XBOX PYTHON TUTORIAL

Written by Alex (aka alx5962) and Alexpoet.

Version 2.0

Please notify us if you notice grammatical or typographical errors in this document. We strive to keep the tutorial readable and effective. However, if you have problems running the scripts contained in this document, reread the instructions or ask for help on an appropriate web forum. Please do not email us directly for support. We don't have the time or resources—that's why we made this tutorial. :)

Also, please note that we wrote this tutorial as an introductory lesson, with ease of understanding a major goal. Experienced developers may find this text too simplistic, but it's not written for them :)

Now, on to the instructions.

I. Introduction from Alx5962 (or, "*This* Python is not a Snake")

Welcome to the Python for XBMC Tutorial! This project began one day when I discovered XBMC (XBox Media Center) supports scripts written in Python. Darkie made the port, and I'd like to thank him for his great work and support! (I harassed him with questions and feature suggestions and he was always nice enough to reply to the questions and to add the features.) Curious, I decided to try to use this scripting language to display some basic stuff.

Before I could begin writing for the XBox, I spent many hours learning the Python language (and, in the process, the snake bit me ;), and reading through all the documentation included in the Windows port. Once I'd finished that, I started to code some very basic scripts. I learned XBMC Python through a lot of trial and error. Now I feel more comfortable coding for the XBox, and so I decided to share my experience.

II. Some Basic Rules (or, "Even so…Be Careful of the Snake!")

In order to script for the XBox, you'll obviously need Python installed with XBMC. Most XBMC releases include a python.rar file containing the necessary scripts. (Some full-scale releases come with Python already included in the main installation.)

So unrar the Python file if you need to, and you'll have two folders: "python" and "scripts." Place both of these in XBMC's root directory.

Note that features are always being added, so we really advise you to have the last version of XBMC *and* of Python, otherwise scripts written using newer versions may not work with your installation.

To run Python scripts on your XBox, use the script launcher, which is based in different locations in XBMC depending on your skin. In Project Mayhem, it can be found under "My Files." So go to "My Files," then scroll down to "scripts" and push "A" (or "Select" on the IR Remote). Now it will show you a list of all the Python scripts—as well as any subfolders—in your "scripts" directory. Select a Python script and hit "A" to run it.

(You'll notice that whenever a script is activated, even if it has no GUI window, the word "running" is added next its filename in the script launcher. This will disappear when the script comes to an end.)

Debug information for Python scripts can be found at the script launcher screen by pressing the white button on your controller. Any **print** statements in the script will print out to this debug screen, as will any errors.

Note that you may need internet access to run some scripts, so be sure to configure your installation of XBMC (instructions can be found in the XBMC documentation) and don't forget to set up your nameserver (aka DNS) to resolve domain names. Of course, if you can't get internet access to work, you can still run any Python scripts that aren't internet-dependent.

Sidenote – Scriptionary: Script Management Utility

I've developed a small utility script as an alternative to XBMC's script launcher. It's called Scriptionary, and is available on my website or at the downloads page linked at the end of this document.

Scriptionary is designed to be a simple interface to provide quick access to the scripts you run often, without cluttering the screen with scripts you *don't* use. It's a product still in development, and at the time I'm writing this tutorial, Scriptionary still certainly has its bugs. But you might consider downloading it and trying it out. This utility just might save you some time and hassle, and make your script development a little bit easier.

Alexpoet.

III. Peculiarities of Python (or, "Pay Attention to the Snake's Behaviour")

Python coding is based on indentation. In many languages, levels of indentation are used as a convention for the sake of readability. In Python, indentation actually describes blocks. So you won't use curly braces—"{" and "}"—to declare the start and end of a function, class, or if statement. Instead, you'll declare a function:

```
def myFunction(params):
```

and follow it with code indented by one level:

```
def myFunction(params):
    variable = False
    doThis(params)
    if variable == True:
        doThat(params)
```

You'll notice a second level of indentation following the if statement. In Python, indentation describes all blocks and sub-blocks.

Everything is an object. This aspect of the language is very nice sometimes, but it can be tricky for beginners! Just remember that, in Python, everything is an object. As you get further into scripting, you'll learn the full implications of this fact.

When assigned, a variable is considered local unless you declare it global. This rule comes up often, although it's not covered within the scope of this tutorial. Still, it's a helpful fact to know, especially when you start reading through other people's scripts.

The goal of this document is not to teach Python, though, but to teach you how to write Python for the XBox. So instead of going into further detail here, we recommend that you read the fine documentation available on www.python.org.

IV. Tutorial: Writing XBMC Python Scripts (or, "The Real Work Begins")

There are two special libraries for Python scripts only available in XBMC: xbmc and xbmcgui. They are dedicated to the user interface, keypad management, and fundamental interaction with XBMC itself. Artificial emulators of these libraries are under development, to allow you to test xbmc- and xbmcgui-dependent code on a PC. For more details, visit the XBMC Python forum and look for the thread "Dev Tool: XBMC Emulator Scripts," or visit Alexpoet's website (the address is at the end of this document).

In the course of this tutorial, we will only address scripts that use a graphical interface, as our primary purpose is to introduce the use of the xbmc and xbmcgui libraries. Console scripts that work without these libraries are outside the scope of this document.

Throughout this text, Python code will be coloured in blue. Tutorial segment headings are shown in **boldface green**.

Creating a Window

The first step in writing a script for XBMC is to import the xbmc libraries:

```
import xbmc, xbmcgui
```

After that, we need to create a class (defined by the keyword "**class**") that will include some of its own functions (defined by the keyword "**def**")

```
class MyClass(xbmcgui.Window):
    print "hello world"
```

MyClass is now set up to act like a Window (as described in the xbmcgui library). Once we've defined the class, we need to initialize the class object (that is, create an instance of it) and then run the library's function doModal, which causes the graphical window to continue displaying on the screen until we close it.

```
mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Note: The final command "**del**" is here for "garbage collection"; it keeps our code clean by deleting the class instance that we created.

Now, put all of this code, in order, into a script, and call it "display.py". **It should look like this:**

```
import xbmc, xbmcgui

class MyClass(xbmcgui.Window):
    print "hello world"

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

(All we did was combine the lines above, to make it easier to read.)

Now, to test your script, you need to copy it to your XBox. We recommend you go to your XBMC installation folder, then find the "scripts" subfolder, and create a folder called "Tutorial" within scripts. Now copy your new display.py script to that folder, as well as the background.gif image that should have come attached to this document. (If you don't have it, don't worry. It's a very simple 720 x 480 image and you can easily make your own. We'll use it later in this text.)

Once you've got the script in place, you can run it through XBMC, using the script launcher (as described above). When you run it, all you'll see is an empty window, which is the same window you

just created with your script! There's one problem, though. Since there's no code to exit the MyClass class, you'll have to reset the XBox to escape the script.

Also notice that the **print** function only displays output in debug mode, not anywhere on the main GUI screen. To access debug mode, once you've exited the script press the white button, and you'll see a list of all the script output generated since you last booted your XBox. Press the white button again to clear this screen, or "Back" to return to the scripts menu.

Responding to the Controller/Keypad (and Creating an "Exit" Option)

Now that we've got a working window taking over our screen, we need to implement a way to exit the display. We'll use the controller/keypad for that, by writing some code that will capture the press of the button on the keypad and respond to it.

It's pretty simple. Open up "display.py" and remove the third line (print "hello world"). We're going to replace it with a function that captures controller actions.

First, add this before you create the class:

```
#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
```

The first line is a comment–comments in Python are preceded by "#"–which means it acts as a note to the programmer, but doesn't actually do anything when the script is run. This one is telling you *where* we found out that the number 10 is what you get when someone pushes the "Back" button on the XBox controller.

Now you have to write a function that will *use* this information. Inside the MyClass block (where the print statement used to be), add this function:

```
    def onAction(self, action):
            if action == ACTION_PREVIOUS_MENU:
                    self.close()
```

Once you've added that, you've got a script that can respond to controller actions. You've also now written an "Exit" option into your script, so users can exit it.

Here's what the full code should look like:

```
import xbmc, xbmcgui

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10

class MyClass(xbmcgui.Window):
    def onAction(self, action):
            if action == ACTION_PREVIOUS_MENU:
                    self.close()

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

You can see we've only really added four lines to the script, but now it's interactive! Oh, sure, it doesn't do much, but it's a beginning. Now, each time we push the "Back" button, we will exit the class and (therefore) the script.

Now's a good time to point out how Python's use of indentation makes these scripts so readable. The flow of the script is clear, and it's easy to find where each logical block starts and ends.

Also, note again that the keyword **def** defines a function. Here we have a special instance, though, because onAction is a function call built into the XBMC library. In any script, XBMC calls the onAction function whenever it receives any keypad-related actions.

Also notice what we use onAction for: to call the command self.close(). This is another function built into xbmc that will close the window and so the class, ending the doModal() loop.

Control Objects - Adding a Text Label

Now, a GUI is obviously about more than input. We'll need some output, too. So it's time for us to learn how to display text on the window we've created.

To do that, we'll use the ControlLabel function. ControlLabel has some properties including position, font colour and transparency, and font size. The following three lines act as a single "block" to create and display a line of text on the window:

```
self.strAction = xbmcgui.ControlLabel(300, 200, 200, 200, "", "font14", "0xFFFFFF00")
self.addControl(self.strAction)
self.strAction.setLabel("BACK to quit")
```

There is a reason we have to write three lines just to create a single line of text on the screen, but that will become more obvious as we get further into the tutorial. For now, think of it like this: First we tell the program what the Control is going to look like (with the xbmcgui.ControlLabel function, and all of its parameters), then we tell the program to *make* that control (with the addControl function), and then we tell the Control what to do (in this case, display the line "BACK to quit").

So, when we start by describing the control, we pass it a bunch of numbers, and a couple of strings. These define the Control object's position, size, and unique characteristics. In this example, we have:

300 is the control's X position on the screen
520 is the control's Y position on the screen
200, 200 is supposed to be size of the element (but it seems to not work with text)
"" is an empty string that can be used to set an initial label text, but we are setting the label later.
"font14" is the font used for this control's text, (the fonts available vary by skin)

The final element is "0xFFFFFF00" – this value may look familiar to you. It represents the colour value and transparency of the label's font, coded in hexadecimal (from 00 to FF). So read this as 0xTTRRGGBB where T is the transparency value, R is red, G is green and as you guessed B is blue. When we add text to the label, it will show up on the screen in the color defined by this value.

Now we have a working script that could include a label telling us how to use its controls (well...just one control, really, but who's counting?). Instead, let's build a slightly different label and associate it with the A button of the keypad, for more practice. First, add this line near the top of your script:

```
ACTION_SELECT_ITEM = 7
```

We used a similar line earlier to identify when the user pushed the "Back" button–that was 10–and now we're using this one to get the "A" button, which is a value of 7.

Now, as you remember from the last segment, the way we use these values is in the special function onAction, by telling the script how to respond when XBMC tells us the "A" button was pushed. So we add this "if" statement to the onAction function we already have:

```
            if action == ACTION_SELECT_ITEM:
                self.strAction = xbmcgui.ControlLabel(300, 200, 200, 200, "", "font14", "0xFF00FF00")
                self.addControl(self.strAction)
                self.strAction.setLabel("Hello world")
```

Once you add these lines to the script, it will show your new label whenever the "A" button is pressed. **The whole script should look like this:**

```
import xbmc, xbmcgui

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
ACTION_SELECT_ITEM = 7

class MyClass(xbmcgui.Window):
    def onAction(self, action):
            if action == ACTION_PREVIOUS_MENU:
                self.close()
            if action == ACTION_SELECT_ITEM:
                self.strAction = xbmcgui.ControlLabel(300, 200, 200, 200, "", "font14", "0xFF00FF00")
                self.addControl(self.strAction)
                self.strAction.setLabel("Hello world")

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Copy this script to your XBox and run it. Try pressing the "A" button and see what happens. Don't forget you have to push "Back" to stop the script!

Control Objects - Removing a Text Label

In “xbmcgui.py” you get a handful of widgets that are called Control objects–these are the Button, Image, Label, FadeLabel, List, and TextBox). All Control objects share certain characteristics. Control object are GUI devices that are drawn over the window, and they’re the main tools you’ll use to interact with the script. As mentioned in the previous segment, you initialize Control objects with a function call (for instance: `self.strAction = xbmcgui.ControlLabel(300, 200, 200, 200, "", "font14", "0xFF00FF00")`), and then tell xbmc to draw them with the addControl function.

Well, we can also tell xbmc to stop drawing any Control, which will remove it from the display. To do this, use the function removeControl.

Let’s stick with the same script, but add another option: pressing the "B" button will remove the Label that was created when you pressed "A". To do that, you have to recognize the "B" button:

```
ACTION_PARENT_DIR = 9
```

and then use these two lines in the onAction function:

```
            if action == ACTION_PARENT_DIR:
                self.removeControl(self.strAction)
```

When you’ve add these three lines, you should have a working script. **The whole script should look like this:**

```
import xbmc, xbmcgui

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
ACTION_SELECT_ITEM = 7
ACTION_PARENT_DIR = 9

class MyClass(xbmcgui.Window):
    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()
        if action == ACTION_SELECT_ITEM:
            self.strAction = xbmcgui.ControlLabel(300, 200, 200, 200, "", "font14", "0xFF00FF00")
            self.addControl(self.strAction)
            self.strAction.setLabel("Hello world")
        if action == ACTION_PARENT_DIR:
            self.removeControl(self.strAction)

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Control Objects - FadeLabel

You also have another way to display text: a Control object called the FadeLabel. If you've used XBMC much at all, you're probably familiar with these objects. A FadeLabel is a segment of your screen that will scroll text from left to right within its borders, and repeat the text when it gets to the end.

You've probably seen this object used for RSS feeds, or perhaps to show the song artist when XBMC is playing an mp3.

You can do the same thing with a fairly simple script. You set up the FadeLabel much like you would a Label, but you don't have to worry about the length of the string—anything that doesn't fit into the space provided will scroll into view.

The FadeLabel object also has a reset() function, which clears out all the text and leaves the FadeLabel in place, so you can still send new text to it later (unlike the removeControl function, which would clear the text from the screen but also destroy the FadeLabel object).

As we said, it's pretty easy to make a FadeLabel. This time we're going to make some big changes from the script we used last time. First, we're going to add a new function called __init__ right after declaring the class. The __init__ function is a special case in Python, it's a built-in function type (similar to a constructor in other languages) that runs automatically whenever you create an instance of a class. We'll talk more about that in the next segment.

For now, we'll just show you how to make one. Add these seven lines right after you create the class:

```
    def __init__(self):
        self.strActionInfo = xbmcgui.ControlLabel(100, 120, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit - A to reset text")
        self.strActionFade = xbmcgui.ControlFadeLabel(200, 300, 200, 200, "font13", "0xFFFFFF00")
        self.addControl(self.strActionFade)
        self.strActionFade.addLabel("This is a fade label")
```

By now, you can probably glance at the first four lines and know what they do. The first declares that it's the beginning of a new function, called __init__ (the "self" inside the parameters means that this function is part of a class, and requires that class instance to run). The next three lines are just like what we were doing two segments earlier: creating a Label.

You can tell what this one does. It gives the user some basic information on how to run the program. The message it will show is "Push BACK to quit - A to reset text." Since it's in the __init__ function, this text will appear as soon as the class is created, which means as soon as the script is run. So the user will know what to do from the very start.

Now, the next three lines define the FadeLabel, and as you can see, they're almost identical. The only difference is the function call xbmcgui.ControlFadeLabel instead of xbmcgui.ControlLabel. That's all you have to do to make a text string that will scroll within the box.

FadeLabel has a very useful function called reset(), though, so let's go ahead and write this script to show you how to use that. You're already recognizing the "A" button press from earlier, so we'll stick with that.

We need to change onAction now, though, since it's doing something different. In your onAction function, leave the first if statement (the one that lets you close the script), but replace the other two with this one:

```
if action == ACTION_SELECT_ITEM:
    self.strActionFade.reset()
```

Now you can press "A" while the script is running to see how a FadeLabel's reset() function works. You should also have a pretty good idea by now how the special onAction function works. Also, you'll notice we no longer have an "if action ==" statement looking for the "B" button, so you can remove that line from the declaration of globals up above.

When you do all that, **the whole script should look like this:**

```
import xbmc, xbmcgui

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
ACTION_SELECT_ITEM = 7

class MyClass(xbmcgui.Window):
    def __init__(self):
        self.strActionInfo = xbmcgui.ControlLabel(100, 120, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit - A to reset text")
        self.strActionFade = xbmcgui.ControlFadeLabel(200, 300, 200, 200, "font13", "0xFFFFFF00")
        self.addControl(self.strActionFade)
        self.strActionFade.addLabel("This is a fade label")

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()
        if action == ACTION_SELECT_ITEM:
            self.strActionFade.reset()

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Try it on the XBox and see what happens. Press the "A" button while you're at it. Now try changing the text of the FadeLabel to something longer, such as "Behold, here you see before you the first and only almighty FadeLabel of Doom which I, in my Pythonic coolness, have created by my very will."

Something restrained and reasonable like that. Then run the script again, and watch it scroll. Press "A". There you go. Not a lot of exciting action, but now you're familiar with the FadeLabel, which is a very useful tool when trying to display long strings.

Sidenote – XBMC Emulator Scripts

Did you try running this script on your PC, using Alexpoet's emulator scripts? If so, you probably got an error. That's because we used the __init__ function for the first time in this script.

When we define an __init__ function in our scripts, it replaces the __init__ function that the Emulator uses to draw the GUI window on your PC. There is still an easy way to get around this problem, but it requires two extra lines in your script that you wouldn't need if you weren't using the emulator.

First, you have to find out whether the script is running on a PC or on the XBox. To do this, add the following lines right after your import statements:

```
try: Emulating = xbmcgui.Emulating
except: Emulating = False
```

Now, as the very first line of your __init__ function, add this line:

```
if Emulating: xbmcgui.Window.__init__(self)
```

That's all it takes. What this does, is tell the script to call the Emulator's __init__ function, too (the one replaced by *your* class's __init__ function), before going on with running. Of course, if you're running the script on the XBox, it will just skip right past and do nothing.

For the sake of those of you using the Emulator in future development, I'm going to include these necessary lines in all of the rest of the tutorial scripts. Note that the Emulator has been developed in the time since Alex first wrote this tutorial, and so these extra lines aren't part of the original tutorial. However, they won't in any way interfere with the scripts' function on the XBox, so I see no harm in including them.

Alexpoet.

Setting up Default Parameters in the Init Function

We already talked some in the last segment about the __init__ function. As we said before, the __init__ function is one that Python automatically runs whenever the class "MyClass" is launched. This is an extremely useful tool for adding elements that we need when the script is first initialized, particularly for adding basic GUI items, such as a background image or text instructions that need to be on screen all the time.

To add these automatic elements, we use the __init__ function, as you've already seen. Let's change the __init__ function from our last script slightly. We'll remove the FadeLabel (now that you know all about them), and have the instructional Label say something different:

```
self.strActionInfo = xbmcgui.ControlLabel(100, 120, 200, 200, "", "font13", "0xFFFF00FF")
self.addControl(self.strActionInfo)
self.strActionInfo.setLabel("Push BACK to quit, A to display text and B to erase it")
```

(We're also adding two more lines—one up above with the globals, and one in the __init__ functions—that will make the PC Emulator scripts work. For details, read the Sidenote above.)

You already know what this part of the script does—we talked about it in the last segment. Of course, now we're changing what the controls do, so we've changed the Label's text.

Let's go ahead and do something new, then. In much the same way as we add Labels, we can also add a background image to help define the script. To do this, we'll add an instance of the ControlImage object to our init function with this one line:

```
self.addControl(xbmcgui.ControlImage(0,0,720,480, "background.gif"))
```

That might look strange to you, because we're doing a lot here in one line. You see we start off with the function self.addControl, and inside the parentheses, instead of giving it the name of a Control object, we create the Control object right there. When you add a Control in one line like this, it has no name, and so there's no way to modify it later. That's why we use three lines to add (and name) a Label, and then modify the Label's text. With a background picture, though, you only show it once, and then leave it unchanged until the script closes.

You could just as well have written this script as two lines:

```
pic = xbmcgui.ControlImage(0,0,720,480, "background.gif")
self.addControl(pic)
```

Either way, it does the same thing, but for Controls you're only drawing and then leaving alone, it's often cleaner to just add them in one line.

But that brings up another issue: where in the __init__ function do we place that line? It's a very important question, and you need to remember to draw the background image before the text, that way the image will be drawn, and then the text placed *over* the image. If we changed that order, the text would appear *behind* the image, so the user would never see it.

Now, for this to work you'll have to *have* an image file by the name of "background.gif" (or, of course, you could replace the filename in the script with the name of a file you have). This tutorial might have come zipped up with a background.gif image, but if not you can create a very simple background just by making a picture 800 pixels by 600 pixels, and saving it with that filename.

You'll have to play with the background colour some to find one that clearly shows the font colour, and still looks good on screen. [Alexpoet's note: Of course, you might end up spending a lot of time doing that once you start developing scripts anyway. I sometimes find myself spending twice as long in Photoshop trying to get good support pics as I spend actually writing these scripts.]

NOTE:

When you enter the filename in the xbmcgui.ControlImage function, you can enter just a filename (and Python will look for the file in the same folder as the script you're running), or you can enter a path. Remember that when you're entering directory paths in Python, you always have to replace "\" with "\\", because Python uses "\" to describe special characters.

So now we know how to build our __init__ function, let's put it into use. Everything else should be simple stuff that you know how to do by now. Put back in the line for recognizing the "B" button, and then change the onAction to draw a friendly message whenever you press "A", then delete it on "B".

Try to write this yourself. If you're having trouble, feel free to look ahead, but this is all stuff we've covered before. When you're done, **the whole script should look like this:**

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
ACTION_SELECT_ITEM = 7
ACTION_PARENT_DIR = 9

class MyClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit, A to display text and B to erase it")

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()
        if action == ACTION_SELECT_ITEM:
            self.strAction = xbmcgui.ControlLabel(300, 300, 200, 200, "", "font14", "0xFF00FF00")
            self.addControl(self.strAction)
            self.strAction.setLabel("Hello world")
        if action == ACTION_PARENT_DIR:
            self.removeControl(self.strAction)

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Creating Basic Dialogs - The "OK" Box

We can also build dialog boxes. XBMCGui provides us with three kinds of dialog box: "ok", "select", and "yesno".

The most basic of these is the "ok" box (although they're all pretty simple). So we'll start with that. First write a function that will create the dialog box:

```
    def message(self):
        dialog = xbmcgui.Dialog()
        dialog.ok(" My message title", " This is a nice message")
```

You can see first we create the dialog instance, and then we make the type in a separate line. You would do the same thing if you were creating a "select" box or a "yesno". They all begin with the line: `dialog = xbmcgui.Dialog()`.

Now we need to build a function that will make the dialog box appear. To do this, change your onAction function to include these lines:

```
        if action == ACTION_SELECT_ITEM:
            self.message()
```

You can also strip out the onAction command for ACTION_PARENT_DIR, since we're not using the "B" button in this script.

Be sure to change the instructions in strActionInfo in your __init__ function, and that's all there is to it. **The whole script should look like this:**

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
ACTION_SELECT_ITEM = 7

class MyClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit or A to display dialog box")

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()
        if action == ACTION_SELECT_ITEM:
            self.message()

    def message(self):
        dialog = xbmcgui.Dialog()
        dialog.ok(" My message title", " This is a nice message ")

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Using Basic Dialogs - More on the "OK" Box

Now that you've seen it in action, let's look more closely at the code that creates an "ok" dialog:

```
dialog = xbmcgui.Dialog()
dialog.ok(" My message title", " This is a nice message ")
```

That's the code from our last segment. You can see the first line tells xbmcgui to create an instance of a Dialog class. The second line creates the actual "ok" dialog, and passes it two strings: one for the title, and one for the box's message.

Once you know how the Dialog Boxes work, they're a very simple way of providing information to the user. Let's rewrite our script to use the same message function in a more general way:

First, change the dialog creation code to look like this:

```
    def message(self, messageText):
        dialog = xbmcgui.Dialog()
        dialog.ok(" My message title", messageText)
```

Notice the difference. In the third line, we're not passing two strings, but a string and a variable. Of course, that variable (messageText) must *contain* a string, or it will break the script when run. But messageText is passed in as one of the parameters of the message function.

Which means we have to change the place where the message function is called, up in onAction:

```
            if action == ACTION_PREVIOUS_MENU:
                    self.message("goodbye")
                    self.close()
            if action == ACTION_SELECT_ITEM:
                    self.message("you pushed A")
```

Compare this onAction with the onAction code from our last segment. We're no longer just calling the function, but passing it a string (which is, basically, the message). So now you know how to call a function inside another function :)

This time we can leave the action globals alone, as well as the strActionInfo instructions. Nothing changes except the way the script runs, and what it outputs. **So the whole script should look like this:**

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
ACTION_SELECT_ITEM = 7

class MyClass(xbmcgui.Window):
    def __init__(self):
            if Emulating: xbmcgui.Window.__init__(self)
            self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
            self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
            self.addControl(self.strActionInfo)
            self.strActionInfo.setLabel("Push BACK to quit or A to display dialog box")

    def onAction(self, action):
            if action == ACTION_PREVIOUS_MENU:
                    self.message("Goodbye")
                    self.close()
            if action == ACTION_SELECT_ITEM:
                    self.message("You pushed A")

    def message(self, messageText):
            dialog = xbmcgui.Dialog()
            dialog.ok("My message title", messageText)

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

This would be a helpful script to change around some on your own, to make sure you know how everything works. Maybe try to write a script that will pop-up a dialog box telling you which button you pressed.

Be careful with the message function, though. Make sure the only values you pass to it are strings. Ints and floats can be converted to strings explicitly (ie: `number = str(num)`), but you can't feed the script a non-string value when it expects a string. It'll throw an error (and on the XBox, that almost certainly means a lock-up).

The only exception to this is Python's built-in print function. `print num` or `print str(num)` will work equally well, but that's designed for debugging output. For any other function that expects a string, passing it ints or floats won't work without an explicit conversion!

Creating Basic Dialogs - The "YesNo" Box

Another fairly standard dialog is the yes / no dialog box, which you'll use when asking for simple user feedback. It works exactly like an "ok" dialog, except that it returns a value of True, if the user selects "Yes" and False if the user selects "No".

So instead of:

```
def message(self, messageText):
        dialog = xbmcgui.Dialog()
        dialog.ok(" My message title", messageText)
```

We type:

```
def goodbye(self):
        dialog = xbmcgui.Dialog()
        if dialog.yesno("message", "do you want to leave?"):
                self.close()
```

You can see the line creating the yesno dialog begins with "if" and ends with a ":". This is the same as saying "`if dialog.yesno("message", "do you want to leave?") == True:`". The line that follows (which, you can see, is indented another level) will only run if the user selects "Yes" from the dialog box.

Now set up onAction to call this function *instead of* self.close() when the "Back" button is pressed. That's really all that needs changing. To keep things clear, let's remove the onAction for the "A" button, and rewrite our strActionInfo instructions. Three cheers for minimalism.

The whole script should look like this:

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10

class MyClass(xbmcgui.Window):
    def __init__(self):
            if Emulating: xbmcgui.Window.__init__(self)
            self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
            self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
            self.addControl(self.strActionInfo)
            self.strActionInfo.setLabel("Push BACK to quit.")

    def onAction(self, action):
            if action == ACTION_PREVIOUS_MENU:
                    self.goodbye()

    def goodbye(self):
            dialog = xbmcgui.Dialog()
            if dialog.yesno("Message", "Do you want to leave?"):
                    self.close()

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Creating Basic Dialogs - The "Select" Box

The final and most complex of the basic dialogs is the "select" box. This dialog contains a list of possible choices. The user can scroll up and down through the list, and select one of them. Drawing the

dialog, just like the other two, is pretty simple. This time you pass it a title, a message (to let the user know what he's choosing), and a Python list containing the possible choices (all as strings, by the way).

So all we *really* need to get information from the "select" box is this:

```
def chooseOne(self):
        possibleChoices = ["Yes", "Probably", "Maybe", "Probably Not", "No"]
        dialog = xbmcgui.Dialog()
        choice = dialog.select("Do you think this will work?", possibleChoices)
```

There we have a list of strings, the creation of a Dialog instance, and then the dialog.select() call which will return the user-selected value (and save it in the variable choice). We can replace the goodbye function with this one.

Here's where the "select" box gets a little tricky, though. The value that it returns is *not* the selected item; it's the list index of the selected item. That means, if we ran the code above, and the user selected "Yes," then choice would be equal to 0, not the word "Yes". (In Python, as in most programming languages, counting begins at 0, so the first item in the list is 0, and the fifth item in a list is 4 – you'll get used to it in time).

If you're familiar with lists and list indexes and not too worried about getting confused by this one, then skip to the whole script at the end of this segment, read through it, and try it on the XBox. But I'm going to take a moment here and explain in greater detail.

Okay...to make things more clear, let's add another Label to our script, so we can see the value of our variable choice. We'll put it up in the __init__ function, then set its value right in the chooseOne function. To begin with, it will be empty, but when a user selects one of the items from the dialog, it will show the value returned.

So up in the script's __init__, add:

```
        self.choiceLabel = xbmcgui.ControlLabel(300, 300, 100, 100, "", "font13", "0xFFFF00FF")
        self.addControl(self.choiceLabel)
```

And then add the label's output line at the end of the chooseOne function:

```
        self.choiceLabel.setLabel("Choice: " + str(choice))
```

That's all we need to use a "select" box and see the returned value, but we've got to change the onAction to make it work. Add ACTION_SELECT_ITEM back into the globals, and change onAction to look like this:

```
def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
                self.close()
        if action == ACTION_SELECT_ITEM:
                self.chooseOne()
```

We're not finished yet, this is just for demonstration, but if you put the whole script together, it would look like this:

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
ACTION_SELECT_ITEM = 7

class MyClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit or A to choose from a select dialog.")
        self.choiceLabel = xbmcgui.ControlLabel(300, 300, 100, 100, "", "font13", "0xFFFF00FF")
        self.addControl(self.choiceLabel)

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()
        if action == ACTION_SELECT_ITEM:
            self.chooseOne()

    def chooseOne(self):
        possibleChoices = ["Yes", "Probably", "Maybe", "Probably Not", "No"]
        dialog = xbmcgui.Dialog()
        choice = dialog.select("Do you think this will work?", possibleChoices)
        self.choiceLabel.setLabel("Choice: " + str(choice))

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Try this out on your XBox. When you select "Yes" in the dialog, the output says "0" (just like I said it would :-). That's how a select dialog works.

It's not a hard problem to fix, though. Whenever you make a "select" dialog, you already have the list available, and the list index tells you where to find the item in the list (that's the whole point).

So let's change the third line in chooseOne to say:

```
        ndex = dialog.select("Do you think this will work?", possibleChoices)
```

And add a line right after that that says:

```
        choice = possibleChoices[ndex]
```

Now run the script again, and see what the label says. It should print out exactly what you selected from the list. You can also remove the "`str(choice)`" from the line after that (so it's just "`choice`"), because the variable choice is now already set to a string (when it was a list index, it was an integer).

The whole script should look like this:

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
ACTION_SELECT_ITEM = 7

class MyClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit or A to choose from a select dialog.")
        self.choiceLabel = xbmcgui.ControlLabel(300, 300, 100, 100, "", "font13", "0xFFFF00FF")
        self.addControl(self.choiceLabel)

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()
        if action == ACTION_SELECT_ITEM:
            self.chooseOne()

    def chooseOne(self):
        possibleChoices = ["Yes", "Probably", "Maybe", "Probably Not", "No"]
        dialog = xbmcgui.Dialog()
        ndex = dialog.select("Do you think this will work?", possibleChoices)
        choice = possibleChoices[ndex]
        self.choiceLabel.setLabel("Choice: " + str(choice))

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Control Objects – Adding a GUI Button

Okay, enough of dialog boxes! It's time to get back to making widgets, and making your scripts do more. Let's build a button on the GUI and see how it works!

Buiding a button works much like any other Control Object, using the xbmcgui.ControlButton function. It takes 5 arguments, like this: `xbmcgui.ControlButton(350, 400, 80, 30, "HELLO")`.

Again, we start with the X and Y positions (350 and 400 respectively) and then the width and height (80 and 30 here) and the last is a text string that will show up as a label on the button.

So we'll add the extra lines necessary to make a working Control Object. Unlike the Labels we've worked with before, after you build a Button it's important to make xbmcgui focus on it:

```
self.button0 = xbmcgui.ControlButton(350, 400, 80, 30, "HELLO")
self.addControl(self.button0)
self.setFocus(self.button0)
```

As you can see, "setFocus" is the function that does it. When you add more than one button to a script, you'll only need to setFocus on one of them (whichever you want to start out selected), but get in the habit of setting the focus at least once per script, if you're using any buttons at all.

Now that you've got a button built, you have to tell the script what to do when the button is activated. A GUI Button is activated by pressing "A" on the controller when a Button (or other usable Control Object, like a List) is selected.

You already know that pushing the "A" button will call the script's onAction function, and pass it the value of the "A" button. If a usable Control Object is selected, it will *also* call the script's onControl function, and pass it the Control Object that was used.

This function (onControl) is another special built-in function that handles events, just like onAction. Most advanced scripts tend to contain both an onAction function to handle controller keypresses and an onControl function to handle GUI events. Here we'll build our first onControl.

First, remove the second "if" statement from your onAction function. You don't *want* to handle the "A" button in your onAction anymore—now you want to let "A" call onControl. We're going to get rid of the chooseOne function, too, and replace it with a simpler function we wrote earlier: the basic message function, that opened an "ok" box. We'll use that for output (which means you can take "choiceLabel" out of the __init__ function, too).

Now add these lines just after the onAction function in your script:

```
def onControl(self, control):
    if control == self.button0:
        self.outputLabel.setLabel( "Button Pushed")
```

And that's a working Button, with working events already triggered! **The whole script should look like this:**

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10

class MyClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit. Push the button if you absolutely must.")
        self.button0 = xbmcgui.ControlButton(350, 400, 80, 30, "Push Me!")
        self.addControl(self.button0)
        self.setFocus(self.button0)

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()

    def onControl(self, control):
        if control == self.button0:
            self.message("You pushed the button.")

    def message(self, messageText):
        dialog = xbmcgui.Dialog()
        dialog.ok(" My message title", messageText)

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Note that you can also remove the button from the screen using removeControl:

```
self.removeControl(self.button0)
```

We've discussed removeControl before. It works the same way with a Button, removing it from the display. Try adding a new command in onAction that will respond to the "B" button by making our Button object disappear.

Control Objects – Associating Multiple Buttons

Of course, one button is rarely enough to keep an interesting script working. You can add as many buttons as you can find room for on the screen, but in order for them to work you have to give the user some way to move among them.

One way, of course, is with the setFocus command. You've seen how that works earlier. You *could* write some complicated script that cycles the focus among the Button objects whenever the user presses "White" or something, but there's a much easier way: associating Control Objects.

Associating Control Objects is done by explicitly telling your script which objects are next to any given object—and in which direction. You do this using the controlUp, controlDown, controlLeft, and controlRight functions on a Control Object.

Let's say you've got a button named "ButtonLeft" with another beside it named "ButtonRight". They're already made, and ready to use, but you decide to associate them. You'd write code that looked something like this:

```
ButtonLeft.controlRight(ButtonRight)
ButtonRight.controlLeft(ButtonLeft)
```

This tells the script that the button to the right of ButtonLeft is ButtonRight. Makes sense. You could even make it circular, so if you go right to select ButtonRight, and then move off-screen to the right, it'll come back around to ButtonLeft. That'd look like this:

```
ButtonLeft.controlRight(ButtonRight)
ButtonLeft.controlLeft(ButtonRight)
ButtonRight.controlRight(ButtonLeft)
ButtonRight.controlLeft(ButtonLeft)
```

You probably wouldn't bother with that, though. Mostly you're only going to set up the least you have to that will still allow users to move around your script. But it helps to think through everything you could do. Also, try to always make associations that make sense graphically on the page. Don't use controlUp to get to a Control directly to your right, unless it somehow makes sense in your script.

Okay, so far these are general examples, but we've explained what really goes into making it work. And nothing else in this segment is particularly new, so we'll skip straight to the end. **The whole script should look like this:**

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10

class MyClass(xbmcgui.Window):
    def __init__(self):
            if Emulating: xbmcgui.Window.__init__(self)
            self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
            self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
            self.addControl(self.strActionInfo)
            self.strActionInfo.setLabel("Push BACK to quit.")

            # Make all the buttons
            self.button0 = xbmcgui.ControlButton(250, 150, 120, 30, "1. Push Me!")
            self.addControl(self.button0)
            self.button1 = xbmcgui.ControlButton(250, 250, 120, 30, "2. Push Me!")
            self.addControl(self.button1)
            self.button2 = xbmcgui.ControlButton(450, 250, 120, 30, "3. Push Me!")
            self.addControl(self.button2)
            self.setFocus(self.button0)
            self.button0.controlDown(self.button1)
            self.button1.controlUp(self.button0)
            self.button1.controlRight(self.button2)
            self.button2.controlLeft(self.button1)

    def onAction(self, action):
            if action == ACTION_PREVIOUS_MENU:
                    self.close()

    def onControl(self, control):
            if control == self.button0:
                    self.message("You pushed the first button.")
            if control == self.button1:
                    self.message("You pushed the second button.")
            if control == self.button2:
                    self.message("You pushed the third button.")

    def message(self, messageText):
            dialog = xbmcgui.Dialog()
            dialog.ok(" My message title", messageText)

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

You can see we added all three Buttons in the __init__ function, and then used setFocus just once to start out with button0 selected. *Then* we added the associations to allow users to switch which Button was selected. And finally we adjusted onControl to make sure it had something to do for any Button pushed.

Control Objects – Lists (Sometimes Called "Listboxes"...but not by XBMC)

The last of our Control Object is the List. A List creates an area on the display capable of taking ListItems (strings that will appear as individual lines within the List's area) which are selectable. A List that has the script's focus will have a horizontal bar showing which item in the list is selected.

Creating a List instance is much like any other Control Object. Here's a sample:

```
self.list = xbmcgui.ControlList(200, 150, 300, 300)
self.addControl(self.list)
self.list.addItem("Item 1")
self.list.addItem("Item 2")
self.list.addItem("Item 3")
self.setFocus(self.list)
```

I don't think you'll have any trouble guessing what this does. The text strings passed to self.list.addItem will appear in the List in the same order they're added in the script. Of course, you could add these items to a Python list, and then use a Python for loop to add them all, like this:

```
self.list = xbmcgui.ControlList(200, 150, 300, 300)
self.addControl(self.list)
items = ["Item 1", "Item 2", "Item 3"]
for item in items:
        self.list.addItem(item)
self.setFocus(self.list)
```

If you're familiar with Python, that will be familiar to you. Of course, if you're familiar with Python, you'd probably already figured that out. If that last set of code doesn't make sense to you, don't worry about it. We'll go ahead and introduce the items one at a time in our official script. But when you get into more advanced scripting, you'll find that most ListItems are added out of a Python list.

So, now we've got a List containing three different items. A List, like a Button, is a usable Control. Any time a List has focus and the user presses "A", XBMC calls the script's onControl function, and passes it the *List*. It's important to realize that onControl is given the List instance that *contains* the selected ListItem—it isn't handed the actual item selected from the List. We have to get that ourselves.

To do that, we use a pair of functions: getSelectedItem and getLabel. First, let's get the script ready. Pull all of the buttons out of the __init__ and add in our new List (as described above). You can use *either* of the methods I typed out, they both do the same thing.

And now that we don't have any Buttons, get rid of your old onControl. Here's where we use getSelectedItem and getLabel. Use this for your new onControl:

```
def onControl(self, control):
        if control == self.list:
                item = self.list.getSelectedItem()
                self.message("You selected : " + item.getLabel())
```

You can see that we first use getSelectedItem to find out which ListItem within the list is selected (a ListItem is a separate class, it's not just a text string), and *then* we use the ListItem's getLable function to find out what the string is. That's most all you need to know, at least for now. Let's see this thing in action.

The whole script should look like this:

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10

class MyClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit, or select an item from the list and push A")

        # Make the List
        self.list = xbmcgui.ControlList(300, 250, 200, 200)
        self.addControl(self.list)
        self.list.addItem("Item 1")
        self.list.addItem("Item 2")
        self.list.addItem("Item 3")
        self.setFocus(self.list)

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()

    def onControl(self, control):
        if control == self.list:
            item = self.list.getSelectedItem()
            self.message("You selected : " + item.getLabel())

    def message(self, message):
        dialog = xbmcgui.Dialog()
        dialog.ok(" My message title", message)

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

That's really all you need to know about the Dialog Boxes and Control Objects, the basic widgets of the xbmcgui library. There's still a lot to teach, though. The next segment will teach you how to get more direct user feedback, using a built-in virtual keyboard. The segments following that will walk you through the process of getting information about the XBox from XBMC. Finally, we'll show you some more advanced tools such as child windows and non-XBMC uses for scripts, like a simple internet file downloader.

Using the Virtual Keyboard

Before we get into the complicated stuff, though, let's do the easy one. XBMC provides a very basic but surprisingly versatile virtual keyboard. You can call it from any script that imports the xbmc library, and your script can get not only what text the user entered, but whether or not the keyboard was canceled. Your script can also provide default text to the keyboard, to save the user time.

The basic keyboard call is very simple, and requires a doModal command to make it appear (just like an xbmcgui.Window). In your display.py script, remove all the List stuff, including the onControl function.

We're going to write a very basic keyboard interface for now. First, add a new Label to your __init__ so we can get some feedback:

```
self.outputLabel = xbmcgui.ControlLabel(100, 300, 200, 200, "", "font13", "0xFFFFFFFF")
self.addControl(self.outputLabel)
```

Now add in the keyboard itself. These two lines make the keyboard (and provide it with a default text).

```
keyboard = xbmc.Keyboard("Entered Text")
keyboard.doModal()
```

After calling doModal, your script will sit and wait for the user to finish with the keyboard. They can finish by typing in new text (or accepting the default) and pressing "Y" to confirm, or by pressing "B" to cancel. Either way, the script will continue, so the first thing we do is add an "if" statement to make sure the keyboard's text is legitimate user input.

```
if (keyboard.isConfirmed()):
    self.outputLabel.setLabel(keyboard.getText())
```

You can see we used the keyboard's function getText to find out what the entered value *was*, once we knew it was legitimate. And now let's add an "else" just so we'll know what counts as an accept, and what's a cancel:

```
else:
    self.outputLabel.setLabel("User Canceled")
```

Now, the way this script works, it will print to the outputLabel any text entered at the keyboard, unless the user cancels the keyboard, in which case it will say so. This script is very simple, but a keyboard's really not hard to use. Once you know how to do it at all, you pretty much know how to do it all, see?

The whole script should look like this:

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10

class MyClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit.")
        self.outputLabel = xbmcgui.ControlLabel(100, 300, 200, 200, "", "font13", "0xFFFFFFFF")
        self.addControl(self.outputLabel)
        keyboard = xbmc.Keyboard("Entered Text")
        keyboard.doModal()
        if (keyboard.isConfirmed()):
            self.outputLabel.setLabel(keyboard.getText())
        else:
            self.outputLabel.setLabel("User Canceled")

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Of course, even if a keyboard is simple, it provides all kinds of possibilities. It might be the clunkiest part of your script—making a user type out a file's path on the XBox or enter a URL—but the keyboard opens up limitless possibilities.

Sidenote – Bitplane's Py9 Dictionary

Every virtual keyboard is going to have its drawbacks—that's why so many people with modded XBoxes add USB ports to plug in *real* keyboards. Even the full-time commercial developers can't do much to improve the problem, as you know if you've played any commercial XBox game that requires you to enter a character name, or anything of the sort.

Bitplane of the XBMC forums is trying to help alleviate the problem some, by porting a concept originally designed for text messaging on cell phones. He has written a module that uses custom dictionaries to recreate the Nokia T9 style predictive text function.

Bitplane's module is designed to be incorporated into keyboard-dependent scripts. It's basic operation is this: when a user begins entering text in the keyboard, the Py9 module will check the beginning of the word against a dictionary and try to predict what whole word the user is entering. If it guesses, right, you can press one button and it will finish the word for you; if it guesses wrong, you keep entering additional letters until it guesses right or you finish spelling out the word.

This can be an incredible time saver. Maybe it won't speed up zip code entry much, for scripts like MovieGuide and XBMC's Weather page, but it might just make things like Instant Messenger and XBox Email a *lot* easier and more efficient.

At the time I'm writing this tutorial, the Py9 Dictionary tool has just been made public on the XBMC Forums, and is still in the early stages of development. If you're thinking about using a keyboard extensively in your scripts, drop by the XBMC forums and do a search for this cool utility. It could turn a tedious chore into a quick and helpful little script.

Alexpoet

Now that you've learned everything there is to know about interacting with the user, let's turn our attention back to the XBox itself. The next three segments will focus on getting information from XBMC, that you can either use in your script, or display to a curious user.

Getting (and Using) the Current Screen Size

The developers of XBMC have worked hard to make sure their program displays well on many different screen sizes. They've been kind enough to provide us, as Python scripters, with the necessary tools to do the same. These are the xbmcgui functions getHeight() and getWidth().

Each of these functions does just what the name implies, returning an integer value equal to the number of pixels available on the current display. Some of the most common values are 720 x 480 (NTSC standard), 1280 x 720, and 1920 x 1080.

To start with, we'll use a simple script that just finds the current display's values and tells them to the user. We can do all of this in the __init__ function, as there's no interaction necessary. Use the following lines:

```
            screenX = self.getWidth()
            screenY = self.getHeight()
            strscreenX = str(screenX)
            strscreenY = str(screenY)
```

You can see in the last two lines we converted the integer values to strings, so we can display them in Labels. Now just add a couple of Labels showing the information we've gotten.

The whole script might look like this:

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10

class MyClass(xbmcgui.Window):
    def __init__(self):
            if Emulating: xbmcgui.Window.__init__(self)
            self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
            self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
            self.addControl(self.strActionInfo)
            self.strActionInfo.setLabel("Push BACK to quit")
            screenX = self.getWidth()
            screenY = self.getHeight()
            strscreenX = str(screenX)
            strscreenY = str(screenY)
            self.widthInfo = xbmcgui.ControlLabel(100, 300, 200, 200, "", "font13", "0xFFFFFFFF")
            self.addControl(self. widthInfo)
            self. widthInfo.setLabel("screen width is " + strscreenX)
            self.heightInfo = xbmcgui.ControlLabel(100, 400, 200, 150, "", "font13", "0xFFFFFFFF")
            self.addControl(self. heightInfo)
            self. heightInfo.setLabel("screen height is " + strscreenY)

    def onAction(self, action):
            if action == ACTION_PREVIOUS_MENU:
                    self.close()

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Of course, telling a user how wide his screen is isn't going to help him out much. These functions are most useful when determining how you're going to layout the GUI. You might use them at the very beginning of your script, as something like:

```
screenX = self.getWidth()
if screenX <= 720:
    background = "Q:\\scripts\\Tutorial\\small_background.gif"
    standardFont = "font12"
elif screenX <= 1280:
    background = "Q:\\scripts\\Tutorial\\medium_background.gif"
    standardFont = "font14"
else:
    background = "Q:\\scripts\\Tutorial\\large_background.gif"
    standardFont = "font16"
```

Then you could include three different background pictures with your script for use on different TV sizes, and use those variables in your __init__ function to make sure you draw the right one on any given TV. This may seem like a lot of work, but trust us: it's worth it.

You'll be surprised how many people are interested in downloading and using any script you make available, and *then* you'll be surprised how many different problems crop up when you have that many people testing out your stuff. Screen resolution differences are an easy problem to avoid (thanks to these two functions), so get in the habit of making dynamically-resizing scripts from the beginning, and you can spend your time fixing much more troublesome issues. :-)

Sidenote – Background Skins and Bitplane's XML-Based Skin Reader

Although this document is focusing on how you *write* scripts--specifically the Python skills and XBMC libraries needed to do that--there's more that goes into making a really useful script than just writing it out. One of the major tools I use in developing scripts is the background skin (like the one you see every time you run "display.py").

Drawing a backgroudn skin allows you to draw stationary graphical elements *behind* your GUI that will visually define the regions of the widgets you draw *in* your GUI. In other words...it can be a shortcut. Background skins are used to make scripts more attractive in appearance, but also to save time defining dialogs and scripting out widget borders.

If you're just starting out in scripting, you probably won't need to use background skins extensively until you get to more advanced scripts, but I wanted to mention them here, briefly, just so you could keep the possibility in mind.

One useful utility for building and organizing background skins is the XML-based Skin Reader developed by Bitplane. It comes attached to his IRC-Client script, and I encourage you to drop by the XBMC forums or the Downloads page linked at the end of this document, and get a copy of it. The Skin Reader comes with very helpful documentation included, so I'm not going to go into detail here on its use, but consider it as a possible aid when you start adding complex backgrounds to your scripts.

Alexpoet

Getting the Current Skin Directory and Localization

Another way you can build flexibility into your scripts is by writing them to interact smoothly with international users. Again, just like the screen sizes, the xbmcgui library provides us with the tools to find out *how* the user is running his XBox, so we can try to make our scripts behave the same way.

This information is found using the xbmc.getLocalizedString() function. It reads information from the user's XML language file and passes it back to your script. The code will look something like this:

```
localtxt1 = xbmc.getLocalizedString(10000)
localtxt2 = xbmc.getLocalizedString(10004)
```

You can also get certain specific information, like the XBMC skin currently in use:

```
myskin = xbmc.getSkinDir()
```

This information can be important, because different skins have different font sizes, different graphics available, and different screen placement of buttons and other GUI items.

Just like the screen size above, this information is really only useful when you build scripts that will react to their environment, displaying properly no matter whose XBox they're running on. It would take too long to explain all the ways these items can be put to use, though, so we're just going to show you how to get the information, and for now (just as an example), we'll also display it on the screen.

Now, you already know all about getting and displaying information during an __init__ function, so this time let's do the same sort of thing in a different way. What we'll do is, from *outside* the MyClass instance, we'll call a function *inside* the class that makes the GUI draw some text on the screen. The way you call a function within a class is by typing the class name first, then a period, then the regular function call, like this:

```
mydisplay.localInfo()
```

We'll add that line to the bottom of our script, after creating the class (`mydisplay = MyClass()`), but before we ever tell it to show up on the screen (`mydisplay.doModal()`). That way the class will be created, the __init__ will run (and will only make *one* Label to appear on the screen), and then *after* that the localInfo() function will run, drawing two additional Labels. *Then* the doModal call will make the screen appear, showing us all thee Labels.

Does that make sense? It doesn't *have* to—we're here to learn how to get information language information using xbmc.getLocalizedString. We just added this functionality so you can get a glimpse at another way to make Python do what you want it to do. You could always write this the same way you've been doing, though, putting the information checks as well as the Label constructors directly in the __init__ function.

So there's no reason to panic if all that talk before didn't make any sense. But you might go ahead and read through it one more time, just to see if you can gain anything from it. The more you learn about Python—and *all* the ways you can make it work—the better your final scripts will be.

Now, we've already discussed our basic plan for the script, so let's put it into code. First, write a localInfo function within MyClass that gets two different text strings out of your XML language file and displays them in a single Label. Then add another label that tells us the name of your XBMC skin. Here's a copy of the function:

```
def localInfo(self):
        localtxt1 = xbmc.getLocalizedString(10000)
        localtxt2 = xbmc.getLocalizedString(10004)
        self.strLanguageInfo = xbmcgui.ControlLabel(100, 300, 200, 200, "", "font13", "0xFFFFFFFF")
        self.addControl(self.strLanguageInfo)
        self.strLanguageInfo.setLabel("Text using your xml language file: " + localtxt1 + " , " + localtxt2)
        myskin = xbmc.getSkinDir()
        self.strSkinInfo = xbmcgui.ControlLabel(100, 400, 200, 150, "", "font13", "0xFFFFFFFF")
        self.addControl(self.strSkinInfo)
        self.strSkinInfo.setLabel("Your skin dir is : /skin/" + myskin)
```

You can see we added "/skin/" to our output on the SkinInfo Label. We did that because getSkinDir only returns the name of the skin's folder, not the path from the XBMC root installation. We know that XBMC skins are supposed to be stored in the "/skin/" subfolder, though, so we add that to the output Label to make the information clearer to our users.

Now all that's left is to add the function call at the bottom of our script, like we described above. **The whole script should look like this:**

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10

class MyClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit")

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()

    def localInfo(self):
        localtxt1 = xbmc.getLocalizedString(10000)
        localtxt2 = xbmc.getLocalizedString(10004)
        self.strActionInfo = xbmcgui.ControlLabel(100, 300, 200, 200, "", "font13", "0xFFFFFFFF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Text using your xml language file: " + localtxt1 + " , " + localtxt2)
        myskin = xbmc.getSkinDir()
        self.strActionInfo = xbmcgui.ControlLabel(100, 400, 200, 150, "", "font13", "0xFFFFFFFF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Your skin dir is: /skin/" + myskin)

mydisplay = MyClass()
mydisplay.localInfo()
mydisplay.doModal()
del mydisplay
```

When you test this on your XBox, you should see the “Text using your XML language file:” Label and then two words shown in your language of choice. And the Label at the bottom should accurately display the script you’re currently using.

It’s easy to test the second one; just change your skin and then run the script again (at the time I’m writing this, apparently only Symbol and Project Mayhem will run scripts). You’d have to be brave (or bilingual) to test the other, but I managed it. Go into your XBMC settings, change the language, and then reboot if necessary. Now go into your scripts folder (I had to do it by memory, but I’ve spent a *lot* of time launching scripts so it wasn’t hard), run display.py, and you’ll see the same words in the new language!

To get the most out of this function, you’ll need to learn the Language file, and which words and phrases are available across languages. Because XBMC is set up like this, it’s *possible* to write one script that will work smoothly in multiple language…but it’s a lot of work. You’ll have to decide just how much you want to support international versions of your scripts. At least now you know how to use the tools, in case you do decide to put in the effort.

Getting Language, IP Address, DVD State, Available Memory, and CPU Temperature

These functions were my first attempt to extend the XBMC Python library and I’m pretty proud of the result. You can use getLanguage, getIPAddress, getDVDState, getFreeMem, and getCpuTemp to access these data (all of which provide popular information to display on skins).

Even more useful, this will show you *how* xbmc is asked for information, and how it provides answers. We can only assume that the devs will continue to provide us with more functions, and access to more information, so learn these five functions, and you'll already be familiar with the basic pattern that will *probably* be used for similar kinds of functions in the future.

The script is very similar to our last one, but we're changing the information shown in the localInfo function. Because there's only really one change (and because you're probably much better at reading through Python scripts by this point), this time I'm just going to post the whole script at once. **It should look like this:**

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10

class MyClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit")

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()

    def localInfo(self):
        infoLanguage = xbmc.getLanguage()
        self.strLanguage = xbmcgui.ControlLabel(100, 150, 200, 200, "", "font13", "0xFFFFFFFF")
        self.addControl(self.strLanguage)
        self.strLanguage.setLabel("Your language is: " + infoLanguage)
        infoIP = xbmc.getIPAddress()
        self.strIPAddress = xbmcgui.ControlLabel(100, 250, 200, 200, "", "font13", "0xFFFFFFFF")
        self.addControl(self.strIPAddress)
        self.strIPAddress.setLabel("Your IP adress is: " + infoIP)
        infoDVD = xbmc.getDVDState()
        self.strDVDState = xbmcgui.ControlLabel(100, 300, 200, 200, "", "font13", "0xFFFFFFFF")
        self.addControl(self.strDVDState)
        dvdstate = ""
        if (infoDVD == 1):
            dvdstate = "DRIVE_NOT_READY"
        if (infoDVD == 16):
            dvdstate = "TRAY_OPEN"
        if (infoDVD == 64):
            dvdstate = "TRAY_CLOSED_NO_MEDIA"
        if (infoDVD == 96):
            dvdstate = "TRAY_CLOSED_MEDIA_PRESENT"
        self.strDVDState.setLabel("DVD state: " + dvdstate )
        infoMemory = xbmc.getFreeMem()
        self.strMemory = xbmcgui.ControlLabel(100, 350, 200, 150, "", "font13", "0xFFFFFFFF")
        self.addControl(self.strMemory)
        self.strMemory.setLabel("Available memory: " + str(infoMemory) + " MB")
        infoTemp = xbmc.getCpuTemp()
        self.strTemp = xbmcgui.ControlLabel(100, 400, 200, 100, "", "font13", "0xFFFFFFFF")
        self.addControl(self.strTemp)
        self.strTemp.setLabel("CPU Temp: " + str(infoTemp) )

mydisplay = MyClass()
mydisplay.localInfo()
mydisplay.doModal()
del mydisplay
```

Creating a Child Window

Now we enter our final lesson on GUIs, and I've saved it this long because it's one that looks pretty complicated. It's really not very hard, and it's all stuff that you know how to do by now, but it might seem tricky to you if you're not a programmer.

Now, any of you who *are* programmers will have pretty much figured this out already, and this segment won't be any kind of a challenge. Read through it, say to yourself, "Oh, yeah, that would work," and then do the same with the next segment and you're done.

But, as I said at the start, I'm not writing this for the experienced programmers, I'm writing it for the beginners. So we'll keep it simple.

What we're going to do now is start a script that then creates a new window, which will pop up above the main display window. It will hang around for a while (until you press "B"), and then go away and leave the main display window there. This is called creating a "child window" (where the "parent window" is the main, original display. Child windows can be very helpful when you have a lot of information to display, and only so much screen space to show it on.

Essentially, a child window is the same thing as a dialog, except that it's raw (that is, you have to tell it what to contain) whereas a dialog is pre-defined to only contain certain things. Also, it gives you a *lot* more room.

So how do we build a child window? The same way we make a main window: by declaring a separate class, calling the class, and then using the doModal command.

First let's do some renaming. Replace MyClass with MainClass, because now there's going to be more than one. :-)

```
class MainClass(xbmcgui.Window):
    def __init__(self):
            if Emulating: xbmcgui.Window.__init__(self)
```

Then go on with your __init__ as normal. Change the strActionInfo Label to tell the user the new rules: Push "Back" to quit, or "A" to open another window. Let's also throw in another Label just to let the user know that the window he's looking at is the parent (or main) window.

Now we'll modify onAction to handle the "A" button. Add these three lines:

```
        if action == ACTION_SELECT_ITEM:
                popup = ChildClass()
                popup.doModal()
                del popup
```

And *that* should look very familiar. It's the same three lines you have had at the bottom of almost every script, although we've changed the variable name, and here we're calling a different class (but one that does almost exactly the same thing).

Can you guess what that class is going to look like? All we want it to do is tell the user to push "Back" to get rid of this window, and show another label making sure you know that this is the child window. Oh, and have it go away when you push "Back". That's *all* stuff you've known how to do ever since page four. Try writing out the child class yourself. I'll give you the declaration line (which should be all the way over to the left of the page):

```
class ChildClass(xbmcgui.Window):
```

And the rest you know. Unless this all seems painfully obvious to you, I encourage you to try to get this script working on your own. It wouldn't be fair to leave you entirely in the dark, though. **So, when you're done, the whole script should look like this:**

```
import xbmc, xbmcgui
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
ACTION_SELECT_ITEM = 7

class MainClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit, or A to open another window.")
        self.strActionInfo = xbmcgui.ControlLabel(300, 300, 200, 200, "", "font13", "0xFFFFFFFF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("This is the first window")

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()
        if action == ACTION_SELECT_ITEM:
            popup = ChildClass()
            popup.doModal()
            del popup

class ChildClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to return to the first window")
        self.strActionInfo = xbmcgui.ControlLabel(300, 300, 200, 200, "", "font13", "0xFFFFFF99")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("This is the child window")

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()

mydisplay = MainClass()
mydisplay.doModal()
del mydisplay
```

That's it. Run it on your XBox and see what it does. At this point, you know everything it takes to make an interactive Python script for your XBox. You can control it, you can access information on the XBox and you know all about how to provide that information to the user. Of course, so far everything we've talked about is how to use the XBMC libraries. But those libraries only exist to allow us to do *other* things—cooler things. In the final segment we'll just touch on how you'll go about putting this information to *use*.

Using Some Handy non-XBMC Functions

Now, it's up to you to come up with the ideas and figure out how to implement them. One very good way to do that is to look through all the other scripts available and see what kinds of things others are doing. You can also *read* through any scripts that are available, to see how someone else solved a problem that's bugging you. So the learning material certainly doesn't stop at the end of this tutorial.

However, to get you started, here's an example of how you could apply the XBMC libraries to write a Python script that would *do* something. Specifically, it downloads a file through HTTP. [Editor's note: This is an unbelievably useful little function—I've copied the logic of it into at least three of my own projects. So pay attention!]

We'll use a special pair of Python methods—Try and Except, which you've seen used before with all our "Emulating = " lines—which work together to test if something can be done. First it tries the instructions in the try block. If those work, the script keeps running. If any of them break, for any reason, the except block allows the script to keep running (instead of throwing an error and crashing), but of course it fails to accomplish whatever it was trying. You can read the Python documentation for more details.

```
import xbmc, xbmcgui, urllib
try: Emulating = xbmcgui.Emulating
except: Emulating = False

#get actioncodes from keymap.xml
ACTION_PREVIOUS_MENU = 10
ACTION_SELECT_ITEM = 7

class MyClass(xbmcgui.Window):
    def __init__(self):
        if Emulating: xbmcgui.Window.__init__(self)
        self.addControl(xbmcgui.ControlImage(0,0,720,480, "Q:\\scripts\\Tutorial\\background.gif"))
        self.strActionInfo = xbmcgui.ControlLabel(100, 200, 200, 200, "", "font13", "0xFFFF00FF")
        self.addControl(self.strActionInfo)
        self.strActionInfo.setLabel("Push BACK to quit - A to download")

    def onAction(self, action):
        if action == ACTION_PREVIOUS_MENU:
            self.close()
        if action == ACTION_SELECT_ITEM:
            webfile = "http://www.google.com/images/logo.gif"
            localfile = "Q:\\scripts\\logo.gif"
            self.downloadURL(webfile,localfile)

    def downloadURL(self,source, destination):
        try:
            loc = urllib.URLopener()
            loc.retrieve(source, destination)
            self.message("Download successful!")
        except:
            self.message("Download failed. Check your internet connection and try again later.")

    def message(self, messageText):
        dialog = xbmcgui.Dialog()
        dialog.ok(" My message title", messageText)

mydisplay = MyClass()
mydisplay.doModal()
del mydisplay
```

Try it out. If you get the message “Download successful!” that means you now have a copy of Google’s logo saved in your scripts folder. Go find it and delete it. :-) And *that* is a successful function. Of course, you can change the “webfile” address to point to something else, or maybe write a “select” dialog to choose among several items to download. Or incorporate the keyboard, and let the user type out a url.

You should know how to do all those things, and much more. At this point, we’ve covered all the basics of programming Python scripts using the XBMC libraries. I hope this tutorial has been of some help to you. You can visit our websites (listed below) or drop by the XBMC Forums (also listed) to offer any encouragement or suggestions. We’re always trying to improve the quality of this document, as well as the quality of any scripts we provide, so please provide any feedback through the appropriate forums.

Thank you for your interest,
Alex (aka alx5962), and Alexpoet

Useful Links:

XBMC Forums:	http://www.xboxmediaplayer.de/cgi-bin/forums/ikonboard.pl
XBox-Scene Forums:	http://forums.xbox-scene.com/index.php?showforum=62
Scripts Download Page:	http://dwl.xboxmediacenter.de/
Alx5962’s website:	http://www.gueux.net/xbmc-scripts/index.html
Alexpoet’s website:	http://members.cox.net/alexpoet/downloads/

You can also email us at these addresses:

Alx5962:	alx5962@yahoo.com
Alexpoet:	script_request@cox.net