Shell scripting :

* Basename :- It gives basename of the file

Ex:- ① basename abc.html .html

o/p:- abc

file.txt
↓ ↓
basename extension name

② basename abc.html html

o/p:- abc.h

[basename → extract a specific character]

* dirname :- It gives directory name of a file

Ex:- dirname /opt/x

o/p:- /opt

* SHELL :- Shell is a command line interpreter, which converts the commands into kernel understandable language.

* Advantages :-

→ Code reusability
→ Execute more commands at a time
→ for executing the scheduled jobs
→ Reducing the code of maintainence
→ customizing the task, Reduces the administrative of task.

Note: pwd
To execute two commands
+ Telnet (or) cygwin
(run linux commands in windows)

* Shell script :- Shell script is a file which is having a group of shell commands, grouped together to perform specific task.

* drawback :- we can execute the script on a particular OS.

* vi first.sh [create a file]

```
date
pwd
who
```
[writes the commands]

* executing the script :-

→ we are having different ways to execute the scripts,

1. By specifying Interpreter name

Ex:- sh first.sh (or) bash first.sh

2. using Source command

Ex:- source /scripts/first.sh [in different path]

Ex:- source first.sh [in same path, shell takes care of the shell name]

(3) . space script name.

Ex:- . first.sh

. /scripts/first.sh

↓

says current shell

Note :- '.' means current shell, it automatically execute the scripts in that shell.

(4) we complete path

Ex:- /scripts/first.sh

Error:- permission denied

Note:- If any user wants to execute the script like without giving any interpretes name. Then they should have execute permission on that.

→ To provide execute permission

eg:- chmod +x first.sh

* When we are executing the script just by giving path, we need execute permission, otherwise no need the execute permission.

./first.sh
↳ scriptname

. → current directory

Copy to bin directory

cp /scripts/first.sh /bin

* .sh [means extension] is not required, it is for understanding purpose.

***

⇒ Hashbang :- It is used to forces the user to execute a script by accessing specific shell.

Syntax :- #! /bin/shellname

Eg:- #! /bin/bash

* echo :- displays given string on the screen.

Ex:- echo hi o/p:- hi

echo bye o/p:- bye

echo date o/p:- date.

** Command Substitution :- backquotes (` `) :-

→ backquotes (` `) used for command substitution

Eg:- #! /bin/bash

echo current working directory is: `pwd`

echo " logged in users : `who | wc -l`

** back slash characters :-

To generate formatted output.

\n → newline

\t → horizontal tab

\v → vertical tab

\b → backspace

\r → carriage return [returns the cursor to the start of the line]

\c → To keep the cursor in same line

\' → To print message in single quote

\" → " " " " double "

\\ → " " " " blackslash.

Using quotes in scripting :-

→ ` ` (back quotes) → command substitution

→ ' ' (single quotes) → It turn-off meaning of special symbols but it processes blackslash characters.

→ " " (double quotes) → double quotes substitute back slash characters & special symbols also [commands].

Syntax :-

" " - echo ---- `cmd` --- {Echo quotes (or) double quotes}
command substitution

' ' - echo 'string. \n ---'

" " - echo "string \n ---- `cmd` ----"

echo ---- \n --- { Single (or) double quotes}

Ex:- ① vi third-sh

#! /bin/bash.

echo -e current working directory is: "pwd" \n currently logged in users: "who | wc -l"

echo -e 'current working Directory is: "pwd" \n currently logged in users: "who | wc -l"'

echo -e " current working Directory is: "pwd" \n currently logged in users: "who | wc -l"

② 1. echo -e hi \t bye

2. echo -e hi \b bye

3. echo -e hi \r bye

4. echo -e hi \c bye → cursor prints before \c statement and ready but not displays the statement

5. echo -e hi \v bye.

o/p:- 1. hi bye

2. hbye

3. bye

4. hi ------ [it moves next line to cursor position]

5. hi
bye

* Shell keywords :- {predefine, reserved, built in words}
The words whose meaning is already defined (already explained to the shell) are keywords.

Keywords :- echo, sleep, read, if, else, for, while, do, then, until.

* Variables :- Variable is an identifier, it identifies data which is stored in the memory location {variable is the memory location to save data}.

→ The value of a variable can change dynamically.

→ We are having 2 types of variables. They are :-

1. Environment variables (or) built-in variables (or) predefined (or) shell variables

2. user defined variables.

1. Environment variables :-

* These are used to change behaviours of the system, behaviour of the system it changes the shell.

Ex:- PATH, SHELL, USER, HOME etc. //* predefined variables are capital letters *//

→ To access the content of a variable we use the special symbol. ($).

Ex:- echo $SHELL /* echo $PATH */.

O/P:- /bin/bash

* env :- This command is used to display all the environment variables.

env ↵ (displays all environment variables).

* User defined variables :-

A variable name should not have any special symbol except underscore (_).

→ Spaces are not allowed in between variable names.

→ keywords cannot be used as variable names.

Ex:- My_variable

Ex:- Echo (because keywords are in small characters).

Categories

1. Local variable
2. Global variable
3. Null variable
4. Constant variable.

* Local Variables :- A local variable is having the scope only to the shell in which it is defined (That is stored in shell memory & shared to that shell only).

Ex:- x=10

echo $x → o/p :- 10

& Now change the shell (sh) then echo $x.

→ Set :- This command is used to display all the variables using by current shell.

Syntax :- set | more

set | grep ^x

* Global variables :-

A global variable can be shared by all the shells.

→ To create the global variables, we use the keyword "export"

Ex:- y=20

* export y (or) export y=10

⇒ To see all the variables → Set (enter)

* Null Variables :-

A variable which is defined without assigning a value.

Syntax :- k= (or) k='' (or) k="

* This is used in functions.

* Constant Variables :-

Constant variables, whose values cannot be changed.

→ "readonly" is the keyword to define constant variables.

Syntax :- readonly x=50

→ If you want to export to global → export x.

/etc/profile.bash

* To remove a variable from shell memory?

Ans:- unset y.

* How to delete empty lines in a file?

Ans:- vi second.sh

#! /bin/bash

echo -e "Sed -i '/^$/d' abc"

* Swap the two files :- { this is used to transfer (or) copy the console content to another file }.

* vi swap.sh

```
mv abc temp-abc
mv xyz abc
mv temp-abc xyz
```

* Shell Input values :-

1. read :- read a line from standard input (or) a file. It reads only one line.

Syntax :-
```
read k
: enter value
→ echo $k
```

Eg :- read a b

10 405

1st value 2nd value.

* expr :- To evaluate expressions.

* Arithmetic operators :-

+, -, *, /, %

Ex :- 2)9(4 → / → mod quotient

8

1 → % → remainder.

Q

Au

Ex

Q:- WAS two swap two columns of a file?

Ans:- vi swap_column

```
#! /bin/bash
awk -f: '{print $1,$4,$3,$2}' abc > tmp_abc
mv tmp_abc abc
```

(or)

```
#! /bin/bash
fn=/scripts/abc
awk '{OFS="\t"}{print $1,$4,$3,$2}' $fn > tmp_abc
rm -rf fn
mv tmp_abc $fn
```

* Read -d:- To specify delimiter

  * read -d: a b

* arithmetic operations :-

Ex:- vi arith.sh

```
echo "Input two values"
read x+y
echo "x+y : `expr $x + $y`"
echo "x*y : `expr $x \* $y`"
```

(or)

```
z=$((x+y))
echo "x+y :$z"
```

* Copy :-

```
#! /bin/bash
echo "Input source file path"
read src
echo "Input destination file path"
read des
cp $src $des
echo "copied Successfully"
```

* positional parameters :-

→ It represents command line arguments passing to a script.

→ command line arguments is the value which is passing to a script (or) a program from command line.

→ positional parameters are

$0 → represents program (or) script name
$1 → " first argument
$2 → " second "
⋮ ⋮ ⋮
$n → " nth argument.

→ $# → represents the no. of argument counts (how many)

→ $*, $@ → " total arguments in a line (or) list

→ $$ → represent process id created for that script (or program.

→ $? → exit status of previous command

↳ 0 → successfull

↳ 1 → not successfull.

* providing password in the command line for ssh connection :-

* sshpass -p "password" sshusername@Ipid cmd -f <path>

→ vi abc.sh

```
echo "scriptname: $0"
ceho "Argument count : $#"
eeho " Argument first: $1"
eeho " Second Argument: $2"
eeho "process Id: $$"
```

run:- ./abc.sh 10 hi

* script to write, count files from one server to another server. 192.168.0.71 192.168.0.72

Ans:- vi count_files.sh

```
#! /bin/bash
Server1 = $1
Server2 = $2
minin = $3
```

count1 = `ssh user@server1 ls -l mydir | grep ^- | wc -l`

count2 = `ssh user2@server2 ls -l mydir | grep ^- | wc -l`

echo "files count on $1 : ${count1}"

echo "files count on $2 : ${count2}"

Q: How to change file1.html to file1.xml

Ans:-

```
#! /bin/bash
fn=$1
bn=`basename ${fn} .html`
mv ${bn}.html ${bn}.xml
```

Run:- ./convert.sh file1.html
↳ create file1.html

* shift :- this command shifts the position of command-line arguments.

Ex:-

```
vi shiftx.sh
echo "No. of arguments : $#"
echo "First Argument : $1"
echo "Second Argument : $2"

shift 2
echo "No of arguments : $#"
echo "First Argument : $1"
echo "Second Argument : $2"
```

o/p:- ./shiftx.sh hi hello welcome bye

* Relational operators :- used for comparision

-lt → lessthan
-gt → greaterthan
-le → lessthan(or) equal to
-ge → greaterthan (or) equal to
-eq → equal to
-ne → not equal to

* logical operators :- used to combine 2 (or) more expressions

-a → and
-o → or
! → not

* file testing operators :- To test file attributes.

-e → True (If file exists returns true)
-f → true If it is false
-d → true " " " directory
-L → true " " " symbolic link (or) soft link
-u → true if the user is the owner (user who is executing the script command)
-g → true if the file belongs to group.
-r → True if the user is having read permission on the file
-w → " " " " write "
-x → " " " " execute "
-s → " " " file having size (not zero byte file).

file1 -ef file2 → true if the files are same (hard link)
file1 -nt file2 → " " " file1 is newer than file2
file1 -ot file2 → " " " " " older " "

* Conditional Control Statement :- To control the flow of execution

if → if is having different forms

① Simple if :-

```
if [ condition/expression ];
then
    statement(s)
fi
```

② If else :-

```
if [ condition/expression ];
then
    statement(s)
else
    statement(s)
fi
```

③ Nested if :-

```
if [ expr ];
then
    if [ expr ];
    then
        statement(s)
    else
        statement(s)
    fi
else
    statement(s)
fi
```

④ Nested else if / ladder else if :-

```
if [ expr ];
then
```

```
    Statement(s)
else if [expr];
  then
    Statement(s)
elseif [expr];
  then
    statement(s)
  else
      Statements
fi
fi
fi
```

else if -> elif

* script to display largest number.

Ans:-

```
if [ $# -ne 2];
  then
    echo "Specify two valid arguments"
    echo "Usage: $0 <value1> <value2>"
    exit
  fi
  val1=$1
  val2=$2
  if [ $val1 -ne $val2];
    then
      if [ $val1 -gt val2];
        then
          echo "$val1 is big"
        else
          echo "$val2 is big"
        fi
    else
        echo "both are equal"
    fi
```

write a script whether the user is existed or not

Ans:-

```
#! /bin/bash
if [ $# -ne 1 ] ;
then
   echo "please specify one user name"
   echo "usage: $0 <username>"
    exit
fi
user=$1
grep -i ^$user /etc/passwd > /dev/null
if [ $? -eq 0 ];
   then
      echo " user is existing"
   fi
```

write a script whether a process is running (or) not?

Ans:-

```
#! /bin/bash
if [$# -ne 1];
  then
    echo " specify one process name"
    echo " usage: $0 <process name>
      exit
  fi
  pname=$1
```

~~ps -ef | grep -i~~

```
ps -ef | grep -i $pname | grep -v grep > /dev/null
if [ $? -eq 0 ];
then
    echo "$pname is running"
else
    echo "$pname is not running"
fi
```

Write a script to check whether the root is having ssh permission.

Ans:-

```
vi ssh_root
key='grep PermissionRootLogin /etc/ssh/sshd_config | head -1 | awk '{print $2}'
if [ $key = "yes" ];
then
    echo "Root is having ssh login permission"
else
    echo "Root is not having ssh login permission"
fi
```

* tr :- To translate given character

Ex:- echo india

o/p:- india

echo india | tr i I

(2) tr mail MAIL < /test
↳ i/p file

(3) tr [a-z] [A-Z] < /test
[small letters to capital letters]

(4) tr [[:lower:]] [[:upper:]] < /test

* Using test :-

```
echo "Enter 2 values"
read x y
if test $x -gt $y      [Instead of square braces we can use test
then
  echo "$x is big"
else
  echo "$y is big"
fi
```

Run :- sh big.sh

* Above "if" Condition can write in single statement as :-

→ test $x -gt $y && echo "$x is big" || echo "$y is big"

&& → True
|| → false.

* write a script for a leap year?

Ans:-

```
y=$1
if [ `expr $y % 4` -eq 0 -a `expr $y % 100`
    -ne 0 `expr $y % 400` -eq 0 ];
then
    echo "$y is a leap year"
else
    echo "$y is not a leap year"
fi
```

* To redirect error messages :-

Syntax :- cmd > /dev/null 2>&1

0-stdin
1-stdout
2-stderr

Eg :- rm a > /dev/null 2>&1

* If you don't give /dev/null 2 is considered as a filename.
* If you (don't) give 1 instead of &1 it creates the files with name 1 and redirect the error message to the filename 1.

Standard file descriptors :-

0-stdin (Standard input)
1-stdout (Standard output)
2-stderr (Standard error).

* Write a script a name is file, directory (or) special file.

Ans:-

```
if [ $# -ne 1 ];
then
    echo "specify file as argument"
    echo "usage: $0 <path>"
    exit
fi

name=$1
if [ -e $name ];
then
    if [ -d $name ];
    then
        echo "$name is a directory"
    elif [ -f $name ];
    then
        echo "$name is a file"
    else
        echo "$name is a special file"
    fi
else
    echo "file not existed"
fi
```

Run :- ./check.sh abc

if 'abc' file is present in current location it gives true otherwise we have to give path

* write a script to check you are the owner of the file & if you are owner check for write permission

Ans:-

```
if [ $# -ne 1 ];
then
    echo "specify file as argument"
    echo "usage: $0 <path>"
exit
fi
name=$1
if [ -o $name ];
then
    if [ -w $name ];
    echo " you are having write permission"
    else
     echo " you are not having write permission"
     fi
else
    echo " you are not owner"
fi
```

* write a script to check whether the file is empty (or) not.

Ans:

```
if [ -s if [ $# -ne 1 ];
        then
        echo "specify file as argument"
        echo "usage: $0 <path>"
        exit
        fi
```

```
name=$1
if [ -s $name ];
then
    echo "$name is not empty"
else
    echo "$name is not empty"
fi
```

To check whether the directory is empty (or) not.

Ans:-

```
if [ $# -ne 1];
then
    echo "Specify directory name"
    echo "usage: $0 <path"
    exit
fi
dirname=$1 → if [ -e $dirname -a -d $dirname ];
then
count=`ls -a $dirname | wc -l`
if [ count -gt 2 ]
then
    echo "$dirname is not empty"
else
    echo "$dirname is empty"
fi
else
    echo "$dirname not existed"
fi
```

* write a script to find out hours..

Ans:- hr = date | awk '{print $4}' | awk F: '{print $2}'

hr = `date '+%H'`

if [$hr -ge 0 -a $hr -le 11];

then

echo "Morning hours"

elif [$hr -ge 12 -a $hr -le 15];

echo "AfterNoon hours"

else

echo "Evening hours"

fi

* case control statements :- It is used to compare single expression with multiple values.

→ case compares only the equality

→ especially this is used for menu operations.

drawbacks :

→ case doesn't supports relational & logical operators.

Syntax :-

```
case expr in
pattern 1) statements
           ;; -> Termination
pattern 2) statements
           ;;
pattern n) statements
           ;;
```

*) Statements (optional)

esac

* Write a script to check for weekday

Ans:

```
If [ $# -ne 1 ];
then
    echo "Specify one argument"
    echo "Usage: $0 <weekday number>"
    exit
fi
w=$1

case $w in
0) echo "Sunday"
   ;;
1) echo "Monday"
   ;;
2) echo "Tuesday"
   ;;
3) echo "Wednesday"
   ;;
4) echo "Thursday"
   ;;
5) echo "Friday"
   ;;
6) echo "Saturday"
   ;;
*) echo "Not a week day"
esac
```

* write a script to check for vowel.

Ans:-

```
echo -e "Input a character : \c"
read ch
case $ch in
[aeiouAEIOU] ) echo "vowel"
               ;;
* ) echo "not vowel"
esac
```

* write a script to check upper case, lowercase, digit

Ans:-

```
echo -e "Input a character : \c"
read ch
case $ch in
[0-9] ) echo "digit"
        ;;
[a-z] ) echo "character is lowercase"
        ;;
[A-Z] ) echo "character is uppercase"
        ;;
* ) echo "other symbol"
esac
```

* To check the root partition / filesystem space usage.

Ans:-

* Mail (or) mailx :-

```
mail -s "subject" -c <ccmailid1, id2 ....>
     -b <bccmailid1, id2 ...:>
     -a <path (attachment>
     <tomailid1, id2 ...>   < "msg"  (or)
                            < "msg filepath"
```

* write a script on file menu operations?

Ans:-

```
if [ $# -ne 1];
then
  echo "please specify file path"
  echo "usage: $0 <file path>"
  exit
fi
fn=$1
if [ ! -e $fn -o ! -f $fn ];
then
  echo "$fn is not existed or it is not a file"
  exit
fi
while [1];
do clear
clear
echo -e "\n\n\t\t\tMENU"
echo -e "\t\t\t****"
```

```
echo -e "\n\t\t 1. Display file"
echo -e "\t\t 2. Append file"
echo -e "\t\t 3. Rename a file"
echo -e "\t\t 4. Remove the file"
echo -e "\t\t 5. Exit"
echo -e "\n\t\t Input choice : \c"
read ch
clear
case $ch in
1)
    if [ -r $fn ];
    then
        echo -e "File content"
        echo -e "------"
        Cat $fn
    else
        echo "Sorry --- you don't have read permission"
    fi
    ;;
2)
    if [ -w $fn ];
    then
        echo "Input Data to Append"
        Cat >> $fn
    else
        echo "Sorry --- you don't have permission
                                to write"
    fi
    ;;
```

```
3)
        echo "Input new file path"
        read p
        mv $fn $p
        ;;
4)
        if [ -w $fn ]
        then
            rm -rf $fn > /dev/null 2>&1
        else
        echo "sorry -- you don't have permission to delete"
        fi
        ;;
5)
        exit
        ;;
*)
        echo "please Enpnt valid option [1-5]"
esac
    echo -e "\n\n\t\tpress any key to continue...\c"
    read key
    done
```

* Loop control Statements :- Loops are used to execute the statements repeatedly.

Advantage :- Saves code development time and execution time of the program.

⇒ Types of Loops :-

1. while
2. until
3. for

1. while :-

Syntax :-

```
while [ expr ];
do
    statement(s)
done
```

* It executes the statements until the expression is true.

* write a script to print 1 to 5 numbers.

Ans :-

```
x=1
while [ $x -le 5 ];
do
    echo $x
    x=$((x+1))    (or)    x=`expr $x + 1`
done
```

* until :- Syntax :-

```
until [ expr ];
do
    statements
```

Eg①:-

```
x=1
until [ $x -gt 5 ];
do
    echo $x
    x=$((x+1))
done
```

②

```
x=1
while [ $x -le 5 ];
do
    y=1
    while [ $y -le $x ];
    do
        echo -e "$y \t \c"
        y=$(( y+1 ))
    done
    echo -e "\n"
    x=$(( x+1 ))
done
```

* for :-

Syntax :-

```
for variable in value1 value2 ... valuen
do
    statement(s)
done
```

Ex:-

```
for x in 1 2 3 4 5        [for list]
do echo $x
done
```

in {1..5} [sequence]

echo $x

Create file /list
hi hello welcome bye

t in `cat /list`

echo $val

specify new entire line as the field separator

IFS: Input field separator.

IFS='
'
for val in `cat /list`
do
  echo $val
done

ng while :-

while read val
do
  echo $val
done < /list

Display no. of files in /opt, /tmp, /var & not sub-directories.

for P in `cat /list`
do
  count=`ls -al $P | grep ^- | wc -l`
  echo "no. of files in $P : $count"