

# Python in Systems Administration: Part VI -- Wrapping up Python



*Cameron Laird*

This is the sixth installment of this series on Python in systems administration. Many of you have emailed me about ways the language has helped in your duties, and suggestions for improvements still needed in Python. In this article, I'll tie up a few loose ends.

In the first article in this series, I introduced Python as the single language that solves the widest variety of problems a systems administrator is likely to meet. A couple of readers objected that my presentation neglected traditional Unix strengths. Paddy McCarthy wrote that it didn't "fit with the concept of 'many smaller utilities doing one thing well, connected via pipes'", and Martin Gadbois made much the same point, that he wishes "you could have created a real pipe in Python".

## Pipelining

They're absolutely right, of course -- pipelining *is* one of Unix's strengths, and I certainly don't want anyone to give it up. It's compatible with Python in several dimensions.

First, Python participates quite nicely in pipelines. Suppose you've written a program in Python that uses Google's Web services interface to suggest corrected spellings for misspelled words. You might pipe:

```
spell corpus | corrections.py | sort | uniq -d | wc -l
```

to generate a count of the number of distinct misspellings that appear to target the same correct word (for example, "exemple" and "exampul" are both misspellings of "example"). Python can read from standard in and write to standard out as well as any other language.

Python also can exploit pipelines "inside-out". Here's an example of what I mean:

```
import commands
count = commands.getoutput(
    "grep ma /etc/passwd | wc -l")
print \
"The number of entries in /etc/passwd which contain 'ma' is %s." \
    % count
```

That is, not only can pipeline-using shell commands invoke Python-coded processes, but Python processes can easily invoke pipeline-using shell commands.

There's a more interesting way to think about pipelines, though, than either of these easy answers. The

simultaneous strength and difficulty with shell pipelines is that they communicate only string data (with a few weak syntactic conventions). If there's an error somewhere in the pipeline, all the processes typically shut down. If the data have more structure than the model of separate lines in a plaintext file...well, it simply gets lost.

Python supports text processing quite well, and so makes a natural partner for the pipelines already mentioned. Python also adds a few other powerful data structures beyond text strings -- principally lists, tuples, and dictionaries. All of these are "first-class" constructs in Python, in the sense that functions and methods can pass and return any of these freely. The consequence is that, as long as you write in Python, you can build up a library of powerful small building blocks that process data, and connect these pieces with a "functional" programming style. The result is programs that look like this:

```
save(render_as_PDF(summarize_report(
    prepare_report(get_data(data_set = x)))))
```

Python's intelligent exception handling also manages errors more gracefully than a conventional pipeline, which ignores many errors and simply halts on others.

Abstract "pipelining" enough, and it's hard to distinguish from the art of writing one-liners. While Python emphasizes clarity over brevity, at least when compared to such languages as shell or Perl, the standard Python library is plenty rich enough to make interesting one-liners possible. Suppose, for example, you want to pull out the human-language names of your host's users -- that is, extract the fourth field of each line of `/etc/passwd`, remove trailing commas, and print the result, one name per line. **awk** and **sed** of course, makes this as easy as:

```
awk -F : '{print $5}' /etc/passwd | sed -e 's/,//g'
```

Python requires a bit more to take the intermediate results apart and reconstitute them:

```
print '\n'.join([line.split(':')[4].strip(',')
    for line in open("/etc/passwd").read()[0:-1].split('\n')])
```

(If your lines are wider than this magazine's, it's ok with Python to have all that on one line of source.)

This example isn't much of an advertisement for Python's "pipelining"; it is almost twice as verbose as the **awk** and **sed** combination, mostly because Python doesn't make abbreviations for the special case of newline-separated character streams. When you need even more complexity, though (such as sorting on an index computed from an associated field) Python requires relatively few more clauses, while the corresponding shell pipeline quickly becomes unsustainably clumsy.

Thus, when I write about the wide applicability of Python, I don't have a serious expectation that herds of systems administrators will abandon shell for Python. I do think the latter offers enough, though, that all of us should at least be familiar with its basics.

## Graphical Variations

The first installment in this series also promoted Python's ability to construct applications with graphical user interfaces (GUIs), and the fourth elaborated the point with a few more examples of Tkinter programming. Among the more-than-a-dozen GUI toolkits Python supports, Tkinter is the only one the standard distribution includes.

Support from Tkinter has improved since those articles were first published. There are now a mailing list and a Wiki dedicated to Tkinter (see <http://tkinter.unpythonic.net/wiki/TkinterDiscuss> for pointers to both). There's also been significant progress in smoothing out rough edges with support of the toolkit for secondary Unixes such as MacOS X, HP-UX, and so on. Tkinter is more portable and inviting than ever before.

At the same time, Tkinter isn't for everyone. As I mentioned previously, alternative toolkits -- including wxPython, anygui, and PythonCard -- have plenty of fans.

One GUI toolkit that's particularly apt for systems administrators is EasyGui (<http://www.ferg.org/easygui/>). EasyGui doesn't require object orientation or event-handling, two programming concepts that often challenge newcomers to GUI development. Instead, as EasyGui's creator Stephen Ferg e-mailed, "I wrote easygui to make it quick and easy to put up a simple GUI interface for a script." EasyGui makes basic dialogues as easy as possible. Would you like to ask a user for the name of a file?

```
from easygui import *
fileName = fileopenbox()
```

is all it takes to invoke a standard file-selector and store its result.

Although Python doesn't build EasyGui into the standard distribution, all you need to run EasyGui is a single Python source file, downloaded into your standard Python library directory. No compilation or other installation complications are necessary. Another part of EasyGui's ease is its enthusiastic support; from all I've seen, Ferg actively follows up with EasyGui users to resolve issues that arise. In fact, he read the first article in this series, and was so struck with systems administrators' need for a simple menu selector that he was "inspired ... to add one new feature to easygui ... It is a multchoicebox() function that puts up a listbox and returns a list of your choices." It makes programming selection of, for example, a collection of user ids as straightforward as:

```
selections = multchoicebox(message="Selected user id-s",
                          title="user id selection",
                          choices= [line.split(':')[0] for
                                   line in open("/etc/passwd").read().split('\n')
                                   if line.count('#') == 0])
```

Most other GUI toolkits require at least separate statements to create a selection box, to populate it, and to retrieve the result when a user selects "Done" or "Cancel". EasyGui does it in one. Of course, you don't *have* to cram all this on one line; if anything, it's more idiomatic Python to write:

```
id_list = []
for line in open("/etc/passwd").read().split('\n'):
    # Ignore lines that begin with a comment.
    if line[0] != '#':
        # The zeroth field holds the id.
        id_list.append(line.split(':')[0])
selections = multchoicebox(message = "Selected user id-s",
                          title = "user id selection",
                          choices = id_list)
```

Even in this more expanded form, the simplicity of **multchoicebox**'s use is evident.

## Batteries Included

The first article also mentioned Python's image among its users as the "batteries-included" language. This testifies to its remarkable compactness; downloading the standard Python distribution takes only a few megabytes, but gives a remarkably complete tool chest of the functions and methods necessary to handle common problems. A posting by open source contributors Mike Fletcher and Thomas Heller to the active comp.lang.python Usenet newsgroup illustrates this aptly. You can read the originals at: <http://groups.google.com/groups?th=f0b4db73789a1df7>.

There, the two team up to produce an "Example script to automate downloading of SourceForge project CVS backups" in a few dozen lines. Many of us have done just this sort of thing in shell, calling out to

lynx, for example, to make the retrieval, and to gzip and basename to unpack an archive and manipulate individual filenames. Python builds in all these capabilities, even the relatively recondite ones of bzip2 decompression and easy timestamp manipulation. The heart of their application is this definition:

```
def retrieve( projectName ):
    """Given a projectName, retrieve and store download to downloadDirectory"""
    os.chdir( downloadDirectory )
    url = "http://cvs.sourceforge.net/cvstarballs/ %(projectName)s-cvsroot.tar.bz2"%locals()
    date = time.strftime( '%Y-%m-%d' )
    fileName = '%(projectName)s-%(date)s-cvsroot.tar.bz2'%locals()
    file = os.path.join( downloadDirectory, fileName )
    def doDots( current, block, total ):
        sys.stdout.write( '.' )
    print 'Retrieving:\n %(url)s\nInto:\n %(fileName)s'%locals()
    urllib.urlretrieve( url, file, doDots )
    print
    print 'Decompressing bzip format'
    data = bz2.BZ2File(fileName, "r").read()
    print 'Recompressing in gzip format'
    tarFile = os.path.splitext(fileName)[0]
    gzip.GzipFile(tarFile + '.gz', "wb", 9).write(data)
    print 'Finished'
```

Write one script, and you can have plenty of confidence that it'll work on all platforms with Python (that is, all platforms you're likely to see in a modern datacenter). As it happens, there is a slight blemish with this particular program, in that `strftime()` has had portability incompatibilities. I hope that these are all solved by the time you read this article.

## Python and Email

The original "killer application" of the Internet, and still the one on which our users most broadly rely, is email. The goal of this series hasn't been how to program in Python so much as how to think about Python, so you can make good choices for possible use of the language. Along with its universality, clarity, and GUI capabilities, one final point you should keep in mind about Python is its special relationship with email.

First, Mailman (<http://www.gnu.org/software/mailman/mailman.html>) is the leading open source mailing-list manager. Its reliability and handy Web management pages have vaulted it past former-standard Majordomo and all other competitors. Mailman is implemented in Python. If you ever have occasion to customize a Mailman installation, you'll almost certainly do it with Python.

Email is the oldest 'Net application, but it remains an area full of research and innovation, much of it recently in response to infection and spam. Python is a common choice among the authors of the most interesting experiments; thus, for example, if you want to run the hugely influential SpamBayes "probability-based mail filter" (<http://spambayes.sourceforge.net/>), you're going to be involved with Python.

As I collect your responses and questions to this series, I'm working up a few additional focused articles on specific uses of Python in systems administration. One likely candidate is Python's use in Unix installers, including many Linux distributions. Please let me know your interests, and I'll return from time to time to address them.

*Cameron Laird, a vice president at consultancy Phaseit, Inc. (<http://phaseit.net/>), is a regular contributor to Sys Admin.*

Copyright © 2001 Sys Admin, [Sys Admin's Privacy Policy](#). Comments about the Web site:  
[webmaster@sysadminmag.com](mailto:webmaster@sysadminmag.com)