Deep Dive

How to script: a BASH course

An easy step-by-step guide to the Bash command-line shell and shell scripting

PAUL VENEZIA

SHUTTERSTOCK

Because Bash has become the de-facto standard shell on most Unix-like operating systems and can generally be found everywhere, it makes an excellent place to start.

Few aspects of Unix system administration are more intimate than the relationship between the admin and their chosen shell. After all, the shell is the most fundamental interface to the system, the conduit for all command-line interactions. Thus, it's important to have a solid foundation in whatever shell you choose.

There are many different shells available, from Ash to Zsh. They all differ in many ways, and some (such as Csh and Tcsh) vary wildly from the others. Each has its selling points. However, because Bash has become the de-facto standard shell on most Unix-like operating systems and can generally be found everywhere, it makes an excellent place to start, even if you ultimately choose a different shell for daily use.

This is not designed to be an exhaustive tutorial, but is structured to introduce you to shells in general and Bash in particular, touching on many common aspects and elements of the shell. It is intended to provide a basic understanding of Bash, from the shell prompt to beginning scripting.

Command-line essentials

When you ssh into a Unix-like system, the system looks up your user account information and spawns a new shell process based on your preferences. This shell process is then connected to the ssh session, and you've logged into the system. The shell then presents you with a prompt:

```
[myname@lab1 ~]$
```

This prompt is fully configurable, and the different distributions have different ways of

Deep Dive

presenting it. Without any configuration, the prompt would simply be `$`. The dollar sign signifies that the shell session has the privileges of a regular user, not a privileged user or administrator. If you log in to a root shell, the dollar sign changes to a hash:

```
[root@lab1 ~]#
```

The prompts above contain the username, the `@` symbol, and the hostname of the system, followed by the current working directory. In this case, we're in our home directory, which is represented by the tilde. There are all kinds of ways that you can modify your prompt to better suit your workflow. The settings are stored in the PS1 environment variable. The prompt shown above is constructed like this:

```
[\u@\h \W]\$
```

This means that the shell expands `\u` to the username, `\h` to the hostname, and `\W` to the working directory. Other special characters can be added to the prompt as well. You can find a list in the Bash man page (more on that later).

But we now have a Bash prompt, and we can start using the shell to navigate around the system.

The basic file system navigation command is `cd`, short for "change directory." This is fairly self-explanatory, though note that `cd ..` will move you down a directory in the tree and just `cd` (when not followed by a directory name) will return you to your home directory. The other most common file system interaction is `ls`, short for "list." The `ls` command will show the files and directories within the current working directory.

Bash supports tab completion, which means that if you type only a few characters of a filename or command, then hit the Tab key, Bash will automatically complete the rest of the name. Tab completion often comes in handy, especially with large file names. Bash is smart about this. It knows when you're trying to enter a command or reference a file, and if you don't type enough unique characters to define a single command or file, it will show you the options that fit the characters you entered.

Bash supports tab completion, which means that if you type only a few characters of a filename or command, then hit the Tab key, Bash will automatically complete the rest of the name.

A few command options come in handy when performing normal file system navigation. The `-la` option causes `ls` to show more information about the files and directories in the current directory. It will show permissions, owner, size, and the time stamp for each file and directory. Another handy command is `-lat`, which will show a list of files with the newest files at the top:

```
[myname@lab1 ~]$ ls -lat
```

You might find that in some directories there are so many files that the list scrolls off the top of the terminal window. This is where one of the most fundamental aspects of shell interaction comes into play: the Unix pipe.

The pipe is represented by the vertical bar, `|`. It functions just as you might guess, allowing the output of one command to be "piped" into the input of another command. This allows you to string together many commands at once to modify the output on the screen, to cause certain actions to be taken on a set of files, or something similar.

For example, we might want to use the less command to introduce pagination into our file listing. Thus, we would enter:

```
[myname@lab1 ~]$ ls -lat | less
```

This would pass the output of the `ls -lat` command to `less`. The `less` command allows us to view the output a page at a time by pressing the space bar or to advance the text by pressing the Enter key.

Another common example of using a pipe is to manipulate entries in a text file. For instance, we might have a file named names.txt that has hundreds or thousands of names entered, one per line, like so:

```
Jack
John
Mary
Steven
Mark
Steven
Mary
```

Deep Dive

Notice that the file contains some duplicates. Let's say we want to alphabetize the list and remove the duplicate entries. This can quickly and easily be done with one line:

```
[myname@lab1 ~]$ sort names.txt | uniq
```

This command causes the shell to invoke the `sort` tool to sort the list, then pipe the output to `uniq`, which removes all the duplicate lines. Now, we probably want that output to go into another file, so we'd instruct the shell to create a new file and place the output there:

```
[myname@lab1 ~]$ sort names.txt | uniq > sortednames.txt
```

Here we've used shell output redirection to create the new file called sortednames.txt with the output of the previous commands. The shell uses `<` and `>` as directional markers, as well as `<<` and `>>`. There's a vital difference between these, as the single `>` will cause a file to be overwritten, whereas `>>` will cause the output to be appended to any data already in the file. It's important to get this straight early on, or you may find that you have unwittingly deleted the contents of a file that you intended to add content to.

The `<` symbol is used to direct text or command output from one file to another or back to a program. For instance, you might have a file that contains some commands that you want another program to run. Thus, you might enter:

```
[myname@lab1 ~]$ myprogram < ./mycommands.txt
```

Many other Bash operators can be used in normal shell sessions or in Bash scripting, but we've covered the major ones. A full listing can be found in the Bash man page, which is accessible by typing `man bash`. It's a lengthy document, but then again, shells do quite a lot, and all of that functionality needs to be documented.

Globs are Bash's method of pattern matching. Bash doesn't have regular expressions, so globs are used to make it easier to work on multiple files that match a single pattern.

Special Bash files

There are a few special files in your home directory that Bash reads when you start a new shell. These are .bash_profile, .bashrc, .bash_history, and .bash_logout. Note that the file names start with a period, which designates them as hidden files. To see them in a directory listing, you have to use the -a switch with `ls`, or `ls -a`.

The .bash_history file contains the commands you've entered into the shell. These can be viewed by running the history command. The .bash_profile and .bashrc files contain special instructions to create your environment by setting aliases, environment variables such as your path, and any other variables you wish to set or commands you want to run. For instance, you might add this line to .bashrc:

```
export PS1='[\u@\h \W - \d]\
```

It would set your prompt to show the date:

```
[myname@lab1 ~ - Mon Feb 25]$
```

You can also set aliases in your .bashrc file. Aliases are a handy way to reduce typing for common commands. For instance, if you find that you're constantly having to type something like cd ~/myfiles/documents/project1/data, you might want add an alias to your .bashrc like so:

```
alias cdp1=' cd ~/myfiles/documents/project1/data'
```

Deep Dive

Then you would only have to type to navigate to the directory.

The purpose of the .bash_logout file is to store commands to be run when you terminate the shell, but it's not used often.

File globbing

Globs are Bash's method of pattern matching. Bash doesn't have regular expressions, so globs are used to make it easier to work on multiple files that match a single pattern.

A simple glob would be `file.*`. The asterisk matches anything, so this expression would match files named file.1 and file.reallylongnamehere. Globs can get more complex and thus more useful. A glob such as `file.?` narrows the options. This would match file.1 or file.2 or file.a, but would not match file.11 or file.ab or file.reallylongnamehere. This is because the ? denotes a single character.

You can also use square brackets to reference multiple options, so `file.[ab]` would match file.a and file.b but not file.c. Further, you can use brackets to select based on ranges of the alphabet or numbers. Thus, `file.[a-d]` would match file.a, file.b, file.c, and file.d but not file.e. You can also negate a match with the ^ character. Thus, file.[^ab] would match neither file.a nor file.b, but it would match file.c.

File globs are very different from regular expressions, but they are very handy when working with many files or in scripting.

File globs are very different from regular expressions, but they are very handy when working with many files or in scripting.

Standard output and standard error

The two main methods that Bash uses to pass output from programs or scripts back to your session are called stdout and stderr. These are used for normal output and error reporting, respectively. If a program is running normally, it will use stdout to communicate with the session, but if something fails, it will send errors through stderr instead. Among other things, this allows you to redirect normal or error output from any program to a file instead of the screen.

You generally don't have to worry too much about stdout and stderr when you're just getting started, but understanding what they are and what they do will be important when you get deeper into Bash.

Background processing and job control

Although the shell might seem one-dimensional, in fact you can run many different jobs at the same time. For instance, say you wanted to run a program that you knew would take a long time to complete, and you had other things to do in the shell session in the meantime -- no need to just sit and wait. Rather, tell Bash to run the process in the background. You do this by adding an ampersand after the command line:

```
[myname@lab1 ~]$ ./longprocess.bin &
[1] 3104
```

The response below the command is Bash's notification that you've started job No. 1, and its process ID is 3104. The process will continue to run in the background until it completes. When it's done, Bash will let you know:

```
[1]+  Done      ./longprocess.bin
```

If you had not added the & when you ran `longprocess.bin`, you would not get a Bash prompt back until the job completed.

You can bring a background job back into the foreground by using the `fg` command. Likewise, you can place a foreground job into the background by using the `bg` command. You can stop a process

Deep Dive

that's running in the foreground by hitting Ctrl-Z, then place it in the background by entering `bg %1`:

```
[myname@lab1 ~]$ ./longprocess.bin
[1]+  Stopped                 ./longprocess.bin
[myname@lab1 ~]$ bg %1
[1]+ ./longprocess.bin
[myname@lab1 ~]$
```

You can view a list of running jobs by using the jobs command:

```
[myname@lab1 ~]$ jobs
[1]+  Running                 ./longprocess1.bin
[2]+  Running                 ./longprocess2.bin
-afG
[3]+  Running                 ./longprocess3.bin -rT
```

One of the major functions that Bash provides is the ability to run commands in a loop.

You can use the `fg` command to bring one of them back into the foreground:

```
[myname@lab1 ~]$ fg %1
./longprocess1.bin
```

Note that any output produced by background processes will be shown in the shell, which can get messy if there's a lot of it. Fortunately, we can use our operators to tell Bash to send the output to `/dev/null`, which will prevent it from displaying in our session:

```
[myname@lab1 ~]$ ./longprocess1.bin > /dev/null &
```

This will still show us error messages sent to stderr, but not output sent to stdout. If we wanted to redirect stderr as well, we would type this:

```
[myname@lab1 ~]$ ./longprocess1.bin > /dev/null 2>&1 &
```

This tells Bash to send stderr (2) to the same place as stdout (1).

Bash loops

One of the major functions that Bash provides is the ability to run commands in a loop. Loops allow you to quickly perform many functions on a file or set of files or just for general output.

For instance, you might use a loop to systematically and, very quickly, rename a large number of files. Let's say we have a directory full of files like so:

```
Info1.txt
Info2.txt
Info3.txt
Info4.txt
Info5.txt
```

These files have served their purpose, and now we need to rename each of them to Info#.txt.old. Rather than manually typing in each name and using the `mv` (or `move`) command, we would be better off using a `for` loop:

Deep Dive

```
for i in 'ls *.txt'; do mv $i $i.old; done
```

This is a very simple one-line loop command that will rename all the files ending in .txt to their original name with .old appended at the end. Thus, our files are now named Info1.txt.old, Info2.txt.old, and so forth.

What this loop is doing is using the output of the `ls *.txt` command as a list of file names, which is represented by the variable `$i`. Bash will take each file name at a time, then run the mv `$i $i.old` command, replacing the variable with the actual filename.

Bash provides other looping methods as well, such as while loops. You can use while loops to keep performing functions until a certain condition is met. For instance, you might want to perform some functions only on a few files, drawing on a text file with the names of those files. The while loop would then be:

```
while read filename; do mv $filename $filename.old; done < ./filelist.txt
```

This command uses the < operator to redirect the contents of the `filelist.txt file` into the loop, where each file name is placed into the $filename variable. As soon as there are no more lines to be read from filelist.txt, the loop exits.

Bash shell scripting is relatively easy to learn, and it can be surprisingly powerful. You can use shell scripts to automate common tasks or perform any other operation that you could normally run from the command line.

Bash shell scripting

Shell scripting plays a large role in the normal functions of a Unix-like system. Shell scripts are used by many distributions to start system services at boot, and by a wide variety of software packages to perform maintenance and configuration tasks. Shell scripting essentially forms the nuts and bolts of a Unix-like system.

Bash shell scripting is relatively easy to learn, and it can be surprisingly powerful. You can use shell scripts to automate common tasks or perform any other operation that you could normally run from the command line. Every shell script starts out with the hashbang, or `#!`.

The hashbang is followed by the path to the executable that should run the script. It would typically be `#!/bin/bash` on most systems. The rest of the script follows.

Let's take a look at a simple script that looks in one directory for files with names matching a glob or filename pattern. It then uses `grep` to check for a particular text string in those files, and if it finds it, it moves the file to a new directory.

```
#!/bin/bash
# We declare two variables for the two directories
firstdir=dir1
seconddir=dir2
# The for loop that moves the right files
for i in 'grep -l matchpattern $firstdir/*'; do
        mv $i $seconddir
        echo $i
done
```

The first thing we do is declare two variables in the script: `firstdir` and `seconddir`. These contain the directory names we'll be working with. Then we create a for loop that uses `grep` to find the names of files that contain the string `matchpattern`. The `-l` switch tells `grep` to output only the name of files that match, rather than also displaying the match pattern itself. Thus, the `$i` variable in the for loop contains the file name of a matching file every time the loop runs. If there are 10 matching files, the loop will run 10 times, with the `$i` variable containing the file name of the next

Deep Dive

matching file.

During each run of the loop, the mv command is used to move the file from `firstdir` to `seconddir`. The script then echoes the file name back to standard output. This means that if you run the script, you will see a list of files that have matched the pattern and have been moved to `seconddir`.

Note that you can use a text editor such as pico, nano, or vim to create these scripts. Once you've created the script (let's name it `script.sh)`, you will need either to make the script executable or to run the script by calling Bash explicitly. You can make an executable script run simply by typing its name on the command line (assuming it has the right permissions).

You make a script executable using the `chmod` command:

```
chmod +x script.sh
```

To run a script by calling Bash explicitly, enter:

```
bash ./script.sh
```

Now, let's make this a little more complex and allow for the match pattern to be specified on the command line:

```
#!/bin/bash
# We declare two variables for the two directories
pattern=$1
firstdir=dir1
seconddir=dir2
# The for loop that moves the right files
for i in 'grep -l "$pattern" $firstdir/*'; do
        mv $i $seconddir
        echo $i
done
```

You can see here that we have a new variable called `$pattern`, which is set to $1. The expression `$pattern=$1` tells Bash to take the first argument given to the script on the command line and place it in the `$pattern` variable. The value of `$pattern` is then used by `grep` to find the files. If we named this script `movefiles.sh`, we would run it like so:

```
./movefiles.sh matchpattern
```

Here's an alternative:

```
./movefiles.sh otherpattern
```

Now we have a script that lets us search for any pattern we like. However, there are no protections in place if the user does not enter an argument on the command line. In order to avoid problems, let's put in a little bit of error checking:

```
#!/bin/bash
if [ -z $1 ]; then
        echo "No pattern given."
```

Deep Dive

```
        echo "Usage: $0 <pattern>"
        exit
fi
# We declare two variables for the two directories
pattern=$1
firstdir=dir1
seconddir=dir2
# The for loop that moves the right files
for i in 'grep -l "$pattern" $firstdir/*'; do
        mv $i $seconddir
        echo $i
done
```

We've added a check at the top to make sure that something was entered on the command line. We use the Bash built-in test mechanism to check if the length of `$1` is nonzero. If the length of `$1` is zero, then clearly nothing was entered. In that case, our if/then statement echoes that there was an error, shows the proper usage of the script, then exits. Note the `$0` here. That is a special variable that contains the name of the script.

The Bash built-in test is represented by the `[` and `]` brackets. Bash will evaluate the statement within the brackets and determine if it's true or false or a variety of other qualifications. In this case, we use the `-z` flag to test the length of the text in the `$1` variable. If the length of `$1` is nonzero (that is, the statement in the brackets is false), the if/then statement skips processing and the script runs normally. Testing is a relatively complex evaluation mechanism, and it can be used in myriad ways. You can learn about the full scope of the tool by running man test.

The more time you spend with Bash -- or with any shell -- the more reflexes you will develop and the more natural it will feel. Eventually you will stop thinking about the specific commands or operators. Instead, you will see only the end goal, and getting there will become a simple matter indeed. We don't think about moving a mouse to click a button in a GUI, and eventually, you won't think about which flags to use with `ls` or `grep`, or how to use pipes and operators and loops to manipulate some data. You'll just do it. ■

The more time you spend with Bash – or with any shell – the more reflexes you will develop and the more natural it will feel.

Paul Venezia *is a veteran *nix system and network architect, and senior contributing editor at InfoWorld, where he writes analysis and reviews.*