



# Python in Systems Administration: Part I -- Better Scripting

Cameron Laird

Sidebar: [How to Handle \(a\) Python](#)



One well-honed item in every sys admin's toolbox is a preferred scripting language -- maybe sh, or ksh, or Perl, or something more unusual, like Rexx or Ruby. I find Python so capable and easy to learn that in my consulting role at Phaseit, I tell newcomers it's the best single language they can learn. Python spans a wider range of applications than any other language I know -- even more than C or Java -- and beginners pick it up quickly.

Over the next several months, *Sys Admin's* series on "Python in Systems Administration" will tackle the following four themes and their relations:

- An introduction to Python
- The "scripting mentality"
- Practical examples of systems administrators' use of Python
- The craft of systems administration and its best practices

## Focus on Flexibility

Of these, the most challenging to explain is what I call the "scripting mentality", or scripting attitude. This mentality suspects an application that meets all nominal requirements will also be too "heavy" and difficult to be used effectively. For scripters, a better strategy for solving problems is a simple scripting interface that allows users to "snap together" their own solutions. The idea is to provide building blocks that are flexible and simple to understand.

This idea can be a hard pill to swallow at times, because it conflicts with the "code re-use" imperative we all learn. Tim Peters is a senior engineer with Zope Corporation whose standing in the Python community is second only to Python's inventor, Guido van Rossum. Peters once made the case in these words: "It's easier to write appropriate code from scratch in Python than to figure out how to *use* a package profligate enough to contain canned solutions for all common and reasonable use cases."

"Find" provides a good example of this contrast in styles. Windows has a "Find" application available through the standard "Start" button. This Find is easy to use, in that end-users regularly take advantage of it without documentation: point, click, and type in a file name, and a list of matches appears.

The Unix **find** utility, on the other hand, is notorious for its obscurity. Few people remember all its options, and beginners frequently mistype even its simplest forms. The command-line utility scales much, much better, though. Windows "Find" makes a few searches very easy, but is frustrating if you need anything beyond the scope of its standard forms. Unix's **find** emphasizes orthogonality, so you

can freely combine predicates in any combination, even those unimagined by the designers of the utility. Once you learn its minimal syntax, it's a small step to request, for example, a list of all files with a certain extension, created or modified in the last week, which are at least a megabyte in size, *except* those owned by a particular user. Windows simply doesn't consider the possibility.

The Unix **find** also plays nicely with other applications; it's easy to combine it with third-party filters. Windows, however, gives no such opportunity, apart from tedious cutting-and-pasting of results.

This emphasis on flexibility and power will recur throughout this series on Python in systems administration. Python rarely gives systems administrators finished solutions; it almost always provides highly capable and reliable tools we can use to solve specific problems in just a few lines of readable code. Enthusiasm for such an opportunity is the "scripting mentality" this series aims to transmit.

## Easy Scripting

A few examples of working code illustrate Python's advantages as a scripting language. Suppose you occasionally need to restore files pulled from a backup tape. From the Unix command line, you might request:

```
tar xvf /dev/nrtape FILE1 FILE2 FILE3
```

In practice, it is often useful to package up even such a simple command as a one- or few-line script; if you don't work with it every day, you might not remember the exact name of the tape device, or the syntax for file names. Generalize the command above, then, into the shell file **my\_restore**:

```
#!/bin/sh
# Usage: "my_restore NAME" retrieves all files in the whole
# directory tree which match NAME; thus "my_restore my_source"
# will find
#     my_source.c
#     project/sources/my_source.c
#     project/sources/my_source.h
# without requiring you to know the tree structure.
#
# A production version of this utility
# would be far more careful about error-handling.

DEVICE=/dev/nrtape
LIST_OF_QUALIFIED_NAMES=tar tf $DEVICE | grep $1
tar xvf $DEVICE $LIST_OF_QUALIFIED_NAMES
```

While Python can do the same, and even without the minor expense of **grep** as an external process, it has no advantage over **sh** for simple pipelines:

```
#!/usr/local/bin/python
# This is my_restore.py.

import os,sys

device = '/dev/nrtape'
    # Invoke the application as, for example, "my_restore.py MYFILE".
pattern = sys.argv[1]

    # tar puts names on separate lines.
    # The count() method substitutes for grep.
list_of_qualified_names = [file for file in \
    os.popen('tar tf %s' % device).read().split('\n') \
    if file.count(pattern) > 0]
```

```

# tar expects white-space--a single blank, for example--to
# separate its input arguments.
files = ' '.join(list_of_qualified_names)
print os.popen('tar xf %s %s' % (device, files)).read()

```

If Python is no more succinct than **sh** for such simple tasks, why should it interest systems administrators? It shouldn't, except that our jobs never stay this simple. As tasks become even slightly more complex, two potent Python advantages become apparent:

- Python's sophisticated data structures -- including dictionaries, first-class functions, full-fledged objects, and more -- are far more capable than the crude string-based values and variables of **sh**.
- Python truly is the "batteries-included" language. Once we've gone to the minimal trouble of writing **my\_restore.py** in Python, all the riches of the built-in Python libraries are available. The next section of this article shows how it takes only a few more lines to add a useful graphical user interface (GUI) to **my\_restore**. It's not just a GUI that Python provides, either; it's equally straightforward to add network services, sophisticated calendar arithmetic, database interfaces, XML processors, host monitors, or much, much more.

Too often, systems administrators resign ourselves to the fiction that there's a sharp boundary between what we do, and "real programming". One of Python's great achievements is to demonstrate a gentle, smooth transition from few-line scripting to sophisticated, polished automation that exploits all a platform offers. Add Python to your personal toolkit, and you immediately expand the range of applications you can complete within a demanding schedule.

### Wrap It Attractively

Python's built-in ability to construct GUIs is a good example. Let's touch up this restore program so that a user can control its operation through a simple GUI. The goal will be to launch it, as before, with a simple command-line invocation such as **my\_restore.py FILENAME**. Rather than extracting all matches for FILENAME, though, this version of **my\_restore.py** presents a list of selections. The user selects only the file instances he truly wants restored (see [Figure 1](#)).

Python requires only a few additional lines to implement such a GUI:

```

#!/usr/local/bin/python

# This is 'my_restore.py'.  Invoke it as, for example,
#   my_restore.py my_source.c
# to show all the archived files named 'my_source.c'
# which the user can select for retrieval.

import os,sys,Tkinter

pattern = sys.argv[1]
archive = '/dev/nrtape'

def esf():
    # Construct the string which lists all selected files,
    #   separated by a single blank.  Use that string to
    #   specify the exact listof files to extract from the
    #   archive.
    command = 'tar xf %s %s' % (archive,
        ' '.join([lb.get(index) for index in lb.curselection()]))
    os.system(command)

# MULTIPLE so that we can select and extract several files in a
#   single operation.

```

```

lb = Tkinter.Listbox(height = 12, width = 30, selectmode = Tkinter.MULTIPLE)
lb.pack()
Tkinter.Button(text = "Extract selected files", command = esf).pack()

# The "[:-1]" says, "ignore the trailing newline tar emits".
for qualified_name in \
    os.popen('tar tf %s' % archive).read().split('\n')[:-1]:
    # Does the basename of this item match the pattern?
    if os.path.basename(qualified_name).count(pattern) > 0:
        lb.insert(Tkinter.END, qualified_name)

# Show the GUI panel.
Tkinter.mainloop()

```

This example dramatizes the contrast with **sh** and other conventional administrative scripting languages. While a few lines in each suffice for simple tasks, requiring a GUI or other common features quickly outstrips the ability of the usual shell languages. If you're coding in Python, though, a solution demands only a few more lines.

Here's the strategy, then -- adopt Python as one of your preferred scripting languages. Use it for the day-to-day chores that wise sys admins solve by scripting. When you return to the same task in eight months, you'll find your well-written Python source so readable that you don't need ancillary READMEs or instruction sheets or recipes. Moreover, when you need more serious development (e.g., a GUI for a little task, or service of remote requests), Python will be ready to express your new version with modest incremental effort.

Over the coming months, I'll compose several scripts that solve specific problems that often arise in systems administration. Future installments will also focus on how Python not only makes essential things easier, but also makes important things *possible*.

In the meantime, you can find more information on the topics mentioned in this article at these sites:

Tutorial on **find** -- <http://www.grymoire.com/Unix/Find.html>

The original christening of Python as the "batteries included" language --

<http://web.archive.org/web/20001013152452/>

<http://sunworld.com/swol-12-1998/swol-12-regex.html>

"Python is an Agile programming language" -- <http://www.oreillynet.com/pub/wlg/3060>

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

<p>Copyright © 2001 Sys Admin, <a href="#">Sys Admin's Privacy Policy</a>. Comments about the Web site:  <a href="mailto:webmaster@sysadminmag.com">webmaster@sysadminmag.com</a></p>
---