



Python in Systems Administration: Part IV -- Python Makes GUIs

Cameron Laird

In the first installment of this series on Python for systems administration, back in December 2003, one of the benefits I mentioned was how easy Python makes development of graphical user interfaces (GUIs). This month, I'll give more attention to Python-based GUIs.

For the purpose of this series, GUIs are applications that rely on bit-mapped windowed images, icons, and the mouse pointer (WIMP), rather than character-based displays. I'll concentrate on X Window clients, although in principle Unix hosts can also use a framebuffer or other interface.

The first step in approaching Python GUI development is to let go of fear. The simplicity and power that make Python so apt for solving problems in systems administration have also made the language a desirable platform for experimentation. The result is that a bewildering variety of "toolkits", "frameworks", and "environments" for GUI work are available through Python. By my count, Python usefully supports a wider span of these libraries than any other language, including even the far better known C and Java: wxPython, PyQt, AnyGUI, ...

Start with the Basics

Don't let that embarrassment of riches distract you. For now, it's enough to concentrate on Tkinter, the GUI-capable library and module distributed with standard Python. If you decide to study Tkinter more deeply, you'll probably want copies of such books as John Grayson's 2000 *Python and Tkinter Programming*, Fredrik Lundh's 1999 *An Introduction to Tkinter*, which is available online at:

<http://www.pythonware.com/library/tkinter/introduction/>

or January 2004's *Learning Python*, Second Edition by Mark Lutz and David Ascher.

Even without such books, though, you can get an idea of what Tkinter can do for you in the sidebar "[Where's Mine?](#)". In the Unix world, systems administrators typically do our jobs from the command line by invoking commands with specific arguments, using command-line-oriented languages such as **sh** and **awk**, and occasionally editing plaintext configuration files. Think for a moment of the occasions when it might be an advantage to have a GUI:

- A small monitor you can park somewhere on your screen, to give you quick access to a pertinent measurement such as free space, network usage, user count, etc.
- A little "control panel" that wraps up common actions you take, so you can pass on temporary responsibility for them to a colleague who covers for you while you travel.
- A small "client-server" application you write to automate an end user process you now do "by hand".

Experience with C or Java might have taught you that GUI work of this sort is "heavy" and best left to

committed developers. How does Python do at such chores?

Monitor Your Environment

Python's greatest benefit in these roles is the ease with which it expresses exactly the requirements specific to your own situation. For this article, though, we'll choose examples that are artificially universal in order to focus on the part Python plays. Keep in mind that the advantages of "GUIfication" multiply as you apply it to your own requirements, rather than trying to fit those requirements to a general-purpose product.

Suppose, then, that you could use a small monitor that tracks uptime. All Unix hosts build in the **uptime** command, so the program that follows works wherever Python is installed. You could look at the top line of **top** output, of course; that would give you the same data. However, it takes only a few extra lines to write:

```
# We need to: access an external process; do easy
# string substitution; and show a GUI. These three
# modules provide such functionality.
import commands, re, Tkinter

# Every second--one thousand milliseconds--this function
# gets the latest 'uptime' result, displays it, then
# reschedules itself for the next cycle.
def one_update():
    # This just collects the result of the external
    # built-in 'uptime' command.
    result = commands.getoutput("uptime")

    # Discard everything in the result string that
    # appears before the load averages themselves.
    pattern = ".*load averages*: "
    display_text.set(re.sub(pattern, "", result))

    # Do it again another second later.
    label.after(1000, one_update)

root = Tkinter.Tk()
# Create a display "buffer".
display_text = Tkinter.StringVar()

# Create a GUI widget which shows the results.
label = Tkinter.Label(root, textvariable = display_text, width = 20)

# Place the Label in the main window.
label.pack()

# Begin the measurements.
one_update()

# Start processing events, that is, turn the GUI "on".
root.mainloop()
```

At this point (see [Figure 1](#)), you have a tiny window that updates every second or so with the freshest **uptime** result. It's well-behaved as a GUI: you can resize it, iconify and restore it, and generally treat it the way you'd treat any other GUI window.

With this as a starting point, you can calculate more interesting quantities to display. What do you need to keep an eye on? Network spikes? Server overloads? Tkinter makes quick work of GUI monitors for

all such values. With just a few lines more, you can render quantities as plots or graphs, rather than textual displays. Later in this series, I'll show an example of such a graphical diagram.

Help When You're Not Around

One value of GUIs is to *limit* action, or, more precisely, to focus it in specified ways. Suppose a junior admin sometimes needs to check the contents of backup media. Let's start with a simplified situation in which we rely on **tar**. **tar** has a couple of liabilities: it's flexible enough that it can be hard for those unfamiliar with it to remember all its arguments; and it's powerful enough to do real damage to existing file systems. An obvious solution to these problems is to wrap up **tar** in a GUI that ensures correct, safe use. This example uses it for read-only access to either of two mass-storage devices:

```
# Pmw includes such conveniences for GUI programming
# as a ScrolledFrame.
import commands, Pmw, Tkinter

device = "/dev/tapeB"
def read_tape():
    text.config(state = Tkinter.NORMAL)
    text.delete(1.0, Tkinter.END)
    text.insert(Tkinter.END,
               commands.getoutput("tar tf %s" % device))
    # Make the display text read-only.
    text.config(state = Tkinter.DISABLED)

root = Tkinter.Tk()
button = {}

# Suppose /dev/rmt1 and /dev/tapeB are the names of
# two valid tape devices.
for (which, particular_device) in [(1, "/dev/rmt1"),
                                   (2, "/dev/tapeB")]:
    button[which] = Tkinter.Radiobutton(root,
                                       variable = "device",
                                       value = particular_device,
                                       text = "tape drive %d" % which)
    button[which].pack()

push = Tkinter.Button(root, text = "Read contents",
                      command = read_tape)
push.pack()

f = Pmw.ScrolledFrame(root)
f.pack(fill = "both", expand = 1)
frame = f.interior()
text = Tkinter.Text(frame, height = 20, width = 60)
text.pack(fill = "both", expand = 1)

root.mainloop()
```

Once you've written this script (see [Figure 2](#)), you can safely give it to an assistant with confidence that it will access real devices in a non-destructive way. These kinds of "control panels" can also be useful to *you*, however, if only to encapsulate argument configurations that are inconvenient to remember or type. Moreover, they package your expertise in a way that's meaningful to managers; a working GUI often makes a point far more effectively than any command-line wizardry.

Friendly to End Users

Finally, GUIs can simplify fragile human processes. Imagine you have an ongoing transient problem with zombie processes on different workstations. To track this down, you've tried to institute the policy of having users email you incident reports that include a process table snapshot, a datestamp, and commentary.

They get it wrong, though; they forget the arguments to `ps`, they don't remember how to put `ps`'s results into an email message, and so on. You've got the right policy, but the users aren't following it.

You're lucky -- this is one of the rare social problems that a technical fix truly solves. Just give your users an application they can run on their own hosts:

```
import commands, os, Tkinter

def report():
    os.system("""mail -s 'Zombie report' MY_ADDRESS < HERE
Commentary:
-----
%s
=====
Process table:
-----
%s
HERE""") % (commentary.get(1.0, END),
            commands.getoutput("ps aux"))

root = Tkinter.Tk()
label = Tkinter.Label(root, text = "Report zombies")
commentary = Tkinter.Text(root, height = 10, width = 40)
button = Tkinter.Button(root, command = report, text = "Send report")
label.pack()
commentary.pack()
button.pack()

root.mainloop()
```

It costs little to write such a "helper", users often find them friendly (see [Figure 3](#)), and they help you get the data you need -- wins all around!

There's a great more to the world of GUIs; however, as a first step, I hope you more clearly see ways that GUIs might help in your work, and how easy it is to use Python to create them. Next month, Part V of this series on Python in systems administration will continue with a look at the language's networking potency. Until then, keep that email coming; it's been very gratifying to receive all the reports of how Python is already helping many of you, as well as the kinds of additional problems you want to solve.

Cameron Laird, a vice president at consultancy Phaseit, Inc. (<http://phaseit.net/>), has written occasionally for Sys Admin in previous years.

Copyright © 2001 Sys Admin, Sys Admin's Privacy Policy . Comments about the Web site: webmaster@sysadminmag.com
