developerWorks®

# Scripting KVM with Python, Part 2: Add a GUI to manage KVM with libvirt and Python

Paul Ferrill

January 17, 2012

This two-part series explores how to use Python to create scripts for managing virtual machines using KVM. In this installment, you learn how to add a GUI to expand on the simple status and display tool.

View more content in this series

Part 1 of this series looked at the basics of scripting Kernel-based Virtual Machine (KVM) using `libvirt` and Python. This installment uses the concepts developed there to build several utility applications and add a graphical user interface (GUI) into the mix. There are two primary options for a GUI toolkit that has Python bindings and is cross-platform. The first is Qt, which is now owned by Nokia; the second is wxPython. Both have strong followings and many open source projects on their lists of users.

For this article, I focus on wxPython more out of personal preference than anything else. I start off with a short introduction to wxPython and the basics proper setup. From there, I move on to a few short example programs, and then to integrating with `libvirt`. This approach should introduce enough wxPython basics for you to build a simple program, and then expand on that program to add features. Hopefully, you'll be able to take these concepts and build on them to meet your specific needs.

## wxPython basics

A good place to start is with a few basic definitions. The wxPython library is actually a wrapper on top of the `C++`-based wxWidgets. In the context of creating a GUI, a *widget* is essentially a building block. Five independent widgets reside at the top-most level of the widget hierarchy:

`wx.Frame`,
`wx.Dialog`,
`wx.PopupWindow`,
`wx.MDIParentFrame`, and
`wx.MDIChildFrame`.

Most of the examples here are based on `wx.Frame`, as it essentially implements a single modal window.

In wxPython, `Frame` is a class that you instantiate as is or inherit from to add or enhance the functionality. It's important to understand how widgets appear within a frame so you know how to place them properly. Layout is determined either by absolute positioning or by using sizers. A *sizer* is a handy tool that resizes widgets when the user changes the size of the window by clicking and dragging a side or corner.

The simplest form of a wxPython program must have a few lines of code to set things up. A typical main routine might look something like Listing 1.

### Listing 1. Device XML definition

```
if __name__ == "__main__":
    app = wx.App(False)
 frame = MyFrame()
 frame.Show()
 app.MainLoop()
```

Every wxPython app is an instance of `wx.App()` and must instantiate it as shown in Listing 1. When you pass `False` to `wx.App`, it means "don't redirect stdout and stderr to a window." The next line creates a frame by instantiating the `MyFrame()` class. You then show the frame and pass control to `app.MainLoop()`. The `MyFrame()` class typically contains an `__init__` function to initialize the frame with your widgets of choice. It is also where you would connect any widget events to their appropriate handlers.

This is probably a good place to mention a handy debugging tool that comes with wxPython. It's called the *widget inspection tool* (see Figure 1) and only requires two lines of code to use. First, you have to import it with:

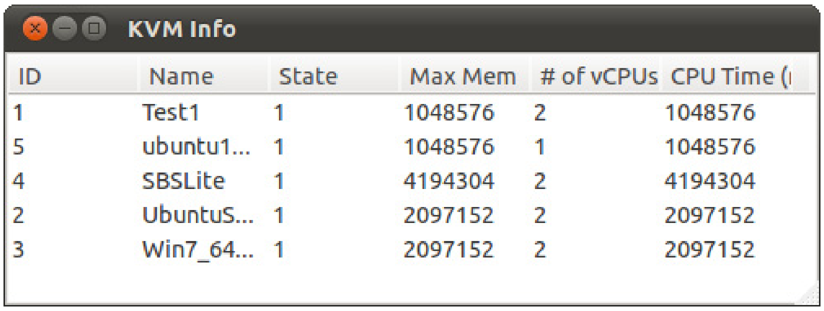
```
import wx.lib.inspection
```

Then, to use it, you simply call the `Show()` function:

```
wx.lib.inspectin.InspectionTool().Show()
```

Clicking the **Events** icon on the menu toolbar dynamically shows you events as they fire. It's a really neat way to see events as they happen if you're not sure which events a particular widget supports. It also gives you a better appreciation of how much is going on behind the scenes when your application is running.

### Figure 1. The wxPython widget inspection tool


KVM Info

| ID | Name | State | Max Mem | # of vCPUs | CPU Time (ı |
|---|---|---|---|---|---|
| 1 | Test1 | 1 | 1048576 | 2 | 1048576 |
| 5 | ubuntu1... | 1 | 1048576 | 1 | 1048576 |
| 4 | SBSLite | 1 | 4194304 | 2 | 4194304 |
| 2 | UbuntuS... | 1 | 2097152 | 2 | 2097152 |
| 3 | Win7_64... | 1 | 2097152 | 2 | 2097152 |

## Add a GUI to a command-line tool

Part 1 of this series presented a simple tool to display the status of all running virtual machines (VMs). It's simple to change that tool into a GUI tool with wxPython. The `wx.ListCtrl` widget provides just the functionality you need to present the information in tabular form. To use a `wx.ListCtrl` widget, you must add it to your frame with the following syntax:

```
self.list=wx.ListCtrl(frame,id,style=wx.LC_REPORT|wx.SUNKEN_BORDER)
```

You can choose from several different styles, including the `wx.LC_REPORT` and `wx.SUNKEN_BORDER` options previously used. The first option puts the `wx.ListCtrl` into Report mode, which is one of four available modes. The others are Icon, Small Icon, and List. To add styles like `wx.SUNKEN_BORDER`, you simply use the pipe character (`|`). Some styles are mutually exclusive, such as the different border styles, so check the wxPython wiki if you have any doubts (see Related topics).

After instantiating the `wx.ListCtrl` widget, you can start adding things to it, like column headers. The `InsertColumn` method has two mandatory parameters and two optional ones. First is the column index, which is zero-based, followed by a string to set the heading. The third is for formatting and should be something like `LIST_FORMAT_CENTER`, `_LEFT`, or `_RIGHT`. Finally, you can set a fixed width by passing in an integer or have the column automatically sized by using `wx.LIST_AUTOSIZE`.

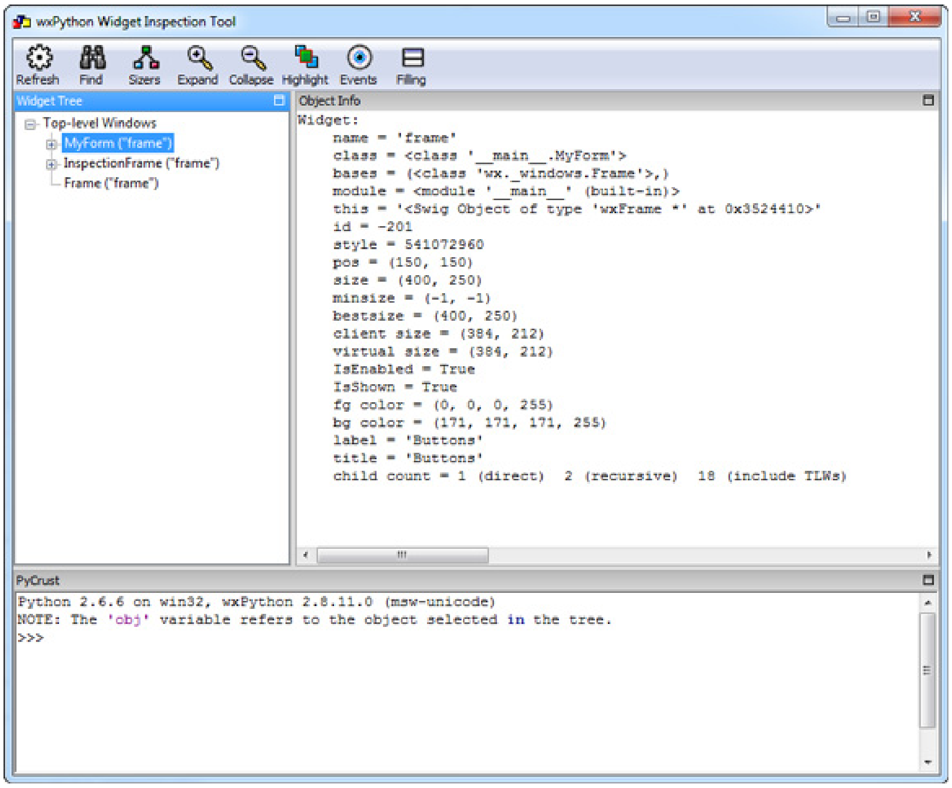
Now that you have the `wx.ListCtrl` widget configured, you can use the `InsertStringItem` and `SetStringItem` methods to populate it with data. Each new row in the `wx.ListCtrl` widget must be added using the `InsertStringItem` method. The two mandatory parameters specify where to perform the insert, with a value of 0 indicating at the top of the list and the string to insert at that location. `InsertStringItem` returns an integer indicating the row number of the inserted string. You can make a call to `GetItemCount()` for the list and use the return value for the index to append to the bottom, as Listing 2 shows.

### Listing 2. GUI version of the command-line tool

```
import wx
import libvirt

conn=libvirt.open("qemu:///system")

class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(None, -1, "KVM Info")
        id=wx.NewId()
        self.list=wx.ListCtrl(frame,id,style=wx.LC_REPORT|wx.SUNKEN_BORDER)
        self.list.Show(True)
        self.list.InsertColumn(0,"ID")
        self.list.InsertColumn(1,"Name")
        self.list.InsertColumn(2,"State")
        self.list.InsertColumn(3,"Max Mem")
        self.list.InsertColumn(4,"# of vCPUs")
        self.list.InsertColumn(5,"CPU Time (ns)")

        for i,id in enumerate(conn.listDomainsID()):
            dom = conn.lookupByID(id)
            infos = dom.info()
            pos = self.list.InsertStringItem(i,str(id))
            self.list.SetStringItem(pos,1,dom.name())
            self.list.SetStringItem(pos,2,str(infos[0]))
            self.list.SetStringItem(pos,3,str(infos[1]))
            self.list.SetStringItem(pos,4,str(infos[3]))
            self.list.SetStringItem(pos,5,str(infos[2]))

        frame.Show(True)
        self.SetTopWindow(frame)
        return True

app = MyApp(0)
app.MainLoop()
```

Figure 2 shows the results of these efforts.

## Figure 2. The GUI KVM info tool



```
Widget:
    name = 'frame'
    class = <class '__main__.MyForm'>
    bases = (<class 'wx._windows.Frame'>,)
    module = <module '__main__' (built-in)>
    this = '<Swig Object of type 'wxFrame *' at 0x3524410>'
    id = -201
    style = 541072960
    pos = (150, 150)
    size = (400, 250)
    minsize = (-1, -1)
    bestsize = (400, 250)
    client size = (384, 212)
    virtual size = (384, 212)
    IsEnabled = True
    IsShown = True
    fg color = (0, 0, 0, 255)
    bg color = (171, 171, 171, 255)
    label = 'Buttons'
    title = 'Buttons'
    child count = 1 (direct)  2 (recursive)  18 (include TLWs)
```

```
Python 2.6.6 on win32, wxPython 2.8.11.0 (msw-unicode)
NOTE: The 'obj' variable refers to the object selected in the tree.
>>>
```

You can enhance the appearance of this table. A noticeable improvement would be to resize the columns. You can do so by either adding the `width =` parameter to the `InsertColumn` call or use one line of code, like this:

```
self.ListCtrl.SetColumnWidth(column,wx.LIST_AUTOSIZE)
```

The other thing you could do is add a sizer so that the controls resize with the parent window. You can do this with a `wxBoxSizer` in a few lines of code. First, you create the sizer, and then you add the widgets to it that you want to resize along with the main window. Here's what that code might look like:

```
self.sizer = wx.BoxSizer(wx.VERTICAL)
self.sizer.Add(self.list, proportion=1,flag=wx.EXPAND | wx.ALL, border=5)
self.sizer.Add(self.button, flag=wx.EXPAND | wx.ALL, border=5)
self.panel.SetSizerAndFit(self.sizer)
```

The last call to `self.panel.SetSizerAndFit()` instructs wxPython to set the initial size of the pane based on the sizer's minimum size from the embedded widgets. This helps to give your initial screen a reasonable size based on the content inside.

## Control flow based on a user action

One of the nice things about the `wx.ListCtrl` widget is that you can detect when a user clicks a specific part of the widget and take some action based on that. This functionality allows you to do things like sort a column alphabetically in forward or reverse order based on the user clicking the column title. The technique to accomplish this uses a callback mechanism. You must provide a function to handle each action that you want to process by binding the widget and processing method together. You do so with the `Bind` method.

Every widget has some number of events associated with it. There are also events associated with things like the mouse. Mouse events have names like `EVT_LEFT_DOWN`, `EVT_LEFT_UP`, and `EVT_LEFT_DCLICK`, along with the same naming convention for the other buttons. You could handle all mouse events by attaching to the `EVT_MOUSE_EVENTS` type. The trick is to catch the event in the context of the application or window you're interested in.

When control passes to the event handler, it must perform the necessary steps to handle the action, and then return control to wherever it was prior to that. This is the event-drive programming model that every GUI must implement to handle user actions in a timely fashion. Many modern GUI applications implement multithreading to keep from giving the user the impression that the program isn't responding. I briefly touch on that later in this article.

Timers represent another type of event that a program must potentially deal with. For example, you might want to perform a periodic monitoring function at a user-defined interval. You would need to provide a screen on which the user could specify the interval, and then launch a timer that would in turn fire an event when it expires. The timer expiration fires an event that you can use to activate a section of code. You might need to set or restart the time, depending again on user preference. You could easily use this technique to develop a VM monitoring tool.

Listing 3 provides a simple demo app with a button and static text lines. Using `wx.StaticText` is an easy way to output a string to the window. The idea is to click the button once to start a timer and record the start time. Clicking the button records the start time and changes the label to **Stop**. Clicking the button again fills in the stop time text box and changes the button back to **Start**.

### Listing 3. Simple app with a button and static text

```
import wx
from time import gmtime, strftime

class MyForm(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, wx.ID_ANY, "Buttons")
        self.panel = wx.Panel(self, wx.ID_ANY)

        self.button = wx.Button(self.panel, id=wx.ID_ANY, label="Start")
        self.button.Bind(wx.EVT_BUTTON, self.onButton)

    def onButton(self, event):
        if self.button.GetLabel() == "Start":
            self.button.SetLabel("Stop")
            strtime = strftime("%Y-%m-%d %H:%M:%S", gmtime())
            wx.StaticText(self, -1, 'Start Time = ' + strtime, (25, 75))
        else:
            self.button.SetLabel("Start")
            stptime = strftime("%Y-%m-%d %H:%M:%S", gmtime())
            wx.StaticText(self, -1, 'Stop Time = ' + stptime, (25, 100))

if __name__ == "__main__":
    app = wx.App(False)
    frame = MyForm()
    frame.Show()
    app.MainLoop()
```

## Enhanced monitoring GUI

Now, you can add functionality to the simple monitoring GUI introduced earlier. There is one more piece of wxPython you need to understand before you have everything you need to create your app. Adding a check box to the first column of a `wx.ListCtrl` widget would make it possible to take action on multiple lines based on the status of the check box. You can do this by using what wxPython calls *mixins.* In essence, a *mixin* is a helper class that adds some type of functionality to the parent widget. To add the check box mixin, simply use the following code to instantiate it:

```
listmix.CheckListCtrlMixin.__init__(self)
```

You can also take advantage of events to add the ability to select or clear all boxes by clicking the column title. Doing so makes it simple to do things like start or stop all VMs with just a few clicks. You need to write a few event handlers to respond to the appropriate events in the same way you changed the label on the button previously. Here's the line of code needed to set up a handler for the column click event:

```
self.Bind(wx.EVT_LIST_COL_CLICK, self.OnColClick, self.list)
```

`wx.EVT_LIST_COL_CLICK` fires when any column header is clicked. To determine which column was clicked, you can use the `event.GetColumn()` method. Here's a simple handler function for the `OnColClick` event:

```
def OnColClick(self, event):
    print "column clicked %d\n" % event.GetColumn()
 event.Skip()
```

The `event.Skip()` call is important if you need to propagate the event to other handlers. Although this need might not be apparent in this instance, it can be problematic when multiple handlers need to process the same event. There's a good discussion of event propagation on the wxPython wiki site, which has much more detail than I have room for here.

Finally, add code to the two button handlers to start or stop all checked VMs. It's possible to iterate over the lines in your `wx.ListCtrl` and pull the VM ID out with just a few lines of code, as Listing 4 shows.

## Listing 4. Starting and stopping checked VMs

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

import wx
import wx.lib.mixins.listctrl as listmix
import libvirt

conn=libvirt.open("qemu:///system")

class CheckListCtrl(wx.ListCtrl, listmix.CheckListCtrlMixin,
                                 listmix.ListCtrlAutoWidthMixin):
    def __init__(self, *args, **kwargs):
        wx.ListCtrl.__init__(self, *args, **kwargs)
        listmix.CheckListCtrlMixin.__init__(self)
        listmix.ListCtrlAutoWidthMixin.__init__(self)
        self.setResizeColumn(2)

class MainWindow(wx.Frame):

    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        self.panel = wx.Panel(self)
        self.list = CheckListCtrl(self.panel, style=wx.LC_REPORT)
        self.list.InsertColumn(0, "Check", width = 175)
        self.Bind(wx.EVT_LIST_COL_CLICK, self.OnColClick, self.list)

        self.list.InsertColumn(1,"Max Mem", width = 100)
        self.list.InsertColumn(2,"# of vCPUs", width = 100)

        for i,id in enumerate(conn.listDefinedDomains()):
            dom = conn.lookupByName(id)
            infos = dom.info()
            pos = self.list.InsertStringItem(1,dom.name())
            self.list.SetStringItem(pos,1,str(infos[1]))
            self.list.SetStringItem(pos,2,str(infos[3]))

        self.StrButton = wx.Button(self.panel, label="Start")
        self.Bind(wx.EVT_BUTTON, self.onStrButton, self.StrButton)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.list, proportion=1, flag=wx.EXPAND | wx.ALL, border=5)
        self.sizer.Add(self.StrButton, flag=wx.EXPAND | wx.ALL, border=5)
        self.panel.SetSizerAndFit(self.sizer)
        self.Show()

    def onStrButton(self, event):
        if self.StrButton.GetLabel() == "Start":
     num = self.list.GetItemCount()
            for i in range(num):
                if self.list.IsChecked(i):
                    dom = conn.lookupByName(self.list.GetItem(i, 0).Text)
                    dom.create()
                    print "%d started" % dom.ID()

    def OnColClick(self, event):
         item = self.list.GetColumn(0)
         if item is not None:
             if item.GetText() == "Check":
                 item.SetText("Uncheck")
                 self.list.SetColumn(0, item)
                 num = self.list.GetItemCount()
                 for i in range(num):
                     self.list.CheckItem(i,True)
             else:
                 item.SetText("Check")
                 self.list.SetColumn(0, item)
                 num = self.list.GetItemCount()
                 for i in range(num):
                     self.list.CheckItem(i,False)

         event.Skip()

app = wx.App(False)
win = MainWindow(None)
app.MainLoop()
```

There are two things to point out here with respect to the state of VMs in KVM: Running VMs show up when you use the `listDomainsID()` method from `libvirt`. To see non-running machines you must use `listDefinedDomains()`. You just have to keep those two separate so that you know which VMs you can start and which you can stop.

## Wrapping up

This article focused mainly on the steps needed to build a GUI wrapper using wxPython that in turn manages KVM with `libvirt`. The wxPython library is extensive and provides a wide range of widgets to enable you to build professional-looking GUI-based applications. This article just scratched the surface of its capabilities, but you'll hopefully be motivated to investigate further. Be sure to check more Related topics to help get your application running.

## Related topics

- `libvirt` website: Check out the entire site for more information.
- Reference Manual for `libvirt`: Access the complete `libvirt` API reference manual.
- Python.org: Find more of the Python resources you need.
- wxPython.org: Get more about wxPython.
- wxPython wiki: Expand your knowledge through the many tutorials found here.
- developerWorks Open source zone: Find extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM products. Explore more Python-related articles.
- developerWorks on Twitter: Follow us for the latest news.