About Full Circle

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

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

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

Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

Welcome to another 'single-topic special'

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

For now, this is a straight reprint of the series **'Programming in Python', Parts 1-8** from issues #27 through #34; nothing fancy, just the facts.

Please bear in mind the original publication date; current versions of hardware and software may differ from those illustrated, so check your hardware and software versions before attempting to emulate the tutorials in these special editions. You may have later versions of software installed or available in your distributions' repositories.

Enjoy!

Find Us

Website:
http://www.fullcirclemagazine.org/

Forums:
http://ubuntuforums.org/forumdisplay.php?f=270

IRC: #fullcirclemagazine on chat.freenode.net

Editorial Team

Editor: Ronnie Tucker
(aka: RonnieTucker)
ronnie@fullcirclemagazine.org

Webmaster: Rob Kerfia
(aka: admin / linuxgeekery-
admin@fullcirclemagazine.org

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

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

Program In Python - Part 1

Written by Greg Walters

SEE ALSO:
N/A

APPLICABLE TO:
ubuntu kubuntu xubuntu

CATEGORIES:
Dev Graphics Internet M/media System

DEVICES:
CD/DVD HDD USB Drive Laptop Wireless

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.

```
#!/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

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

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

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

Hello.  I am a python
program.

What is your name? Ferd
Burphel

Hello there, Ferd Burphel!

greg@earth:~/python_examples$
```

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

```
#!/usr/bin/env python
```

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

```
print 'Hello. I am a python
program.'
```

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

```
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

instance, simply takes the input and stores it in the "name" shoe-box for use later in the program.

```
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.

>>>
```

You are now in the python shell. From here, you can do a number of things, but let's see what we got before we go on. The first thing you should notice is the python version -- mine is 2.5.2. Next, you should notice a statement indicating that, for help, you should type "help" at the prompt. I'll let you do that on your own. Now type:

```
print 2+2
```

and press enter. You'll get back

```
>>> 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:

```
>>> Print 2+2
  File "<stdin>", line 1
    Print 2+2
          ^
SyntaxError: invalid syntax
>>>
```

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:

```
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:

```
print var
```

and press enter.

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

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

```
>>> print var * 2
8
>>>
```

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

```
>>> 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:

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

>>> print strng

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

>>>
```

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

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!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
4
```

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
>>>
```

The string (s), which is '4', has now been converted to an integer and then multiplied by 4 to give 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
```

```
2
3
4
5
6
7
8
9
greg@earth:~/python_examples$
```

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:

```
#! /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.

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 2

Written by Greg Walters

SEE ALSO:
FCM#27 - Python Part 1

APPLICABLE TO:
ubuntu kubuntu xubuntu

CATEGORIES:
Dev Graphics Internet M/media System

DEVICES:
CD/DVD HDD USB Drive Laptop Wireless

Correction To Part1
I received an email from David Turner who suggested that using the Tab-key for indentation of code is somewhat misleading as some editors may use more, or less, than four spaces per indent. This is correct. Many Python programmers (myself included) save time by setting the tab key in their editor to four spaces. The problem is, however, that someone else's editor may not have the same setting as yours, which could lead to ugly code and other problems. So, get into the habit of using spaces rather than the Tab-key.

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

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:

```
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:

```
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:

```
print len(strng)
```

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

```
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:

```
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:

```
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.

Literal Substitution

There 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:

```
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:

```
Months =
['Jan','Feb','Mar','Apr','May
','Jun','Jul','Aug','Sep','Oc
t','Nov','Dec']
DaysInMonth =
[31,28,31,30,31,30,31,31,30,3
1,30,31]
for cntr in range(0,12):
    print '%s has %d
days.' %
(Months[cntr],DaysInMonth[cnt
r])
```

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:

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

or like this:

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

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

```
st = 'He said he's on his way'
```

you will get a syntax error. You need to assign it like this:

```
st = "He said he's on his way"
```

Think of it this way. To define a string, you must enclose it in some kind of quotes – one at the beginning, and one at the end – and they must match. If you need to mix quotes, use the outer quotes to be the ones that aren't in the string as above. You might ask, what if I need to define a string like "She said "Don't Worry""? In this case, you could define it this way:

```
st = 'She said "Don\'t Worry"'
```

Notice the backslash before the single quote in 'Don't'. This is called an escape character, and tells Python to print the (in this case) single-quote – without considering it as a string delimiter. Other escape character sequences (to show just a few) would be '\n' for new line, and '\t' for tab. We'll deal with these in later sample code.

Assignment verses 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):

```
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):

```
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:

```
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:

```
# 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:

```
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:

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

and in Python we would say:

```
if x == y:
      do something
else:
      do something else
      more things to do
```

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:

```
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:

```
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.

```
loop = 1
while loop == 1:
      response = raw_input("Enter something or 'quit' to end => ")
      if response == 'quit':
            print 'quitting'
            loop = 0
      else:
            print 'You typed %s' % response
```

FIG. 1

In this example, we are combining the if statement, while loop, raw_input statement, newline escape sequence, assignment operator, and comparison operator – all in one 8 line program.

Running this example would produce:

```
Enter something or 'quit' to
end
 => FROG
You typed FROG
Enter something or 'quit' to
end
 => bird
You typed bird
Enter something or 'quit' to
end
 => 42
You typed 42
Enter something or 'quit' to
end
 => QUIT
You typed QUIT
Enter something or 'quit' to
end
 => quit
quitting
```

Notice that when we typed 'QUIT', the program did not stop. That's because we are evaluating the value of the response variable to 'quit' (response == 'quit'). 'QUIT' 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 usrname 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(usrname) 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.

```
#-----------------------------------------------
#password_test.py
#    example of if/else, lists, assignments,raw_input,
#    comments and evaluations
#-----------------------------------------------
# Assign the users and passwords
users = ['Fred','John','Steve','Ann','Mary']
passwords = ['access','dog','12345','kids','qwerty']
#-----------------------------------------------
# Get username and password
usrname = raw_input('Enter your username => ')
pwd  = raw_input('Enter your password => ')
#-----------------------------------------------
# Check to see if user is in the list
if usrname in users:
     position = users.index(usrname) #Get the position in the list of the users
     if pwd == passwords[position]:  #Find the password at position
          print 'Hi there, %s.  Access granted.' % usrname
     else:
          print 'Password incorrect.  Access denied.'
else:
     print "Sorry...I don't recognize you.  Access denied."
```

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

Written by Greg Walters

SEE ALSO:
FCM#27-28 - Python Parts 1-2

APPLICABLE TO:

ubuntu kubuntu xubuntu

CATEGORIES:

DEVICES:

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.

```
#=======================================
# random_example.py
# Module example using the random module
#=======================================
import random
# print 14 random integers
for cntr in range(1,15):
     print random.randint(1,10)
```

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:

```
from random import randint
```

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

```
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.

```
#silly example
print 'Adding the two numbers %d and %d = %d ' % (1,2,1+2)
print 'Multiplying the two numbers %d and %d = %d ' % (1,2,1*2)
print 'Subtracting the two numbers %d and %d = %d ' % (1,2,1-2)
print '\n'
print 'Adding the two numbers %d and %d = %d ' % (1,4,1+4)
print 'Multiplying the two numbers %d and %d = %d ' % (1,4,1*4)
print 'Subtracting the two numbers %d and %d = %d ' % (1,4,1-4)
print '\n'
print 'Adding the two numbers %d and %d = %d ' % (10,5,10+5)
print 'Multiplying the two numbers %d and %d = %d ' % (10,5,10*5)
print 'Subtracting the two numbers %d and %d = %d ' % (10,5,10-5)
print '\n'
```

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

```
#silly example 2...still silly, but better
def DoTwo(num1,num2):
      print 'Adding the two numbers %d and %d = %d ' % (num1,num2,num1+num2)
      print 'Multiplying the two numbers %d and %d = %d ' % (num1,num2,num1*num2)
      print 'Subtracting the two numbers %d and %d = %d ' % (num1,num2,num1-num2)
      print '\n'

DoTwo(1,2)
DoTwo(1,4)
DoTwo(10,5)
```

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.

```
'+==============================+'
'| Item 1                 X.XX  |'
'| Item 2                 X.XX  |'
'|------------------------------|'
'| Total                  X.XX  |'
'+==============================+'
```

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 '%2.f'. 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,item1,10)
```

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.

Save the code as 'pprint1.py' and run it. Your output should look something like the text shown above right.

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

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

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

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

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

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:

```
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.

```
#item1 = 3.00
#item2 = 15.00
itms =
[['Soda',1.45],['Cand
y',.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.

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

print TopOrBottom('=',40)

total = 0 #NEW LINE
for cntr in range(0,4): #NEW LINE
     print Fmt(itms[cntr][0],30,itms[cntr][1],10) #NEW LINE
     total += itms[cntr][1] #NEW LINE
print TopOrBottom('-',40)
print Fmt('Total',30,total,10) #CHANGED LINE
print TopOrBottom('=',40)
```

I 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.

```
+======================================+
| Soda                           1.45  |
| Candy                          0.75  |
| Bread                          1.95  |
| Milk                           2.59  |
+--------------------------------------+
| Total                          6.74  |
+======================================+
```

If you wanted to get wild and crazy, you could add a line for tax as well. Handle it close to the same way we did the total line, but use (total * .086) as the cost.

```
print
Fmt('Tax:',30,total*.086,10)
```

If you would like to, you can add more items to the list and see how it works.

That's it for this time. Next time we'll concentrate on classes. **Enjoy!**

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

Program In Python - Part 4

Written by Greg Walters

SEE ALSO:
FCM#27-29 - Python Parts 1-3

APPLICABLE TO:

ubuntu kubuntu xubuntu

CATEGORIES:

Dev Graphics Internet M/media System

DEVICES:

CD/DVD HDD USB Drive Laptop Wireless

```
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
```

I 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.

```
        /--Beagle
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.

```
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.

This one I've made more flexible. Depending on the mood of the dog, the bark will change. Shown on the next page is the full class code so far.

So, when we run this we'll get

```
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
```

Now, that takes care of the grumpy old Beagle. However, I said earlier that I have 3 dogs. Because we coded the class

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

carefully, all we have to do is create two more instances of our dog class.

```
Lab =
Dog('Nina','Black','Medium','
Heavy','Laid Back',7)
Heeler =
Dog('Bear','Black','Tall','Sk
inny','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
My Name is Nina
```

```
My color is Black
My Mood is Laid Back
I am hungry = False
Yawn...ok...Woof
Bark Bark Bark Bark Bark
Bark Bark
```

Now that you have the basics, your homework will be to expand our dog class to allow for more methods, such as maybe Play or EncounterStrangeDog or something like this.

Next time, we will start discussing GUI or Graphical User Interface programming. We will be using ***Boa Constructor*** for this.

```
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 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
        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'
        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()
```

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 5

SEE ALSO:
FCM#27-30 - Python Parts 1-4

APPLICABLE TO:
ubuntu kubuntu xubuntu

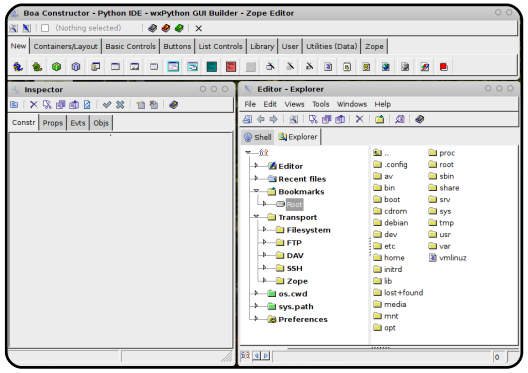
CATEGORIES:
Dev Graphics Internet M/media System

DEVICES:
CD/DVD HDD USB Drive Laptop Wireless

If 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.

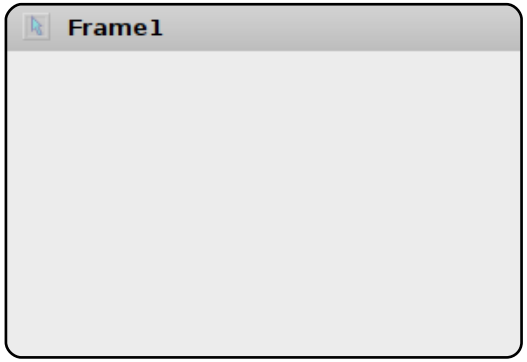
Here are a few things to remember as we are getting ready for our first app. There are a few bugs in the Linux version. One is that SOME controls won't allow you to move them in the designer. Use the <Ctrl>+Arrow keys to

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.

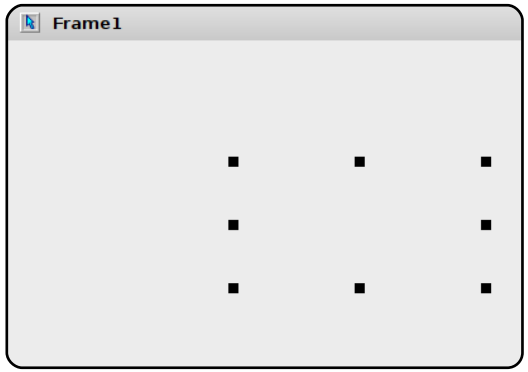
Okay, here we go. Under the 'New' tab of the tool frame, select wx.App (5th button from the left). This will create two new tabs in the editor frame: one named "*(App1)*", the other named "*(Frame1)*". Believe it or not, the VERY first thing we want to do is save our two new files, starting with the 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:

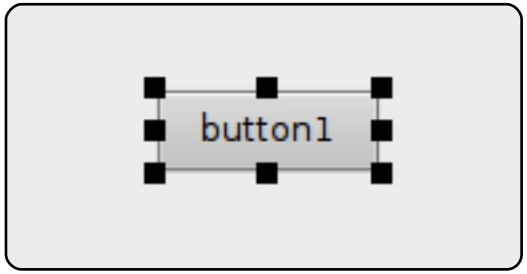


Remember when I warned you about the bugs? Well, this is one of them. Don't worry. See the 8 little black squares? That's the limits of the panel. If you wanted, you could click and drag one of them to resize the panel, but for this project what we want is to make the panel cover the entire frame. Simply 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.

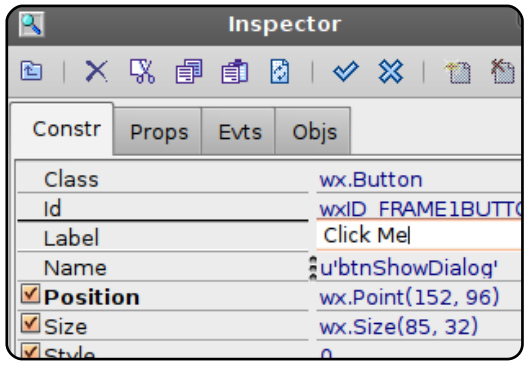
The Check button: 

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.

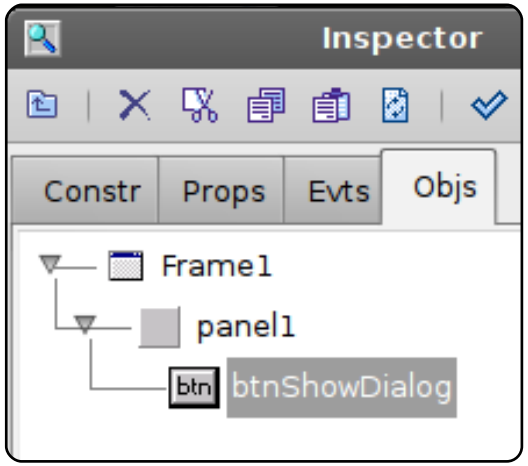
Go back into the Design mode. Click the 'Buttons' tab on the Tool frame and then click the first button on the left, the wx.Button. Then add it somewhere close to the middle of your frame. You'll have something that looks close to this:



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'.

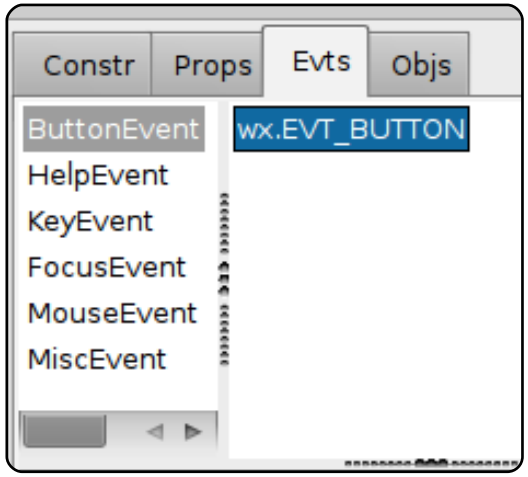


Now, let's skip over all the rest of that tab and go to the Objs tab. This tab shows all the controls you have and their parent/child relationships. As you can see, the button is a child of panel1, which is a child of Frame1.



Post (check button) and save your changes. Go back to the designer once again, and notice that (assuming you still have the 'Objs' 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 11).

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_ctrls method. Notice the comment just under the definition line. Don't edit the code in this section. If you do, you 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.

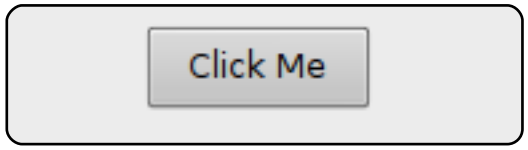
Next, look at the __init__ routine. Here you can put any calls to initializing code. Finally, the OnBtnShowDialogButton routine. This is where we will put our code that will do the work when the user clicks the button. Notice that there is currently an event.Skip() line there. Simply stated, this says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

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!

```
#Boa:Frame:Frame1
import wx
def create(parent):
    return Frame1(parent)
[wxID_FRAME1, wxID_FRAME1BTNSHOWDIALOG, wxID_FRAME1PANEL1,
] = [wx.NewId() for _init_ctrls in range(3)]

class Frame1(wx.Frame):
    def _init_ctrls(self, prnt):
        # generated method, don't edit
        wx.Frame.__init__(self, id=wxID_FRAME1, name='', parent=prnt,
              pos=wx.Point(543, 330), size=wx.Size(458, 253),
              style=wx.DEFAULT_FRAME_STYLE, title=u'Our First GUI')
        self.SetClientSize(wx.Size(458, 253))
        self.panel1 = wx.Panel(id=wxID_FRAME1PANEL1, name='panel1', parent=self,
              pos=wx.Point(0, 0), size=wx.Size(458, 253),
              style=wx.TAB_TRAVERSAL)
        self.btnShowDialog = wx.Button(id=wxID_FRAME1BTNSHOWDIALOG,
              label=u'Click Me', name=u'btnShowDialog', parent=self.panel1,
              pos=wx.Point(185, 99), size=wx.Size(85, 32), style=0)
        self.btnShowDialog.Bind(wx.EVT_BUTTON, self.OnBtnShowDialogButton,
              id=wxID_FRAME1BTNSHOWDIALOG)

    def __init__(self, parent):
        self._init_ctrls(parent)
    def OnBtnShowDialogButton(self, event):
        event.Skip()
```

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 6

SEE ALSO:
FCM#27-31 - Python Parts 1-5

APPLICABLE TO:
ubuntu kubuntu xubuntu

CATEGORIES:
Dev Graphics Internet M/media System

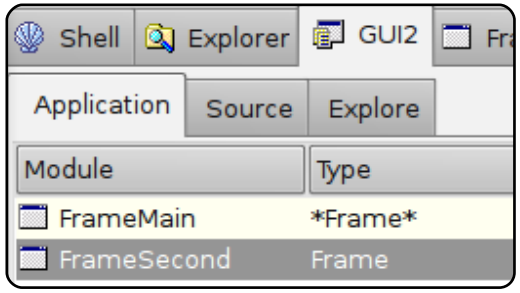
DEVICES:
CD/DVD HDD USB Drive Laptop Wireless

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...

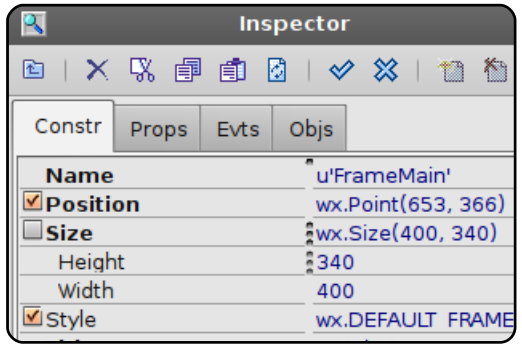
Start up Boa Constructor and close all tabs in the Editor frame with the exception of Shell and Explorer by using the (Ctrl-W) key combination. This ensures that we will be starting totally fresh. Now create a new project by clicking on the wx.App button (see last time's article if needed).

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 400x340 by clicking on the Size check box. This drops down to show height and width. Height should be 400 and width should be 340:

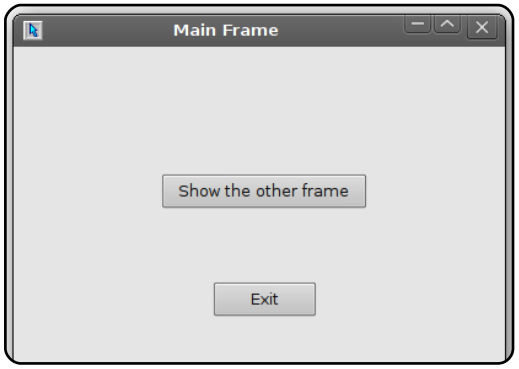


Now click on the Props tab. Click on the Centered property and set it to wx.BOTH. Click the post check-mark and save your work. Now run your application by clicking on the button with the yellow arrow. Our application shows up in the center of the screen with the title of "Main Frame". Now close it by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

Bring FrameMain back into the designer. Add two wx.Buttons 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".

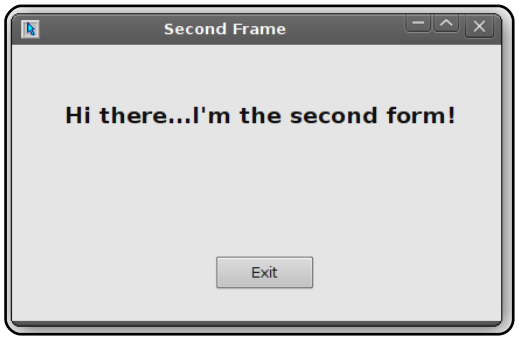
Post, save, and run to see your changes. Exit our app and go back to the designer. We are going to add button click events. Select the top button, and in the inspector frame, select the Evts tab. Click on ButtonEvent, then double click on wx.Evt_BUTTON. Notice you should have "OnBtnShowNewButton" below. 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:

```
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:

```
self.parent.Show()
self.Hide()
```

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 following page):

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

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()
```

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 _init_ctrls in range(4)]

class FrameMain(wx.Frame):
    def _init_ctrls(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.panel1 = wx.Panel(id=wxID_FRAMEMAINPANEL1,
name='panel1',
              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.panel1, 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.):

```
        self.btnExit =
wx.Button(id=wxID_FRAMEMAINBTNEXIT, label=u'Exit',
              name=u'btnExit', parent=self.panel1,
pos=wx.Point(162, 191),
              size=wx.Size(85, 29), style=0)
        self.btnExit.SetBackgroundColour(wx.Colour(225,
218, 91))
        self.btnExit.Bind(wx.EVT_BUTTON,
self.OnBtnExitButton,
              id=wxID_FRAMEMAINBTNEXIT)

    def __init__(self, parent):
        self._init_ctrls(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_FRAMESECONDBTNFSEXIT,
wxID_FRAMESECONDPANEL1,
 wxID_FRAMESECONDSTATICTEXT1,
] = [wx.NewId() for _init_ctrls in range(4)]

class FrameSecond(wx.Frame):
    def _init_ctrls(self, prnt):
        # generated method, don't edit
        wx.Frame.__init__(self, 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.panel1 = wx.Panel(id=wxID_FRAMESECONDPANEL1,
name='panel1',
              parent=self, pos=wx.Point(0, 0),
size=wx.Size(419, 236),
              style=wx.TAB_TRAVERSAL)

        self.btnFSExit =
wx.Button(id=wxID_FRAMESECONDBTNFSEXIT, label=u'Exit',
              name=u'btnFSExit', parent=self.panel1,
pos=wx.Point(174, 180),
              size=wx.Size(85, 29), style=0)
        self.btnFSExit.Bind(wx.EVT_BUTTON,
self.OnBtnFSExitButton,
              id=wxID_FRAMESECONDBTNFSEXIT)

        self.staticText1 =
wx.StaticText(id=wxID_FRAMESECONDSTATICTEXT1,
              label=u"Hi there...I'm the second form!",
name='staticText1',
              parent=self.panel1, pos=wx.Point(45, 49),
size=wx.Size(336, 23),
              style=0)
        self.staticText1.SetFont(wx.Font(14, wx.SWISS,
wx.NORMAL, wx.BOLD,
              False, u'Sans'))

    def __init__(self, parent):
        self._init_ctrls(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.

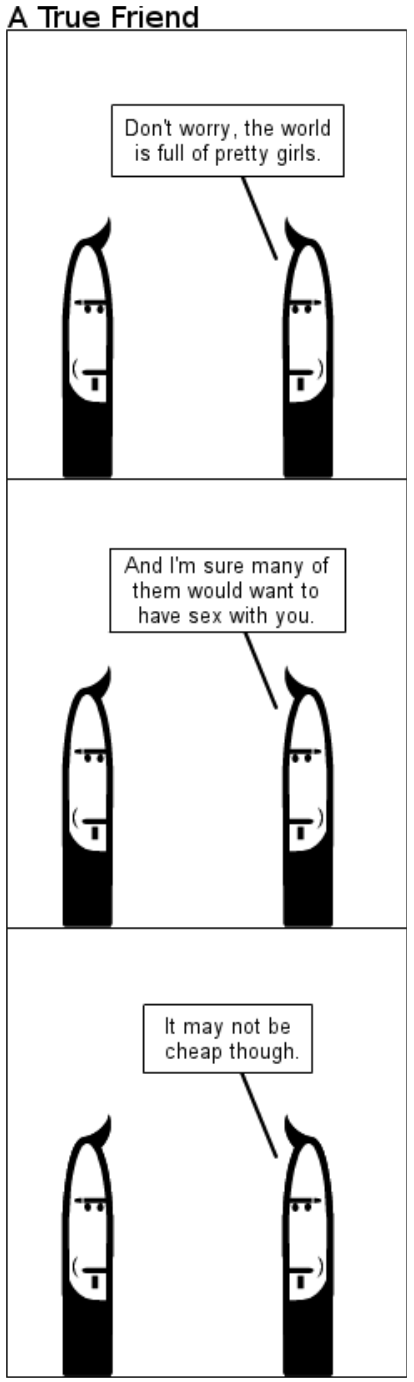
I promised you we'd discuss naming conventions. Remember way back, we discussed commenting your code? Well, by using well-formed names for GUI controls, your code is fairly self-documenting. If you just left control names as staticText1 or button1 or whatever, when you are creating a complex frame with 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_
```

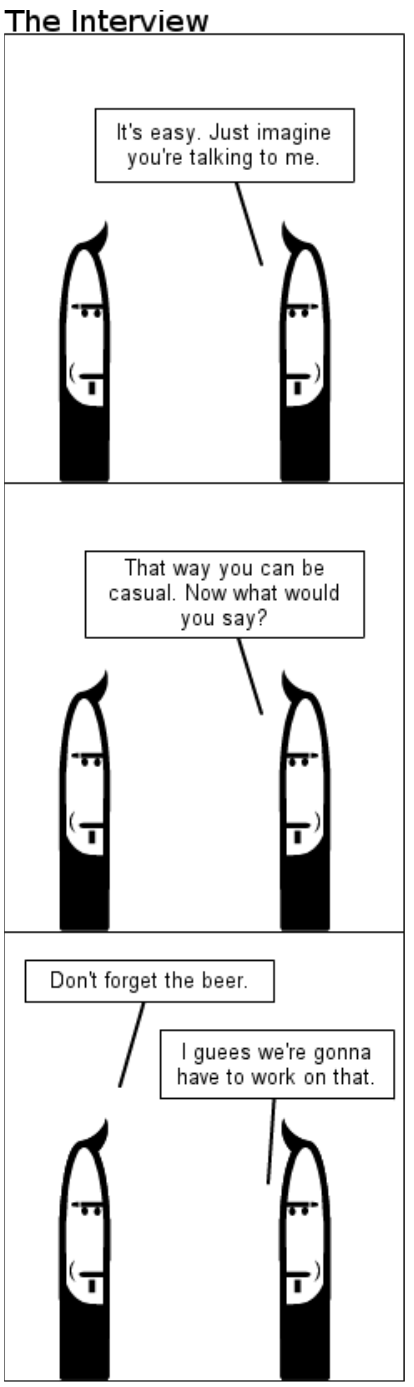
You can come up with your own ideas for naming conventions as you grow as a programmer, and in some instances your employer might have conventions already in place.

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.



by Richard Redei



by Richard Redei

contents ^

Program In Python - Part 7

Written by Greg Walters

SEE ALSO:
FCM#27-32 - Python Parts 1 - 6

APPLICABLE TO:

ubuntu kubuntu xubuntu

CATEGORIES:

Dev Graphics Internet M/media System

DEVICES:

CD/DVD HDD USB Drive Laptop Wireless

Good morning Boys and Girls. It's story time. Everyone get settled and comfy. Ready? Good!

Once upon a time, the world was ruled by paper. Paper, paper everywhere. They had to make special homes for all that paper. They were called filing cabinets, and were big metal things that would take rooms and rooms and rooms at businesses to house all the paper. In each filing cabinet was something called a file folder, which attempted to organize relevant papers together. But after time, they would get over-stuffed, and fall apart when they got old or opened too many times.

Using these filing cabinets properly required a college degree. It could take days to find all the papers that were in the various cabinets. Businesses suffered horribly. It was a very dark time in the history of man- and woman-kind.

Then one day, from the top of a mountain somewhere (I personally think it was Colorado, but I'm not sure), came a lovely fairy. This fairy was blue and silver - with beautiful wings and white hair, and was about 1 foot tall. Her name, believe it or not, was See-Quill. Isn't that a funny name? Anyway, See-Quill said that she could fix everything having to do with all the paper and filing cabinets and wasted time, if only people would believe in computers and her. She called this power a "Database". She said that the "Database" could replace the entire filing system. Some people did, and soon their lives were very happy. Some didn't, and their lives stayed the same, lost in mountains of paper.

All fairy promises, however, come with some sort of requirement. That requirement was that whoever wanted to use the power of See-Quill needed to learn a bit of a different language. It wouldn't be too difficult a language to learn. In fact, it was much like the one the people already used. It just has a different way of saying things, and you had to think about things very carefully BEFORE you said them - to use the power of See-Quill.

One day, a young boy named, curiously enough, User, came to see See-Quill. He was very impressed with her beauty, and said "See-Quill, Please teach me to use your power." See-Quill said that she would.

She said, "First, you have to know how your information is laid out. Show me your papers."

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

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:

```
# 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.

```
RECIPES
------------
pkID (Integer Primary Key)
name (Text)
source (Text)
serves (Text)
```

```
INSTRUCTIONS
----------------------
pkID(Integer Primary Key)
recipeID (Integer)
instructions (Text)
```

```
INGREDIENTS
--------------------
pkID (Integer Primary Key)
recipeID (Integer)
ingredients (Text)
```

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

Next we have to tell ASPW to actually do the sql command:

```
cursor.execute(sql)
```

Now we create the other tables:

```
sql = 'CREATE TABLE
Instructions (pkID INTEGER
PRIMARY KEY, instructions
TEXT, recipeID NUMERIC)'

cursor.execute(sql)

sql = 'CREATE TABLE
Ingredients (pkID INTEGER
PRIMARY KEY, ingredients
TEXT, recipeID NUMERIC)'

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

automatically entered for us, so we don't include that in the list of fields in our insert statement. Since we will be using the field names, they can be in any order, not just the order they were created in. As long as we know the names of the fields, everything will work correctly. The insert statement for our recipe table entry becomes

```
INSERT INTO Recipes (name, serves, source) VALUES (“Spanish Rice”,4,”Greg Walters”)
```

Next we need to find out the value that was assigned to the pkID in the recipe table. We can do this with a simple command:

```
SELECT last_insert_rowid()
```

However, it doesn't just come out as something we can really use. We need to use a series of statements like this:

```
sql = "SELECT last_insert_rowid()"

cursor.execute(sql)

for x in cursor.execute(sql):
    lastid = x[0]
```

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:

```
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)
```

Notice that we are using the variable substitution (%s) to place the pkID of the recipe (lastid) into the sql statement. Finally, we need to put each ingredient into the ingredient table. I'll show you just one for now:

```
sql = 'INSERT INTO Ingredients (recipeID,ingredients) VALUES ( %s,"1 cup parboiled Rice (uncooked)")' % 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.thedesignatedgeek.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.

Program In Python - Part 8

SEE ALSO:
FCM#27-33 - Python Parts 1 - 7

APPLICABLE TO:
ubuntu kubuntu xubuntu

CATEGORIES:
Dev Graphics Internet M/media System

DEVICES:
CD/DVD HDD USB Drive Laptop Wireless

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

```
#!/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 '                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.'
```

our 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:

```
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:

```
/usr/bin/python -u
 "/home/greg/python_examples/APSW/cookbook/cookbook_stub.py"
==========================================================
                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 ->
```

```
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
```

For a number of the menu items, we will want to print out all of the recipes from the Recipe 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.

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

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

```
    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
        cbk.PrintAllRecipes()
    elif response == '5': # Add a recipe
        pass
    elif response == '6': # Print a recipe
        cbk.PrintAllRecipes()
    elif response == '0': # Exit the program
        print 'Goodbye'
        loop = False
    else:
        print 'Unrecognized command.  Try again.'
```

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:

```
Item Name    Serves    Source

-------------------------
```

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'
    cntr = 0
    for x in cursor.execute(sql):
        cntr += 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 = cntr
```

The cntr 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'
        cntr = 0
        for x in cursor.execute(sql):
            cntr += 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 = cntr
```

```
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 Try|Except 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 pkID/name/servings/source, we can use x[0],x[1],x[2] and x[3] as the detail. Then, we want to select everything from the ingredients table where the recipeID (our key into the recipes data table) is equal to the pkID we just used. We loop through the tuple returned, printing each ingredient, and then finally we get the instructions from the instructions table – just like we did for the ingredients table. Finally, we wait for the user to press a key so they can see the recipe on the screen. The code is shown on the next page.

Now, we have two routines

```
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:

```
SELECT * FROM Recipes WHERE
name like '%rice%'
```

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%%'. Sorry if this is as clear as mud. The third query is called a Join statement. Let's look at it a bit closer:

```
sql = "SELECT
r.pkid,r.name,r.servings,r.so
urce,i.ingredients FROM
Recipes r Left Join
ingredients i on (r.pkid =
i.recipeid) WHERE
i.ingredients like '%%%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

```
  def PrintSingleRecipe(self,which):
        sql = 'SELECT * FROM Recipes WHERE pkID = %s' %
str(which)
        print
'~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
        for x in cursor.execute(sql):
            recipeid =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 ''
        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 -> ')
```

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

```
  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%%' GROUP BY r.pkid" %response
            try:
                if parm == '3':
                    print '%s %s %s %s %s'
%('Item'.ljust(5),'Name'.ljust(30),'Serves'.ljust(20),'Source'.ljust(30),'Ingredient'.ljust(30))
                    print '--------------------------------------------------------------------------------'
                else:
                    print '%s %s %s %s' %('Item'.ljust(5),'Name'.ljust(30),'Serves'.ljust(20),'Source'.ljust(30))
                    print '------------------------------------------------------------------------------------'
                for x in cursor.execute(sql):
                    if parm == '3':
                        print '%s %s %s %s %s'
%(str(x[0]).rjust(5),x[1].ljust(30),x[2].ljust(20),x[3].ljust(30),x[4].ljust(30))
                    else:
                        print '%s %s %s %s' %(str(x[0]).rjust(5),x[1].ljust(30),x[3].ljust(20),x[2].ljust(30))
            except:
                print 'An Error Occured'
            print '------------------------------------------------------------------------------------'
            inkey = raw_input('Press a key')
```

anything, just fall through. Notice that I used "!=" as Not Equal To instead of "<>". Either will work under Python 2.x. However, in Python 3.x, it will give a syntax error. We'll cover more Python 3.x changes in a future article. For now, start using "!=" to make your life easier to move to Python 3.x in the future. Finally, we "pretty print" again our output. Let's look at what the user will see, shown right.

You can see how nicely the program prints the output. Now, the user can go back to the menu and use option #3 to print whichever recipe they want to see. Next we will add recipes to our database. Again, we just have to add one line to our menu routine, the call to the EnterNew routine:

```
cbk.EnterNew()
```

The code that needs to replace the stub in the Cookbook class for EnterNew() is at: http://pastebin.com/f1d868e63.

We start by defining a list named "ings" – which stands

```
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 ing 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 1 tag for the title, the

```
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.DeleteRecipe(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 DeleteRecipe(self,which):
       resp = raw_input('Are You SURE you want to
Delete this record? (Y/n) -> ')
        if string.upper(resp) == 'Y':
            sql = "DELETE FROM Recipes WHERE pkID = %s"
% str(which)
            cursor.execute(sql)
            sql = "DELETE FROM Instructions WHERE
recipeID = %s" % str(which)
            cursor.execute(sql)
            sql = "DELETE FROM Ingredients WHERE
recipeID = %s" % str(which)
            cursor.execute(sql)
            print "Recipe information DELETED"
            resp = raw_input('Press A Key -> ')
        else:
            print "Delete Aborted - Returning to menu"
```

<H2> tag for servings and source. We then use the 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 web site, www.thedesignatedgeek.com to get the code.

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

```
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"
```