First Shell Script- Hello World Shell Script

- Scripting is not a programming it's a series of commands.
- You can write shell script in a file and extension can be anything, best practice is to create file with **`.sh`** it will highlight the syntax and errors in our shell script file.

- To run shell script, use - **`./script_name.sh`** if file trying to run from current directory where script is stored.

./ → because we are running our shell from current work directory, **'./'** represents the PWD.

We can mention the full path of the script file as well, if script is not in present work directory.

Eg: **`/home/user/script.sh`**

- Simple shell script for hello world →

```
echo "Hello World"
echo "this is our first shell script."
```

- If you get **permission denied error** while executing shell script, change the file permission using the chmod command, give executable permissions, so that our script can be executed.

```
$ chmod a+x script_name.sh
```

and then simply run script-

`./script_name.sh`

Shebang Shell Script- How to use shebang in Shell

The first line in shell script is - `#!/bin/bash` used to tell which shell use for execution, it is nothing but a bash interpreter who converts high level language into machine level language.
The `#!` syntax is used in scripts to indicate an interpreter for execution under **UNIX/Linux** operating systems.
The **sharp sign** (#) and the **bang sign** (!) that's why it is called as the **shebang (sharp-bang).**
Shebang starts with `#!` characters and the path to the bash or other interpreter of your choice. Make sure the interpreter is the full path to a binary file. For example: /bin/bash.

```
#!/bin/bash
sleep 300       #sleep 300 seconds used to add wait time in shell script
```

- If a script does not contain a shebang the commands are executed using your shell.
- you can print SHELL variable to show which shell you are using.

```
#!/bin/bash
echo $SHELL
```

- Different shells have slightly varying syntax.
- you can see all the available shells in **/etc/shells** file in Linux operating system.
- It makes shell scripts more like actual executable files.
- If you do a 'ps' while script is running, the real name of script appears instead of 'sh' or 'bash'. Likewise, system accounting is done based on the real name. but if we use #!/bin/bash it will show as /bin /bash ./script.sh
- It will allow other interpreters to fit in more smoothly.

```
$ cat /etc/shells
#!/bin/bash
#!/bin/python
#!/bin/sh
#!/usr/bin/tcl
#!/bin/sed -f
```

let's run a below python script with below snippet, save python script as pyscript.sh

```
#!/usr/bin/python           #used python interpreter instead of #!/bin/bash
print("Hello this is python script")
```

```
$ chmod +x pyscript.sh

$ ./pyscript.sh
Hello this is python script
```

Comments and Escape Character (\)

Comments are the useful information that the developers provide to make the reader understand the source code.

There are two types of comments:

- Single-line comment
- Multi-line comment

Single-line comments:

A single-line comment starts with a hashtag symbol with no white spaces (#) and lasts till the end of the line. If the comment exceeds one line, then put a hashtag on the next line and continue the comment.

SYNTAX

```
#!/bin/bash
# This is a comment.
echo “Hello World” #This is another comment
```

Multi-line comments

There are some tricky ways to do multi-line comments in shell script let me show you some tricks for the same

Use, `: '` to start and `'` to end multiline comment in shell scripting- (make sure you have space between `:` and `'`)

```
#!/bin/bash
: '
This is
multiline comment
in shell script
'
```

Escape Character

A Backslash(‘\’) is the bash escape character. Any character immediately following the backslash loses its special meaning and any letter following the backslash gains its special meaning. Enter the following commands in the terminal.

```
#!/bin/bash
# purpose: print some echo commands
echo this is Prashant user          # in line comment
echo 'this is our first                  shell script'   # one more comment
echo "my
name
is
Prashant"

echo "
########### Script Starting ########
purpose: going to install nginx
###################################
"


# strong quotes.
echo 'my \
```

```
name \
is \
root'   # Escape character \ taken literally because of strong quoting.

echo " my \
name \
is \
root \" # Newline escaped.

echo -e "this is prashant\t karne\t test name"
echo -e "this is prasahnt \v karne \v test name"
echo -e "this is prashant \n karne \n test name"
```

let's run the above script and see the output

```
$ ./echo.sh
this is Prashant user
this is our first                  shellscript
my
name
is
Prashant


########### Script Starting ########
purpose: going to install nginx
####################################


 my name is prashant
this is prashant   karne         test name
this is prashant
                karne
                       test name
this is prashant
 karne
 test name
```

Must use `-e` for the escape character to work echo command.

\t	→	Tab
\n	→	New line
\v	→	Vertical Tab

Difference between **'strong quotes'** and **"normal quotes"**

```
#!/bin/bash
echo 'my \
name \
is \
prashant'

echo " my \
name \
is \
```

```
prashant \
"
```

Output:

```
[root@ip-172-31-92-242 mnt]# ./new.sh
my \
name \
is \
prashant

my name is prashant
```

Echo command - print in different colours in Shell Script

echo Command

All the parameters to the echo command are printed on the screen. with the help of echo command, we can print the text in a different colour.

Start writing \033 and then [after that 0 and then ; then codes and end with m to change properties of text to be print on terminal. Use \033[0;31m to access the text properties in echo command, and to end command use \033[m

```
#!/bin/bash
echo 'this is our           shellscript'
echo -e "\033[0;31m fail message"
echo -e "\033[0;32m success message"
echo -e "\033[0;33m warning message here"
```

now let's run the above script

```
$ ./echo.sh
this is our            shellscript
fail message here
success message here
warning message here
```

you will see fail message here in red colour
success message here in green colour
warning message here in yellow colour

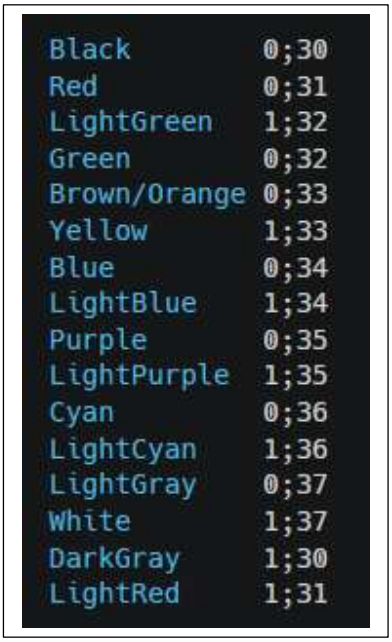
Print range:

```
echo {1..10}
```

==========colours==========

Text colour range – 30			**background colour range - 40**
Black	:	0;30	background : 40
Red	:	0;31	background : 41
Green	:	0;32	background : 42
Brown/Orange		0;33	background : 43
Blue	:	0;34	background : 44
Purple	:	0;35	background : 45
Cyan	:	0;36	background : 46

Use **\033[m** to stop using colour in same line



Variables in Shell Script- How to Define Variable?

In general word variable is a container in which we can store or to which we can assign a value and that value can be used or changed while performing various operations in our script.
A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Rules to for a variable name:

- The name of a variable can contain **only letters [a to z & A to Z]**, **numbers [0 to 9]** or the **underscore character [_]**
- Variable names cannot be reserved words (reserved words are defined for system use)
- Variable names cannot have whitespace in between
- The variable name cannot have special characters.
- The first character of the variable name cannot be a number.
- By convention, Unix shell variables will have their names in UPPERCASE.

Below are some examples are valid variable names –

```
_VARIABLE_NAME
VARIABLE_NAME
VARIABLE_1_NAME
vARIABLE_2_NAME
```

Below are some examples of invalid variable names –

```
2_VARNAME
-VARIABLENAME
VARIABLENAME-SOMENAME
SOMENAME_A!
```

The reason you cannot use other characters such as **!**, ***, or **-** is that these characters have a special meaning for the shell.

Defining Variables

Variables are defined as follows –

```
variable_name=variable_value        --- Don't leave blank space before or after =
```

For example –

```
MY_MESSAGE="Hello World"            # here variable name is –'MY_MESSAGE' and 'Hello World' is a value
A=10
number=125
class="DevOps"
```

Note that there must be no spaces around the **"="** sign: **VAR=value** works; **VAR = value** doesn't work. In the first case, the shell sees the **"="** symbol and treats the command as a variable assignment. In the second case, the shell assumes that VAR must be the name of a command and tries to execute it.

How to call a variable:

We can use $ sign followed by variable name to call the variable.
$variable_name and ${variable_name} both have same meaning.
– prefer ${variable_name} to isolate variable from other part of script.

SCRIPT:

```
#!/bin/bash
name="Prashant"
age="30"
echo ${name}
echo "my name is ${name} and my age is ${age}"
# echo 'my name is ${name} and my age is ${age}'
work="programm"
var="ing"
echo "i am $working"
echo "i am ${work}ing"
echo "i am ${work}${var}"
```

OUTPUT:

```
$ ./variable.sh
Prashant
my name is Prashant and my age is 30
i am
i am programming
i am programming
```

Variables in Linux are **'case sensitive'**

SCRIPT:

```
#!/bin/bash
_variableName="first variable"
variable2Name="second variable"
name="root"
NAME="Prashant"
nAmE="Karne"

echo "${name} ${NAME} ${nAmE}"
echo "${_variableName}"

echo "${variable2Name}"
variable_name="vartest"
echo "${variable_name}"

# 3namevariable="myname"
# echo "${3namevariable}"

my-name="root"
echo "${my-name}"
```

OUTPUT:

```
root Prashant Karne
first variable
second variable
vartest
./variable_name.sh: 22: my-name=root: not found
name
```

System Defined Variables

Created and maintained by Linux bash shell itself. This type of variable 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.

These are special types of variables. They are created and maintained by Linux Shell itself. These variables are required by the shell to function properly.

There are several commands available that allow you to list and set environment variables in Linux:

- **env** – The command allows you to run another program in a custom environment without modifying the current one. When used without an argument it will print a list of the current environment variables.
- **printenv** – The command prints all or the specified environment variables.
- set – The command sets or unsets shell variables. When used without an argument it will print a list of all variables including environment and shell variables, and shell functions.
- **unset** – The command deletes shell and environment variables.
- **export** – The command sets environment variables.

Below are some of the most common environment variables:

- **USER** - The current logged in user.
- **HOME** - The home directory of the current user.
- **EDITOR** - The default file editor to be used. This is the editor that will be used when you type edit in your terminal.
- **SHELL** - The path of the current user's shell, such as bash or zsh.
- **LOGNAME** - The name of the current user.
- **PATH** - A list of directories to be searched when executing commands. When you run a command, the system will search those directories in this order and use the first found executable.
- **LANG** - The current locales settings.
- **TERM** - The current terminal emulation

```
#!/bin/bash
# System define variables.
# echo ${SHELL}
echo ${HOME}  # will show the home directory of current user
echo ${OSTYPE} # type type of operating system.
echo $PATH # A list of directories to be searched when executing commands.
echo ${$} # process id
echo ${PPID} # parent process id
echo $PWD # present working directory
echo $HOSTNAME # hostname of machine.
echo $UID # user id
# UID="5000"
echo $UID
sleep 5
```

Now let's run the above shell script and see the output.

```
$./variable.sh
/root
linux-gnu
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
87788
68426
/var/log/amazon/ssm
aws-linux
1000
1000
```

Read Input from User in Shell Script

We can simply get user input from the `read` command in BASH.

read command provides a lot of options and arguments along with it more flexible usage

Read Basic Value from User

```
#!/bin/bash
read name
echo "Hello ${name}"
```

```
#!/bin/bash
read name
read age
echo "Hello ${name}, and your age is ${age}"
```

Read With Prompt Message

```
#!/bin/bash
read -p "please enter your name " name
read -p "please enter your age " age
echo "Hello ${name}, and your age is ${age}"
```

Read Secret Text with Prompt Message

```
#!/bin/bash
read -p "please enter your password " -s password                → # -S: Secure
echo "your password is ${password}"
```

let's create a file with below script and execute it.

```
#!/bin/bash
read -p "please enter your name " name
read -p "please enter your age " age
read -p "please enter your password " -s password
echo "Hello ${name}, and your age is ${age} and your password is ${password}"
```

What is Built-in, Keywords and Commands in Shell Script

Shell Built-in

Shell Built-in does not request another program to run a process because it is a shell built-in. So use shell built-in if it is available. A built-in command is simply a command that the shell carries out itself, instead of interpreting it as a request to load and run some other program. This has two main effects. First, it's usually faster, because loading and running a program takes time. Of course, the longer the command takes to run, the less significant the load time is compared to the overall run time (because the load time is fairly constant).

Keywords

Keywords are the words whose meaning has already been explained to the shell. the keywords cannot be used as variable names because it is a reserved word containing reserved meanings.

Sequence

when you run a command bash will search a function with the same name if the function with the same name is not present then bash will search it in built-ins. if built-in is also not available then it will search the command at PATH locations.

function → built-in → PATH location

```
$ uptime
01:37:53 up 2 min,  2 users,  load average: 0.60, 0.59, 0.26

$ type -a uptime
uptime is /usr/bin/uptime
uptime is /bin/uptime

$ type -a echo
echo is a shell builtin
echo is /usr/bin/echo
echo is /bin/echo

$ type -a pwd
pwd is a shell builtin
pwd is /usr/bin/pwd
pwd is /bin/pwd

$ type -a if
if is a reserved word

$ echo $PATH
/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/
usr/games:/home/kali/.dotnet/tools
```

Command Line Arguments in Shell Script

Arguments are inputs that are necessary to process the flow. Instead of getting input from a shell program or assigning it to the program, the arguments are passed in the execution part.

Positional Parameters

Command-line arguments are passed in the positional way i.e. in the same way how they are given in the program execution.

let's see it with an example

script:

```
#!/bin/bash

name=${1}
age=${2}

echo "Name is ${name} and age is ${age}"
```

output:

```
$ ./arguments.sh psk 20
Name is psk, and age is 20
```

Note: Make sure to use ${1} instead of $1.

```
#!/bin/bash
name=${1}
age=${2}
echo ${1}
echo ${2}
echo ${3}
echo ${4}
echo ${5}
echo ${6}
echo ${7}
echo ${8}
echo ${9}
echo ${11}
echo ${12}
echo ${13}
echo "my name is ${name}, and my age is ${age}"
echo $#
echo $@
echo $*
```

Special Variables:

`${0}` will represent the shell script name itself.

`${#}` or `$#` will show us the Total number of arguments and it is a good approach for loop concepts.

`$*` or `${*}` In order to get all the arguments as double-quoted, we can follow this way

`$@` or `${@}` Values of the arguments that are passed in the program. This will be much helpful if we are not sure about the number of arguments that got passed.

Assign a Command's Output to a Variable in Shell Script

Suppose we want to store a command output into a variable so that we can use that in our script.

Way 1: we can use back tick `

```
VARIABLE_NAME=`command_here`
```

Way 2: we can use the round brackets (recommend way) – with a parentheses '( )'

```
VARIABLE_NAME=$(command_here)
```

Example:

```
#!/bin/bash
CURRENT_WORKING_DIR=$(pwd)
VARIABLE_SECOND_METHOD=`pwd`
echo "${CURRENT_WORKING_DIR}"
echo "${VARIABLE_SECOND_METHOD}"
date_time=$(date)
echo "${date_time}"
```

output:

```
./assing-command-output-to-variable.sh
/home/kali/shellscript-
/home/kali/shellscript-
07/13/22-07:58:50
```

How to create a read-only variable in Shell Script

We can create read-only variable in shell script using readonly.

Syntax:

```
$ readonly VARIABLE_NAME
```

let's take below example:

```
 #!/bin/bash
name="hinal"
readonly name
echo "${name}"
unset name    #check if this works
name="root"
echo "${name}"
```

Set Default Value to Shell Script Variable

If parameter is unset or null Set Default Value

```
${parameter:-word}
```

If parameter is unset or null, the expansion of word is substituted. Otherwise, the value of parameter is substituted.

If parameter is unset then Set Default Value

```
${parameter-word}
```

If parameter is unset, the expansion of word is substituted. Otherwise, the value of parameter is substituted.

let's create a script and execute it for practice purpose.

```
#!/bin/bash

read -p " please enter your name " name

name=${name:-World}

echo "Hello ${name^}"

yourname=${unsetvariable-Nisha}
echo $yourname

myname=""
mytestname=${myname:-Sasha}
echo ${mytestname}
```

Output:

```
$ ./defaultvalue.sh
 please enter your name Root
Hello Root
Nisha
Sasha
```

Return Codes | Status Exit Code | Exit Status code*IMP

To check if command has successfully run or not
$? – Exit status, value 0 denotes successful execution of command, value other than 0 denotes error
We can check exit status codes with the help of man command
echo $?- Will print last command is successfully run or not

Exit Status

- Every command returns an **exit status **(sometimes referred to as a *return status *or exit code).
- A successful command returns a 0, while an unsuccessful one returns a non-zero value that usually can be interpreted as an error code.
- Well-behaved UNIX commands, programs, and utilities return a 0 exit code upon successful completion, though there are some exceptions.
- When a script ends with an exit that has no parameter, the exit status of the script is the exit status of the last command executed in the script (previous to the exit).

```
$ echo "Aws-linux"
Aws-linux


$ echo "echo: Command Not Found"
echo: Command Not Found


$ echo $?
0 #getting exit status is zero means last command executed successfully.


$ asdasd
asdasd: command not found


$ echo $?
127 # getting exit status as non-zero means last command not executed
successfully.
```

Test Command

- A test command is a command that is used to test the validity of a command or test the command.
- It checks whether the command/expression is true or false.
- It is used to check a number, string and file expression
- It is used to check the type of file and the permissions related to a file.
- Test command returns 0 as a successful exit status if the command/expression is true, and returns 1 if the command/expression is false.
- Test is used by virtually every shell script written. It may not seem that way, because test is not often called directly. test is more frequently called as [. [is a symbolic link to test, just to make shell programs more readable. It is also normally a shell built-in.

```
$ a=5 # assign a value(5) to variable a.


$ test $a -eq 5 # checking that a = 5 or not


$ echo $?
0 # in last command our expression is true so its return a successful status
means zero.


# now let's try with a=4
$ test $a -eq 4


# above express if not true so return non zero (1) exit status
$ echo $?
1
```

Check if command line argument (value of variable) is passed or not *IMP

We can check If variables value is passed or not, without the help of if condition.
How to check if variables value is not set and if we want to exit from script? without using exit, no conditional statement?

Null Command – " **:** "
: this command does nothing.
Exit status of this command is always successful.

Command to check if variable value is set or not:

```
: ${variable_name:?"message goes here..."}
```

Script:

```
#!/bin/bash
name=${1}
: ${name:?" please set variable values. "}
echo "i am here."
```

Output:

```
#output of above script when argument is passed
$ ./if-variable-not-set.sh Root
i am here.

#output of above script when argument is not passed
./if-variable-not-set.sh
./if-variable-not-set.sh: line 4: 1:  please set variable values.
```

In above script if we haven't set any value for variable name, it would have thrown an error, but we can now get a custom message for our error.

Best use of this command is to check if variable has value or passed command line argument or not?

Question:
Write a shell script to accept name and password from user and print message if name or password value is not passed:

```
#!/bin/bash
read -p "Enter a name: " name
read -p "Enter password: " -s password
: ${name:?"No value entered for name"}
: ${password:?"No value entered for password"}
```

Arithmetic Operations in Shell Script

We can do arithmetic operations in shell script in a serval way (using let command, using expr command) but I will recommend using brackets for that.

Different ways to compute Arithmetic Operations in Bash

1] Using Double Parenthesis

2] Using let command

3] using expr command

Let's see these methods one-by-one:

1] Using Double Parenthesis

Addition

```
Sum=$((20+2))
echo "Sum = ${Sum}" # output will be 22
```

Subtraction

```
sub=$((29-2))
echo "sub = ${sub}" # output will be 27
```

Multiplication

```
mul=$((20*4))
echo "Multiplication = ${mul}"  # output will be 80
```

Division

```
div=$((10/3))
echo "Division = ${div}" # output will be 3
```

Modulo

```
mod=$((10%3))
echo "Modulo = ${mod}"  # output will be one.
```

Exponentiation

```
exp=$((10**2))
echo "Exponent = ${exp}" # output will be 100.
```

let's see a shell script that will perform some arithmetic operations and some increment and decrement operations.

```
#!/bin/bash
a=5
b=6
echo "$((a+b))"
echo "$((a-b))"
echo "$((a*b))"
echo "$((a/b))" # 5/6
echo "$((a%b))"
echo "$((2**3))" # 2*2*2
((a++)) # a=a+1
echo $a
((a+=3)) # a=a+3
echo $a
```

output:

```
$ ./arithmetic.sh
11
-1
30
0
5
8
6
9
```

2] Using let Command

let command is used to perform arithmetic operations.

```
#!/bin/bash
x=10
y=3
let "z = $(( x * y ))"  # multiplication
echo $z
let z=$((x*y))
echo $z

let "z = $(( x / y ))"  # division
echo $z
let z=$((x/y))
echo $z
```

Output:

```
30
30
3
3
```

3] expr command with backticks

The arithmetic expansion could be done using backticks and 'expr ' command:

```
#!/bin/bash
a=10
b=3

# There must be spaces before/after the operator (arithmetic operator)
sum=`expr $a + $b`
echo $sum

sub=`expr $a - $b`
echo $sub

mul=`expr $a \* $b`
echo $mul

div=`expr $a / $b`
echo $div
```

Output:

```
13
7
30
3
```

Conditional Statement | Loops in Linux

if

if else,

nested if else

elif

IF with command

* check manual for test command once it has all the expressions like gt, eq, lt etc
This block will process if the exit status of COMMAND is zero (or a command executed successfully).

```
if COMMAND
then
# This block will only execute if above COMMAND's exit status is 0
# Your code here.
# Your code here.
fi
```

Example: let's create a file with the name if-condition.sh with the below content.

```
#!/bin/bash
if grep -i localhost /etc/hosts > /dev/null
#if you don't want to print standard output of any command use > /dev/null
then
  echo "Grep Command Executed successfully"
fi

echo "I am Here"
```

let's execute that file and see the output.

```
./if-condition.sh
Grep Command Executed successfully
I am Here
```

Now let's do some changes in the if-condition.sh file and try to search for another word that is not present in the /etc/hosts file.

```
#!/bin/bash
if grep -i rootyt /etc/hosts > /dev/null
then
  echo "Grep Command Executed successfully"
fi
echo "I am Here"
```

now let's check the output

```
./if-condition.sh
I am Here
```

Now you can see that our if block did not execute because the grep command did not return a zero status.

```
$ echo "hello world" > dev/null   # This command will not print output on terminal as we have used null device to print the output but exit status or this command will be 0

$ echo $?
$ 0
```

Example:

```
#!/bin/bash
if grep -i pskarne /etc/hosts > /dev/null
then
  echo "pskarne">/etc/hosts
fi
echo "done"
```

Assignment: Write a shell script to check if file is present or not, if file is present then print "File already exists" if file is not present then create a file:

```
#!/bin/bash
function check(){
        if ls ${1} /etc/hosts>/dev/null
        then
                echo "File ${1} already exists"
        else
                touch ${1}
                echo "File ${1} created succesfully"
        fi
}

read -p "enter file name: " fname
check "${fname}"
```

DevOps Use Cases:

1. Let's say we have to add one entry in /etc/hosts file then we will first check if entry is resent in the file first with grep command.

2. Check if software is already present if not then install the service on Linux machine.

3. Let's say we are working with docker and we have created a swarm service so we can check if service is already present or not if not present then we can update service image.

IF With Number

Now that you have understand how to use if statement with command and use status exit code with if command, in this section we are going to see, how to user **[** for comparison:

We can use IF with the test command.

```
#!/bin/bash

number=5

if test $number -eq 5
then
  echo "number is euqal to five"
fi
```

We can use **[** & **]** also instead of Test command

eq → equals to
lt → less than
gt → greater than
le → less than and equals to
ge → greater than and equals to
ne → not equals to

```
INTEGER1 -eq INTEGER2
       INTEGER1 is equal to INTEGER2

INTEGER1 -ge INTEGER2
       INTEGER1 is greater than or equal to INTEGER2

INTEGER1 -gt INTEGER2
       INTEGER1 is greater than INTEGER2

INTEGER1 -le INTEGER2
       INTEGER1 is less than or equal to INTEGER2

INTEGER1 -lt INTEGER2
       INTEGER1 is less than INTEGER2

INTEGER1 -ne INTEGER2
       INTEGER1 is not equal to INTEGER2

FILE1 -ef FILE2
       FILE1 and FILE2 have the same device and inode numbers

FILE1 -nt FILE2
       FILE1 is newer (modification date) than FILE2

FILE1 -ot FILE2
       FILE1 is older than FILE2
```

```
#!/bin/bash

number=15
# If number is equal to 5 then the below condition will become true.
if [ $number -eq 5 ]
then
  echo "number is eq 5"
fi

# If number is less than 11 then the below condition will become true.
if [ $number -lt 10 ]
then
  echo "number is less than 10"
fi

# If number is greater than 4 then the below condition will become true.
if [ $number -gt 4 ]
then
  echo "number is greater than 4"
fi

# If number is greater than or equal to 5 then the below condition will become
true.
if [ $number -ge 5 ]
then
  echo "number is greater than or equal to 5"
fi
# If number is less than or equal to 5 then the below condition will become
true.

if [ $number -le 5 ]
then
  echo "number is less than or equal to 5"
fi

# Is number is not equals to 5 then below condition will become true.
if [ $number -ne 5 ]
then
  echo "number is not equal to five."
Fi
```

Assignment 1 : Write a shell script to accept name and age from user, validate if arguments are passed or not without, if age is greater than 18 then print user is illegible to vote else print user is not illegible to vote.

```
#!/bin/bash

read -p "Enter your name: " name
read -p "Enter your age: " age

: ${name:?"No name has been entered, please enter a name"}
: ${age:?"No age has been entered, please enter a age"}

if test ${age} -gt 18       #simmilar to if [ ${age} -gt 18 ]
then
        echo -e "\n${name} is illegible to vote"
else
        echo -e "\n${name} is not illegible to vote"
fi
```

Assignment 2:

Write a script to accept username and password from user, check if user name is entered or not, if user name or password is empty then exit the script with message, if password and user name is correct then print “WELL-COME” if not then print “NOT AUTHORISED”

```
#!/bin/bash

user_name="admin"
password="admin@123"

while true
do
        echo
        read -p "Enter username: " name
        : ${name:?"no name is entered, exiting script"}
        read -p "Enter password for ${name}: " -s passwd
        : ${passwd:?"no password is entered, exiting script"}
        if [[ ${user_name} == ${name} && ${password} == ${passwd} ]]
        then
                echo -e "\n\t'WELL-COME ${name}'"
        else
                echo -e "\n\t'NOT AUTHERIZED\'"
        fi
done
```

IF with strings

'[' vs '[[' in Shell script and String Comparison

if we want to compare two string that is equal or not then we can use == sign.

```
#!/bin/bash
if [ "awsdevops" == " awsdevops" ]
then
  echo "both strings are equal"
fi

Output:
$ ./if-string.sh
both strings are equal
```

let's remove double quotes "" from condition and check the output again:

```
#!/bin/bash
if [ awsdevops == awsdevops]
then
  echo "both strings are equal"
fi

Output:
$ ./if-string.sh
both strings are equal
```

now let's create a variable with name as 'name'

```
#!/bin/bash
name=awsdevops
if [ name == awsdevops ]
then
  echo "both strings are equal"
fi

Output:
$ ./if-string.sh
both strings are equal
```

now let's modify variable value with space

```
#!/bin/bash
name=aws devops
if [ $name == aws devops ]
then
  echo "both strings are equal"
fi

Output: # Now you will get an error
$ ./if-string.sh
./test.sh: line 2: aws: command not found
./test.sh: line 3: [: devops: binary operator expected
```

to solve this error let's put the double quotes " "

so now the script will be like the below script.

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

```
name="aws devops"
if [ "$name" == "aws devops" ]
then
 echo "both strings are equal"
fi
```

Output:

```
$ ./test.sh
both strings are equal
```

now let's create one more variable and compare them

```
#!/bin/bash
name="aws devops"
othername="aws devops"
if [ "${name}" == "${othername}" ]
then
 echo "both strings are equal"
fi
```

Output:

```
$ ./if-string.sh
both strings are equal
```

now let's remove the " form if the condition

```
#!/bin/bash
name="aws devops"
othername="aws devops"
if [ ${name} == ${othername} ]
then
 echo "both string are equal"
fi
```

Output:

```
$ ./if-string.sh
./test.sh: line 4: [: too many arguments
```

now let's add [[in if condition and check the output.

```
#!/bin/bash
name="aws devops"
othername="aws devops"
if [[ ${name} == ${othername} ]]
then
 echo "both string are equal"
fi
```

Output:

```
$ ./if-string.sh
both string are equal
```

so here we get our desired output. and [[is the advance version of [.

Difference between [and [[

- [is the same as the test built in and works like the test binary (man test)
- works about the same as [in all the other sh-based shells in many UNIX-like environments
- only supports a single condition. Multiple tests with the bash && and || operators must be in separate brackets.

- doesn't natively support a 'not' operator. To invert a condition, use a ! outside the first bracket to use the shell's facility for inverting command return values.
- == and != are literal string comparisons
- [[is a bash
- is bash-specific, though other shells may have implemented similar constructs. Don't expect it in an old-school UNIX sh.
- == and != apply bash pattern matching rules, see "Pattern Matching" in man bash
- has a=~ regex match operator
- allows the use of parentheses and the !, &&, and || logical operators within the brackets to combine subexpressions
- With [[no need to use “” for string comparison
- Use [[instead of [

IF with Files

We know that everything in Linux is a file, directory, regular file, any block device is also a file.
Suppose we want to check that file is a regular file or directory, the file has read, write or execute permission then again we have to use test command or ([[or [) with if condition can also tell us the type of file:

When we run command,

```
$ ls -ltr
```

we can see file types by first character in the details:
- d : Directory
- f : File
- c : Character device
- l : Symbolic link (or shortcut)
- b : block device

```
$man test
```

-b FILE
FILE exists and is block special
-c FILE
FILE exists and is character special
-d FILE
FILE exists and is a directory
-e FILE
FILE exists
-f FILE
FILE exists and is a regular file
-g FILE
FILE exists and is set-group-ID
-G FILE
FILE exists and is owned by the effective group ID
-h FILE
FILE exists and is a symbolic link (same as -L)
-k FILE
FILE exists and has its sticky bit set
-L FILE
FILE exists and is a symbolic link (same as -h)
-O FILE
FILE exists and is owned by the effective user ID
-p FILE
FILE exists and is a named pipe
-r FILE
FILE exists and read permission is granted
-s FILE
FILE exists and has a size greater than zero
-u FILE
FILE exists and its set-user-ID bit is set
-w FILE
FILE exists and write permission is granted
-x FILE
FILE exists and execute (or search) permission is granted

here is an example.

```
#!/bin/bash
file_full_path="/home/kali/abc.txt"

# check file is a directory.
if [[ -d $file_full_path ]]
then
   echo  "${file_full_path} is a dir"
fi

# -b means file is block device.
if [[ -b $file_full_path ]]
then
   echo  "${file_full_path} is a block device"
fi

#check, file is a char device.
if [[ -c $file_full_path ]]
then
   echo  "${file_full_path} is a char device"
fi

#check, file exists.
if [[ -e $file_full_path ]]
then
   echo  "${file_full_path} is a exist device"
fi

#check, file have read permission.
if [[ -r $file_full_path ]]
then
   echo  "${file_full_path} have read permission"
fi

# check, file have write permission
if [[ -w $file_full_path ]]
then
   echo  "${file_full_path} have write permission"
fi
# check file have execute permission.
if [[ -x $file_full_path ]]
then
   echo  "${file_full_path} have execute permission"
fi
```

IF-Else in Shell Script

If the specified condition is not true in the if part then the else part will be executed.

Syntex:

```
if [ expression ]
then
   statement1
else
   statement2
fi
```

Example:

```
#!/bin/bash
name=""
othername="prashant karne"

if [[ -n ${name} ]]
then
    echo "length of string is non zero"
else
    echo "length of string is zero"
fi

if [[ -z ${name} ]]
then
    echo "length of string is zero -two"
else
    echo "length of string is non zero. = two"
fi

if [[ ${name} == ${othername} ]]
then
    echo "both string are equals - three"
else
    echo "both string are not equals. - three"
fi

if [[ ${name} != ${othername} ]]
then
    echo "both string are not equals -four"
else
    echo "both strings are equals -four"
fi
```

Output:

```
$ ./if-else.sh
length of string is zero
length of string is zero -two
both strings are not equals. - three
both strings are not equals -four
I am Here
```

Nested If Else in Shell Script

Example:

Accept a number from user and tell if number is greater, lesser or equal to 10

```
#!/bin/bash
read -p "Enter a number: " num
: ${num:?"No value is entered"}
if [[ $num -eq 10 ]]
then
       echo -e "\nNumber is equal to 10"
else
       if [[ $num -lt 10 ]]
       then
              echo -e "Number is less than 10"
       else
              echo -e "Number is greater than 10"
       fi
fi
```

we can define if-else inside if-else.

A nested if-else block can be used when one condition is satisfied then it again checks another condition.

Example:

```
#!/bin/bash
number=9
if [[ ${number} -gt 10 ]]
then
    if [[ $number -gt 50 ]]
    then
        if [[ ${number} -gt 100 ]]
        then
            echo "number is greater than 100"
        fi
    else
        echo "number is in between 11 to 50"
    fi
else
    echo "number is less than or equal to 10"
fi
```

```
$ ./nested-if-else.sh
number is less than or equal to 10
```

Elif in Shell Script

if..elif..else..fi statement (Else If ladder)

To use multiple conditions in one if-else block, then elif keyword is used in shell. If the expression1 is true then it executes statements 1 and 2, and this process continues. If none of the conditions is true then it processes else part.

Syntax:

```
if [ expression1 ]
then
   statement1
```

```
   statement2
   .
   .
elif [ expression2 ]
then
   statement3
   statement4
   .
   .
else
   statement5
fi
```

Example:

```
#!/bin/bash
number=4

if [[ ${number} -eq 10 ]]
then
  echo "number is 10"
elif [[ ${number} -lt 10 ]]
then
  echo "number is less then 10"
elif [[ ${number} -lt 5 ]]
then
  echo "number is less than 5"
else
  echo "number is greater than 10"
fi
```

Output:

```
$ ./elif.sh
number is less than 10
```

Example:

```
#!/bin/bash

read -p "Enter number: " num
: ${num:?"No numer is entered, exiting script"}
if [[ ${num} -eq 1 ]]
then
       echo -e "\nNumber= 1"
elif [[ ${num} -eq 2 ]]
then
       echo -e "\nNumber= 2"
elif [[ ${num} -eq 3 ]]
then
       echo -e "\nNumber= 3"
       else
       echo -e "\nNumber is greater than 3"
fi
```

Assignment:

Ask user “Do you want to continue (Y/y/yes)?” and on the basis of input from user print you have selected Y, y or yes:

```
#!/bin/bash

read -p "Do you want to continue (Y/y/yes): " ch
: ${ch:?"choice cant be null"}
if [[ ${ch} == "Y" ]]
then
       echo -e "\nYour choice is: 'Y'"
elif [[ ${ch} == "y" ]]
then
       echo -e "\nYour choice is: 'y'"
elif [[ ${ch} == "yes" ]]
then
       echo -e "\nYour choice is: 'yes'"
else
       echo -e "\nYour choice is incorrect!"
fi
```

Operators in Shell Script

Not Operator (!)

The NOT logical operator reverses the true/false outcome of the expression that immediately follows..
For example, if a file does not exist, then display an error on the screen.

```
#!/bin/bash
name="awsdevops"
othername="aws devops"
if [[ ! ${othername} == ${name} ]]
then
    echo "both are same"
fi

Output:
$ ./if-not.sh
both are same
```

And (&&) Operator in Shell Script

We can check multiple commands in a single if condition statement, The second command will only execute if the first command has executed successfully. i.e, its exit status is zero. This operator can be used to check if the first command has been successfully executed. This is one of the most used commands in the command line.

Syntax:

```
command1 && command2
```

command2 will execute if command1 has executed successfully. This operator allows us to check the exit status of command1. and it returns true only and only if all commands execute successfully.

```
$ ping -c 1 8.8.8.8>/dev/null && echo "Internet working fine."
Internet working fine.
```

Example:

```
#!/bin/bash
# os == linux && user == root
OS_TYPE=$(uname)
if [[ ${OS_TYPE} == "linux" &&  ${UID} -eq 0 ]]
then
    echo "user is root user and os is linux."
fi
```

let's execute the above program as a non-root user and see the output

```
$ ./if-and-operator.sh
```

now let's run the same program as a root user.

```
$./if-and-operator.sh
user is root user and os is linux.
```

OR (||) Operator in Shell Script

This is logical OR. If one of the operands is true, then the condition becomes true.

Syntax:

```
command1 || command2
```

command2 will execute if command1 has failed.
and it returns false only and only if all commands return not zero exit code.

```
$ ping -c 1 8.8.8.8>/dev/null || echo "Internet is not working."
Internet is not working.
```

Example:

```
#!/bin/bash
OS_TYPE=$(uname)
if [[ ${OS_TYPE} == "Linux" || ${UID} -eq 0 ]]
then
  echo "user is root user or os is linux."
fi

Output:
$ ./if-or-operator.sh
user is root user or os is linux.
```

now let's run the same program as a root user.

```
$./if-or-operator.sh
user is root user or os is linux.
```

Example 2:

```
#!/bin/bash
read -p "do you want to continue (Y/y/yes) " uservalue
if [[ ${uservalue,,} == 'y' || ${uservalue,,} == 'yes' ]]
then
  echo "you want it"
else
  echo "you dont want it."
fi
```

executing the above script four-time and supply different outputs and check the output in the below section.

```
$ ./if-or-operator.sh
do you want to continue (Y/y/yes) y
you want it

$ ./if-or-operator.sh
do you want to continue (Y/y/yes) Y
you want it

$ ./if-or-operator.sh
do you want to continue (Y/y/yes) yes
you want it

$ ./if-or-operator.sh
do you want to continue (Y/y/yes) Yes
you want it
```

Use Case: Suppose we want to install any service on system but we have to check if service is present on our system or not if service is not present then we will install the service, we can achieve this with OR statement. In first condition we will check if service is installed is output is 0 then script will not go for next condition.

Eg: `$ kubectl get svc servicename || kubectl create svc`

Example:

print authorised if given name is prashant or ashish only!

```
#!/bin/bash

while true
do
        echo
        read -p "Enter your name: " name
        : ${name:?"No name entered, exiting script!"}
        if [[ ${name} == "prashant" || ${name} == "ashish" ]]
        then
                echo -e "\nAUTHORIZED :D'"
        else
                echo -e "\nUN-AUTHORIZED :('"
        fi
done
```

While Loop in Shell Script

A while loop is a statement that iterates over a block of code till the condition specified is evaluated to true. We can use this statement or loop in our program when do not know how many times the condition is going to evaluate to false before evaluating to true.

This repeats until the condition becomes false.

Syntax:

```
while [[ condition ]]
do
    # statements
    # commands
done

while [ condition ]
do
    # statements
    # commands
Done
```

Example-1

```
#!/bin/bash
while [[ $answer != “yes” ]]
do
    read -p “please enter yes “ answer
done
```

Example-2

```
#!/bin/bash
# example of infinite loop
while true
do
    echo "this is test"
done
```

Output:

```
this is test
this is test
...
...
```

Example-3

```
#!/bin/bash
read -p "please enter a number " number
initNumber=1
while [[ ${initNumber} -le 10 ]]
do
    echo $((initNumber*number))
    ((initNumber++))
done
```

output:

```
$ ./while-loop.sh
please enter a number 2
2
4
6
8
10
12
14
16
18
20
```

Example:

Print number until user want:

```
#!bin/bash

while true
do
        echo
        read -p "Enter a number: " num
        : ${num:?"No number is entered"}
        number=1
        while [[ ${number} -le ${num} ]]
        do
                echo -e "\t${number}"
                sleep 1
                ((number++))
        done
done
```

Debug Mode

Command:

```
$ bash -x <script_name>
```

Example:

```
#print number until user want
#!bin/bash

while true
do
        echo
        read -p "Enter a number: " num
        : ${num:?"No number is entered"}
        number=1
        while [[ ${number} -le ${num} ]]
        do
                echo -e "\t${number}"
                sleep 1
                ((number++))
        done
done
```

Debug Mode:

```
[root@ip-172-31-39-151 mnt]# bash -x main.sh
+ true
+ echo

+ read -p 'Enter a number: ' num
Enter a number: 2
+ : 2
+ number=1
+ [[ 1 -le 2 ]]
+ echo -e '\t1'
        1
+ sleep 1
+ (( number++ ))
+ [[ 2 -le 2 ]]
+ echo -e '\t2'
        2
+ sleep 1
+ (( number++ ))
+ [[ 3 -le 2 ]]
+ true
+ echo

+ read -p 'Enter a number: ' num
```

Read File in Shell Script

We can read a file with the help of read and while.
The return code of read command is zero, unless the end-of-file is encountered.

If we want to read file line by line best way to use while loop:

```
#!/bin/bash
file_path="/etc/passwd"
while read line
do
    echo "$line"
    sleep 0.20
done < $file_path    #standard input
```

OR

```
#!/bin/bash
cat /etc/passwd | while read line  #another way of reading file
do
    echo "$line"
    sleep 0.20
done
```

Until Loop

The Until loop is used to iterate over a block of commands until the required condition is false. Same as while loop but **while was running till value was true**, until will run till the value is false as soon as the value becomes true until will stop iterating.

Syntax:

```
until [ condition ];
do
 block-of-statements
done
```

Here, the flow of the above syntax will be --

- Checks the condition.
- if the condition is false, then executes the statements and goes back to step 1.
- If the condition is true, then the program control moves to the next command in the script.

Example

```
#!/bin/bash
read -p "please enter a number" number
initNumber=1

until [[ initNumber -eq 11 ]]
do
    echo $((initNumber*number))
    ((initNumber++))
done
```

Output:

```
$ ./until-loop.sh
please enter a number3
3
6
9
12
15
18
21
24
27
30
```

Example:

```
#!/bin/bash
x=10
until [[ ${x} -eq 20 ]]
do
        echo -e "${x}"
        ((x++))
        sleep 0.50
done
```

For Loop

The for loop moves through a specified list of values until the list is exhausted.

- Keywords are **`for, in, do, done`**
- List is a list of variables which are separated by spaces. If list is not mentioned in the for statement, then it takes the positional parameter value that were passed into the shell.
- varname is any variable assumed by the user.

```
#!/bin/bash

for variableName in item1 item2 item3 item4 item5 item6
do
 echo "${variableName}"
done
```

output:

```
item1
item2
item3
item4
item5
item6
```

We can use range in for loop. * range {1..10}

```
#!/bin/bash

read -p "please enter a number " number
for variableName in {1..10}
do
  echo $((variableName*number))
done
```

output:

```
please enter a number 2
2
4
6
8
10
12
14
16
18
20
```

```
#!/bin/bash
for variableName in "Prashant karne" "Ashsih karne" "Abhinay karne"
do
 echo "${variableName}"
done
```

output:

```
Prashant karne
```

```
Ashish karne
Abhinay karne
```

```
#!/bin/bash
for i in *    #will list all files from current directory
do
  echo $i
done
```

output: will print all the files and folder name of present working directory.

```
#!/bin/bash
for i in $(ls *.txt)
do
  echo "$i"
done
```

output: it will print all the file name with txt extension

Assignment: Write a shell script to print all even numbers between 1 - 50:

```
for num in {1..50}
do
        if [[ $((num%2)) == 0 ]]
        then
                echo -e "\n${num}"
        fi
done
```

Assignment: Write a program to print table of 2:

```
#!/bin/bash
num=2
for i in {1..10}
do
        echo $((num*i))
        sleep 0.50
done
```

Break Statement in Shell script

The break statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

Syntax

The following break statement is used to come out of a loop.

```
break
```

The break command can also be used to exit from a nested loop using this format.

```
break n
```

Here n specifies the nth enclosing loop to the exit from.

Example:

```
#!/bin/bash
initNumber=1
while [[ ${initNumber} -lt 10 ]]
do
    echo ${initNumber}
    if [[ ${initNumber} -eq 5 ]]
    then
      echo "condition is true number is ${initNumber} going to break the loop."
      break;
    fi
    ((initNumber++))
done
```

Case in Shell Script

You can use multiple if..elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Shell supports case...esac statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

The case statement saves going through a whole set of if .. then .. else statements. Its syntax is really quite simple:

Example:

```
#!/bin/bash
action=${1,,}
# start,stop,restart,reload
case ${action} in
    start)
        echo "going to start"
        echo "actionone two"
        ;;
    stop)
        echo "going to stop"
        ;;
    reload)
        echo "going to reload"
        ;;
    restart)
        echo "going to restart"
        ;;
    *)
        echo "please enter correct command line args."
esac
```

Let's run the above program with one command line argument.

```
$ ./casestatement.sh START
going to start
actionone two



$ ./casestatement.sh start
going to start
actionone two



$ ./casestatement.sh stop
going to stop



$ ./casestatement.sh sToP
Going to stop
```