# UNIT 13 (EXTRA). BASH SCRIPTING

ADAPTED FROM:

**Computer systems**
**CFGS DAW**

# Linux Shell Scripting Tutorial - A Beginner's handbook

Written by Vivek Gite <vivek@nixcraft.com> and Edited By Various Contributors

Alfredo Oltra
Sergio García

## Licencia

This book is a remix of “Linux Shell Scripting Tutorial - A Beginner's handbook”, obtained in http://bash.cyberciti.biz/guide/Main_Page and http://nixcraft.com/.

# INDEX

# UD013. BASH SCRIPTING (BASED ON VIVEK GITE WORK)

## 1. INTRODUCTION

Wikipedia definition of a shell script is:

"A **shell script** is a computer program designed to be run by the Unix shell, a command-line interpreter.[1] The various dialects of shell scripts are considered to be scripting languages.

Typical operations performed by shell scripts include file manipulation, program execution, and printing text"

Shell script can be written in several languages. One of the most common is bash shell script. Bash is included in the most important Linux distributions.

In this unit, we are going to talk about bash shell scripts, but you have to know that a lot of sys admins prefer to write shell scripts using Perl, Python,….

There are a lot of shell script manuals. For this reason, we have choose a creative commons manual "Linux Shell Scripting Tutorial - A Beginner's handbook", obtained in

http://bash.cyberciti.biz/guide/Main_Page and http://nixcraft.com/ and we have summarized it to provide a proper way to code shell scripts.

Also there are other tutorials that can support you creating shell scripts:

https://www.shellscript.sh/

http://linuxcommand.org/lc3_writing_shell_scripts.php

## 2. HELLO WORLD IN SHELLSCRIPT

Create a text file like the next.

```
#!/bin/bash
echo "Hello, World!"
echo "Knowledge is power."
```

Save and close the file. You can run the script in the Linux console as follows:

```
./hello.sh
```

Sample outputs:

```
bash: ./hello.sh: Permission denied
```

It will not run script since you've not set execute permission for your script hello.sh.

To execute this program, type the following command:

```
chmod +x hello.sh
./hello.sh
```

Sample Outputs:

```
Hello, World!
Knowledge is power.
```

## 3. SHELL COMMENTS

Take look at the following shell script:

```
#!/bin/bash
# A Simple Shell Script To Get Linux Network Information
# Vivek Gite - 30/Aug/2009
echo "Current date : $(date) @ $(hostname)"
echo "Network configuration"
/sbin/ifconfig
```

The first line is called a shebang or a "bang" line. The following are the next two lines of the program:

# A Simple Shell Script To Get Linux Network Information

# Vivek Gite - 30/Aug/2009

- A word or line beginning with # causes that word and all remaining characters on that line to be ignored.
- These lines aren't statements for the bash to execute. In fact, the bash totally ignores them.
- These notes are called comments.
- It is nothing but explanatory text about script.
- It makes source code easier to understand.
- It helps other sys admins to understand your code, logic and it helps them to modify the script you wrote.

# 4. VARIABLES IN A SHELL

You can use variables to store data and configuration options. There are two types of variable as follows:

## 4.1 System Variables

Created and maintained by Linux bash shell itself. This type of variable (with the exception of auto_resume and histchars) is defined in CAPITAL LETTERS. You can configure aspects of the shell by modifying system variables such as PS1, PATH, LANG,HISTSIZE,and DISPLAY etc.

To see all system variables, type the following command at a console

```
set
```

OR

```
env
```

Some of the most important system variables are:

- BASH_VERSION: Holds the version of this instance of bash.
- HOSTNAME: the name of your computer.
- CDPATH: The search path for the cd command.
- HISTFILE: The name of the file in which command history is saved.
- HOME: The home directory of the current user.
- PATH: The search path for commands. It is a colon-separated list of directories in which the shell looks for echo $PATH commands.

## 4.2 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.

## 4.3 Assign values to Variables

Creating and setting variables within a script is fairly simple. Use the following syntax:

```
varName=someValue
```

someValue is assigned to given varName and someValue must be on right side of = (equal) sign. If someValue is not given, the variable is assigned the null string.

## 4.4 How do I display the value of a Variable?

Use echo command to display variable value. To display the program search path, type:

```
echo "$PATH"
```

All variable names must be prefixed with $ symbol, and the entire construct should be enclosed in quotes. Try the following example to display the value of a variable without using $ prefix:

```
echo "HOME"
```

To display the value of a variable with echo $HOME:

```
echo "$HOME"
```

You must use $ followed by variable name to print a variable's contents.

The variable name may also be enclosed in braces:

```
echo "${HOME}"
```

This is useful when the variable name is followed by a character that could be part of a variable name:

```
echo "${HOME}work"
```

## 4.5 Common Examples

Define your home directory:

```
myhome="/home/v/vivek"
echo "$myhome"
```

Set file path:

```
input="/home/sales/data.txt"
echo "Input file $input"
```

Store current date (you can store the output of date by running the shell command):

```
NOW=$(date)
echo $NOW
Set NAS device backup path:
BACKUP="/nas05"
echo "Backing up files to $BACKUP/$USERNAME"
```

## 4.6 Generating output with echo command

Use echo command to display a line of text or a variable value. It offers no formatting option. It is a good command to display a simple output when you know that the variable's contens will not cause problems.

echo Command Examples

```
#!/bin/bash
# Display welcome message, computer name and date
echo "*** Backup Shell Script ***"
echo
echo "*** Run time: $(date) @ $(hostname)"
echo
# Define variables
BACKUP="/nas05"
NOW=$(date +"%d-%m-%Y")
# Let us start backup
echo "*** Dumping MySQL Database to $BACKUP/$NOW..."
# Just sleep for 3 secs
sleep 3
# And we are done...
echo
echo "*** Backup wrote to $BACKUP/$NOW/latest.tar.gz"
```

### 4.7 Getting user input via keyboard

You can accept input from the keyboard and assign an input value to a user defined shell variable using read command.

```
read -p "Prompt" variable1 variable2 variableN
```

Where,

- -p "Prompt" : Display prompt to user without a newline.
- variable1 : The first input (word) is assigned to the variable1.
- variable2 : The second input (word) is assigned to the variable2.

To test it you can create a test script called as follows:

```
#!/bin/bash
read -p "Enter your name : " name
echo "Hi, $name. Let us be friends!"
```

Sample Outputs:

```
Enter your name : Vivek Gite
Hi, Vivek Gite. Let us be friends!
```

Other example:

```
#!/bin/bash
# read three numbers and assigned them to 3 vars
read -p "Enter number one : " n1
read -p "Enter number two : " n2
read -p "Enter number three : " n3
# display back 3 numbers - punched by user.
echo "Number1 - $n1"
echo "Number2 - $n2"
echo "Number3 - $n3"
```

### 4.8 Perform arithmetic operations

Arithmetic expansion and evaluation is done by placing an integer expression using the following format:

```
$((expression))
$(( n1+n2 ))
$(( n1/n2 ))
$(( n1-n2 ))
```

Example

```
#!/bin/bash
read -p "Enter two numbers : " x y
ans=$(( x + y ))
echo "$x + $y = $ans"
```

# 5. CONDITIONALS

## 5.1 Test command

The test command is used to check file types and compare values. Test is used in conditional execution. It is used for:

- File attributes comparisons
- Perform string comparisons.
- Arithmetic comparisons.

Test command syntax:

```
test condition
```

OR

```
test condition && command executed when true
```

OR

```
test condition || command executed when false
```

OR

```
test condition && true-command || false-command
```

Type the following command at a shell prompt (is 5 greater than 2? ):

```
test 5 > 2 && echo "Yes"
test 1 > 2 && echo "Yes"
```

Sample Output:

```
Yes
Yes
```

The result is wrong. Rather than test whether a number is greater than 2, you have used redirection to create an empty file called 2. To test for greater than, use the -gt operator (see numeric operator syntax):

```
test 5 -gt 2 && echo "Yes"
test 1 -gt 2 && echo "Yes"
```

Sample output:

```
Yes
```

You need to use the test command while make decision. Try the following examples and note down its output:

```
test 5 = 5 && echo Yes || echo No
test 5 = 15 && echo Yes || echo No
test 5 != 10 && echo Yes || echo No
test -f /etc/resolv.conf && echo "File /etc/resolv.conf found." || echo
"File /etc/resolv.conf not found."
test -f /etc/resolv1.conf && echo "File /etc/resolv1.conf found." ||
echo "File /etc/resolv1.conf not found."
```

## 5.2 IF structures

Now, you can use the if statement to test a condition. if command The general syntax is as follows:

```
if condition
then
           command1
           command2
           ...
           commandN
fi
```

OR

```
if test var == value
then
           command1
           command2
           ...
           commandN
fi
```

OR

```
if test -f /file/exists
then
           command1
           command2
           ...
           commandN
fi
```

OR

```
if [ condition ]
then
           command1
           command2
           ....
           ..
fi
```

If given condition is true than the command1, command2..commandN are executed. Otherwise script continues directly to the next statement following the if structure.

```
#!/bin/bash
read -p "Enter a password" pass
if test "$pass" == "jerry"
then
           echo "Password verified."
fi
```

### 5.3 If. .else. .fi

if..else..fi allows to make choice based on the success or failure of a command. For example, find out if file exists(true condition) or not (false condition) and take action based on a condition result.

```
if command
then
           command executed successfully
           execute all commands up to else statement
           or to fi if there is no else statement
else
           command failed so
           execute all commands up to fi
fi
```

Example:

```
#!/bin/bash
read -p "Enter a password" pass
if test "$pass" = "jerry"
then
           echo "Password verified."
else
           echo "Access denied."
fi
```

Other example

```
#!/bin/bash
read -p "Enter number : " n
if test $n -ge 0
then
           echo "$n is positive number."
else
           echo "$n number is negative number."
fi
```

### 5.4 Multilevel if-then-else

if..elif..else..fi allows the script to have various possibilities and conditions. This is handy, when you want to compare one variable to a different values.

```
if condition
then
           condition is true
           execute all commands up to elif statement
elif condition1
then
           condition1 is true
           execute all commands up to elif statement
elif condition2
then
           condition2 is true
           execute all commands up to elif statement
elif conditionN
then
           conditionN is true
           execute all commands up to else statement
else
           None of the above conditions are true
           execute all commands up to fi
fi
```

### 5.5 Nested ifs

You can put if command within if command and create the nested ifs as follows:

```
if condition
then
           if condition
           then
                      .....
                      ..
                      do this
           else
                      ....
                      ..
                      do this
           fi
else
           ...
           .....
           do this
fi
```

## 5.6 Conditional expression using [

The test command is used to check file types and compare values. You can also use [ as test command. It is used for:

- File attributes comparisons
- Perform string comparisons.
- Arithmetic comparisons.

Syntax

```
[ condition ]
```

OR

[ ! condition ]

OR

```
[ condition ] && true-command
```

OR

```
[ condition ] || false-command
```

OR

```
[ condition ] && true-command || false-command
```

Examples

```
[ 5 == 5 ] && echo "Yes" || echo "No"
[ 5 == 15 ] && echo "Yes" || echo "No"
[ 5 != 10 ] && echo "Yes" || echo "No"
[ -f /etc/resolv.conf ] && echo "File /etc/resolv.conf found." || echo
"File /etc/resolv.conf not found."
[ -f /etc/resolv1.conf ] && echo "File /etc/resolv.conf found." || echo
"File /etc/resolv.conf not found."
```

## 5.7 String comparison

String comparison can be done using test command itself.

The strings are equal

```
STRING1 = STRING2
```

Example

```
#!/bin/bash
read -s -p "Enter your password " pass
echo
if test "$pass" = "tom"
then
            echo "You are allowed to login!"
fi
```

The strings are not equal

Use the following syntax:

```
STRING1 != STRING2
```

Example

```
#!/bin/bash
read -s -p "Enter your password " pass
echo
if test "$pass" != "tom"
then
          echo "Wrong password!"
fi
```

The length of STRING is zero

Use the following syntax (this is useful to see if variable is empty or not):

```
-z STRING
```

Example

#!/bin/bash

```
read -s -p "Enter your password " pass
echo
if test -z $pass
then
          echo "No password was entered!!! Cannot verify an empty password!!!"
exit 1
fi
if test "$pass" != "tom"
then
          echo "Wrong password!"
fi
```

## 5.8 File attributes comparisons

Use the following file comparisons to test various file attributes. You can use the test command or conditional expression using [.

```
-f file
```

True if file exists.

Example

```
[ -f /etc/resolv.conf ] && echo "File found" || echo "Not found"
```

```
-d dir
```

True if file exists and is a directory.

Example

```
#!/bin/bash
DEST=/backup
SRC=/home
# Make sure backup dir exits
[ ! -d $DEST ] && mkdir -p $DEST
# If source directory does not exits, die...
[ ! -d $SRC ] && { echo "$SRC directory not found. Cannot make backup
to $DEST"; exit 1; }
# Okay, dump backup using tar
echo "Backup directory $DEST..."
echo "Source directory $SRC..."
/bin/tar zcf $SRC $DEST/backup.tar.gz 2>/dev/null
# Find out if backup failed or not
```

## 6. THE EXIT STATUS

### 6.1 The exit status of a command

Each Linux command returns a status when it terminates normally or abnormally. You can use command exit status

in the shell script to display an error message or take some sort of action. For example, if tar command is unsuccessful, it returns a code which tells the shell script to send an e-mail to sys admin.

Exit Status

- Every Linux command executed by the shell script or user, has an exit status.
- The exit status is an integer number.
- The Linux man pages stats the exit statuses of each command.
- 0 exit status means the command was successful without any errors.
- A non-zero (1-255 values) exit status means command was failure.
- You can use special shell variable called ? to get the exit status of the previously executed command. To print ? variable use the echo command:

```
echo $?
date # run date command
echo $? # print exit status
foobar123 # not a valid command
echo $? # print exit status
```

### 6.2 How Do I See Exit Status Of The Command?

Type the following command:

```
date
```

To view exist status of date command, enter:

```
echo $?
```

Sample Output:

```
0
```

### 6.3 Exit command

The syntax is as follows:

```
exit N
```

- The exit statement is used to exit from the shell script with a status of N.
- Use the exit statement to indicate successful or unsuccessful shell script termination.
- The value of N can be used by other commands or shell scripts to take their own action.
- If N is omitted, the exit status is that of the last command executed.
- Use the exit statement to terminate shell script upon an error.
- If N is set to 0 means normal shell exit.

```
#!/bin/bash
echo "This is a test."
# Terminate our shell script with success message
exit 0
```

## 7. SHELL COMMAND LINE PARAMETERS

### 7.1 What is a command line argument?

A command line argument is nothing but an argument sent to a program being called. A program can take any number of command line arguments.

For example, type the following command:

```
ls grate_stories_of
```

Sample Outputs:

```
grate_stories_of: No such file or directory.
```

ls is the name of an actual command and shell executed this command when you type command at shell prompt. The first word on the command line is:

- ls - name of the command to be executed.
- Everything else on command line is taken as arguments to this command.

Consider the following example:

```
tail +10 /path/to/file.txt
```

- tail : command name.
- +10 /path/to/file.txt : The arguments.

### 7.2 How to use positional parameters

All command line parameters ( positional parameters ) are available via special shell variable $1, $2, $3,...,$9.

Create a simple shell script:

```
#!/bin/bash
echo "The script name : $0"
echo "The value of the first argument to the script : $1"
echo "The value of the second argument to the script : $2"
echo "The value of the third argument to the script : $3"
echo "The number of arguments passed to the script : $#"
echo "The value of all command-line arguments (\$* version) : $*"
echo "The value of all command-line arguments (\$@ version) : $@"
```

## 8. FOR LOOP

### 8.1 The for loop statement

Bash shell can repeat particular instruction again and again, until particular condition satisfies. A group of instruction that is executed repeatedly is called a loop. Bash supports:

- The for loop
- The while loop

Each and every loop must:

- First, the variable used in loop condition must be initialized, then execution of the loop begins.
- A test (condition) is made at the beginning of each iteration.
- The body of loop ends with a statement that modifies the value of the test (condition) variable.
- Repeatedly execute a block of statements.

## 8.2 The for loop syntax

The for loop syntax is as follows:

```
for var in item1 item2 ... itemN
do
            command1
            command2
            ....
            ...
            commandN
done
```

The for loop variable's contents syntax:

```
for var in $fileNames
do
            command1
            command2
            ....
            ...
            commandN
done
```

The for loop command substitution syntax:

```
for var in $(Linux-command-name)
do
            command1
            command2
            ....
            ...
            commandN
done
```

The for loop three-expression syntax ( this type of for loop share a common heritage with the C programming language ):

```
for (( EXP1; EXP2; EXP3 ))
do
            command1
            command2
            command3
done
```

The above syntax is characterized by a three-parameter loop control expression; consisting of an initializer (EXP1), a loop-test or condition (EXP2), and a counting expression (EXP3).

## 8.3 For examples

Example

```
#!/bin/bash
for i in 1 2 3 4 5
do
           echo "Welcome $i times."
done
```

Example

```
#!/bin/bash
# A simple shell script to print list of cars
for car in bmw ford toyota nissan
do
           echo "Value of car is: $car"
done
```

Example

```
#!/bin/bash
# A simple shell script to run commands
for command in date pwd df
do
           echo "*** The output of $command command >"
           #run command
           $command
done
```

Example

```
#!/bin/bash
# A shell script to verify user password database
files="/etc/passwd /etc/group /etc/shadow /etc/gshdow"
for f in $files
do
           [ -f $f ] && echo "$f file found" || echo "*** Error - $f file
missing."
done
```

### 8.4 Nested for loops

Nested for loops means loop within loop. They are useful for when you want to repeat something several times for several things.

For example:

```
#!/bin/bash
# A shell script to print each number five times.
for (( i = 1; i <= 5; i++ )) ### Outer for loop ###
do
           for (( j = 1 ; j <= 5; j++ )) ### Inner for loop ###
           do
                       echo -n "$i "
           done
echo "" #### print the new line ###
done
```

## 9. WHILE LOOP

### 9.1 The while loop syntax

The syntax is:

```
while [ condition ]
do
           command1
           command2
           ..
           ....
           commandN
done
```

Command1..commandN will executes while a condition is true.

While loop example:

```
#!/bin/bash
# set n to 1
n=1
# continue until $n equals 5
while [ $n -le 5 ]
do
           echo "Welcome $n times."
           n=$(( n+1 )) # increments $n
done
```

Infinite While loop example:

```
#!/bin/bash
while true
do
        echo "Do something; hit [CTRL+C] to stop!"
done
```

## 10. USEFUL PROGRAMS TO CREATE SCRIPTS

There are a several of programs that are widely used to create scripts.

All these programs are very powerful and require a detailed analysis that escapes the content of this documentation. Research on them if you think they can be useful in your scripts.

The list is very long, but by way of summary, we can include:

- **grep:** it is used to search for text from a file or another command's output. It can return the lines where it finds a match or the lines where it doesn't.

```
> cat file.txt
THIS IS THE FIRST LINE
Second line
I like Computer Systems

> grep "Computer" file
I like Computer Systems
```

- **cut**: it removes or "cut out" sections of each line of a file or files. Usually we indicate “delimitator” (-d) and which field (-f n) we want to obtain.

```
> cut -d" " -f 3 file.txt
THE

Computer
```

Note: THE is 3rd field of line 1, in line 2 there isn’t second field and in line 3 Computer is the 3rd field.

- **ps:** report a snapshot of the current processes in the system.

```
ps aux
```

- **tr**: tr utility copies the standard input to the standard output with substitution or deletion of selected characters.

```
> tr "[:lower:]" "[:upper:]" < file.txt

> cat file
THIS IS THE FIRST LINE
SECOND LINE
I LIKE COMPUTER SYSTEMS
```

Example of tr to replace a character

```
> tr 'I' 'A' <file.txt

> cat file
THAS AS THE FARST LANE
Second line
A like Computer Systems
```

- **kill**: kill is used to send a signal to a process. Although it's name, it is not only for killing programs, there can be signals of other kinds. The most used signal is signal 9 (to kill a program).

```
Kill -9 300
```

Note 300 is PID (Process Identifier) of a running program.

- **awk:** AWK is an interpreted programming language which focuses on processing text. It was designed to execute complex pattern-matching operations on streams of textual data

```
awk 'length($0) < 12' file.txt
Second line
```

## 11. REGULAR EXPRESSIONS

Regular expressions are combinations of characters that can be expanded matching complex patterns. It is one of the most powerful utilities of Linux.

Regular expressions are very powerful and require a detailed analysis that escapes the content of this documentation. Research on them if you think they can be useful in your scripts.

Examples of regular expressions:

- ^ matches start of string

```
> grep ^S < file.txt
Second line
```

Lines starting in S.

- $ matches end of string

```
> grep E$ < file.txt
THIS IS THE FIRST LINE
```

Lines ending in E.

[ ] Indicates a range of characters. For example to indicate all those lines ending with a lowercase letter or a number.

```
> grep [a-z0-9]$ < file.txt
Second line
I like Computer Systems
```

Lines ending in lowercase letter or number.

## 12. ADDITIONAL MATERIAL

[1] Glossary.

[2] Exercises

## 13. BIBLIOGRAPHY

[1] “Linux Shell Scripting Tutorial - A Beginner's handbook”, obtained in http://bash.cyberciti.biz/guide/Main_Page

[2] https://www.shellscript.sh/

[3] http://linuxcommand.org/lc3_writing_shell_scripts.php