CHAPTER-1 IMPROVE COMMAND-LINE PRODUCTIVITY

Write Simple Bash Scripts

Create and Execute Bash Shell Scripts

A Bash shell script is an executable file that contains a list of commands, and possibly with programming logic to control decision-making in the overall task. When well-written, a shell script is a powerful command-line tool on its own, and you can use it with other scripts.

Shell scripting proficiency is essential for system administrators in any operational environment. You can use shell scripts to improve the efficiency and accuracy of routine task completion. Although you can **use any text editor**, advanced editors **such as vim or emacs** understand Bash shell syntax and can provide color-coded highlighting. This highlighting helps to identify common scripting errors such as improper syntax, unmatched quotes, parentheses, brackets, and braces, and other structural mistakes.

Specify the Command Interpreter

The **first line** of a script begins **with the #! notation,** which is commonly referred to **as she-bang or hash-bang,** from the names of those two characters, sharp or hash and bang. This **notation is an interpreter directive that indicates the command interpreter** and optional command options that are needed to process the remaining lines of the file. For normal Bash syntax script files, the first line is this directive:

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

Execute a Bash Shell Script

A shell script file must have execute permissions to run it as an ordinary command. Use the chmod command to modify the file permissions. Use the chown command, if needed, to grant execute permission only for specific users or groups.

If the **script is stored in a directory that is listed in the shell's PATH environmental variable**, then you **can run the shell script by using only its file name, similar to running compiled commands**. Because **PATH parsing runs the first matching file name that is found**, always **avoid using existing command names** to **name your script files**. If **a script is not in a PATH** directory, **run the script using its absolute path name**, which can be determined by querying the file with the which command. Alternatively, run a script in your current working directory using the. Directory prefix, such as ./scriptname.

```
[user@host ~]$ which hello
~/bin/hello
[user@host ~]$ echo $PATH
/home/user/.local/bin:/home/user/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/
sbin:/usr/local/bin
```

Quote Special Characters

A number of characters and words have special meaning to the Bash shell. When you need to use these characters for their literal values, rather than for their special meanings, you will escape them in the script**. Use the backslash character** (\), single quotes ('), or **double quotes ("") to remove (or escape)** the **special meaning of these characters**.
The **backslash character removes the special meaning of the single character** that **immediately follows the backslash**.

For example, to use the echo command **to display the # not a comment literal string**, the **# hash character must not be interpreted as a comment.**

The following example shows the **backslash character (\) modifying the hash** character so it is not interpreted as a comment.

```
[user@host ~]$ echo # not a comment

[user@host ~]$ echo \# not a comment
# not a comment
```

To escape more than one **character in a text string,** either use **the backslash character multiple times or enclose the whole string in single quotes (')** to interpret literally. Single quotes preserve the literal meaning of all characters that they enclose. Observe the backslash character and single quotes in these examples:

```
[user@host ~]$ echo # not a comment #

[user@host ~]$ echo \# not a comment #
# not a comment
[user@host ~]$ echo \# not a comment \#
# not a comment #
[user@host ~]$ echo '# not a comment #'
# not a comment #
```

Use **double quotation marks** to **suppress globbing** (file name pattern matching) and shell expansion, but still allow command and variable substitution. Variable substitution is conceptually identical to command substitution, but might use optional brace syntax.

Observe the following examples of various quotation mark forms. Use single quotation marks to interpret all enclosed text literally. Besides suppressing globbing and shell expansion, single quotations also direct the shell to suppress command and variable substitution. The question mark (?) is included inside the quotations, because it is a metacharacter that also needs escaping from expansion.

```
[user@host ~]$ var=$(hostname -s); echo $var
host
[user@host ~]$ echo "***** hostname is ${var} *****"
***** hostname is host *****
```

```
[user@host ~]$ echo Your username variable is \$USER.
Your username variable is $USER.
[user@host ~]$ echo "Will variable $var evaluate to $(hostname -s)?"
Will variable host evaluate to host?
[user@host ~]$ echo 'Will variable $var evaluate to $(hostname -s)?'
Will variable $var evaluate to $(hostname -s)?
[user@host ~]$ echo "\"Hello, world\""
"Hello, world"
[user@host ~]$ echo '"Hello, world"'
"Hello, world"
```

Provide Output from a Shell Script

The echo command displays arbitrary text by passing the text as an argument to the command.

```
[user@host ~]$ cat ~/bin/hello
#!/usr/bin/bash

echo "Hello, world"

[user@host ~]$ hello
Hello, world
```

Loops and Conditional Constructs in Scripts

Process Items from the Command Line

In Bash, the for loop construct uses the following syntax:

```
for VARIABLE in LIST; do
COMMAND VARIABLE
done
```

The **loop processes** the **strings** that you provide **in LIST** and exits after processing the last string in the list. The **for loop temporarily stores each list string** as the **value of VARIABLE**, then **executes the block of commands** that **use the variable**. The variable name is arbitrary. Typically, you reference **the variable value with commands** in the command block.
Provide **the list of strings** for the **for loop from a list** that the **user enters directly**, or that is generated from shell expansion, such **as variable, brace**, or file name expansion, or command substitution.

Bash Script Exit Codes

After **a script interprets and processes** all of **its content**, the **script process exits and passes control back to the parent process** that called it. However, a script can be exited before it finishes, such as when the script encounters an error condition. Use the exit command to immediately leave the script, and skip processing the remainder of the script.

Test Logic for Strings and Directories, and to Compare Values

To ensure that unexpected conditions do not easily disrupt scripts, it is recommended to verify command input such as command-line arguments, user input, command substitutions, variable expansions, and file name expansions. You can check integrity in your scripts by using the Bash test command.

To see the exit status, view the $? Variable immediately following the execution of the test command. **An exit status of 0 indicates a successful exit with nothing to report**, and **nonzero values indicate some condition or failure**. Perform tests by using various operators **to determine whether a number is greater than (gt), greater than or equal to (ge), less than (lt), less than or equal to (le), or equal (eq) to another number.**

Use operators to test whether a **string of text is the same (= or ==)** or **not the same (!=)** as another string of text, or whether the string has zero length (z) or has a non-zero length (n). You can also test if a regular file (-f) or directory (-d) exists and some special attributes, such as if the file is a symbolic link (-L) or if the user has read permissions (-r).

The following examples demonstrate the test command with Bash numeric comparison operators:

```
[user@host ~]$ test 1 -gt 0 ; echo $?
0
[user@host ~]$ test 0 -gt 1 ; echo $?
1
```

Perform tests by using the **Bash test command syntax**, [<TESTEXPRESSION>] or the newer extended test command syntax, [[<TESTEXPRESSION>]], which provides features such as **file name globbing** and **regex pattern matching**. In most cases you should use the [[<TESTEXPRESSION>]] syntax.

The following examples demonstrate the Bash test command syntax and numeric comparison operators:

```
[user@host ~]$ [[ 1 -eq 1 ]]; echo $?
0
[user@host ~]$ [[ 1 -ne 1 ]]; echo $?
1
[user@host ~]$ [[ 8 -gt 2 ]]; echo $?
0
[user@host ~]$ [[ 2 -ge 2 ]]; echo $?
0
[user@host ~]$ [[ 2 -lt 2 ]]; echo $?
1
[user@host ~]$ [[ 1 -lt 2 ]]; echo $?
0
```

The following examples demonstrate the Bash string comparison operators:

```
[user@host ~]$ [[ abc = abc ]]; echo $?
0
[user@host ~]$ [[ abc == def ]]; echo $?
1
[user@host ~]$ [[ abc != def ]]; echo $?
0
```

The following examples demonstrate the use of Bash string unary (one argument) operators:

```
[user@host ~]$ STRING=''; [[ -z "$STRING" ]]; echo $?
0
[user@host ~]$ STRING='abc'; [[ -n "$STRING" ]]; echo $?
0
```

Conditional Structures

Simple shell scripts represent **a collection of commands** that **are executed** from **beginning to end**. Programmers incorporate **decision-making** into shell scripts using **conditional structures**. A script can execute specific routines when stated conditions are met.

Use the If/Then Construct

The simplest of the conditional structures is the if/then construct, with the following **syntax:**

```
if <CONDITION>; then
      <STATEMENT>
      ...
      <STATEMENT>
fi
```

Use the If/Then/Else Construct

You can further **expand the if/then construct** so that it takes different **sets of actions** depending on whether a condition is met. **Use the if/then/else construct to accomplish** this behavior, as in this example:

```
if <CONDITION>; then
      <STATEMENT>
      ...
      <STATEMENT>
else
      <STATEMENT>
      ...
      <STATEMENT>
fi
```

The following code section demonstrates an **if/then/else statement** to start the psacct service if it is not active, and to stop it if it is active:

```
[user@host ~]$ systemctl is-active psacct > /dev/null 2>&1
[user@host ~]$ if  [[ $? -ne 0 ]]; then \
sudo systemctl start psacct; \
else \
sudo systemctl stop psacct; \
fi
```

Use the If/Then/Elif/Then/Else Construct

Expand an if/then/else construct to test more than one condition and execute a different set of actions when it meets a specific condition. The next example shows the construct for an added condition:

```
if <CONDITION>; then
      <STATEMENT>
      ...
      <STATEMENT>
elif <CONDITION>; then
      <STATEMENT>
      ...
      <STATEMENT>
else
      <STATEMENT>
      ...
      <STATEMENT>
fi
```

Match Text in Command Output with Regular Expressions

Write Regular Expressions

Regular expressions provide a pattern matching mechanism that facilitates finding specific content. The vim, grep, and less commands can use regular expressions. Programming languages such as Perl, Python, and C also support regular expressions, but might have minor differences in syntax.

Option	Description
.	The period (.) matches any single character.
?	The preceding item is optional and is matched at most once.
*	The preceding item is matched zero or more times.
+	The preceding item is matched one or more times.
{n}	The preceding item is matched exactly n times.
{n,}	The preceding item is matched n or more times.
{,m}	The preceding item is matched at most m times.
{n,m}	The preceding item is matched at least n times, but not more than m times.
[:alnum:]	Alphanumeric characters: `[:alpha:]` and `[:digit:]`; in the 'C' locale and ASCII character encoding, this expression is the same as `[0-9A-Za-z]`.

Option	Description
[:alpha:]	Alphabetic characters: `[:lower:]` and `[:upper:]`; in the 'C' locale and ASCII character encoding, this expression is the same as `[A-Za-z]`.
[:blank:]	Blank characters: space and tab.
[:cntrl:]	Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (DEL).
[:digit:]	Digits: 0 1 2 3 4 5 6 7 8 9.
[:graph:]	Graphical characters: `[:alnum:]` and `[:punct:]`.
[:lower:]	Lowercase letters; in the 'C' locale and ASCII character encoding: a b c d e f g h i j k l m n o p q r s t u v w x y z.
[:print:]	Printable characters: `[:alnum:]`, `[:punct:]`, and space.
[:punct:]	Punctuation characters; in the 'C' locale and ASCII character encoding: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~.
[:space:]	Space characters: in the 'C' locale, this is tab, newline, vertical tab, form feed, carriage return, and space.
[:upper:]	Uppercase letters: in the 'C' locale and ASCII character encoding: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z.
[:xdigit:]	Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f.
\b	Match the empty string at the edge of a word.
\B	Match the empty string provided that it is not at the edge of a word.
\<	Match the empty string at the beginning of a word.
\>	Match the empty string at the end of a word.
\w	Match word constituent. Synonym for `[_[:alnum:]]`.
\W	Match non-word constituent. Synonym for `[^_[:alnum:]]`.
\s	Match white space. Synonym for '[[:space:]`].
\S	Match non-white space. Synonym for `[^[:space:]]`.

NOTES: for details tutorials on script please check Scripting Tutorials