Python for System Administration

Jerry Feldman

Boston Linux and Unix

Portions of this presentation have been borrowed with permission from Jonathan Voris, Columbia University

Presentation prepared in LibreOffice.org Impress.

Background

- I've worked as a computer programmer/software engineer almost continually since 1972. I have experience developing applications and system software in C/C++, Python, TCL/TK, shell scripting and FORTRAN.
- I first got into Python at work where someone produced a module that tied into our product. At the time I had a program on the BLU server that extracted user names and passwords from mailman and converted them to htpasswords on the web server so the websites associated with the listservs could have private members-only sections. Our first cut was JABR wrote it as a Perl script, but it ran to slow. I rewrote as a C++ program which was acceptable, but rewriting it in Python was a significant time savings because it could access mailman structures directly.

Overview

- Python was developed by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the 1980s and first published in 1991.
-
- Guido has the title of "Benevolent Dictator For Life" (BDFL)
-
- Python has been used in many applications from some of the Red Hat systems administration scripts to major applications such as mailman.
-
- It is effective both as an ad-hoc script as well as an object-oriented programming language.

- **Please feel free to interrupt and ask questions at any time.**

What is Python

- Interpreted, Interactive, Object-Oriented
  - Actually modules get compiled (talk about later)
- Multi-paradigm, yet sparse syntax
- PythonicPhilosophy:
  - Do the simplest thing that can possibly work
  - Correctness and clarity before speed
- Fully Open Source

Syntactical Summary

- Indentation or “the whitespacething”
  - Indentation is used to delineate blocks of code similar to the use of the curly braces in C/C++. This actually makes the code naturally more readable.
- Many built-in data types such as strings, lists, tuples, maps, dictionaries
- Dynamic typing
- Everything is an object

Some very nice shortcuts

- Python has a neat shortcut for lists and dictionaries called comprehensions
- List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a

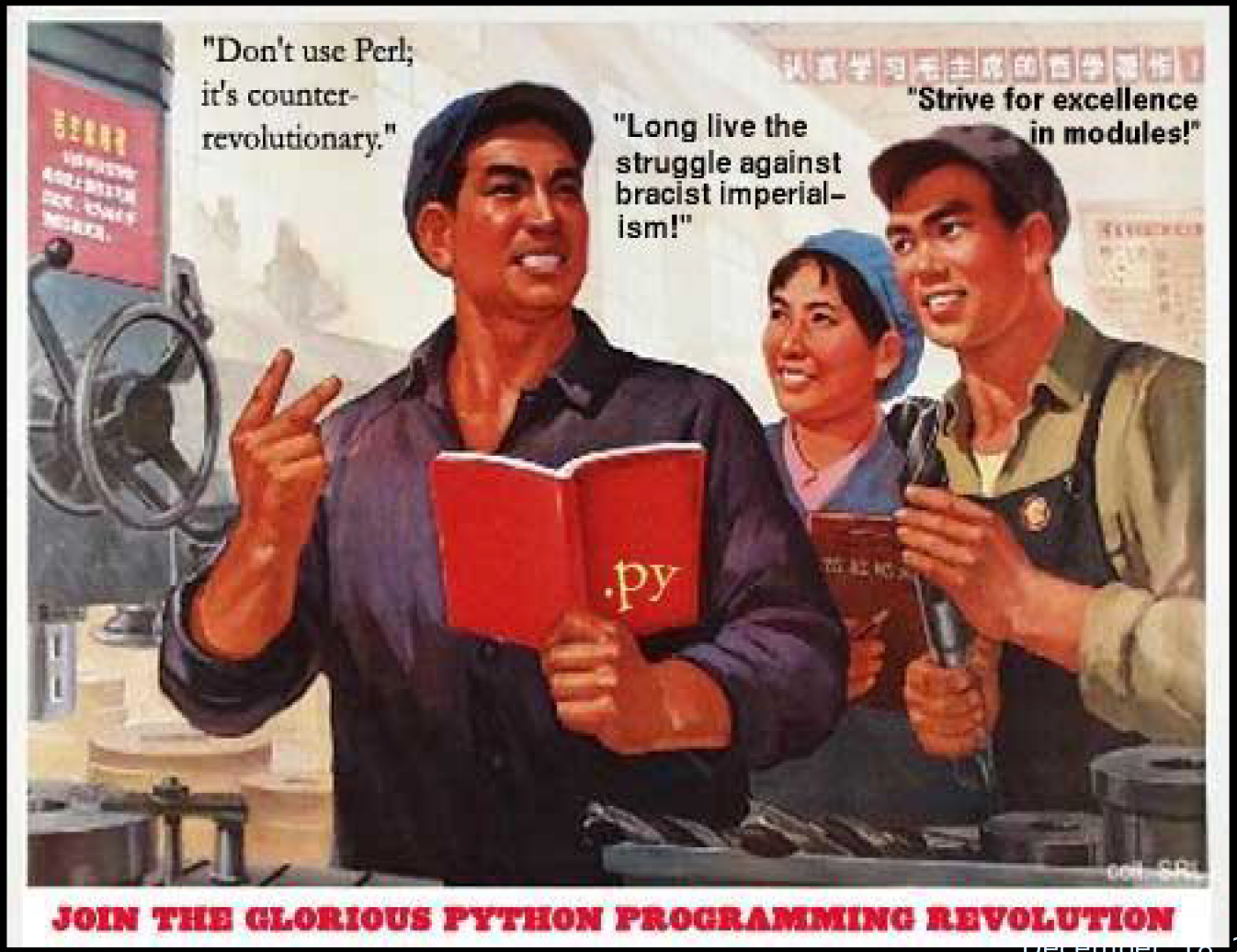
List comprehesion example

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The above can be done as:

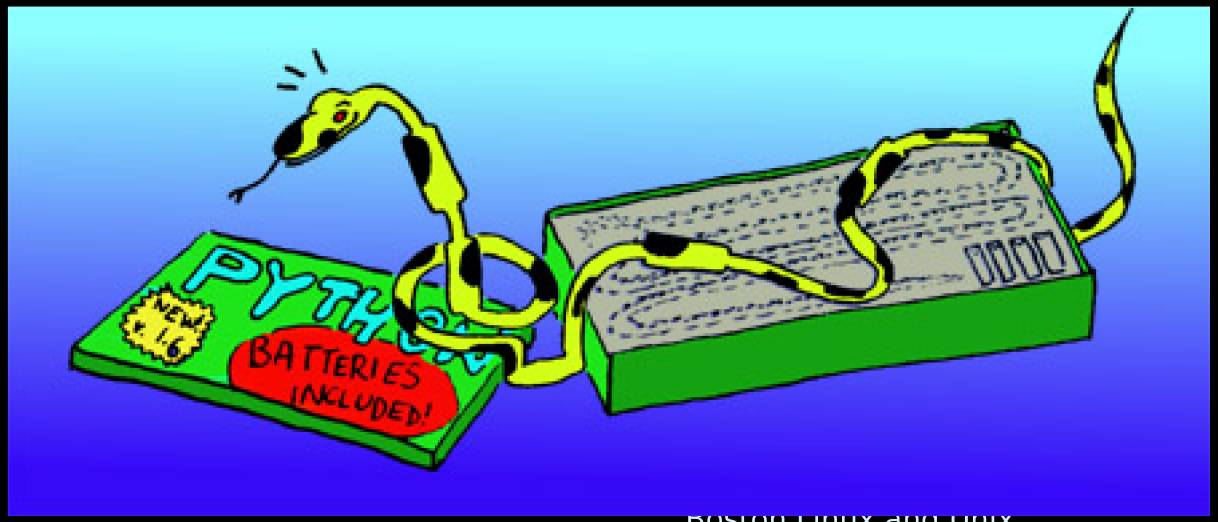
```
>>> squares = [x**2 for x in range(10)]
```

Couldn't resist this



Python Modules

- Python code organized into modules
- Large built-in standard library
- Several standard modules helpful for system programming



Python Standard Library

- Python has 2 major versions, Python 2 and Python 3. The largest body of code is Python 2.n.
  - http://docs.python.org/2/library/
  - http://docs.python.org/3/library/
- The following slides list a few of the many useful modules included with a Python distribution.

Python Modules: sys

System-specific parameters and functions

- Allows for interpreter interaction
  - Get command line arguments:

    sys.argv
  - Exit the program:

    sys.exit([arg])
  - Redirect stdout:

```
class FileFaker: #Define a clase
    def write(self, string): # define a function
in that class
        #Use the string
sys.stdout= FileFaker() # Use that class
print someObject
```

Python Modules: OS

Miscellaneous operating system interfaces

- Imports an OS-specific module (posix, nt, mac, etc)
- os.environ – Dictionary containing environment variables.
- Modify the file system:
  - os.mkdir("test")
  - os.rmdir("test")

Python Modules: subprocess Subprocess management

- subprocess.call() - Run a command and wait for completion.
- subprocess.popen() - The popen class allows you to call a process in the background.
- Example to source a BASH script into os.environ:

```
## Source the environment
    def source(self, script): #this is a method in a class in this case
        # source the script and print env(1) to stdout
        pipe = subprocess.Popen(". %s; env" % script,
stdout=subprocess.PIPE, shell=True)
        data = pipe.communicate()[0] # grab the output.
        # convert it into a py dictionary using what is known as
comprehension
        env = dict((line.split("=", 1) for line in data.splitlines()))
```

Python Modules:re Regular Expression operations

- Modeled on perl'soperations, with an OO twist
  - Create a regexobject:

    regex= re.compile('[a-z]+')
  - Perform a search:

    matches = regex.search(strToSearch)
  - Get results:

    print matches.group()

    print matches.start(), matches.end()

Some other modules I use frequently

- datetime — Basic date and time types
- glob — Unix style pathname pattern expansion
- csv — CSV File Reading and Writing
- ConfigParser — Configuration file parser
- getopt — C-style parser for command line options
- Tkinter (and TIX) — Python interface to Tcl/Tk
- pydoc — Documentation generator and online help system
- urllib: Open arbitrary resources by URL

Python vs. Other examples

- Simple programming task –count the number of times a string occurs in a file
- How do the solutions in different languages compare?

Python Source

```
1.fileName= "cleesebio.txt"
2.strToFind= "Cleese"
3.
4.try:
5.fileObject= open(fileName)
6.except:
7.print "Unable to open file " + fileName+ "."
8.sys.exit(1)
9.
10.occurances= 0
11.try:
12.for line in fileObject:
13.occurances+= line.count(strToFind)
14.finally:
15.fileObject.close()
16.
17.print "The string " + strToFind+ " occurs " + str(occurances) + " times in the file " + fileName+ "."
```

Pros:

- Extremely clear
- Fast to code
- Objects make life easier

Cons:

- A little on the slow side
- Could be more concise

VS. C

```
1.#include <stdio.h>
2.
3.intmain() {
4.FILE *infile;
5.char *fileName= "cleesebio.txt";
6.char *strToMatch= "Cleese";
7.intnumMatches= 0;
8.
9.if((infile= fopen(fileName, "r")) == NULL)
10.{
11.printf("ErrorOpening File.\n");
12.exit(1);
13.}
14.
15.char *cur;
16.while( fgets(cur, 2, infile) != NULL )
17.{
18.if (*cur == strToMatch[0])
19.{
20.fgets(cur, 6, infile);
21.if (strcmp(cur, &strToMatch[1]) == 0)
22.{
23.numMatches+= 1;
24.printf("|%i|%s|\n", numMatches, cur);
25.}
26.}
27.}
28.printf("Thestring %s occurs %i times in the file %s.\n", strToMatch, numMatches, fileName);
29.fclose(infile);
30.}
```

Pros:

- Super fast
- Very fine grain control

Cons:

- Need to compile
- Way too much code
- Pointers make Jon sad, but I like pointers but I am a C guy

vs. BASH

```
1.fileName="cleesebio.txt"
2.strToFind="Cleese"
3.occurrences=(`tr" " "\n" < $fileName| grep$strToFind| wc-w`)
4.echo "The string $strToFindoccurs $occurrences times in the file $fileName."
```

Pros:

- Extremely concise -only four lines of code!

Cons:

- Requires three separate function calls. (echo is a builtin)
- Took Jon three hours to write

vs. Perl

```
1.$fileName= "cleesebio.txt";
2.$strToFind= "Cleese";
3.
4.local $/;
5.open SLURP, $fileNameor die "can't open $file: $!";
6.
7.$data = <SLURP>;
8.
9.if (@matches = $data =~ /$strToFind/g)
10.{
11.$numMatches= @matches;
12.print "The string $strToFindoccurs $numMatchestimes in the file
$fileName.\n";
13.}
14.
15.close SLURP or die "cannot close $file: $!";
```

Pros:

- Moderately concise
- Also fast to code
- Very flexible syntax

Cons:

- Interpreted slowness
- Hard to read –what exactly is happening here?

Some personal comments

- Several years ago JABR wrote some code in either BASH or Perl to convert mailman users+passwords to htpasswords and it took about an hour to run.
  - I rewrote that in C++ and it was much faster.
  - I rewrote that in Python and it was faster by a factor of 10, not because Python is faster than C++ but because mailman is written in Python and I could access mailman's structures where C and Perl required some very slow function calls. Also, the Paramiko Python library made it easier to push the htpasswords over to the other machine.

Some personal comments

- Personally, I find Python easy to learn since I know both BASH and C and C++ and regularly code in these.
- Python is very readable and includes a number of features to easily document.
- At work I had a bash script to manage all of our servers, but I then created it in TCL/TK, and then in Python. I modified this script for the BLU servers, and I'll demo it. now.

Some additional features

- Extending and Embedding the Python Interpreter – This is a way to extend Python and link Python to C/C++ code.
- Python has a well documented C/C++ API
- Python has a number of graphics interfaces including QT, wxWidgets (GTK), and others like Tkinter, GTK+, and OpenGL.

And Finally – almost time for CBC

- Python is
  - Object Oriented (if you want to)
  - Versatile (very)
  - Clear and readable
  - Fun. I really enjoyed adapting several existing TCL and BASH applications to Python
  - Stands up against the competition. While Python is interpretive it also is compiled (see the .pyc and .pyo files)
  - Large standard library is well suited for System Administration
- See http://python.org for more information