python Pt. 57

CROSS STITCH PATTERN GENERATOR – PART 4 – UNDERSTANDING pyFPDF

Sorry for missing so many months. I still can’t sit for long periods of time, so this article might be shorter than what you are used to. My original plan was to jump right into the PDF output portion of the program, but there is so much to understand about this library, I decided to use this installment as a tutorial on pyFPDF and then tackle the PDF output next time. So let’s get started.

FPDF stands for Free PDF. A VERY minimal example would be as follows:

```python
from fpdf import FPDF
pdf = FPDF()
pdf.add_page()
pdf.set_font('Arial', 'B', 16)
pdf.cell(40,10,'Hello From Python')
pdf.output('example1.pdf', 'F')
```

The first line imports the library file. The next creates an instance of the FPDF object. We use the default values for this example, which are:
- Portrait
- Measure Unit = Millimeters.
- Format = A4

If you need to use ‘US’ standards, you could do it this way:

```python
pdf=FPDF('P','in','Letter')
```

Notice the parameters are FPDF(orientation, units, format):
- Possible values for orientation are “P” for Portrait and “L” for Landscape.
- Possible values for units are: ‘pt’ (points), ‘mm’ (millimeter), ‘cm’ (centimeter), ‘in’ (inches).
- Possible values for format are: ‘A3’, ‘A4’, ‘A5’, ‘Letter’, ‘Legal’ or a tuple containing the width and height expressed in the unit given in the unit parameter.

The third line creates a page to enter data into. Notice a page is not automatically created when we create the instance of the object. The origin of the page is the upper-left corner, and the current position defaults to 1 cm from the margin. The margin can be changed with the SetMargins function.

Before you can actually print any text, you must call pdf.set_font() to define a font. In the line above, we are defining Arial Bold 16 point. Standard valid fonts are Arial, Times, Courier, Symbol and ZapfDingbats.

Now we can print a cell with the pdf.cell() call. A cell is a rectangular area, possibly framed, which contains some text. Output is at the current position which is specified (40,10 cm) in the above example. The parameters are:

```python
df.cell(Width, Height, text, border, line, align, fill, link)
```

Where:
- Width is length of cell. If 0, width extends to the right margin.
- Height is the height of the cell.
- Text is the string of text you want to print.
- Border is either 0 (no border(default)), 1 is border, or a string of any or all of the following characters: "L", "T", "B", "R"
- Line is where the current position should go after printing the text. Values are 0 (to the right), 1 (to the beginning of the next line, 2 (below). Default is 0, and putting 1 is equivalent to putting 0 and calling ln() immediately after.
- Align allows to center or align the text within the cell. Values are "L" (left), "C" (center), "R" (right).
- Fill sets the background to be painted (true) or transparent (false). Default is false.
Link is a url or identifier returned by addlink().

Finally, the document is closed and sent to the file with Output. The parameters are fpdf.output(name,dest). If file is not specified, the output will be sent to the browser. Options for destination are "I" (inline to browser(default)), "F" (local file given by name), "D" (to the browser and force a file download with the name passed), and "S" (return the document as a string).

Since we will be sending our cross stitch images to the pdf file, we will have to understand the image function.

The function is called like this:

\[
\text{pdf.image(name, x=\text{None}, y=\text{None}, w=\text{0}, h=\text{0}, type=\text{"\", link=\"\")}
\]

This function puts the image. The size it will take on the page can be specified in different ways:
- Explicit width and height or
- One explicit dimension

Supported formats are JPEG, PNG, and GIF. If you wish to use GIF files, you must get the GD extension.

For JPEGs, all flavors are allowed:
- gray scale
- true colours (24 bits)
- CMYK (32 bits)

For PNGs, the following are allowed:
- gray scales on at most 8 bits (256 levels)
- indexed colors
- true colors (24 bits)

Note: interlacing is not allowed, and if you are using a version of FPDPD prior to 1.7, Alpha channel is not supported.

I stole this example (shown right) from the pyFPDF tutorial.

You have been around long enough that you should be able to look at the program and understand what is going on. But in this example the line we are REALLY interested in is the fourth line:

\[
\text{this.image('img1.png',10,8,33)}
\]

In this instance, we are calling the image function with the filename, the x position of where the picture will go on the page, the y position, and the width of the picture.

Now that you have a gross grasp of the library, we will start our PDF code next time.

Until then, have a good month. See you soon.

```python
from fpdf import FPDF

class PDF(FPDF):
    def header(self):
        # Logo - replace with a small png of your own
        this.image('img1.png',10,8,33)
        # Arial bold 15
        this.set_font('Arial','B',15)
        # Move to the right
        this.cell(80)
        # Title
        this.cell(30,10,'Title',1,0,'C')
        # Line break
        this.ln(20)

        # Instantiation of inherited class
        pdf=PDF()
        pdf.alias_nb_pages()
        pdf.add_page()
        pdf.set_font('Times','',12)
        for i in range(1,41):
```

**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
F

irst, let me thank all the readers who sent me emails of hope and wishes for a quick recovery. They were very kind and helpful. I also want to thank Ronnie, our wonderful editor, for his support and patience during that painful period. I still have issues with sitting for long periods of time, so this is being done over the course of a number of days, so I hope the continuity that I’m trying for works. Now on with “the show”...

Not too long ago, I was walking to the time clock and the General Manager of my “day job” called me into his office. Hoping it was just a “how’s it going” talk, I went in and sat down. He then started the meeting with “I’m having a problem with my spreadsheet program, and was hoping you could help me”.

As my vision darkened and the three-note ominous orchestral string hits “Da Da DAAAAAAA” that we all know from the horror flicks of the 70’s and 80’s rang through my mind, rather than running screaming from the room, I innocently asked what was wrong. He responded that there was something wrong with one of the macros and “the thing just quits in the middle of the calculations”. As I whipped out my white cowboy hat, I said in my best hero voice “Don’t worry citizen. We’ll have you up and running in no time.” Within a short while, I discovered the reason the spreadsheet was unceremoniously crashing was that one cell in one of 35 workbooks was getting a divide by zero error due to an expected value not being entered in another cell in yet another one of the 35 workbooks. Let me make this perfectly clear, it was not my boss’s fault. All he had asked for was a simple way to get the higher-up values from the data. (The previous two sentences have absolutely nothing to do with the fact that my boss may read this article! Or maybe it does.)

As I walked back to my work area, brushing the spurious bits of computer code from my white hat, I realized that this would be an excellent teaching moment. So, here we are. But first, let’s revert back to 1979 when Apple introduced Visicalc. That was the first “Free Form Calculation type system” to really make a hit in the marketplace. While there were many bugs in the software, the world loved the idea and clones (bugs and all) began to pop up on other computer systems, like the Commodore Pet and other Apple competitors (including Microsoft in 1981 with a program called Multiplan). Finally, in 1983, a company called Lotus Development Corp. introduced Lotus 1-2-3. While very close to Visicalc in many aspects, including the menu structure, it was written completely in x86 assembly language, which made it very fast, and many of the bugs of Visicalc were fixed. Lotus 1-2-3 was so popular that it became a common benchmark to test a machine for “PC Compatibility”.

The advent of the Free Form Calculation systems, allowed the “normal” person to deal with numbers in a way that previously was in the realm of the programmer. Almost anyone could, in a few hours or so, make sense of numbers, create charts and graphs, and share that information with coworkers. Shortly after that, the ability to automate some portions of the spreadsheet through Macros and Basic-like embedded languages gave these non-programmer users even more power over their destiny. They could get the answers themselves, and pretty charts and graphs as well, without having to wait in the queue for I.T. assistance. However, as we all learned from Peter Parker’s uncle Ben...
With great power, comes great responsibility.

Soon the spreadsheet was taken into areas that were better suited for databases than spreadsheets. We now had workbooks upon workbooks that relied on other workbooks, and if one little number along the way didn’t happen to get updated... well, we had the old “house of cards” effect.

While I don’t think that every spreadsheet is evil, there are some (read this to say ‘many’) that should have been converted to databases many years ago. They just became too large and unwieldy for their own good. If someone had just sat down with the programmers and said, “Please help”, the world would be a kinder, gentler place.

Now as I step down from my soapbox, we come to the real reason for this month’s article. Every good Python programmer should have a way to deal with spreadsheets in their arsenal of tools. You never know when you will be called upon to pull data from a spreadsheet and manipulate it. While there are many ways to get data from spreadsheets like using CSV files, which has its own drawbacks, sometimes you need to read and write directly from and to a ‘live’ spreadsheet. After looking around, I settled on a very nice library to access my boss’s problematical spreadsheet.

Let’s create an excel spreadsheet that we can use to examine the functionality of XLRD. Either open excel, or openoffice or libreoffice calc. In the first column (A), enter the numbers 1 to 5 going down. In the next column (B), enter 6 to 10. It should look something like this:

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
```

Now save the spreadsheet as “example1.xls” in the folder you will use to save the test code. This way, we won’t have to worry about paths.

Now download and install XLRD (https://pypi.python.org/pypi/xlrd). We can use it like is shown below.

```python
import xlrd
def OpenFile(path):
    # Open and read excel file
    book = xlrd.open_workbook(path)
    # Get number of active workbooks
    print "Number of workbooks: ",book.nsheets
    # Get the names of those workbooks
    print "Workbook names: ",book.sheet_names()
    first_sheet = book.sheet_by_index(0)
    cell = first_sheet.cell(1,1)
    print "Cell at 1,1: ",cell
    print "Cell Value at 1,1: ",cell.value

if __name__ == "__main__":
    path = "example1.xls"
```

Save the file as example1.py in the same folder as the spreadsheet. Since the code is so short, we will simply discuss it here. Of course, the first line imports the library. Then we create a function called OpenFile and pass the name (and path if needed) of the spreadsheet to the function.

Now we call the open_workbook method and get back a ‘book’ object. Then we use the nsheets attribute to return the number of ACTIVE workbooks. We can also get the name of the workbooks. In this case, they are the default. We use the sheet_by_index method to get Sheet1 into the first_sheet object. Now we can...
start getting data. We get the information from the cell at position (1,1) which translates to cell position B2 (it’s Zero based, so cell A1 would be (0,0)). We print the data from there, both what the cell contains and the value, so we could use it in a calculation if we wish.

That was really easy, wasn’t it? Now, let’s do something a bit more useful. Enter the code shown on the next page (top right) and save it as ‘example2.py’. This example will print out the contents of the workbook.

Since we already used the first four lines of code in the first example, we’ll skip them. By using the ‘sheet.nrows’ and ‘sheet.ncols’ attributes, we get the number of rows and columns. This can be helpful, not only so we know what we are dealing with; we can write “generic” routines that use those values in our calculations as you will see. In fact, we use ‘rows’ in a for loop to obtain each row’s information.

Notice the line that has

\[ \text{import xlrd} \]
\[ \text{def OpenFile(path):} \]
\[ \text{book = xlrd.open_workbook(path)} \]
\[ \text{first_sheet = book.sheet_by_index(0)} \]
\[ \text{# Get the number of rows in this workbook} \]
\[ \text{rows = first_sheet.nrows} \]
\[ \text{# get the number of columns in this workbook} \]
\[ \text{cols = first_sheet.ncols} \]
\[ \text{print "There are \%d rows in this workbook." \% rows} \]
\[ \text{print "There are \%d cols in this workbook." \% cols} \]
\[ \text{for r in range(0,rows):} \]
\[ \text{cells = first_sheet.row_slice(rowx=r,start_colx=0,end_colx=cols)} \]
\[ \text{print cells} \]
\[ \text{if __name__ == "_main_":} \]
\[ \text{path = "example1.xls"} \]

`first_sheet.row_slice`. This gets a block of cells of a given row. The syntax is as follows:

\[ \text{X = first_sheet.row_slice(\text{RowInQuery})} \]

`estion, Start_Column, End_Column)`

So we have used the number of rows and the number of columns in calculations. The output from our program should look something like this...

There are 5 rows in this workbook.
There are 2 cols in this workbook.
\[ \text{[number:1.0, number:6.0]} \]
\[ \text{[number:2.0, number:7.0]} \]

We’ll do one more example before we end this month’s article. Go to the spreadsheet and in column C put some dates. Here’s what my spreadsheet looks like now:

You can use any dates you like. Now let’s re-run our
example2.py program. Here is the output from mine.

There are 5 rows in this workbook.
There are 3 cols in this workbook.
[number:1.0, number:6.0, xldate:41649.0]
[number:2.0, number:7.0, xldate:42109.0]
[number:3.0, number:8.0, xldate:31587.0]
[number:4.0, number:9.0, xldate:23284.0]
[number:5.0, number:10.0, xldate:36588.0]
Press any key to continue ...

Well, that’s not what we expected. It seems that excel holds dates as a value that is simply formatted for whatever we ask it to. This might be helpful for sorting and calculations, but, for showing the actual data, this won’t do. Luckily, the writers of the library already thought of this. Delete the line that says “print cells” and replace it with the code shown below.

Here, we go through each cell in the cells list and check the type of the cell to see if it is considered a XL_CELL_DATE. If it is, then we convert it to a tuple. It is stored as YYYY,MM,DD. We simply pretty it up to print it as MM/DD/YYYY. Here is the output of our new program...

There are 5 rows in this workbook.
There are 3 cols in this workbook.
1.0
6.0
1/10/2014
2.0
7.0
4/15/2015
3.0
8.0
6/24/1986
4.0
9.0
9/30/1963
5.0
10.0
3/3/2000
Press any key to continue ...

Just for your information, there is a library from the same wonderful people called XLWT, which allows you to write to excel files. There is a wonderful tutorial and documentation on these two libraries at http://www.python-excel.org/.

The source code for example3.py is on pastebin at http://pastebin.com/bWz7beBw.

Hopefully, I’ll see you next month.

---

**Greg Walters** is owner of RainyDay Solutions, LLC, a consulting company in Aurora, Colorado, and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family. His website is...
last time, we discussed reading and using data from an Excel file directly. If you remember, my boss (from my “day” job) had a massive spreadsheet that if one calculation failed, it caused the entire process to abort. Well, I created a database from that spreadsheet that was easy to get a report from. However, the original spreadsheet created pretty charts and graphs that his bosses liked to see. So I undertook the task to create charts so everyone would be happy.

After spending about two days digging into the existing charting/graphing packages already available for Python, most free, most of them output directly to a file, like a pdf file or some sort of a graphics (jpg, png, svg) file. What I was looking for was one that would go directly to a wxPython frame or panel so it can be displayed inside a GUI program. I found one solution, but it required so many interdependent libraries that the possibility of just giving the application on a flash drive quickly became nil.

So, being the pig-headed, tenacious, never-say-die kinda guy that I am, I decided to write one on my own. The original goal was that it was to do (at least) bar-charts and maybe in the future line charts and/or other types. It also should eventually be able to do colours, but just plain black bars would suffice for the time being. It should be standalone in general so that it could be called as a library. It wasn’t supposed to be so generic that it gets complicated, just dates along the horizontal (bottom) axis, values along the vertical axis and bars that represent the daily sales for that period. In order to keep the chart somewhat neat, the dates should be angled so that they don’t overwrite each other. So, what I came up with will be presented here. Left, is a sample output of the code.

Again, not fancy, not terribly pretty, but it does the job. If it needs to be prettier later on, then I can work on it down the road.

The first thing I had to do was pull out my wxPython documentation to remember the graphic commands. In able to draw graphics, we use the a “dc” or Device Context. It’s sort of like a blank canvas that we can draw points, lines, and text to. wxPython offers 9 different types of dc objects and I chose the wx.PaintDC which works from the OnPaintEvent. We will use some very basic commands to do our drawing and painting. These are:

dc.DrawLine
dc.SetPen
dc.SetFont
dc.DrawLine
dc.DrawRectangle
dc.DrawRotatedText
dc.GetFullTextExtent

Those are the only wxPython routines we will use, though there are many others that would make our program much prettier. We will combine these commands into our own “logical” routines like, DrawBars, DrawAxis, DrawValues, and so on. While I could have done it in one or two large routines, I wanted to break them out into routines that make sense for the teaching moment. So let’s get started looking into the code. Create a file called mygraph.py. I couldn’t come up with anything pithy, since PyChart, PyGraph and the like are all taken. Maybe if I had a bit more time, I’d come up with something, but that’s not important. Let’s get started. First, we’ll do the imports as we...
always do.

#!/usr/bin/python
# mygraph.py

import wx
from datetime import date, timedelta
import time
import math

Obviously, we need to import the wxPython library and the math library will help us with some of the calculations. The datetime and time libraries are used to do the date calculations for the horizontal axis labels.

Something to keep in mind as we go from here...When you think about drawing on a context, the upper left corner of the container window (our dc) is X=0, Y=0. X is the horizontal axis and Y is the vertical axis. The closer we get the lower right corner, both numbers go higher. In our program, we will actually start by drawing a box that defines our charting area which starts at upper left X=10, Y=10 and end with lower right at X=800, Y=700. However, before we get to that part, we have to define a class to handle the routines and the __init__ routine. Hopefully you remember these from earlier sessions.

Top right is the class definition and the __init__ routine.

Our class is called Line and we will be creating a wxFrame to do our drawing. This could also be a panel within a frame or any number of other options. My choice was to have a Frame pop up with our chart data on it. When the class is first instantiated, the __init__ routine is called with the name of the parent object, the id of that object, the title of the frame (in the title bar), the data that we want to chart and finally the title of the chart itself. Next we create the wx.Frame object that is 1024x768 pixels in size. Next we bind the paint event (which is called everything the frame is created, moved, covered, uncovered, etc.) to our event routine OnPaint. Remember, since this is inside of a class we use the “self” to say the routine belongs to the class not somewhere else. We set some variables (BoxWidth, BoxHeight, ChartTitle, data) for use later. After we set self.data to an empty list, we call a routine called SetData to find our data scale, which we will discuss further down. Finally, we set the frame to be centered in the screen and call the Show routine. This will automatically

class Line(wx.Frame):
    def __init__(self, parent, id, FrameTitle, IncomingData, ChartTitle):
        wx.Frame.__init__(self, parent, id, FrameTitle, size=(1024, 768))
        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.BoxWidth = 790
        self.BoxHeight = 690
        self.ChartTitle = ChartTitle
        self.setData = []
        self.SetData(IncomingData)
        self.Centre()

def DrawBox(self,dc):  #Horizontal
    dc.DrawLine(10,10,800,10)
    dc.DrawLine(10,700,800,700)  #Vertical
    dc.DrawLine(10,10,10,700)
    dc.DrawLine(800,10,800,700)
    
This is fairly simple. We pass the dc of the frame, then draw four lines. The DrawLine function parameters are:

def DrawAxis(self,dc):
    #Horizontal
    dc.DrawLine(60,580,700,580)
    #Vertical
    dc.DrawLine(60,580,60,80)
call the OnPaint routine since we are creating the Frame.

Next (above) we will write a routine that will create a box that shows the area that we want to confine our graph to. This is not a clipping or constraining box, it is simply to draw the eye to what we want the user to look at.

Not really difficult. We will be using the DrawLine function several times throughout the program. Next we will create a routine that will draw the X (horizontal) and Y (vertical) axis lines on the screen. We again pass the dc of the frame into the routine.

Since we just discussed the DrawLine method, there’s nothing very out of the ordinary here. We are drawing a line 580 pixels down the Frame that starts at X=60 and ends at X=700. Then we draw a line that starts at X=60 Y=580 and goes up to X=60 Y=80. This one is drawn from the bottom up, but we could have drawn it from the top down.

Next we will deal with the DrawTitle routine. Once again, we pass the dc of the frame as well as the text we want to draw. During this process, think of drawing text rather than printing text. It’s a very minor thing, but it will help.

This routine is longer than most of the others, but part of that is the comments I put in. The first two lines set the font and the pen style that we will be using. In the first line (SetFont), we define the font to be the “default” font, 20 points, not italic and bold. Next we set the colour of the pen to black and the width to be 20. Now we need to figure out the width of the text that we will be drawing so we know how to center it in the box. We get this information by calling the GetFullTextExtent with the text that we will be drawing using the font, font size, pen width and so on that we just defined. The tuple that is returned contains Width, Height, Decent (how far down letters like “g” and “y” will go below the base line) and any leading space. For our purposes, all we are concerned with the width. If you remember, we defined the

def DrawTitle(self,dc,text):
   dc.SetFont(wx.Font(20,wx.DEFAULT,wx.NORMAL,wx.BOLD))
   dc.SetPen(wx.Pen(wx.NamedColour('black'),20))
   #Get the length of the text to draw
   vals = dc.GetFullTextExtent(text)
   # Returned (Width,height,Decent,externalLeading)
   #Get the left position (x) to draw centered text
   txtleft = (self.BoxWidth-vals[0])/2
dc.DrawText(txt,txtleft,30)

def DrawDateTicks(self,dc,dcount):
   for cntr in range(1,dcount+1):
      dc.DrawLine(65+(cntr*20),580,65+(cntr*20),600)

width of the box back in the _init__ function as 790. To find the center of our text within our box we take the box width minus the width of the text and then divide it by 2. That will be the X value we use to draw our text. Finally, we reset the pen size and colour. Rather than use some default values we pick out of nowhere, we could have called the dc.GetPen function before we started, but when I started the project, I didn’t think about it.

Our next routine will draw the tic lines along the horizontal axis at the bottom of the chart. We want them to be equidistant along the line. We pass (as usual) the dc and a value I called dcount which is the number of dates we want to show. Since the number of days in any given month can range from 28 to 31, I wanted to be a bit dynamic. We simply use a for loop to count the number of lines to draw, which one to draw and where. If you have been carefully paying attention, we will start the lines at position 85 and it will be 20 pixels high and they will be 20 pixels apart.

def DrawRotText(self,dc,txt,x,y):
   dc.SetFont(wx.Font(10,wx.DEFAULT,wx.NORMAL,wx.BOLD))
   dc.SetPen(wx.Pen(wx.NamedColour('black'),20))
   dc.DrawRotatedText(txt,x,y,-45)
HOWTO - PYTHON

When we get around to drawing the dates into the chart, we want to draw the text on an angle. That way, the text doesn’t draw over itself and, well let’s admit it, looks cool. For this we will use the DrawRotatedText function. The function takes the text that we want to have drawn, the X and Y location as a starting point and the angle we want the text to be drawn. In this case, we want the text to be rotated anti-clockwise by 45 degrees which we enter as “-45”. We will set the font and pen parameters each time the text is drawn. We’ll deal with the actual draw date function in a little bit.

We will also want to draw the values along the vertical axis showing tics along the way. If we had the same range of data each time, it would be very easy to do. However, reality shows that the data range of our chart could vary from run to run. One time, the highest value could be 300. The next time it could be 3000. How could we create a generic routine that would account for this? I will try to explain my mindset here.

You might have wondered why I chose the value of 500 for the vertical axis when we drew the line from 80 to 580 (or actually 580 to 80). I chose to use a 500 pixel “view port” to contain our values. That way, we can create a scaling value based on an offset of 500.

Let’s say that for a given run that our highest value is 395. We can simply draw a bar that is 395 pixels high to represent that value. The next run, our highest value is 2,345. If we try to draw the bar to its full height, it would disappear off the top of the chart. In order to show the value, we can round the value to the nearest 500, which would be 2500 and then set that as the top value of our axis. We then can scale the value by dividing 2500 by 500 which gives us a “scaling factor” of 4. Now if we take our data values and divide each one by our scaling factor, we can then plot the values that they will fit within our graph.

So (shown top right) we need to find the highest value within our data and round that up to the nearest 500. So 375 would be 500, 3750 would be 5000 and so on.

Next, we need to decide what kind of data we are going to use. You will see further down the program that I provide two different types of data in lists. One assumes that the date range we will use, along the X axis, is data for October, but you can easily follow that code (shown in a

```python
# Round up to the nearest 500
# ==============
def roundup(self,x):
    return int(math.ceil(x/500.0))*500
```

```python
def SetData(self,DataToUse):
    if type(DataToUse[1]) is tuple:
        self.DateList=[]
        self.ValList=[]
        for l in DataToUse:
            self.DateList.append(l[0])
            self.ValList.append(l[1])
        self.HiValue = self.roundup(max(self.ValList))
        self.ScaleValue = self.HiValue/500
    else:
        self.DateList=[]
        self.ValList=[]
        for l in DataToUse:
            self.ValList.append(l)
        self.HiValue = self.roundup(max(self.ValList))
```

```python
def DrawValues(self,dc):
    c2 = 0
    for cntr in range(580,30,-50):
        dc.SetPen(wx.Pen(wx.NamedColour('black'),1))
        dc.DrawLine(60,cntr,50,cntr)
        dc.SetFont(wx.Font(10,wx.DEFAULT,wx.NORMAL,wx.BOLD))
        dc.SetPen(wx.Pen(wx.NamedColour('black'),20))
        dc.DrawText(str(c2),26,cntr-7)
        c2 = c2 + (50 * self.ScaleValue)
```
few moments) and change it to whatever month you wish. The second data list is more
generic and provides both a date and a value as a list of
tuples. This allows for data to be passed for any time period.
The date is a string and the value is either an integer or a
float. The SetData function will look at the first value within the
data list and to determine if it is a
tuple. If it is, we assume that
the data structure of the list is
the second option, if not, it is
the first.

If it is a tuple, we create two
lists, one for the dates and one
for the values. We then walk
the list splitting the data
between the two lists. Once we
have that done, we then find
the highest value
(max(self.ValList)) and send it
the roundup function (shown
above) so we can get our
scaling value. If the data isn’t in
tuples, then we clear BOTH lists
and do the same steps as
above.

Now that we have our scale
value we can draw our tics and
the values that will represent
our vertical axis. We again use

a for loop, this time from 580
to 30 with a step of -50 to work
our way up the line and draw a
10 pixel line. Next we set the
font (just in case it gets
changed somehow) and draw
the value of each of our values.

Now we get into the routines
that will create the date tics
along the X axis if we choose to
have a simple list of data
without including the dates. We
have two support routines, one
called DateToStamp and the
other Timestamp2Date (Yes, I
got lazy when I wrote this one).
Rather than going through a
bunch of complicated DateTime
routines to determine the
number of days in any given
month, I’m going to use a start
date and an end date, convert
both of those to Unix
timestamps to get the proper
day of month within the
sequence. I’ve shown you the
DateToStamp routine before
and the Timestamp2Date
simply reverses the process.

The next routine takes the
start date and end date, as we
moment ago, converts them to
Unix timestamps, then adds
86400 (the number of ticks in a
24 hour period) to make sure
we get the last date in the
sequence, then uses another
for loop to draw the rotated
text where we want it.

We are now at the OnPaint
event handler that calls all the
helper routines we dealt with so...
far. Remember, by using the PaintDC, every time the frame is moved, re-sized, covered or uncovered, the OnPaint event handler is called, thereby assuring our graph will be persistent.

First (shown on the next page, top left) we get an instance of our dc, and then we call the DrawBox, DrawAxis, DrawTitle, and the DrawDateTicks routines. We then determine if the DateList list (created in the SetData routine called from __init__ routine) is empty or if it has dates for us to draw. If so, we call the DrawDates routine with the proper values. We then call the DrawValues routine and finally the DrawBars routine. Now you should understand why I broke everything down into little bitty chunks.

The last thing we have to look at is the runtime routine. You probably remember that the 'if __name__ == "__main__"' runs if we are calling the program as a standalone rather than as a library. The next two lines are the dummy data that I used to test the program. You could comment out the first one and run it with the second data line which is the one that uses the tuple. The last three lines will instantiate the wxPython routines, then the Line class and finally call the app.MainLoop wxPython routine to get the frame to run.

So there it is. Our own graphing/charting program and library. I've put the full code up on pastebin at http://pastebin.com/m2feeh5P.

Until next time, have fun coding.

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
First, let me say Happy 100 to Ronnie and the crew. It’s a privilege to be part of this milestone.

This time I thought that I’d share some information on my new obsession. I’ve started repairing and building stringed musical instruments like guitars and violins. Believe it or not, there is a lot of math involved in musical instruments. Today, we will look at some of the math involved with the length of strings and where the frets should be placed on the fretboard.

Take a look at the picture of the guitar. I annotated various items in the image. The important things to look at are the Nut near the top of the fingerboard, the Frets, the Bridge near the bottom, and the white “line” within the bridge called the Saddle. The purpose of the frets is to create a perfect spot to change the length of the string to create a note that is in tune. The positions of these frets are not arbitrary, but mathematically determined.

Now, the physics of vibrating strings tells us that if you half the vibrating string length of a theoretically perfect string, you will double the frequency of the vibrations. In the case of a guitar, this string length is between the nut and the saddle. This distance is referred to the Scale Length of the guitar. The half-point that allows for the doubled frequency is fret # 12. If correctly done, just by lightly placing your finger on the string at this location, you get a pleasing tone. There are a few other positions that this will happen, but the 12th fret should be the perfect location for this doubling, making the note go up one octave.

Different scale lengths will create different feel and tones. For example, guitars like the Fender Stratocasters® have a scale length of 25½”, which produces a rich and strong bell-like tone. On the other hand, Gibson guitars often use a scale length of 24¾”. This creates a lower string tension which makes an easier playing feel and a warmer tone. Other guitar manufacturers decided that a scale length of 25” makes a clearer tone than either of the other two “standard” scale lengths.

So with the ability of a guitar maker to come up with their own scale length, the spacing of the frets will have to be recalculated. This is something that luthiers (guitar makers) have been dealing with for hundreds of years.

In the past, there was a technique called the rule of 18 which involves successively dividing the scale length minus the offset to the previous fret by 18. While this kind of worked, the tones were off, the higher up the fingerboard the player went. These days, we use a different constant. This constant is 17.817. By using this “new” constant, the 12th fret or octave is at the exact position to be half the scale.
HOWTO - PYTHON

length of the string.

Now, these calculations are easy enough to do by paper and pencil or even a simple calculator, it's just as easy to create a Python program to do the calculations for us in just a second. Once you have the positions, you simply saw a slot for the fret at the correct positions and then hammer in the frets.

So, let's take a look at the program.

We want to create a program that will prompt for the scale length of the guitar (or bass), do the calculations and then print out the distances. The calculations and all returned lengths are all in inches, so all our friends that use metric measurements, please add the proper conversion calculations. After almost 5 years, you should be able to do this with ease.

We don’t need to import any libraries for this so we will start off by defining a couple of variables.

```python
ScaleLength = 0
CumulativeLength = 0
```

Next we will create a routine (top right) that will be called repeatedly as we “travel down” the fingerboard. We will pass two values into this routine. One is the scale length and the other is the cumulative distance from the nut to the previous fret.

In this routine, we take the scale length, subtract the cumulative distance and assign that value to BridgeToFret. We then take that value, divide it by our constant (17.817), add back in the cumulative distance and then return that value to our calling routine. Remember, we could simply have returned the calculated value without assigning it to a variable name. However, if we ever want to inspect the calculated values, it’s easier to do if we assign the value before we return it.

Now we will make our worker routine. We’ve done this kind of thing many times in the past. We will pass it the scale length and it will loop for up to 24 frets (range(1,25)). Even if your project has less than 24 frets, you will have the correct positions of all the frets you do have. I chose 24 because that’s the maximum of frets for most guitars. When we get into the loop, we check the fret number (x) and if it is 1, we pass the cumulative length as 0, since this is the first calculation. Otherwise, we pass the last cumulative length in and it becomes the result from the calculation routine. Finally, we print each fret number followed by a formatted version of the cumulative length.

Finally, we have the code that does the prompting for the scale length. I’m sure you will remember the format for the raw_input routine, since we have used it so many times before. Something you might not remember: that raw_input always returns a string, so when we pass it off to the DoWork routine, we have to pass it as a floating point number so the routine will work correctly. Of course, we could simply pass it as a string, but we would have to deal with the conversion in the DoWork routine.

```python
def CalcSpacing(Length, NTF):
    BridgeToFret = Length - NTF
    NutToFret = (BridgeToFret/17.817) + NTF
    return NutToFret

def DoWork(ScaleLength):
    CumulativeLength = 0
    for x in range(1,25):
        FretNumber = x
        if FretNumber == 1:
            CumulativeLength = CalcSpacing(ScaleLength, 0)
        else:
            CumulativeLength = CalcSpacing(ScaleLength, CumulativeLength)
        print(“Fret=%d,NutToFret=%.3f” % (FretNumber, CumulativeLength))
```

full circle magazine 270 The Compleat Python
ScaleLength =
raw_input("Please enter Scale
Length of guitar --> ")

DoWork(float(ScaleLength))

You might wonder what good this program will do if you aren’t going to build a guitar from scratch. It can be valuable when you're looking at buying a used guitar or trying to tweak a guitar with a floating bridge. Also, if you are a guitar player, this might have been something you didn’t know about guitars.

Of course, the code is available from pastebin at http://pastebin.com/A2RNECT5.
Welcome fellow pythoners. As the kids here in the central parts of the U.S. say, “What’s Shakin’ Bacon?” I’m not exactly sure what that’s supposed to mean, but I assume it’s a good thing.

You might notice the new header. I decided that I’ve taught you all the basics of Python that I can for “general” programming, so now we are going to delve into using Python to talk to other types of computers and controllers, like the Raspberry Pi and the Arduino micro controller. We’ll look at things like temperature sensors, controlling motors, flashing LEDs and more.

This issue we will be focusing on what we’ll need to do this and focus on a few of the projects we will be looking at in the future. Next issue, we will start the first project.

One of the things we will talk about next time will be the Raspberry Pi. The Pi is a credit-card sized computer that natively runs Linux on an SD card. Its output goes to your TV set via HDMI. It also has an Ethernet connection for Internet access.

You can find out more at the official site https://www.raspberrypi.org. If you want to follow along with the projects, you will need a Pi, SD card, Keyboard, Mouse, a 5volt DC power supply like the ones on modern cell phones, and access to an HDMI monitor or TV. Eventually, you should also consider getting a breadboard and some connecting wires for when we start to interface to the outside world. You can find any number of places that sell the Pi on the Internet. Here in the U.S., we can get them for around $35.

One other thing about the Pi is that it provides access to a series of pins that support GPIO (General Purpose Input/Output). Basically, this means that you can write programs that will send signals to the output pins and read the signals from the input pins. This can be used to interface to things like LEDs, sensors, push buttons, etc. Many people have made home automation systems, multiple processor systems (by linking 40 or so Pi computers together to emulate a supercomputer), weather stations, even drones. So you can imagine that the possibilities are endless. That’s why I decided to start with it for this series of articles.

After a while, we will begin to work with the Arduino, which according to the official website (https://www.arduino.cc): “Arduino is an open-source electronics platform based on easy-to-use hardware and software. It’s intended for anyone making interactive projects”.

Once again, this is an exciting device to work with. In this part of the series, we will look at talking to the Arduino, first in its native scripting language, and then in Python and eventually interfacing the Pi with the Arduino.

I know this month’s article is fairly short, but I’ve been doing poorly health-wise, so I’m saving my strength for the next article. Until then, grab some electronics and get ready for 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.