

Fundamentals of UNIX & Linux for System Administrators

FUL-02: Shell Scripting

Samuel K. Macharia

System Administrator, KENET

February, 2024.

Shell Scripting

Anything you can run normally on the command line can be put into a script. Anything you can put into a script can run normally on the command line.

What is a Shell?

- A UNIX Shell is a program or a command line interpreter that interprets the user commands which are either entered by the user directly or which can be read from a file, and then pass them to the operating system for processing.
- There are different types of shells available in Linux OS, these includes;
 1. Bourne Shell
 2. C Shell
 3. Korn Shell
 4. GNU Bourne Shell

What is a Shell?

- To know which shell types are supported by your OS;
==> `cat /etc/shells`
- To know where bash is located in your OS
==> `which bash`

Bash Script

- A script is a set of commands for an appropriate run time environment which is used to automate the execution of tasks.
- A Bash Shell Script is a plain text file containing a set of various commands that we usually type in the command line.
- It might include a set of commands, or a single command, or it might contain the hallmarks of imperative programming like loops, functions, conditional constructs

Advantages of Scripts

- Automate frequently performed operations
- Run multiple commands at once
- Easy to use
- Executed easily on any UNIX/LINUX based OS
- Saves time

Bash Script Syntax

- All bash scripts must have a **.sh** extension.
- All scripts may start with the first line known as Shebang.
- Shebang indicates the interpreter that is being used to execute a script (**#!/bin/bash**).
- Sample script for printing the word hello;

<snip>

#!/bin/bash

echo "hello"

</snip>

Bash Script Syntax

- Bash allows single-line and multi-line comments.
- For single-line comment, we use a hash (#) symbol at the start of the comment.
- For multi-line comment, either enclose the comment between <<COMMENT and COMMENT or :' and '

Bash Variables

- Variables are the containers which store data or a useful piece of information as the value inside them.
- A variable name can only contain letters (a-z, A-Z), numbers (0-9) or an underscore (_). The name can only start with alphabets and underscore.
- A variable name must have a variable value.
- Sample definitions for variables;
variable_name=variable value
VAR1="Bash Scripting"
VAR2=100

Bash Variables

- Sample script with variables defined;

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

```
VAR1="Server Collocation"
```

```
VAR2="KENET"
```

```
VAR3="5000"
```

```
echo "The price of $VAR1 at $VAR2 per U is $VAR3 Kenya  
shillings".
```

Bash Variables (read-only)

- Bash shell provides a way to mark variables as read-only by using the read-only command.
- After a value is marked read-only, its value cannot be altered.

Sample script with read-only

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

```
NAME="KENET"
```

```
readonly NAME
```

```
Name="Kenet"
```

Bash Variables (read-only)

- Working with readonly variables

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

```
NAME="KENET"
```

```
echo "NAME"
```

```
readonly NAME
```

```
Name="Kenet"
```

```
echo "NAME"
```

Bash Variables (Unsetting)

- Unsetting a variable informs the shell to remove the variable from the lists of variable it tracks.
- Once you unset or delete a variable, you cannot access the stored value in the variable.
- Syntax for unset command is; unset variable_name

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

```
NAME="KENET"
```

```
echo "NAME"
```

```
unset NAME
```

```
echo "NAME"
```

Bash Special Variables

- `$#` - Number of command-line arguments
- `$ _` - Set after the shell and contains the absolute file name of the script being executed as passed in the argument list.
- `$ -` - Expands to the current option flags as specified, by the set built-in command, set by the shell itself)like -i option)
- `$?` - Exit value or status of last executed command
- `$` - Process number of the shell
- `$!` - Process number of last background command
- `$ 0` - First word, that is, the command name which will have the full pathname if it was found via a PATH search
- `$ n` - Individual arguments (positional parameters). (n=1-9); Bash allows n to be greater than 9 if specified as `${n}`
- `$ *, $ @` - All arguments on command line (`$ 1 $ 2 ...`
- `"$*"` - Expands all arguments on the command line as one string (`"$ 1 $ 2 ..."`). Values are separated by first character in `$ IFS` .
- `"$@"` - All arguments on the command line that are individually quoted (`"$ 1" "$ 2" ...`)

Bash Arrays

- Similar to other arrays, bash arrays have numbered indexes. The values can be defined altogether or one by one.

Syntax of bash array; ARR = (Hello world)

- Working of Bash Arrays

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

```
ARRAY1=(cat dog mouse fish)
```

```
echo "First parameter of the array is = ${ARRAY1[0]}"
```

```
echo "Third parameter of the array is = ${ARRAY1[2]}"
```

Bash Operators

- Arithmetic Operators

Used to perform normal mathematical operations such as + (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulus for finding the remainder), ++ (increment), --(decrement).

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

```
read -p "enter 1st value" a
```

```
read -p "enter 1st value" b
```

```
#addition
```

```
echo "addition:
```

```
    a+b = $((a + b))
```

```
"
```


Bash Operators

- Arithmetic Operators

Used to perform normal mathematical operations such as + (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulus for finding the remainder), ++ (increment), --(decrement).

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

```
read -p "enter 1st value" a
```

```
read -p "enter 1st value" b
```

```
#division
```

```
echo "Division:
```

```
    a/b = $((($a/$b))
```

```
"
```

Bash Operators

- Arithmetic Operators

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

```
read -p "enter 1st value" a
```

```
read -p "enter 1st value" b
```

```
#unary increment operator
```

```
echo "initial value of a is $a"
```

```
echo "incremented value of a is $((++a))"
```

```
#unary decrement operator
```

```
echo "initial value of b is $b"
```

```
echo "incremented value of b is $((--b))"
```

Bash Operators

- Relational Operators

they define the relation between two operands. Result in either a true or false depending on the relation. such as ==, !=, <, <=, >, >=.

```
#!/bin/bash

STRING1="$1"
STRING2="$2"

if [ $STRING1 == $STRING2 ]
then
    echo "Both strings are the same"
fi

if [ $STRING1 != $STRING2 ]
then
    echo "Both strings are different"
fi
```

Bash Operators

- Boolean Operators

used to perform logical operations, eg logical AND (&&), logical OR (||), Not Equal to (!).

`#!/bin/bash` **Bash Operators**

```
read -p 'enter a value a: ' a
```

```
read -p 'enter a value b: ' b
```

```
if (($a == 'true' && $b == "true"))
```

```
then
```

```
    echo "both are true"
```

```
fi
```

```
if (($a == "true" || $b == "true"))
```

```
then
```

```
    echo "one is true"
```

```
fi
```

```
if ((! $b == "true"))
```

```
then
```

```
    echo "b was initially true"
```

```
fi
```

Bash Operators

- Bitwise Operators

used to perform logical operations, eg logical AND (&&), logical OR (||), Not Equal to (!).

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

```
read -p "a: " a
```

```
read -p "b: " b
```

```
echo "bitwiseAND = $((a&b))"
```

```
echo "bitwiseOR = $((a|b))"
```

```
echo "bitwiseCOMPLEMENT = $((~a))"
```

```
echo "bitwiseLEFTSHIFT = $((a<<1))"
```

```
echo "bitwiseRIGHTSHIFT = $((a>>1))"
```

Bash Operators

- File Test Operators

File Test Operators - test a particular property of a file, such as;

- -b operator; checks whether a file is a block special file or not. if exist returns true.
- -c operator; checks whether a file is a character special file or not. if exist returns true.
- -d operator; checks if the given directory exists or not. if exist returns true.
- -e operator; checks whether the given file exists or not. if exist returns true.
- -r operator; checks whether the given file has read permissions or not. if exist returns true.
- -w operator; checks whether the given file has write permissions or not. if exist returns true.
- -x operator; checks whether the given file has execute permissions or not. if exist returns true.
- -s operator; checks the size of the given file. if size is greater than 0 it returns true.

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

Bash Operators

```
if [ -b $1 ]
```

```
then
```

```
echo "$1 is a block device"
```

```
else
```

```
echo "$1 is not a block device"
```

```
fi
```

```
if [ -d $2 ]
```

```
then
```

```
echo "$2 is a directory"
```

```
else
```

```
echo "$2 is not a directory"
```

```
fi
```

```
if [ -e $3 ]
```

```
then
```

```
echo "$3 file exists"
```

```
else
```

```
echo "$3 file does not exists"
```

```
fi
```


User Input

- At times, you may want to input data and assign it to a variable directly while the script is running.
- To achieve this, we use the built-in Bash command `read`.
- The `read` only reads a single line from the Bash shell.

Syntax for reading user input is as follows;

```
read variable_name
```

User Input

- Syntax for reading user input

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

```
# Read the user input
```

```
echo "Enter the user name: "
```

```
read first_name
```

```
echo "The Current User Name is $first_name"
```

```
echo
```

```
echo "Enter other users'names: "
```

```
read name1 name2 name3
```

```
echo "$name1, $name2, $name3 are the other users."
```

User Input

- Syntax for reading user input using a prompt

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

```
read -p "username:" user_var
```

```
echo "The username is: " $user_var
```

If Statements

- a conditional statement, also known as conditional expression.
- an if statement executes a code when the given conditions are met or satisfied.
- if statement checks for the Boolean condition to evaluate as True or False.
- any operator that returns a True or False value can be used between two conditions.

syntax of the statement

if [condition]

then

statement

fi

If Statements

- Working with if-statements

```
#!/bin/bash  
read -p "enter value for a:"a  
read -p "enter value for b:"b  
if [$a -eq $b]  
then  
    echo "a equal to b"  
fi
```

If-else Statements

- if the specified condition in the if statement is not valid, the else condition will be executed.
- it is used when you specifically require an output irrespective of whether the condition is satisfied or not.
- The syntax

if [condition]

then

statement

else

statement

fi

If-else Statements

- Working with if-else statements

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

```
a=10
```

```
b=20
```

```
if [$a == $b]
```

```
then
```

```
    echo "a and b are the same"
```

```
else
```

```
    echo "a and b are not equal"
```

```
fi
```

else-if ladder

- It is used when there are multiple conditions to be checked. If any of the condition is satisfied, the instructions in that block is executed. It is used with or without an else statement at the end.
- The syntax

```
if [condition1]  
then  
    statement1  
    statement2  
elif [condition2]  
then  
    statement3  
else  
    statement4  
fi
```


Scheduling scripts

- In Linux, a cron job allows us to schedule tasks that we want to run multiple times a day, or on specific days and months.
- The general syntax of a cron job line in crontab is ;
* * * * * command_to_execute

The asterisks represent

Minute (0-59)

Hour (0 - 23)

Day of the month (1 - 31)

Month (1 - 12)

Day of the week (0 - 6) (0 is Sunday)

THANK YOU

www.kenet.or.ke

Jomo Kenyatta Memorial
Library, University of Nairobi
P. O Box 30244-00100, Nairobi.
0732 150 500 / 0703 044 500

support@kenet.or.ke / smacharia@kenet.or.ke