COREY BRUNE

Python: an untapped resource in system administration

Corey Brune is currently a Master Consultant in Plano, Texas. He has been a UNIX Engineer for over 13 years, specializing in systems development, networking, architecture, and performance and tuning. His latest project is the development of parsers for performance tuning and capacity planning.

cbrune@sim10tech.com

HISTORICALLY, SYSTEM ADMINISTRA-tors chose Perl as their preferred scripting language. However, the functionality of Python may surprise those not familiar with the language. I hope to illustrate the benefits of Python within system administration while highlighting script functionality within a password-file-to-LDAP conversion script.

Python is an ideal language for beginning and expert programmers. Organizations such as Google, the New York Stock Exchange, and NASA have benefited from Python. Furthermore, Python is used behind Red Hat menu systems as well as BitTorrent clients. As a system administrator, I find Python to be an exciting and rewarding open-source language. Many first-time users are surprised at the speed at which the code falls into place when beginning to program. However, it is in the large and demanding projects that you will find Python most beneficial. This is where you will find increased manageability and time savings as opposed to other languages. Not only does Python aid in rapid deployment, but its functionality, ease of use, portability, and dependability result in hassle-free administration. Furthermore, it is compatible with all standard operating systems.

Python's ease of use is achieved primarily in the language's maintainability and elimination of code redundancy. Elimination of code redundancy is important, since duplicating code consumes time and resources. In regard to maintainability, you will notice that reading scripts is made easy. For example, Perl can be time-consuming and difficult to maintain, primarily with large programs. You want a script that can be easily read and supported, especially when modifying past or unfamiliar programs. Syntax is easy to use, read, and understand. This is primarily achieved with the language's straightforward code and similarity to the English language. Since it is object-oriented, it lends itself to easily creating modules and reducing code duplication. Python also supports Functional Programming (FP), leaving it up to the programmer to use Object Oriented Programming (OOP) or FP. Furthermore, a user can easily reuse previously created code, and specific modules can be tested rather than the entire program.

My goal in highlighting this script is to illustrate the ease in which the code falls together while spotlighting process interactions. Furthermore, I hope to demonstrate code simplicity while defining methodology.

Password-to-LDAP Conversion Script

You can find the entire listing for this script online [1]. In this article, I will just cover the highlights of the script as a way of describing Python syntax. I will also show how modules make it easy to perform system administration tasks.

Similarly to Perl and Java, Python's extensive library contains built-in modules that may be used to simplify and reduce development time. These include import pwd, sys, os, csv, and subprocess.

Note that Python statements do not end in a terminating semicolon; rather, the terminator is the end of the logical line itself.

The def keyword is used to declare a method:

```
def main(argv):
```

The main declaration accepts a list named argv. argv is similar to Perl's @ARGV or C language's **argv and contains the command-line arguments passed to the program.

Exception handling is how errors are managed within a program. For example, consider the following:

```
try:

    <code>
except IOError, (errno, strerror):
    sys.exit('I/O error (%s): %s' % (errno, strerror))

except ValueError:
    sys.exit('Could not find x in string %s: %s' % (pwLine, sys.exc_info()[0]))
```

When an exception is thrown, the programmer decides whether the script will exit, call another method, or prompt for user action. Multiple exceptions may be nested in a try block. The sys.exc_info() method returns the last exception caught.

Parsing Comma Separated Value (csv) files is simplified with the csv module. Instead of using methods such as split() to parse these files, the csv.reader() or csv.writer() methods allow for a standard mechanism for reading or writing csv files:

```
fd = csv.reader(open('shadow','r'), delimiter=':')
```

The delimiter argument allows reading or writing different csv files. Notice that the results of open() are used as the argument to csv.reader(). This syntax is common usage throughout Python.

Here is an example of file IO with Python:

```
ldifOut = open(ldifFile, 'w+')
ldifOut.close()
```

The first argument is the filename, and the second argument is the mode. There are different modes available depending on the type of operation required: read (r), read-append(r+), write (w), write-append (w+), or append (a+). The return value is a file object assigned to the variable ldifOut. The close() method closes the file object.

Lists are one of the most dynamic data types in Python. Open and close brackets with comma-delimited values declare a list. The example here declares an empty list, pwLine:

```
pwLine = []
```

Lists may be indexed, split, and searched. Furthermore, they may be used as a stack or queue and may contain other data types.

In the following code:

```
for row in fd:
    if row[1] in '*LK*' 'NP' '*' '!!':
       continue
```

the for loop iterates through the file object fd until the end of file. Conditionals and loops are terminated with a colon. Unlike other languages, loops, conditionals, and other statements are "grouped by using indentation" (python.org). The if statement says if row*[1]* matches *LK*, NP, **, or !!, to continue to the next line in the file, since these are local accounts such as root or nobody.

Python contains many of the C standard UNIX/Linux system functions and usage is similar to C functions. The pwd.getpwnam() method generates the list of users to be converted to the LDAP script:

```
String = pwd.getpwnam(line[0])
```

The argument passed is line[0], which is the username.

The pwd.getpwnam() method returns a tuple. Tuples, like strings, are read-only or immutable data types. To modify the tuple, we convert the tuple to a list:

```
pwLine = list(String)
```

Python offers many types of conversion methods, such as int(), float(), and str().

The code:

```
index = pwLine.index('x')
pwLine.pop(index)
pwLine.insert(index, line[1])
```

illustrates some of the methods available for list manipulation. pwLine.index('x') returns an integer value to where the value was found. If the value is not found, a ValueError exception is thrown. pwLine.pop(index) removes and returns the value at index. pwLine.insert(index, line[1]) inserts the encrypted password (line[1]) into the list at index.

Using the write() method allows for updating or writing files:

```
ldifOut.write('dn: cn=' + pwLine[0] + ',' + fullDN + '\n')
```

The arguments to write() illustrate how to use string concatenation with the + sign. You can access individual elements in a list or string by using brackets []. In the example here, pwLine[0] accesses the first data element. Note that lists and strings start at index number 0.

The subprocess module is the preferred mechanism when interacting with processes:

```
output = subprocess.Popen(ldapStr, shell=True, stdout=subprocess.PIPE)
```

```
output.wait()
stdout_value = output.communicate()[0]
```

subprocess.Popen() is used to invoke ldapadd and the associated arguments. shell=True indicates that the command will be passed to the shell; otherwise os.execvp() is executed. stdout=subproces.PIPE contains a pipe to control the output. Other pipes may be created for stderr and stdin. The variable output is assigned a Popen object. wait() is called to allow for the process to finish. Process output is then retrieved with the communicate()[0] method.

Every module or method has many convenient built-in methods. These are denoted by underscores on either side of the name, for example:

```
if __name__ == '__main__':
   main(sys.argv[1:])
```

The build-in method __name__ defines the module's name. The next line passes sys.argv[1:] to main(). List elements may be accessed by listname[start index:end index]. In this example, the list sys.argv[1:] will pass elements starting at index 1 through the last element.

General Notes

Python uses indentation for code blocks, such as loops and conditionals, rather than semicolons, braces, or parentheses. Statements do not require semicolons for termination.

Python contains built-in data types referred to as *numbers*, *strings, lists, tuples*, and *dictionaries*. These data types are represented by characters such as parentheses, brackets, and braces. Every data type is an object and has associated methods for manipulation.

File parsing is made simple in Python with the module re. If you are familiar with Perl, you will notice that Regular Expression Syntax is similar. Python has the capability to handle large files such as XML, CSV, binary, and text files.

Conclusion

Although I have only skimmed the surface of Python's functionality and syntax, I hope to have provided a foundation for further exploration. The application range for Python crosses over many domains such as system administration, game programming, Web programming, and research and development. The extensive library and maintainability of code make this a versatile language. Examples of functionality are highlighted in the interaction of processes, file parsing, and exceptions. If you have dabbled with Python in the past, this is an opportunity to revisit the language. Soon after programming in Python, you may find its range spreading into all aspects of administration. For further information and resources visit www.python .org/.

REFERENCE

[1] http://www.sim10tech.com/code/python/passwd2ldap.py.