Shell Scripting Notes

Photo by Chris Ried on Unsplash

What is Shell Script?

- In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.
  Normally the shell scripts has the file extension .sh

- To create a shell script, you use a text editor. There are many, many text editors available for your Linux system, both for the command line environment and the GUI environment. Here is a list of some common ones:
  1) vi or vim is the command line interface text editor.
  2) gedit is the GUI text editor.

Following steps are required to write shell script:

Step 1) Use any editor like vi or gedit to write shell script.
Syntax: vi << Script file name >>
Example:
`#vi hello.sh` // Start vi editor

Step 2) After writing shell script set execute permission for your script as follows
Syntax: chmod << permission >> <<script file name >>

Examples:

```
# chmod +x <script_file_name>
# chmod 755 <script_file_name>
```

Note: This will set read, write and execute (7) permission for owner, for group and other
permissions are read and execute only (5).
Step 3) Execute your script as
Syntax: bash script file name

```
sh <script_file_name>
./<script_file_name>
```

Example for writing script:
vi shub.sh // Start vi editor

```
#!/bin/sh
echo “Hello welcome to The DevOps Learning!”
```

Give the permissions as follows

```
chmod +x shub.sh
```

Run the hello.sh script file as follows.

```
./shub.sh
```

Output:

```
Hello welcome to The DevOps Learning!
```

. . .

First Shell script

File Name: `shubdevops.sh`

```
#!/bin/bash
#Purpose : To display the message.
#Author: Shubham Londhe.
#Date: 31st December 2022.
echo “Hello Welcome to the DevOps Learning”
```

- The first line is called a **shebang or a “bang”** line. It is nothing but the absolute path to the Bash interpreter. It consists of a number sign and an exclamation point character (#!), followed by the full path to

the interpreter such as `/bin/bash.` If you do not specify an interpreter line, the default is usually the `/bin/sh.`

- In order to execute the shell script make sure that you have execute permission. Give execute permissions as follows.

`#chmod +x shub.sh (OR) chmod 111 shub.sh`

There are different ways to run a shell script:

`1) bash shub.sh`

`2) sh shub.sh`

`3) ./shub.sh`

`4) . shub.sh`

. . .

Case Sensitivity

- As you know Linux is case sensitive, the file names, variables, and arrays used in shell scripts are also case sensitive.

Check out this Example:

```
#!/bin/bash
string1=10
String1=20
echo “The value of string1 is $string1 “
echo “The value of String1 is $String1 “
```

File Naming conventions:

1. A file name can be a maximum of 255 characters
2. The name may contain alphabets, digits, dots and underscores
3. System commands or Linux reserve words can not be used for file names.
4. File system is case sensitive.
5. Some of valid filenames in Linux are

shubham.txt
Shubham.sh
Shubham_08112013.sh
Shubham05.sh

Comments

- Comments are used to escape from the code.
- This part of the code will be ignored by the program interpreter.
- Adding comments make things easy for the programmer, while editing the code in future.
- Single line comments can be do using #
- Multilane comments can be using HERE DOCUMENT feature as follows

```
<<COMMENT1
your comment 1
comment 2
blah
COMMENT1
```

File name : `comments.sh`

```
#! /bin/bash
# This is a basic shell script by Shubham.
echo “Hello, welcome to The DevOps Learning…” # It will display the message
```

File name : `multiline_comments.sh`

```
#!/bin/bash:wq
echo “We are commenting multiple line”
<<COMMENT1
I am commenting here multiple lines
using HERE DOCUMENT
feature
COMMENT1
echo “Multilane comment done”
```

Variables

There are two types of variables in Linux shell script.
Those are

1. System variables
2. User defined variables (UDV).

Naming conventions for variables:

- Variable name must begin with alphanumeric character or underscore character (_).
- By convention, environment variables (**PAGER, EDITOR, ..**) and internal shell variables (**SHELL, BASH_VERSION, ..**) are capitalized. All other variable names should be lower case.
- Remember that variable names are case-sensitive; this convention avoids accidentally overriding environmental and internal variables.

System defined Variables:

Created and maintained by Linux bash shell itself. This type of variable is defined in **CAPITAL LETTERS.**
There are many shell inbuilt variables which are used for administration and writing shell scripts.
To see all system variables, type the following command at a console / terminal:

```
env or printenv
```

Use echo command to display variable value as follows.
File name : `system_variables.sh`

```
#! /bin/bash
echo ‘BASH=’$BASH
echo ‘BASH_VERSION=’$BASH_VERSION
echo ‘HOSTNAME=’ $HOSTNAME
echo ‘TERM=’$TERM
echo ‘SHELL=’$SHELL
echo ‘HISTSIZE=’$HISTSIZE
echo ‘SSH_CLIENT=’$SSH_CLIENT
echo ‘QTDIR=’$QTDIR
echo ‘QTINC=’$QTINC
echo ‘SSH_TTY=’$SSH_TTY
echo ‘RE_HOME=’$JRE_HOME
echo ‘USER=’$USER
echo ‘LS_COLORS=’$LS_COLORS
echo ‘TMOUT=’$TMOUT
echo ‘MAIL=’$MAIL
echo ‘ATH=’$PATH
echo ‘PWD=’$PWD
echo ‘JAVA_HOME=’$JAVA_HOME
echo ‘LANG=’$LANG
echo ‘SSH_ASKPASS=’$SSH_ASKPASSecho ‘HISTCONTROL=’$HISTCONTROL
```

```
echo ‘SHLVL=’$SHLVL
echo ‘HOME=’$HOME
echo ‘LOGNAME=’$LOGNAME
echo ‘TLIB=’$QTLIB
echo ‘CVS_RSH=’$CVS_RSH
echo ‘SSH_CONNECTION=’$SSH_CONNECTION
echo ‘LESSOPEN=’$LESSOPEN
echo ‘_BROKEN_FILENAMES=’$G_BROKEN_FILENAMES
echo ‘OLDPWD=’$OLDPWD
```

User defined Variables

Created and maintained by user. This type of variable defined may use any valid variable name, but it is good practice to avoid all uppercase names as many are used by the shell.

File name : `user_defined_variables.sh`

```
#! /bin/bash
trainingCourse=DevOps
echo ‘Displaying the user defined the varibale (trainingCourse)
value is: ‘$trainingCourse
```

Command Line arguments

During shell script execution, values passing through command prompt is called as command line arguments.
For example, while running a shell script, we can specify the command line arguments as `“sh scriptfile.sh arg1 arg2 arg3”`

While using command line arguments follow the below important points.

- We can specify n number of arguments, there is no limitation.
- Each argument is separated by space.

Following example describes the command line arguments.
File name: `commandlineargs.sh`

```
#!/bin/sh
#Number of arguments on the command line.
echo ‘$#:’ $#
#Process number of the current process.
echo ‘$$:’ $$
#Display the 3rd argument on the command line, from left to right.
echo ‘$3:’ $3
```

```
#Display the name of the current shell or program.
echo ‘$0:’ $0
#Display all the arguments on the command line using * symbol.
echo ‘$*:’ $*
#Display all the arguments on the command line using @ symbol.
echo ‘$*:’ $@
```

Run: `# sh commandlineargs.sh Github Bluemix UCD Jenkins Java Linux WebSphereApplicationServer`

Output:

```
$#: 4

$$: 16955
$3: WebSphereApplicationServer
$0: commandlineargs.sh
$*: Java Linux WebSphereApplicationServer Android
$@: Java Linux WebSphereApplicationServer Android
```

Difference between `$*` and `$@`

The collection of arguments in `$*` is treated as one text string, whereas the collection of arguments in `$@` is treated as separate strings.

Strings

Strings are scalar and there is no limit to the size of a string.
Any characters, symbols, or words can be used to make up your string.

How to define a string?

We use single or double quotations to define a string.

“Double Quotes”—Anything enclose in double quotes removed meaning of that characters (except \ and $).

‘Single quotes’—Enclosed in single quotes remains unchanged.
`Back quote`—To execute command

Example:

FileName: `quotes.sh`

```
#!/bin/bash
single=’Single quoted’
double=”Double quoted”
```

```
echo $single
echo $double
Save the above example script as quotes.sh
chmod 755 quotes.sh
./quotes.sh
```

String Formatting

Character Description
-n Do not output the trailing new line.
-e Enable interpretation of the following backslash escaped characters in the strings:
\a alert (bell)
\b backspace
\c suppress trailing new line
\n new line
\r carriage return
\t horizontal tab
\\ backslash

. . .

User Interaction using read command

In some cases the script needs to interact with the user and accept inputs.
In shell scripts we use the **read** statement to take input from the user.

read : read command is used to get the input from the user (Making scripts interactive).

Example:

File Name: `readName.sh`

```
#!/bin/bash
#Author: Shubham Londhe.
#Date: 31st Decemer 2022.
echo “Please enter your name:”
read userName

echo The name you entered is $userName
Run the above script as follows:
#./readName.sh
```

Output:

```
Please enter your name:  Shubham Londhe
The name you entered is  Shubham Londhe
```

. . .

Control commands

if control statement:

Syntax:

`if` condition

`then`

Display commands list if condition is true.

`else`

Display commands list if condition is false.

`fi`

Note: if and then must be separated, either with a << new line >> or a semicolon (;). Termination of the if statement is fi.

Example:

Following example demonstrates the find biggest number.

File Name: `FindBiggestNumber.sh`

```
if [ $# -ne 3 ]
then
echo “$0: number1 number2 number3 are not given” >&2
exit 1
fi
if [ $n1 -gt $n2 ] && [ $n1 -gt $n3 ]
then
echo “$n1 is the biggest t number”
elif [ $n2 -gt $n1 ] && [ $n2 -gt $n3 ]
then
echo “$n2 is the biggest t number”
elif [ $n3 -gt $n1 ] && [ $n3 -gt $n2 ]
then
echo “$n3 is the biggest number”
elif [ $1 -eq $2 ] && [ $1 -eq $3 ] && [ $2 -eq $3 ]
then
echo “All the three numbers are equal”
else
```

```
echo “I can not figure out which number is bigger”
fi
```

Run:

```
sh findbiggestNumber.sh 1 2 3
```

Output:

```
3 is the biggest number
```

Example—2

File Name: `FileExists.sh`

```
#! /bin/bash
echo -e “Enter the name of the file: \c”
read file_name
if [ -f $file_name ]
then
if [ -w $file_name ]
then
echo “Type something, To Quit type Ctrl +d”
cat >> $file_name

19
else
echo “The file do not have write permissions”
fi
else
echo “$file_name not exists”
fi
```

for loop

Syntax:

`for` (condition)
`do`
execute here all command/script until the condition is
not satisfied.(And repeat all statement between do and done)
`done`

Example:

FIleName: `for_loop.sh`

```
echo “Can you see the following:”
```

```
for (( i=1; i<=5; i++ ))
do
echo $i
echo “”
done
```

Output:

```
Can you see the following:
1
2
3
4
5
```

while loop

Syntax:

```
while [ condition ]
do
```

command1
command2
command3
..
....

```
done
```

Example:

```
i=5
while test $i != 0
do
echo “$i”
echo “ “
i=`expr $i–1`
```

Output:

```
5
4
3
2
1
```

switch case

The case statement is good alternative to multilevel `if-then-else-fi` statement. It enables you to match several values against one variable. It's easier to read and write.

Syntax:

```
case $variable-name in
pattern1) command
…
..
command;;
pattern2) command
…
..
command;;

patternN) command
…
..
command;;
*) command
…
..
command;;
esac
```

Example:

Filename: `switch_case.sh`

```
#!/bin/sh
echo “Enter a number between 1 and 10. “
read NUM
case $NUM in
1) echo “You entered is one” ;;
2) echo “You entered is two” ;;
3) echo “You entered is three” ;;
4) echo “You entered is four” ;;
5) echo “You entered is five” ;;
6) echo “You entered is six” ;;
7) echo “You entered is seven” ;;
8) echo “You entered is eight” ;;
9) echo “You entered is nine” ;;
10) echo “You entered is ten” ;;
```

```
*) echo “INVALID NUMBER!” ;;
esac
```

. . .

Advanced Text Processing Commands

- grep
- sed
- awk

grep & egrep

- **grep:** Unix utility that searches a pattern through either information piped to it or files.
- **egrep:** extended grep, same as grep –E
- **zgrep:** compressed files.

Usage: `grep <options> <search pattern> <files>`

- Options:

`-i` → ignore case during search

`-r,-R` → search recursively

`-v` → invert match i.e. match everything except pattern

`-l` → list files that match pattern

`-L` → list files that do not match pattern

`-n` → prefix each line of output with the line number within its input file.

`-A num` → print `num` lines of trailing context after matching lines.

`-B num` → print `num` lines of leading context before matching lines.

grep Examples

Search files containing the word bash in current directory

```
grep bash *
```

Search files NOT containing the word bash in current directory

```
grep -v bash *
```

Repeat above search using a case insensitive pattern match and print line number that matches the search pattern

```
grep -in bash *
```

Search files not matching certain name pattern

```
ls | grep -vi fun
```

grep OR

```
grep 'Man\|Sales' employee.txt
```

```
-> 100 Shub Manager Sales $5,000
```

```
300 Ankur Sysadmin Technology $7,000
```

```
500 Naveen Manager Sales $6,000
```

grep AND

```
grep -i 'sys.*Tech' employee.txt
```

```
-> 100300 Ankur Sysadmin Technology $7,000
```

sed

- "stream editor" to parse and transform information – information piped to it or from files
- line-oriented, operate one line at a time and allow regular expression matching and substitution.
- `s` substitution command

sed commands and flags

Flags	Operation	Command	Operation
`-e`	combine multiple commands	`s`	substitution
`-f`	read commands from file	`g`	global replacement
`-h`	print help info	`p`	print
`-n`	disable print	`i`	ignore case
`-V`	print version info	`d`	delete
`-r`	use extended regex	`G`	add newline
		`w`	write to file
		`x`	exchange pattern with hold buffer
		`h`	copy pattern to hold buffer
		`;`	separate commands

sed Examples

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

```
# My First Script
```

```
echo “Hello DevOps!”
```

sed Examples (2)

Delete blank lines from a file

```
sed ’/^$/d’ shub.sh
```

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

```
# My First Script
```

```
echo “Hello DevOps!”
```

Delete line `n` through `m` in a file

```
sed ’2,4d’ shub.sh
```

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

```
echo “Hello DevOPs!”
```

awk

The awk text-processing language is useful for tasks such as:

- Tallying information from text files and creating reports from the results.
- Adding additional functions to text editors like “vi”.
- Translating files from one format to another.
- Creating small databases.
- Performing mathematical operations on files of numeric data.

awk has two faces:

- It is a utility for performing simple text-processing tasks, and
- It is a programming language for performing complex textprocessing tasks.

How Does awk Work

- awk reads the file being processed line by line.
- The entire content of each line is split into columns with space or tab as the delimiter.
- `$0` Print the entire line
- `$1, $2, $3, …` for each column (if exists)
- `NR` number of records (lines)
- `NF` number of fields or columns in the current line.
- By default the field delimiter is space or tab. To change the field delimiter use the `-F<delimiter>` command.

awk Syntax

```
awk pattern {action}
```

pattern decides when **action** is performed

Actions:

- **Most common action:** print
- Print file `dosum.sh: awk '{print $0}' dosum.sh`

- Print line matching files in all .sh files in current directory: `awk '/bash/{print $0}' *.sh`

Awk Examples

Print list of files that are bash script files

```
awk '/^#\!\/bin\/bash/{print $0, FILENAME}' *
```

```
! #!/bin/bash Fun1.sh
```

```
#!/bin/bash fun_pam.sh
```

```
#!/bin/bash hello.sh
```

```
#!/bin/bash parm.sh
```

. . .

For Practice Shell scripts you can refer this article

https://medium.com/@sankad_19852/shell-scripting-exercises-5eb7220c2252