Bash Scripting

Somenath Sinha

April 2018

Contents

I Bash Scripting Fundamentals **3**

1 Creating Your First Shell Script **4**

1.1 Why Scripting in Bash Makes Sense 4

1.1.1 Understanding Scripting 4

1.2 Choosing an Editor 4

1.3 Core Bash Script Ingredients 5

1.3.1 exit statement 5

1.3.2 Script to print Hello World 5

1.4 Storing and Running the Script 6

1.4.1 Executing a script in a directory within $PATH 6

1.4.2 Executing a script in a directory NOT within $PATH 6

1.4.3 Multiple files with same name in $PATH 7

2 Working with Variables and Parameters **9**

3 Transforming Input **10**

4 Using Essential External Tools **11**

5 Using Conditional Statements **12**

6 Using Advanced Scripting Options **13**

7 Script Debugging and Analyzing **14**

8 Scripting by Example **15**

II Advanced Bash Scripting 16

9 Reviewing Basics 17

10 Scripting Best Practices 18

11 Understanding Syntax Differences 19

12 Using Advanced awk, sed, and Regular Expressions 20

13 Analyzing Advanced Scripts 21

14 Writing a Complex Script 22

15 Scripting for Performance 23

16 Beyond Bash - Scripting in Python 24

Part I

Bash Scripting Fundamentals

Chapter 1

Creating Your First Shell Script

1.1 Why Scripting in Bash Makes Sense

On most versions of Linux and **nix*es (e.g., Unix), **bash** is the most common shell. While other alternative shells are available, this is the one that's present almost on all such operating systems. In fact, many parts of the software running on the OS and many scripts on the OS are written with bash as well! Some prominent examples are the scripts in `/etc/init.d` and the file `/etc/profile` (makes changes to the environment for all users after login).

1.1.1 Understanding Scripting

A script is a simple program that can be directly executed and doesn't need to be compiled before use. At the most basic level, a script can contain a list of commands to be executed sequentially, such as:

```
#/bin/bash
clear
ls -l
df -h
```

However, it may also contain variables, flow control statements (loops and conditional statements), processing of user input, etc. Using shell scripts, we can automate repetitive tasks which can reduce our workload significantly.

1.2 Choosing an Editor

While any text editor works for writing shell scripts, **vim** editor (or the older *vi* editor) is available on almost every *nix OS by default and it supports syntax highlighting, which makes writing scripts easier. To initiate the syntax highlighting, we should enter the *shebang* (`#/bin/bash`) on the first line, then close and reopen the file. The syntax highlighting is especially useful in bash since it's a scripted language and unexpected highlighting may reveal syntax errors.

1.3 Core Bash Script Ingredients

The first line of a bash script should always be the **shebang**, a statement that defines which program should be used to execute the script. In our case, the shebang for bash scripts looks like:

```
#/bin/bash
```

Note that the # character doesn't start a comment in this case, unlike everywhere else (since it's the first character on the very first line)! Thus, the shebang for bash is: `#!/bin/bash`, shebang for perl is `#!/usr/bin/perl`, and so on.

The *shebang* is followed by comment lines describing the function of the script, as well as comments throughout the rest of the script wherever needed to explain functionality. Blank lines and spaces/tabs should also be included and consistently added to keep the script readable.

1.3.1 exit statement

Every command on Linux has an exit status that is `0` if the command was successful or any other number if not. This exit status can be accessed with `#?` environment variable:

```
# ls
Apps   Desktop    Downloads  Notes     Public  rpmbuild  Templates  Videos
Cloud  Documents  Music      Pictures  Repo    Scripts   tmp        VM
# echo $?
0
# ls doesntExist
ls: cannot access 'doesntExist': No such file or directory
# echo $?
2
```

If we do not want the exit status of the last command in the script to be used as the exit status of the script itself, we can use an exit statement:

```
exit 1 # indicates some error occurred
```

1.3.2 Script to print Hello World

The command to print something to the terminal is `echo`:

```
#!/bin/bash
#
# Demo script that greets the world
# Usage: ./hello.sh

clear
echo "Hello, World!"

exit 0
```

Typically, `exit 1` is used to indicate some error has occurred, and `exit 2` onwards is used to indicate a specific error determined by the programmer.

1.4 Storing and Running the Script

Every script needs to have the executable permission set for the current user:

```
# chmod u+x script.sh
```

Now, how a particular script is executed from the command line depends on the location of the storage of the script. Specifically, whether the location of the script is a directory within the **$PATH** variable or not.

1.4.1 Executing a script in a directory within $PATH

When bash encounters a command, it searches all the folders within the `$PATH` variable sequentially to see if a command (i.e., executable file) with that name exists in the folder. If it does, searching stops and the subsequent folders in the `$PATH` variable are ignored, and the command is executed. If not, an error is returned. The `$PATH` variable typically has a value such as:

```
echo $USER
somu
$ echo $PATH
/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/home/somu/.local/bin:/home/somu/bin
```

For our own scripts, an apt location is `/usr/local/bin` or `$USER/bin`, (i.e., `/home/somu/bin` for the user 'somu'). While `$USER/bin` would make the script available to only the specific user, `/usr/local/bin` would make the script available to all users on the system. In either case, this would allow us to run our scripts from anywhere within the file system. For example:

```
#!/bin/bash
#
# Prints the present working directory
# Usage: whereami

pwd
```

This would have the output:

```
# which whereami
/usr/local/bin/whereami
# whereami
/home/somu
# cd ~
# whereami
/root
```

The distinct advantage of this method is that the script can be executed from anywhere without specifying the location of the script.

1.4.2 Executing a script in a directory NOT within $PATH

If however, we choose to store the script elsewhere, i.e, a directory not mentioned within the `$PATH` variable, then the program won't be found when used as in the last section. The shell doesn't realize that the file we want to execute in the present directory *because the present directory is not within the $PATH variable* in Linux for security reasons. Thus, we have to explicitly mention that we want a script to be executed by providing the location of the script, which for the present directory is:

```
$ test.sh
bash: test.sh: command not found...
$ chmod u+x test.sh
$ ./test.sh
This is a test
```

The above output is for a file `/home/somu/test.sh` with the content:

```
#!/bin/bash
echo "This is a test"
```

Note that instead of `./test.sh`, we could also have used `/home/somu/test.sh` but the former is just more convenient. An important point to note is that if the `./` is not given in the name of the executable, then bash would assume the script is present within a directory in `$PATH` and in the best case won't work, or it'll execute some other script with the same name but within a directory in `$PATH`, thus producing unexpected results! For example if our script is called test:

```
$ chmod +x test
$ test
$ which test
/usr/bin/test
$ ./test
This is a test
```

The first command with `test` didn't execute the original test file in the present working directory, but the bash function `test` located within `/usr/bin/test` where `/usr/bin` is in the path variable for all users.

Note that if the script is started as an argument to the bash shell (i.e., create a new sub-shell that runs the script), the script itself doesn't need to have executable permissions set:

```
# bash script.sh
```

1.4.3 Multiple files with same name in $PATH

We know the system provided binary `/usr/bin/test` is in a directory in the $PATH variable. If however, we put our `test` script in suppose, the `/home/somu/bin` directory, we'll find the system still uses the `/usr/bin/test` file, since `/usr/bin` appears before `/home/somu/bin` in the `$PATH` variable: `/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/home/somu/.local/bin:/home/somu/bin`.

Again, once bash finds a file name within one of the `$PATH` folders, it won't look any further within the remaining `$PATH` folders to find other files with the same name! Thus, it's best to ensure that our script name is *unique* for the present system. To ensure that there are no clashes, even before we start creating the script, we can use the `which` command to find out if any other executable with that name exists in the `$PATH` folders. If our script name is `doesntYetExist`, the expected output will be:

```
$ which doesntYetExist
/usr/bin/which: no doesntYetExist in
 ↪  (/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/home/somu/.local/bin:/home/somu/bin)
```

If however, there is already a command with that name, our command becomes:

```
$ which script
/usr/bin/script
```

1.5 Using Bash Internal Commands versus External Commands

In a script, we can use both bash **internal** and **external** commands.

1.5.1 Internal Commands

An internal command is any command that is a part of the bash shell. Such a command is faster because it doesn't have to be loaded from the disk. A list of all bash internal commands can be viewed with:

```
$ help
GNU bash, version 4.4.19(1)-release (x86_64-redhat-linux-gnu)
These shell commands are defined internally.  Type ‘help’ to see this list.
Type ‘help name’ to find out more about the function ‘name’.
Use ‘info bash’ to find out more about the shell in general.
Use ‘man -k’ or ‘info’ to find out more about commands not in this list.

A star (*) next to a name means that the command is disabled.

job_spec [&]                            history [-c] [-d offset] [n] or hist>
(( expression ))                        if COMMANDS; then COMMANDS; [ elif C>
. filename [arguments]                  jobs [-lnprs] [jobspec ...] or jobs >
:                                       kill [-s sigspec | -n signum | -sigs>
[ arg... ]                              let arg [arg ...]
[[ expression ]]                        local [option] name[=value] ...
alias [-p] [name[=value] ... ]          logout [n]
bg [job_spec ...]                       mapfile [-d delim] [-n count] [-O or>
bind [-lpsvPSVX] [-m keymap] [-f file>  popd [-n] [+N | -N]
break [n]                               printf [-v var] format [arguments]
builtin [shell-builtin [arg ...]]       pushd [-n] [+N | -N | dir]
caller [expr]                           pwd [-LP]
case WORD in [PATTERN [| PATTERN]...)>  read [-ers] [-a array] [-d delim] [->
cd [-L|[-P [-e]] [-@]] [dir]            readarray [-n count] [-O origin] [-s>
command [-pVv] command [arg ...]        readonly [-aAf] [name[=value] ...] o>
compgen [-abcdefgjksuv] [-o option] [>  return [n]
complete [-abcdefgjksuv] [-pr] [-DE] >  select NAME [in WORDS ... ;] do COMM>
compopt [-o|+o option] [-DE] [name ..>  set [-abefhkmnptuvxBCHP] [-o option->
continue [n]                            shift [n]
coproc [NAME] command [redirections]    shopt [-pqsu] [-o] [optname ...]
declare [-aAfFgilnrtux] [-p] [name[=v>  source filename [arguments]
dirs [-clpv] [+N] [-N]                  suspend [-f]
disown [-h] [-ar] [jobspec ... | pid >  test [expr]
echo [-neE] [arg ...]                   time [-p] pipeline
```

```
enable [-a] [-dnps] [-f filename] [na>  times
eval [arg ...]                          trap [-lp] [[arg] signal_spec ...]
exec [-cl] [-a name] [command [argume>  true
exit [n]                                type [-afptP] name [name ...]
export [-fn] [name[=value] ...] or ex>  typeset [-aAfFgilnrtux] [-p] name[=v>
false                                   ulimit [-SHabcdefiklmnpqrstuvxPT] [l>
fc [-e ename] [-lnr] [first] [last] o>  umask [-p] [-S] [mode]
fg [job_spec]                           unalias [-a] name [name ...]
for NAME [in WORDS ... ] ; do COMMAND>  unset [-f] [-v] [-n] [name ...]
for (( exp1; exp2; exp3 )); do COMMAN>  until COMMANDS; do COMMANDS; done
function name { COMMANDS ; } or name >  variables - Names and meanings of so>
getopts optstring name [arg]            wait [-n] [id ...]
hash [-lr] [-p pathname] [-dt] [name >  while COMMANDS; do COMMANDS; done
help [-dms] [pattern ...]               { COMMANDS ; }
```

1.5.2 External Commands

Contrastingly, an external command needs to be loaded from a file on the OS. This is because internal commands are loaded with the bash shell, and are stored in the memory. Another point of interest is the fact that the `which` command doesn't return any information about internal commands! For example, there are two test commands:

```
$ help | grep test
disown [-h] [-ar] [jobspec ... | pid >  test [expr]
$ which test
/usr/bin/test
```

So, there's an internal command `test` and an external command `/usr/bin/test`. To find out if there's an internal command with a particular name, we use:

```
$ type test
test is a shell builtin5
```

The bash shell first checks if a command exists as an internal command for the given command name. If it does, it's executed. If it doesn't exist as an internal command, only then is the `$PATH` searched for an external command of the same name. Thus, in cases of conflict between internal and external command, internal commands always win.

1.6 Finding Help About Scripting Components

1.6.1 man bash

The `man bash` command has the documentation for everything used in a bash script and thus contains all the help we'll need with scripting syntax, etc. but it's also very large. We can search the man page using the `/<search String>` using vim, but better ways might exist.

1.6.2 help command

For help regarding the bash internal commands, we can use the `help <internal Command>` to get details. For example, for the `trap` internal command, the documentation is:

```
$ help trap
trap: trap [-lp] [[arg] signal_spec ...]
Trap signals and other events.

Defines and activates handlers to be run when the shell receives signals
or other conditions.
...
```

External commands typically have their own man pages and some even have a `--help` option to display the syntax and usage at a glance.

1.6.3 Online Resources

While there are tons of resources regarding every domain of shell scripting on the internet, a particularly useful one is the The Advanced Bash-Scripting Guide on `tldp.org` (The Linux Documentation Project). It's authoritative (i.e., tries to contain all information about each component), but it's also large and might not be easy to understand.

Chapter 2

Working with Variables and Parameters

Chapter 3

Transforming Input

Chapter 4

Using Essential External Tools

Chapter 5

Using Conditional Statements

Chapter 6

Using Advanced Scripting Options

Chapter 7

Script Debugging and Analyzing

Chapter 8

Scripting by Example

Part II

Advanced Bash Scripting

Chapter 9

Reviewing Basics

Chapter 10

Scripting Best Practices

Chapter 11

Understanding Syntax Differences

Chapter 12

Using Advanced awk, sed, and Regular Expressions

Chapter 13

Analyzing Advanced Scripts

Chapter 14

Writing a Complex Script

Chapter 15

Scripting for Performance

Chapter 16

Beyond Bash - Scripting in Python