Bash Shell Scripting

Monday, July 24, 2023 10:04 AM

➢ Shell Scripting Concepts:

1. Variables
2. Filters like grep, , awk and sed Commands
3. Conditional Statements
4. Loops
5. Functions
6. Job Scheduling and Many more

➢ Steps to Write and Execute a Bash Shell Script:

Taking example to find docker & nginx version:

1. Get manual commands for the requirement — docker -v
nginx -v
2. Select command line editor (Vi)
3. Create a file with extension .sh to represent shellscript — demo.sh
write first line as shebang line (which shell we are using to run the script) — #!/bin/bash
Paste all the required commands in sequence — docker -v
nginx -v
4. Execute Shell Script

Provide execution permissions using chmod	chmod u+x demo.sh
run with ./filename.sh (use . if script in same location, else provide complete path)	./demo.sh
Note : Running without execution permissions	bash demo.sh

➢ Steps to Configure Gmail on Ubuntu Server:

step1:	Login Into Gmail
Step2:	Enable 2 step Verification (Manage Google Account --> Security --> Signing in to Google)
Step3:	Get App Password (yourGMAIL@gmail.com appPassword: generateIT)
Step4:	login into ubuntu and switch to root using: sudo su -
Step5:	Run below commands:
	apt-get update -y apt-get install sendmail mailutils -y

Step6:	Create authentication file
	cd /etc/mail mkdir -m 700 authinfo cd authinfo/ vi gmail add the below conntent AuthInfo: "U:root" "I:your-mail@gmail.com" "P:your-password" Now edit your mail id and password

Step7:	create hash map of the file:
	makemap hash gmail < gmail

Step8:	Got to /etc/mail and open sendmail.mc
	then Add the following lines to sendmail.mc file right above MAILER_DEFINITIONS #GMail settings: define(`SMART_HOST',`[smtp.gmail.com]')dnl define(`RELAY_MAILER_ARGS', `TCP $h 587')dnl define(`ESMTP_MAILER_ARGS', `TCP $h 587')dnl define(`confAUTH_OPTIONS', `A p')dnl TRUST_AUTH_MECH(`EXTERNAL DIGEST-MD5 CRAM-MD5 LOGIN PLAIN')dnl define(`confAUTH_MECHANISMS', `EXTERNAL GSSAPI DIGEST-MD5 CRAM-MD5 LOGIN PLAIN')dnl FEATURE(`authinfo',`hash -o /etc/mail/authinfo/gmail.db')dnl

Step9:	Now run below two command from /etc/mail
	make /etc/init.d/sendmail reload

Step10:	Verify the test mail using
	echo "Hello, I am from $(hostname)" | mail -s "Testing Gmail Setup on Ubuntu Server" abhishekraovelichala@gmail.com

➢ Creating a newuser in Ubuntu:

1. sudo su — (superuserdo switch user)
useradd [username] — (creating username)
passwd [username] — (creating password)
2. Directory is not created for new user automatically. We need to create manually in home directory
3. chwon -R username: username path(/home/username) — change owner and group for the new user
If not in default user use sudo to perform actions. — ~ indicates default user
4. Change to bash shell:

echo $SHELL	to see the present assigned shel
bash	to set bash shell (but it is not)
chsh	to change shell

➢ Redirection Operations:

1. Output Redirection Operator:

>	To create a new file and add content
>>	To append the content to exisiting content

2. Input Redirection Operator:

<	To Provide the input

3. Combining redirection Operator:

|	To send the output of one command to another command input

➢ StndIn, StndOut, StndErr:

StndIn	0
StdOut	1
StdErr	2
cmd > filename	(default) Stores the output in filename
cmd 1>filename	Stores output in filename
cmd 2>filename	stores err in filename
cmd 1>filename 2>filename	Stores both output & error in filename
cmd 1>filename 2>&1	means store error where 1 is storing
cmd &>filename	store both output & error in filename

➢ Commands to read a File Content:

1. Read a file content by opening it

vi filename	Using vi/vim/nano editors

2. Read a file content without Opening it

cat filename	outputs the file contents
less filename	display content partially by pressing enter to quit enter q
more filename	display content partially by pressing enter but cannot go previous content. To quit enter q

3. Read a file content with conditions

more -n filename	displays only first n lines and wait for next lines
more +n filename	starts from nth line
first filename	displays first 10 lines
first -n filename	displays only first n lines
tail filename	displays last 10 lines
tail -n filename	displays last n lines

4. Read, display output on specific range

awk 'NR>=m && NR<=n {print} filename	displays line from range m to n
sed -n 'm,np' filename	displays line from range m to n
head -n filename | tail -m+1	displays line from range m to n

➢ Grep Command:

grep [options] "pattern/string" filename	Used to filter the content

1.

cat filename | grep filter	displays the content having filter
grep filter filename	displays the content having filter
grep filter filename1 filename2	searches for filter in file 1 & 2
grep "string" filename	to search for a string in file
grep filter *	search in all files in current directory

2. Options in grep command:

-i	ignore case sensitive
-w	exact word ex: line is given, op: lines is not matched only line matched
-v	not having that filter
-o	to print exactly that word not whole line
-n	display matched line numbers
-c	display matched number of lines
-r	to look in current directory and sub directory

-A n	print n lines after match
-B n	print n lines before match
-C n	print n lines before and after match
-l	display only filenames
-h	to hide file names
-f	grep -f searchFilename filename compares the strings in searchFilename to filename we can search multiple patterns by placing strings one per line in searchFilename.
-e "pattern"	grep -e "pattern" -e "pattern" ... filename searches for multiple patterns in a file without saving them into a file
-E "str1|str2|.."	searches for the multiple strings in filename
Note:	Combination of strings is called pattern

3. Rules to create Patterns:

xy|pq	serches for string xy or pq
^xyz	lines starting with xyz
xyz$	lines ending with xyz
^$	match for empty lines
\	to remove purpose of special symbols
.	matches any one character Ex. grep -E "t . . s" filename
\b	having space at edge of word
?	preceding character is matched at 0 or once
*(star)	preceding character is mateched at 0 or more times
+	preceding character is matched at 1 or more times
[xyz]	within square bracket searches for either x or y or z
[a-d]	within square bracket searches in the range of a/b/c/d
[a-ds-z]	within square bracket searches in range a/b/c/d/s/t/u/v/w/x/y/z
^[abc]	within square bracket line starting with a/b/c
{N}	matches precedding character n times
{M,N}	matches preceeding character of m/n times
{M,}	matches precedding character min of M times no max limit
[[:alnum:]] [[:alpha:]] [[:blank:]] [[:digit:]] [[:lower:]] [[:space:]] [[:upper:]]	– Alphanumeric characters. – Alphabetic characters – Blank characters: space and tab. – Digits: '0 1 2 3 4 5 6 7 8 9'. – Lower-case letters: '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'. – Space characters: tab, newline, vertical tab, form feed, carriage return, and space. – Upper-case letters: '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'.

➢ Cut Command:

- Tool to extract parts of each line of a file
- It is based on
  - ○

Byte position	-b
Character Position	-c
Fields based on delimiter (by default delimiter is tab)	-f

1. **Based on character:**

-

cut -c n filename	extract chacracter c from the nth position from every line
cut -c m,n filename	characters at m,n position
cut -c m-n filename	characters in m,n range
cut -c -n filename	from beginning to nth position
cut -c m- filename	from position m to last

2. **Based on Fields(Columns):**

cut -f n filename	f indicates fields. extracts the content in field(column) number n
cut -d "seperator" -f n filename	d is delimiter with seperator extracting content in field no. n
cut -d "seperator" -f n filename --output-delimiter="space/ single symbol or character "	in output fields are seperated by space or character
cut -sf n filename	use sf to extract only having seperator or delimiter

➢ awk Command:

1. awk [options] '[selection criteria] {action}' filename
2. Default seperator is space
3.

Ex: awk -F '[/]''/string/ {print $n}' filename	search for string and seperators as space & /, then print the field number n
NR==n	work on only line number n Used instead of a string
$0	prints entire file
{print NR}	gives line number
{print NR, $0}	prints all content with line numbers
{print NR, $0, NF}	prints row & column numbers
{print NR, $NF}	prints last column of content

➢ tr Command:

1. Used to convert or delete given set of characters
2.

tr [options] '[set1]' '[set2]'<input file	find set1 & replace with set2
tr '[:upper:]' '[:lower:]' <filename	change uppercase to lowercase
tr 'A' 'a' <filename	change capital A to small a
tr -d "character"	delete character

➢ tee Command:

1. Used to display output and store that into a file
2. used to create log files
3.

command | tee file.txt	store op from command into file.txt
command | tee -a file.txt	append output into file.txt

➢ echo Command:

1.

echo string/message	prints the string/message to screen
echo "$(command)"	prints command output
echo -e "Line1\nLine2…"	print line1 and line 2 in different lines
echo -e"line1\tline2	tab space (horizontal tab)
echo -e"line1\vline2"	vertical tab
echo -e"line1\bline2"	backspace
echo -e"line1\rline2"	goes to starting position of line
echo -e"line1\line2'	escapes one character
echo -n "line1" echo "line2	Both lines are printed in same line

➢ Variables:

1. variables are used to store data in shell script and later we can use it
2.

s = "string"	to store the value of string in variable s
$s	use $s where we want to use variable value
echo $s	print s value

3. Variables are of two types:
   a. System Variables:
      i. Created and mainted by system itself
      ii. always in CAPITAL letters
      iii. can see them with set command
   b. User defined variables:
      i. created and maintained by user
      ii. generally in lowercase or camel case
      iii. no special characters expect (_ underscore)
      iv. length should be less than or equal to 20 characters
      v. case senstitive
      vi. don't provide space while defining variables on either side of = symbol
      vii. use quotes if string contains spaces
4.

variable=$(command)	stores command output in variable
variable=`command`	stores command output in variable

variable1=$variable2	assign on variable value to another variable

➢ MutliLine Block:

1. heredoc is used to write multi-line block

anyCommand <<DELIMITER Line1 Line2 Line3 DELIMITER	cat << EOF The user is $USER The home is $HOME EOF	we can use any string in place of delimiter. But it should start and end with same
anyCommand <<DELIMITER > filename Line1 Line2 Line3 DELIMITER	cat << EOF >filename The user is $USER The home is $HOME EOF	send output to a file
anyCommand <<DELIMITER | command Line1 Line2 Line3 DELIMITER	cat << EOF | grep user The user is $USER The home is $HOME EOF	using pipes

2. herestring is useful when working with one line string

command<<<string	tr [a-z] [A-Z]<<<"Hello World"	change the string case from lower to upper
command<<<$variable	tr [a-z] [A-Z]<<<$s	change the s value from lower to upper case
command<<<$(command)	tr [a-z] [A-Z]<<<$(docker -v)	change the chars in docker version from lower to upper

➢ Comments in Bash Shell Scripting:

1. Human readable explanation of script.
2. Comments two types:
   a. Single Line Comments

#comment	#starting script	use # at starting to make it comment

   b. Multi Line Comments

<< DELIMITER comment1 comment2 .. DELIMITER	<<EOF created by created on EOF	use delimiter at starting and ending to make it a multi line comment

➢ Debugging Bash Shell Scripts:

1. Two types of errors
   a. Syntax Error
      i. Stops execution
   b. Runtime Error
      i. did not stop execution
2. set command for debugging
3.

set [options]	write at the starting of code
set No options	list system defined variables
set -n	no execution, purely for syntax check
set -x	prints the op of command before executing it in script
set -e	exits script if any command fails

➢ Exit Status:

If op is 0(Zero) successfully executed
other than 0(zero)-(1-255)- not executed successfully

echo $?	if 0 previously executed command is sucessful else (1-255)not executed ssuccessful
var=$?	value in variable
127	command not found
1	command failed during execution
2	incorrect command usage

➢ Basic String Operations:

x="string"	assigning a value to string

x=$(cmd)	assigning op of a cmd to varaible x
echo $x	displaying value of x
x=${#string}	display length of string
xy=xy	concats of strings
xUpper=${x^^}	convert x to capital
xLower=${y,,}	convert y to lower case
newY=${y/find/replace}	find and replace
${var_name:start_position:length}	${x:2:2}

String operations on Path:

realpath filename	gives the path of the file but it doesnot validate
base path/filename	removes path and gives filename
base path/file.sh .sh	gives filename without extension
dirname path/filename	it removes the value after last / i.e., filename

➢ Inputs:

read var	input the value into variable from the command line during execution
read -p "Enter name" var_name	p indicates prompt gives instructions to input
read -p "Enter name" echo $REPLY	if no variable is given the default variable is REPLY

➢ Arithmetic Operators:

1. Default data type is a string
2. To perform integer(not float) operations include them b/w ((var))

	((sum=x+y))	declares x,y as integers and add the summation to variable sum
3.	bc<<<"scale=n;$x+$x"	to perform float arthemetic operations use bc(bash calculator) n refers to how many decimals to get and include $ before variables.

➢ Case Statement:

1.Syntax:

case $opt in opt1) statements ;; opt2) statements ;; *) statements ;; esac	compares the value of variable opt to the options opt1 or opt2 and run those statements. If value doesn't match with any options *) option is executed

➢ Test Command:

1. Used to validate conditions

	test command or [command]	syntax
2.	test 4 -eq 4 | echo $?	if op is 0 then condition is true
	[4 -eq 4] | echo $?	if op is 0 then condition is true

3. Conditional Operators:

Numbers:

```
[[ int1 -eq int2 ]]        – It return true if they are equal else false
[[ int1 -ne int2 ]]        – It return false if they are not equal else true
[[ int1 -lt int2 ]]        – It return true if int1 is less than int2 else false
[[ int1 –le int2 ]]        – It return true if int1 is less than or equal to int2 else false
[[ int1 -gt int2 ]]        – It return true if int1 is greater than int2 else false
[[ int1 –ge int2 ]]        – It return true if int1 is greater than  or equal to int2 else false
[[ ! int1 –eq int2 ]]      –  It reverse the result
```

Strings:

```
[[ -z str  ]]        – It return true if the length of the str is zero else false
[[ -n str  ]]        – It return true if the length of the str is no-zero else false
[[ str1 == str2 ]]   – It return true if both the strings are equal else false
[[ str1 != str2 ]]   – It return true if both the strings are equal else false
```

4. File Test Operators:

```
[[ -d file ]]          -- It return true if the file/path is directory else false
[[ -f file ]]   -- It return true if the file/path is a file else false
[[ -e file ]]   -- It return true if the file/path is exists else false
[[ -r file ]]   -- It return true if the file/path is readable else false
[[ -w file ]]    -- It return true if the file/path is writable else false
[[ -x file ]]   -- It return true if the file/path is executable else false
```

➢ Command Chaining Operators:

1. Used to combine commands
2. Four types:

;	semi colon operator	cmd1;cmd2;cmd3;…	execute multiple commands at a time.
&&	Logical AND Operator	cmd1&&cmd2	cmd2 runs only if cmd1 runs
||	Logical OR operator	cmd1||cmd2	run cmd2 if cmd1 fails
&& ||	Logical AND OR operator	cmd1 &&cmd2||cmd3	run cmd2 is cmd1 runs else run cmd3
&& or -a	use -a when [] used to compare	use && when [[]] used to compare	-
|| or -o	use -o when [] used to compare	use || when [[]]used to compare	-
!	Logical NOT operator		

➢ Executing a block of code:

{ cmd1; cmd2; cmd3 }	a block of code

➢ If and if else conditions:

if cmd1 then cmd2 cmd2 fi	simple if condition
1. if cmd1 then cmd2 else cmd3 fi	if-else condition

➢ Elif Condtion:

```
if cmd1
then
        cmd2
elif
        cmd3
elif
        cmd4
else
        cmd4
fi
```

➢ Arrays:

1. Array are two types:
   a. Index based arrays
   b. Associative arrays
2. **Index based arrays:**
3. | | |
   |---|---|
   | arr=(ls date whoami) | collection of commands or variables seperated by space |
4. Defining an array:

arr =()	Empty array
arr=("ls -lrt" "sudo su -")	use " "
declare -a arr arr =(1 2 3 ls pwd)	declaring a new array and assigning the values.

5. Zero-based array.

echo "${x[* or @]}"	to print all elements of an array
echo "${x[index]}"	print element at index

echo "${x[@]:index}"	print all elements starting from index
echo "${x[@]:index:n}"	print n elements starting from index
echo "${#x[@]}"	prints length of array
arr=($(cmd))	to get the cmd op into array
unset arr	to delete array
arr =(1 2 3) arr+=(4 5 6)	update array with exisiting values
read -a arryname	to give input from the command line
read -p "Enter array" -a arr	to give input instructions to array

6.

7. **Associative Arrrays:**
   a. has index values as strings
   b. declare before using them.
   c.

declare -A arr	to declare associative array use capital A
arr=([name]="bash" [version]="x.x.x.x.x"	here name and version are indexes

   d. called key-pair arrays

➢ For Loop:

1.

for each in x1 x2 x3 x4 do cmd1 cmd2 cmd3 done	for x in 1 2 3 4 5 do echo "$x" done
for each in $(ls) do .... done	

2. C type for loop:

```
for(init; condition;increment)
do
        cmd1
        cmd2
done
```

3. Infinity for loop:

```
for(;;)
do
        cmd1
        cmd2
done
```

4. break: used to terminate the execution by exiting the iteration
5. continue: used to skip the particular iteration and move to next iteration

➢ While Loop:

1. while command
2. do
   cmd1
   cmd2
3. done

➢ Functions:

a block of code that performs a specific task and can be reusable.

1. Syntax:

```
fun_name()
{
statements
...
..
}

(or)
function functionname
{
statements
..
..
```

```
}
```

2. Variables in functions:
   two types:
   a. global (default)
   b. local - declare as local var_name
3. return statement
   return $x
   while printing use $?
   works only for int not for strings
4. Passing arguments to functions:

```
var()
{
m=$1
n=$2
result=$((m+n))
echo $r
}
var $x $y
```