The Beginner Guide to Writing Linux Shell Scripts

Joshua Price [1] 30th Jun 2011 Linux [2] 4 Comments

For starters – let's clarify that headline. Linux has more than one possible shell, and scripting any of them is a subject that can easily pack a full book [3]. What we're going to be doing is covering the basic elements of a *bash* script. If you don't know what shell you're using, it's probably *bash*. The process will be familiar to anyone who's worked with DOS's *bat* files, it's essentially the same concept. You just put a series of commands into a text file and run it. The difference comes from the fact that bash scripts can do a LOT more than batch files. In fact, bash scripting isn't all that far from a full-fledged language like Python. Today we'll be covering a few basics like input, output, arguments and variables.

Note: *If we want to get really technical, bash is not a Linux-only shell. Much (though possibly not all) of the following would apply to any UNIX-type system, including Mac OSX [4] and the BSDs.*

Hello World

It's tradition to begin a new "language" by creating a simple script to output the words "Hello World!". That's easy enough, just open your favorite text editor and enter the following:

```
#!/bin/bash
echo Hello World!
```

With only two lines, it couldn't be a whole lot simpler, but that first line, *#!/bin/bash*, may not be immediately obvious. The first two characters (often called a hashbang) are a special signal. It tells Linux that this script should be run through the /bin/bash shell, as opposed to the C shell or Korn shell or anything else you might have installed. Without it, there's no easy way for Linux to tell exactly what type of shell script this is. A Python script, for example, would likely start with something like *#!/usr/bin/python*.

After that is just the *echo* statement, which prints the words after it to the terminal (technically, to *standard output*).

Running Your Script

As is often the case with Linux, there are multiple ways to do this job. The most basic way would be to call bash manually and feed it the script file, as in

```
#Filename can be anything, .sh is a common practice for shell scripts.
bash myscript.sh
```

Clever readers may be thinking "*But wait, didn't we put that hashbang thing in so it would know to use bash? Why did I have to run bash manually?*" and the answer is "*You didn't*". At least, you wouldn't have if we had taken a moment to make the script executable on its own.

In the previous example, we launched bash and sent it the script. Now we'll save ourselves some future time by making the script executable so we dont need to

run bash manually. That's as easy as a single command.

```
chmod +x myscript.sh
```

And now it can be run with the filename directly.

Variables and Arguments

Variables in bash can be a little more confusing than some other scripting languages, partly because they sometimes need to be prefaced with a *$* character and sometimes not – depending on what you're doing. Take the following example.

```
PATH=$PATH:/home/josh/scripts
```

We refer to the same variable, PATH, two times. Once there's no *$*, but the other time there is. There are a few ways that you can remember when a $ is appropriate, but this author uses a "talking" metaphor. If I'm talking TO the variable (such as assigning it a new value) I call it by the short name, in this case PATH [5]. If I'm talking ABOUT a variable (such as getting its current value) it gets a more formal title ($PATH). The precise reasoning and inner workings of this design are beyond the scope of this guide, so just try to remember that you need to include a $ if you're trying to fetch the information in a variable.

Now we're going to use a variable in our script. Change the second line to look like the following:

```
#!/bin/bash
echo Hello $1!
```

And re-run your script, but this time include your name after the script name.

Bash auto-assigns certain variables for you, including a few such as $1, $2 etc which hold each of the arguments passed to the script. Variables can be reassigned and renamed any way you wish, so you could rewrite the previous script as

```
#!/bin/bash

firstname=$1
lastname=$2

echo Hello $firstname $lastname!
```

As you can see, there are no $ signs when assigning the value to the variable, but you do need them when pulling the info out.

Conditionals and Loops

No script could get very far without the ability to analyse or loop through data. The most common method of determining a course of action is to use the if statement. It works much like you'd expect – IF something THEN do stuff ELSE do something different. This example compares the string of characters that we stored in the variable *firstname* and compares it to some hardcoded text. If they match, it prints special output. Otherwise, it continues as normal.

```
#!/bin/bash
```

```
firstname=$1
lastname=$2

if [ "$firstname" == "Josh" ]
then
  echo "What a great name"
else
  echo Hello $firstname $lastname!
fi
```

Finally, the next core component is bash's ability to loop over data. The normal looping mechanisms for bash are FOR, WHILE, and UNTIL. We'll start with while, as it's the simplest.

```
#!/bin/bash
         counter=0
         #While the counter is less than 10, keep looping
         while [  $counter -lt 50 ]; do
             echo $counter
             let counter=counter+1
         done
```

That example creates a *counter* variable, begins a *while* loop, and continues looping (and adding one to the counter) until it reaches the limit, in this case 50 [6]. Anything after the *done* statement will execute once the loop is complete.

UNTIL operates similarly, but as the reverse of WHILE. A while loop will continue as long as its expression is true (counter less than 50). The until loop takes the opposite approach, and would be written as

```
until [  $counter -gt 50 ]; do
```

In this example, "while less than 50" and "until greater than 50" will have nearly identical results (the difference being that one will include the number 50 itself, and the other will not. Try it out for yourself to see which one, and why.)

Conclusion

As stated above, it would take a lot more than a single Getting Started article to fully demonstrate the power of bash scripting. The pieces shown here can be seen as the core components of how bash operates, and should suffice to show you the basic principles behind shell scripting in Linux. If you really want to get into the guts and start making some great scripts, check out GNU's official bash reference guide here [7]. Happy scripting!

1. http://www.maketecheasier.com/author/joshuaprice/
2. http://www.maketecheasier.com/category/linux-tips/
3. http://viglink.pgpartner.com/rd.php?r=5316&m=1280615957&q=n&rdgt=1388597327&it=1388770127&et=1389202127&priceret=8.99&pg=~~3&k=9f6ce158a46d9dc7b177ef4a8a62b9e5&source=feed&url=http%3
4. http://viglink.pgpartner.com/rd.php?r=5316&m=971612622&q=n&rdgt=1388591925&it=1388764725&et=1389196725&priceret=97.00&pg=~~3&k=4f71efc0278ee5aef791a1b6810bbf0e&source=feed&url=http%3
5. http://viglink.pgpartner.com/rd.php?r=5316&m=1442192701&q=n&rdgt=1388604110&it=1388776910&et=1389208910&priceret=8.45&pg=~~3&k=a424f912d9d136329b147af421559eb4&source=feed&url=http%3
6. http://viglink.pgpartner.com/rd.php?r=610&m=88926625&q=n&rdgt=1388590183&it=1388762983&et=1389194983&priceret=811.39&pg=~~3&k=f18eb5f530d75fd2a6b31cb62194ea69&source=feed&url=http%3A
7. http://www.gnu.org/software/bash/manual/bashref.html