



# Python in Systems Administration: Part II -- Step up from Shell

*Cameron Laird*

What kinds of problems are suited to Python? As a first approximation, think of Python the way you do Perl. Although far more Unix systems administrators currently work in Perl, the two languages are roughly comparable, for our purposes. Many of the differences between them are subjective, in the sense that experienced programmers simply find the features of one or the other fit their own habits of thinking better, although they're equally capable or provide the same formal functionality. This second installment in my series on "Python in Systems Administration" explains the parallels, then spotlights instances where Python might serve you better.

An example will show how similar they are, and how they differ from shell. Here I'm using "shell" to mean the language `/bin/sh` interprets, which is also compatible with `bash`, `ksh`, and other alternatives.

## Reporting for Duty

In her popular book, *Essential System Administration*, Aileen Frisch lauds Perl for its ability, among other things, "to generate attractive reports". She demonstrates that (pages 368-369 of the Second Edition), with a simple table that tells the disk-space usage of different home directories.

Knowledgeable systems administrators *do* commonly turn to Perl when preparing reports. Many times, as in this case, it's for work that shell can do -- but with more difficulty. Python's just as easy as Perl, though.

To warm up to this problem, first consider this translation into shell of the Perl implementation Frisch provides:

```
#!/bin/sh

PASSWD=/etc/passwd
if [ ! -f $PASSWD ]
then
    echo "Can't open $PASSWD."
    exit 1
fi

FORMAT="%-18s  %-17s  %10s  %-9s\n"
printf "$FORMAT" "" "" "Disk " ""
printf "$FORMAT" "Username (UID)" "Home Directory" \
        Space Security
echo "-----\
-----"
export IFS=:
```

```

cat $PASSWD | while read UNAME PASS xUID xGID JUNK \
                    HOME_DIR JUNK2
do
  case $UNAME in
    root|nobody|uu*)
      continue;;
  esac
  if [ $xUID -le 100 ] && [ $xUID -gt 0 ]
  then
    continue
  fi
  if [ $PASS != '!' ] && [ $PASS != '*' ] && [ $PASS != 'x' ]
  then
    WARN="** CK PASS";;
  elif [ $xUID -eq 0 ] && [ $UNAME != root ]
  then
    WARN="** UID=0*"
  else
    WARN=""
  fi
  if [ -d $HOME_DIR ] && [ $HOME_DIR != / ]
  then
    # It's an error in the original that it
    # lacks stderr redirection.
    DISK='du -s -k $HOME_DIR 2>/dev/null | \
          awk '{ print $1 }''
    if [ x$DISK == x ]
    then
      DISK=unknown
    else
      DISK=${DISK}K
    fi
  else
    DISK=skipped
  fi
  printf "$FORMAT" "$UNAME ($xUID)" $HOME_DIR \
          $DISK $WARN
done

```

The output is the same report:

Username (UID)	Home Directory	Disk Space Security
lpd (104)	/	skipped
sanders (464)	/home/sanders	725980K
stein (0)	/chem/1/stein	4982K ** UID=0
...		

as Frisch's Perl version produced. Does Perl improve on this? It's a few lines shorter. Perl -- and Python, as I'll illustrate -- pull ahead of shell when there's string-"wrangling" to do, or arithmetic. Frisch used a substr() function in her implementation, but I think the shell's wild-card match against "uu\*" is even clearer than her coding.

Shell's branching is clumsy, which accounts for all Perl's advantage in conciseness in this example. Perl, Python, and other general-purpose languages have at least two capabilities that shell lacks entirely or exhibits only in a difficult form: data structuring, so that "records" or data values more complicated than primitive numbers and strings can be easily manipulated; and direct connections to OS-level operations. Among other things, the latter means that Perl and Python can easily operate on multiple files simultaneously.

Here's a Python script that generates the same report:

```
#!/usr/local/bin/python

import os,sys

passwd="/etc/passwd"
if not os.path.isfile(passwd):
    print "Can't open %s." % passwd
    sys.exit(1)

format="%-18s %-17s %10s %-9s"
print format % ("", "", "Disk ", "")
print format % ("Username (UID)", "Home Directory",
               "Space", "Security")
print "-----\
-----"
for line in open(passwd).readlines():
    (uname, xpass, uid, gid, junk, home_dir, junk2) = line.split(':')
    if uname == 'root' or uname == 'nobody' or uname[0:2] == 'uu':
        continue
    uid = int(uid)
    if uid <= 100 and uid > 0:
        continue
    if uid == 0 and uname != 'root':
        warn = "*** UID=0"
    elif xpass != '!' and xpass != '*' and xpass != 'x':
        warn = "*** CK PASS"
    else:
        warn = ""
    if os.path.isdir(home_dir) and home_dir != '/':
        disk = os.popen("du -s -k %s 2>/dev/null" %
                       home_dir).read().split('\t')[0]
        if disk == '':
            disk = "unknown"
        else:
            disk += "K"
    else:
        disk = "skipped"
    print format % ("%s (%s)" % (uname, uid), home_dir, \
                   disk, warn)
```

Is this an improvement on shell and Perl? I think so. Python's line count happens to be smaller than that for either of the other two languages, and I think it's a bit "cleaner", too.

## Six-month Test

I recognize that these are largely subjective matters. While ease of learning and "comprehensibility" were explicit design goals of Python from its origin in a way they never were for shell and Perl, experts in all three languages have no trouble reading and understanding the scripts we're considering in this series. A rigorous demonstration of the general superiority of one syntax or another is possible, I believe, but it would take a great deal of research and discipline.

Far less expensive is simply to try these languages for yourself. The first installment in this series (<http://www.samag.com/documents/s=8964/sam0312a/0312a.htm>) explained how easy it is to begin experimenting with Python. You don't have to be an expert on how Perl and Python compare for all systems administrators; learn instead which one suits *you* best.

If you're like many other systems administrators I've met, you'll find that Python particularly suits you for its success at "the six-month test". A significant portion of the people who program only occasionally with Perl complain to me that they have trouble reading and therefore maintaining what they've written after the fact. They're good enough with the language to write what they need, but find that when they return to a particular script -- six months later, perhaps -- it no longer makes sense.

Quite a few of these systems administrators cherish Python because they find it easy to pick up again after an extended interval away from it. Maybe it will have that advantage for you.

### One-liners

Perl is justly celebrated for its success in a variety of other roles, including Web scripting, network programming, and database development. As with report generation, Python substitutes well in each of these roles. One that's particularly helpful to systems administrators is "one-liners" -- the brief, disposable small scripts that take care of the miscellaneous chores that arise during typical administration work.

Python often does in three or five lines what Perl does in one. Perl builds in clever devices that make it easy to iterate over filenames, and to combine operations on a single line. Python emphasizes regularity more, and often demands explicitness where Perl builds in commonly used defaults. A file-renaming example illustrates that.

We often need to move files around within a file system, for administrative purposes. Here's a typical task -- move all files with the .jpg suffix found in directory DIR1 to DIR2, while simultaneously changing their names according to the model:

```
bridge.jpg -> bridgeB.jpg  
airplane.jpg -> airplaneB.jpg
```

In Python, a good way to accomplish this is with:

```
import glob, os  
  
os.chdir(DIR1)  
for file in glob.glob('*.jpg'):  
    (base, extension) = os.path.splitext(file)  
    os.rename(file, os.path.join(DIR2, base + 'B.jpg'))
```

How does that compare with the way you've been doing such tasks *without* Python?

### Immediate Execution

Part II of "Python in Systems Administration" has emphasized that Python can be used for jobs where you'd first think of shell or Perl. A final aspect of this similarity has to do with interactive use.

Much of our work is "on the command-line" -- rather than compose a script saved in a file, which we run "batch", we just type commands and have the shell interpret them on the spot. Perl has this capability, by the way, although very few Perl users seem to know it. It's possible to enter a Perl shell, and have Perl statements executed interactively. It's an unusual way to work, though; most systems administrators think of immediate execution only in terms of their base shell. For more on this subject, see the sidebar ["Notes on Interactive Perl"](#).

Python changes that. Python users frequently work with the base Python shell -- just type "python" and all your commands will be interpreted interactively. This makes it handy to experiment.

Are you trying out different ideas for making a temporary file with a random component in the name?

Try out your ideas "live":

```
# python
Python 2.2 (#1, 11/12/02, 23:31:59)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import random, string
>>> def a(head):
...     return head + ''.join([random.choice(string.digits) for \
        i in range(4)])
...
>>> a("name")
'name4943'
>>> a("name")
'name2707'
>>> a('longname')
'longname6525'
```

When you have a clear goal, but aren't sure how to reach it, this sort of "exploratory" programming can help you reach your goal incrementally.

It's even possible to take this to the extreme of doing *all* your work in Python. IPython (<http://ipython.scipy.org/>) repackages the standard Python language so it can realistically substitute for more standard shells such as **bash** and **ksh**. IPython's only for enthusiasts, I think; tossing out standard shell isn't a step to take lightly. Its availability does underline, however, Python's flexibility and aptness for the kinds of jobs usually given to shell.

In all these different ways, then, Python can substitute for the shell and Perl processors systems administrators more commonly use. Next time you start to solve a problem with Perl or shell, take a moment to consider whether Python might do the job just as well, and leave you with a readable and maintainable solution afterward.

Next month, in Part III in this series, I'll look at how Python solves the thorny problems involving password automation that often arise in systems administration. Would you like to learn a Python answer for a question you face? Write me at: [claird@phaseit.net](mailto:claird@phaseit.net).

*Cameron Laird, a vice president at consultancy Phaseit, Inc. (<http://phaseit.net/>), is a frequent contributor to Sys Admin magazine, and a columnist for UnixReview.com.*

Copyright © 2003 UnixReview.com, [UnixReview.com's Privacy Policy](#). Comments about the Web site: [webmaster@unixreview.com](mailto:webmaster@unixreview.com)