Redirection Operators

Saturday, December 30, 2023 2:52 PM

1. Input operation redirection
   " <"
2. Output redirection operation
   " > " to create a new file
   ">>" to append the content
3. Combining redirection operators
   " | "

Q] How to separate stdout and stderr
A] file decriptors
0: STDIN
1: STDOUT
2: STDERR

ls 1>succ.txt 2>err.txt
java -version 1>jav_ver.txt 2>jav_ver.txt
java -version 1>jav_ver.txt 2>&1
java -version &> jav_ver.txt

Commands to read a file content

Saturday, December 30, 2023 3:07 PM

1. **read a file by opening it**

vi filename
:set umber --> set line number

1. **read a file content without openning it**

A] using cat, less, more

ctrl + L/n --> clear screen
cat -n rohan.txt --> set line number
more rohan.txt --> here more command will only give you option to scroll down, you cannot scroll up.

1. **read a file content with conditions**

A] more, head ,tail, grep, awk, sed

more -n xyz.txt --> will read only specified lines. if you want to read more lines you can press enter.
more +n xyz.txt --> display from specified lines

head xyz.txt
head -2 xyz.txt

tail xyz.txt
tail -4 xyz.txt

1. **read file content for a given range of lines**

A] with head and tail combination
with awk command
with sed command

head -12 xyz.txt | tail -7
awk 'NR>=6 && NR<=12 {print}' xyz.txt
sed -n '6,12p' xyz.txt

"NR" --> line number

Grep command

Saturday, December 30, 2023 3:08 PM

Grep command is a filter command, it is used to search a string in a given file.

grep "rohan" abc.txt
grep "rohan" abc.txt def.txt
cat abc.txt | grep "rohan"

basic options: -i -w -v -o -n -c -A -B -C -r -l -h

-i --> avoid case sensitivity

-w --> to match a whole word

-v --> display lines other than the specified string

-o --> print only matched part of the matched line

-n --> display the matched line number

-no
-c --> count the number of lines

-A --> display n lines after the match
grep -A 2 "rohan" abc.txt

-B --> display n lines before the match
grep -B 2 "rohan" abc.txt

-C --> display n lines around the match
grep -C 2 "rohan" abc.txt

-r --> search for current dir and sub directory

-l --> display the file name

-h --> to hide file name

Advanced options:
-f -e -E

-f -->takes search string/pattern from a file, one per line and searches it in abc.txt
grep -f my_search abc.txt

-e --> to search multiple string
grep -e "rohan" -e "patkar" abc.txt

-E -->to search as given below
grep -E "rohan|patkar|shell|python" abc.txt

Rules to create a pattern:
^xyz --> matches for lines starting with xyz
grep -E "^Listen" /etc/httpd/conf --> line starting with Listen

xyz$ --> matches for lines ending with xyz

^$ --> matches for lines which are empty
grep -E "^$" abc.txt

\ --> to remove special purpose of any symbol
grep -E "\^" abc.txt

. --> matches any one character
grep -E "r…n" abc.txt

\. --> matches exactly with .
grep -E "\." abc.txt

\b --> matches empty string at edge of word

? --> preceding character is optional and will be matched at most once
grep -E "yf?" abc.txt

* --> preceding character will be matched 0 or more times
grep -E "yf*" abc.txt

+ --> preceding character will be matched one or more times

[xyz] --> matches for the lines which are having x y or z

[a-d]

^[abc] --> matches for lines which are starting with a/b/c

[^abc] --> matches for liine which are not starting with a/b/c

{N} --> preceding string matched exactly N time
grep -E "xf{4}" abc.txt

[[:alnum:]] --> alpha numeric characters

[[:alpha:]] --> alphabetic characters

[[:blank:]] -->blank characters: space and tab

[[:digit:]] --> digits: '0 1 2 3 4 5 6 7 8 9'

[[:lower:]] --> lower case letters

[[:upper:]] --> upper case letters

[[:space:]] --> tab, newline, vertical tab, space

Practice with grep command

ls -lrt | grep -E "^d"
ls -lrt | grep -E "^-"

Cut Command

Saturday, December 30, 2023 3:10 PM

To extract parts of each line from a file.
based on:
byte position
character position
field based on delimiter(by default delimiter is the tab)

- **Cut command syntax:**
  - cut [options] <position/fields/range> <input file>
  - cat file | cut [options] <position/fields/range> <input file>
  - Options: -b -c and -f Ranges:
    2 only second byte/character/filed
    2- second byte/character/filed to last
    -7 first to seven
    3,5 third and fifth

Cut command for Byte/Character Position:

- To cut out a section of a line by specifying a byte/character position use the -b/-c option.
- Syntax:
  - cut -b <position/range> file
  - cut -c <position/range> file
  - Position's: 3,5,10 - Range of Position's: 3-7, 6-10

- Ex: mytext.txt
- cut -b 2 mytext.txt
- cut -b 3,7 mytext.txt
- cut -b 5-9 mytext.txt
- cut -b 5- mytext.txt
- cut -b -7, 9 mytext.txt
- Use --complement to complement the output

Cut command for field Position:

To cut out a section of a line by specifying a field position use the -f option.
Assume fields are like columns, by default cut command will separates columns based on tab(delimiter).
If we want to use different filed separator use -d (delimiter).
Syntax:
cut -f <position's/range of position's> file
cut -f <position's/range of position'ss> [-d ':'] [--output-delimiter='**'] file
- -d is a delimiter like @ , : / etc….

Position's: 3,5,2
Range of Position's: 3-7, 6-10

Ex: mytext.txt ➢ cut -f 2 mytext.txt

cut -f 3,7 mytext.txt Use -s option with –f to Ignore the line that do not contain a delimiter
cut -f 5-9 mytext.txt
cut -f 5- mytext.txt
cut -f -7, 9 --output-delimiter=“ “ mytext.txt

cut -d ' ' -sf 1 abc.txt
cut -sf 1 abc.txt
cut -d ':' -f -4 /etc/passwd --output-delimiter="|'
cut -d ':' /etc/passwd --output-delimiter=" "

awk command

Saturday, December 30, 2023 10:01 PM

The awk command is a powerful method for processing or analyzing text or data files , which are organized by lines (rows or records) and columns(fields).
we can use awk as a linux command and also as a scripting language like bash shell scripting.

Simple awk command syntax:

awk [options] '[selection _criteria] {action }' input-file
cat input-file | awk [options] '[selection _criteria] {action }' input-file

Awk can take the following options:

-F fs To specify a field separator. (Default separator is tab and space)
-f file To specify a file that contains awk script.
-v var=value To declare a variable.

Selection criteria: pattern/condition
Action: It is a logic to perform action on each row/record
Action: Action is a logic to perform action on each record.

- Example: print $1 print first filed from each line

Some of the default variables for awk:
$0 → Entire file
$1 → First field from each line/record

Simple awk command syntax:
awk '{action}' input-file

$2 → Second field from each line/record
NR → It will print line or record number
NF → It will print number of filed from each line/record

httpd -v | awk -F '[/]' ' /version/ { print $4}'

httpd -v | awk -F '[/]' ' NR==1 { print $4}'
awk '{ print $1 }' abc.txt
awk '{print NR,$0 }' abc.txt
awk '{print NR,$0,NF }' abc.txt

	here NR=line number
	NF= number of field

tr command

Saturday, December 30, 2023 10:49 PM

```
tr '[:upper:]' '[:lower:]' < abc.txt
tr '[:lower:]' '[:upper:]' < abc.txt
tr 'l' 'i' < abc.txt
tr 'i' '_' < abc.txt
cat abc.txt | tr " " "_"
tr -d "," <abc.txt
tr -d "[,:]"
```

tr: short for translate
tr is useful to translate or delete given set of characters from the input.
Syntax:
tr [options] [SET1] [SET2] <inputFile
Some Command | tr [options] [SET1] [SET2]
No Option: For translation
examples for SET1/SET2: [:lower:], [:a-z:], [:upper:], [:A-Z:], [:digit:], [:0-9:], [:space:]

- -d : deletes given set of characters

tee command

Saturday, December 30, 2023 11:01 PM

date | tee abc.txt
uptime | tee abc.txt

date | tee -a abc.txt
uptime | tee -a abc.txt
here -a is used to append

-
- tee command is used to display the output and store that output into a file. (it does both the tasks simultaneously).
- it is useful to create logs for shell scripting
  syntax:
  command | tee output file.txt
  command | tee -a output file.txt

Advanced Usage of echo command

Sunday, December 31, 2023 1:37 AM

Advanced Usage of echo command
echo command is used to display string/message or variable value or command result. ➢ Simple syntax:
echo message/string
echo “message/string”
echo “message/string with some variable $xyz”
echo “message/string/$variable/$(command)”

➢ Advanced usage (to execute escape characters):
    ➢ echo -e “Message/String or variable”
➢ Escape Characters:
    \n New Line
    \t Horizantal Tab
    \v Vertical Tab
    \b Backspace
    \r Carriage Return etc…
    To display message in colors.
    echo -n “message/string/$variable/$(command)”

Multiline Block

Sunday, December 31, 2023 1:48 AM

Here doc is very useful for wiriting multiline block

cat << DELIMITER
LINE-1
LINE-2
LINE-3
LINE-4
DELIMITER

Note: Here DELIMITER can be any string
Heredoc is mostly used with the combination of cat command.
Display multi-lines using cat command.

We can also redirect this heredoc result into a file or as a input for another command.

```
cat << DELIMITER  | grep  Line-1
  Line-1
  Line-2
  Line-3
DELIMITER
```

Here string usage

Sunday, December 31, 2023 1:54 AM

```
echo ""hello this is rohan"
echo ""hello this is rohan" | tr [a-z] [A-Z]
tr [a-z] [A-Z] <<<"hello this is rohan"
docker -v | tr [a-z] [A-Z]
tr [a-z] [A-Z] <<<$(docker -v)
```

Writing Comment

Sunday, December 31, 2023 1:21 PM

```
#!/bin/bash
#single line comment can be done like this

<< DELIMITER
this is how
you can
comment
multiple lines
DELIMITER


: '
this is how
you can
comment
multiple lines
'

echo "this is inventory script"
```

Debugging a shell script

Sunday, December 31, 2023 3:00 PM

#set -n
#set -x
#set -e
#set -v

Debugging is determining the cause which fails the script.

We can go with set command and We have different options with set command.
Syntax:
set [options]
No Options: To list system defined variables

- set -n No Execution, Purely for syntax check.

set -x Prints the command before executing it in script
set -e Exit Script if any command fails

How Interpreter(Bash) executes script(s) ?
It reads and executes our code/script line by line.
Sometimes it throws an exception/error while executing code/script.
This is because of two type of errors.

Runtime Errors

Syntax Errors stops our script execution and run time errors don't stop our script.
Here: Debugging is simply, determining the cause which fails the script.

Exit status

Sunday, December 31, 2023 3:14 PM

Each linux command returns a status when it is executed.

we can display the exit status of a command with echo $?
x=$? (storing exit status of a command into a variable)
0 exit status means the command was successful without any errors.

A non-zero (1-255 values) exit status means command was failure.

example:
127 - command not found
1- command failed during execution
2- incorrect command usage etc …

Basic String Operations

Sunday, December 31, 2023 4:05 PM

Defining a string variable
x=shell / y="Shell scripting" / cmdOut=$(date)

Displaying the string variable value
echo $x / echo ${x}

Finding the length of a string
xLength=${#x}

Concatenation of strings
xyResult=xy

Convert Strings into lower/upper case
xU=${x^^}, yL=${y,,}

Replacing the part of the string using variable
newY=${y/Shell/Bash Shell} or we can also use sed command

Slicing the string/sub-string
${variable_name:start_position:length}

String operation on path

Sunday, December 31, 2023 6:50 PM

realpath : Converts each filename argument to an absolute pathname but it do not validate the path.

realpath abc.txt

- basename:

  Strips directory information

  Strips suffixes from file names

basename /home/ec2user/abc.txt

basename /home/ec2user/abc.txt .txt --> abc

mypath= /home/ec2user/abc.tar.gz

basename $mypath .tar.gz --> abc

dirname : It will delete any suffix beginning with the last slash character and return the result

dirname /home/ec2user/abc.txt --> /home/ec2user

Test command

Sunday, December 31, 2023 7:58 PM

It is a command to judge conditions.

test condition or [condition] or [[condition]] ▪ Note: [[ ]] works with bash/ksh/zsh shells. ▪ It will return exit status as 0 or 1. (echo $?)

0 -- Condition is true or test is successful

1 -- Condition is false or test is failed

How to make condition to work with test command ?

Comparison Operators

File Test Operators

Comparison Operators with test command ▪ 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

File test Operators with test command

[[-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

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

---

Advanced file test operators are listed below:

a : True if the file exists.
b : True if the file exists and is a block special file.
c : True if the file exists and is a character special file.
d : True if the file exists and is a directory.
e : True if the file exists.
f : True if the file exists and is a regular file.
g : True if the file exists and its SGID bit is set.
h : True if the file exists and is a symbolic link.
k : True if the file exists and its sticky bit is set.
p : True if the file exists and is a named pipe (FIFO).
r : True if the file exists and is readable.
s : True if the file exists and has a size greater than zero.
t : True if file descriptor is open and refers to a terminal.
u : True if the file exists and its SUID (set user ID) bit is set.
w : True if the file exists and is writable.
x : True if the file exists and is executable.
O : True if the file exists and is owned by the effective user ID.
G : True if the file exists and is owned by the effective group ID.
L : True if the file exists and is a symbolic link.
N : True if the file exists and has been modified since it was last read.
S : True if the file exists and is a socket.

==========

Command Chaining Operators

Sunday, December 31, 2023 8:17 PM

- This concept is useful to write simple and short shell scripts.
- Chaining of Linux commands means, combining several commands and make them execute based upon the behavior of operator used in between them.
- The different Command Chaining Operators are:
- Semi-colon Operator ;
- Logical AND Operators &&
- Logical OR Operator ||
- Logical AND-OR Operators && ||

Note:

- cmd1 ; cmd2 – Run cmd1 and then cmd2, regardless of the success or failure of cmd1
  ls;pwd

- cmd1 && cmd2 – Run cmd2 only if cmd1 succeeded

  which docker 2>&1 1>/dev/null && docker -v --> here both the success and error output are nullified

  ls && pwd && date

- cmd1 || cmd2 – Run cmd2 only if cmd1 failed
- cm1 && cmd2 || cmd3 – Run cmd2 if cm1 is success else run cmd3

Executing block of code using {}

Sunday, December 31, 2023 9:02 PM

```
#!/bin/bash

#{ ls ; pwd ; date ; }

#which docker && { echo "docker is installed" ; echo "docker version is $(docker -v) ; }

#which apache2 && { echo "apache is installed" ; echo "apache version is $apache2 -v" ; } || echo "apache is not installed"

which apache2 2>&1 1>/dev/null && { echo "apache is installed" ; echo "apache version is $apache2 -v" ; } || echo "apache is not installed"
```

---> here both the success and error output are nullified

If else condition

Sunday, December 31, 2023 9:43 PM

```
#!/bin/bash
<<<ifelse
if systemctl status docker 2>&1 1>/dev/null
then
        echo "docker is already running
else
        echo "starting docker service"
        systemctl start docker
        echo "successfully started docker service"
fi
ifelse



<<<logicalor
read -p "enter your confirmation to start docker: (say yes or no)" cnf

if [[ $cnf == "yes" || $cnf == "y" ]]
then
        echo "starting docker service"
        sudo systemctl start docker
else
        echo "skipping……."
fi
logicalor



if [[ $(id -u) -ne 0 ]]
then
        echo "you are not allowed to run this script"
        exit 1
fi
```

Handling command line argument

Sunday, December 31, 2023 10:38 PM

- There are two ways to provide inputs for a shell script
  - using read command
  - using command line arguments

```
#!/bin/bash
<<<service
#read -p "enter your service to execute your action on  it: " servicename
#read -p "enter your action to execute on your service: $servicename " action
sudo systemctl $action ${servicename}
service

<<<simplelogic
servicename=$1
action=$2
sudo systemctl $action ${servicename}
simplelogic


<<<simplelogic
if [[ $# -ne 2 ]]
then
        echo "hey adminplease run this script as follows"
        echo "usage: $0 <servicename> <actiontoexecuteonservice>"
        echo "valid actiontoexecuteonservice are: start stop restart status"
        exit 1
fi
simplelogic
```

at command

Monday, January 1, 2024 10:48 AM

Scheduling job with at command

- at command is very useful for scheduling one time table
- example:
  - shutdown system at the specified time
  - taking a one time backup
- syntac:
  - echo "bash backup.sh" | at 9:00 AM
    or
  - run first: at 9:00 AM then enter and give the cmd or script to run and press ctrl+D

atq --> list of jobs that are qued
atrm n --> to remove jobs

at now + 1 hour
at now + 30 minutes
at now + 1 week
at now + 2 weeks
at now + 1 year
at midnight

Crontab

Monday, January 1, 2024 10:55 AM

crontab -e --> to schedule a job
crontab -l --> to list the jobs (crontab -u user_name -1)
contab -r --> to remove jobs

0 9,21 * * * /hoem/ec2user/abc.sh

auto mail alert when ram gets low

Monday, January 1, 2024 11:18 AM

```
#!/bin/bash
To="rohan@gmail.com"
TH_L=400
free_ram=$(free -mt | grep -E "Total" | awk '{print $4}')

if [[ $free_ram -lt $Th _L ]]
then
        echo -e "server is running with low ram size. Available ram size is $free_ram" | /bin/mail -s "RAM
        INFO $(date)" $TO
fi


crontab -e
```

Array

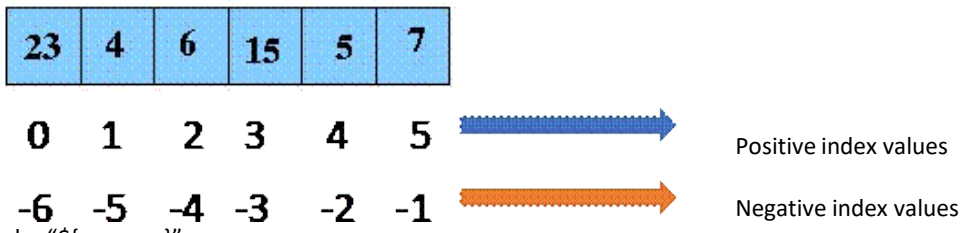
Monday, January 1, 2024 12:13 PM

Concepts of Arrays
What is an Array ?
How to define array ?
How to access Array Values ?
Different Types of Arrays
How to store the command output into an array ?
How to delete and update exiting array with new values ?
How to read array using read command ?
What is an Array and How to define or declare it ?
What is an array ?
An Array is the data structure of the bash shell, which is used to store multiple data's.
Simple array: myarray=(ls pwd date 2 5.6) #No limit for length of an array ➢ How to Define/declare an array ?

There are different ways to define an array in bash shell scripting.
Empty Array: myArray=()
mycmds=(ls pwd date 2 5.6)
myNewArray=(ls -lrt hostname -s)
myNewArray=("ls –lrt" "hostname –s")
declare -a NewArray
NewArray=(1 3 4 5 bash scripting)
How to access Array values/elements ?
Basically, Bash Shell Array is the zero-based Array (i.e., indexing start with 0)

Then what is an index ? ➢ myarray=(23 4 6 15 5 7)
myarray

echo "$myarray"



echo "${myarray}"
echo "${myarray[*]}" echo "${myarray[@]"
echo "${myarray[0]}" Prints first value
echo "${myarray[-1]}" Prints last value
Positive Index Values or Positive Indices Negative Index Values or Negative Indices

How to access Array values/elements ?...
echo "${myarray[*]:0}" Prints all the values starting from index-0 echo "${myarray[*]:1}" Prints all the values starting from index-1
echo "${myarray[*]:0:2}" Prints two values starting from index-o echo "${myarray[*]:1:2}" Prints two values starting from index-1
echo "${!myarray[*]}" → Prints index values of array
echo "${#myarray[*]}" Find the length (number elements) of array
We can also customize index numbers: ➢ newarray[5]="bash"
newarray[9]="shell scriting"

Or

newarray=([5]="bash" [9]="shell scripting")
Note: We can also take indices as strings and that array is called Associative Array.
Different Types of Arrays
We have two types of arrays in Bash Shell Scripting.

They are:
Index Based Arrays or Arrays ➢ Associative Arrays.

How to store the command output into an array ?
Storing the output of a command into array:
arraywithcmd=($(command))

How to delete and update an exiting array ?
Delete an array or even normal variable:
unset variable/arrayvariable
Updating an exiting array: ➢ myarray=(1,2,3)
myarray+=(4,5,6)

How to read an array using read command ?
Syntax:
read -a myarray
read –p “Enter your array” -a myarray

Associative array

Monday, January 1, 2024 12:29 PM

We already know about Normal arrays or Indexed Arrays or Index Based Arrays and for these arrays Index values or Indices are Numbers (Integer Number)

Associative arrays are the arrays with index values as strings.

Generally no need to declare normal arrays before using them but we have to declare Associative arrays before using them.

```
declare -A myassarray
```

Defining Associative Arrays:

```
myassarray=([name]="bash shell scripting" [version]=4.4)

Or

myassarray[name]="bash shell scripting"

myassarray[version]=4.4
```

Simply Associative Arrays are called key-value pair representation.

Associative Arrays should be declare before using them (declare –A myassarray).

echo "${indexarray}" → will print first value by default

echo "${myassarray}" → wont print any value

Indexed Array Indices are numbers

Associative Array Indices are strings

Loops

Monday, January 1, 2024 1:29 PM

```
#!/bin/bash
if [[ $# -ne 1 ]]
then
        echo "usage: $0 <any_path>"
        exit
fi


give path=$1
for each in $(ls $given_path)
#for each in httpd_info.sh httpd_ver_port.sh
do
        if [[ -x $each ]]
        then
                echo "$each is having execution permission"
        else
                echo "$each is not having execution permission"
        fi
done
```

for loop to install packages

Monday, January 1, 2024 5:37 PM

```
<<<forloop
#!/bin/bash
#installing multiple packages using for loop

if [[ $(id -u) -ne 0 ]]
then
        echo "please run from root user or with sudo priviliages"
        exit 1
fi

for each_pkg in vim httpd nginx
do
        if which $each_pkg &> /dev/null
        then
                echo "already $each_pkg is installed"
        else
                echo "installing $each_pkg……."
                yum install $each_pkg -y &> /dev/null
                if [[ $? -eq 0 ]]
                then
                        echo "successfully installed $each_pkg pkg"
                else
                        echo "unable to install $each_pkg"
                fi
        fi
done
forloop


<<<commandlineargument
#!/bin/bash
if [[ $@ -eq 0 ]]
then
        echo "usage: $0 pkg1 pkg2 ……."
        exit 1
fi

if [[ $(id -u) -ne 0 ]]
then
        echo "please run from root user or with sudo priviliages"
        exit 1
fi

for each_pkg in $@
do
        if which $each_pkg &> /dev/null
        then
```

```
                echo "already $each_pkg is installed"
        else
                echo "installing $each_pkg……."
                yum install $each_pkg -y &> /dev/null
                if [[ $? -eq 0 ]]
                then
                        echo "successfully installed $each_pkg pkg"
                else
                        echo "unable to install $each_pkg"
                fi
        fi
done
commandlineargument
```

While loop

Monday, January 1, 2024 8:23 PM

Infinity while loop:

- while true
  do
  statement/commands
  done

- while:
  do
  statements
  done

While loop with command:

- while command
  do
  statement/command
  done

- while [[3 -gt 5]]
  do
  statements
  done

different ways to use while loop:

- Reading a file content:

  while read line
  do
  statement/command
  done < file name

- reading command output:

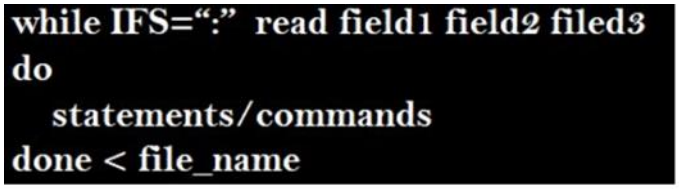
  command | while read line
  do
  statements/commands
  done

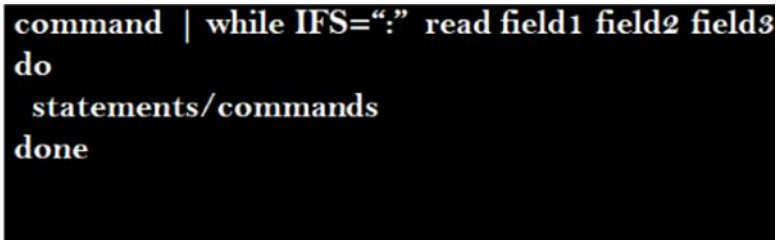
while loop with IFS

Monday, January 1, 2024 8:26 PM

scripting
IFS: Internal Field Separator, which is one of the shell or environment variable.
The IFS variable is used as a word separator (token) for the loops.
If we are going to change the default IFS, then it is a good practice to store the original IFS in a variable.

While syntax with IFS:





- while read f1 f2f f3 f4 f5
  do
  echo "$f2"
  done < server_info.txt

- while IFS="," read f1 f2f f3 f4 f5
  do
  echo "$f2"
  done < server_info.txt

- cat servers_info.txt | awk 'NR!=1 {print}' | while IFS="," read f1 f2f f3 f4 f5
  do
  echo "$f2"
  done

login to remote server from local server using ssh

Monday, January 1, 2024 7:08 PM

Using Passwrod:

ssh user_name@remote_ip

ssh remote_ip (here remote user name is same as local terminal user)

vi /etc/ssh/sshd_config (Make it; PasswordAuthentication yes in remote server)
==========

Using passwordless:

Step1: On local server generate keys using ssh-keygen
Step2: go to user_home/.ssh then here you will find two files
id_rsa (private key, it should be safe)
id_rsa.pub (public, this has to share with remote servers)
Step3: use below command to share public key with remtoe server, it will ask password
ssh-copy-id username@remote_server_ip
Step4: if step3 is success then use below command to login with remote server, it wont ask password now
ssh username@remote_server_ip

multiple command on multiple servers

Monday, January 1, 2024 8:01 PM

```
#!/bin/bash

<< my_com
echo "The date command output on the server: 100.26.187.33"
sshpass -f pass ssh -o StrictHostKeyChecking=No automation@100.26.187.33 "date"
echo "------------------------------------------------------"
echo "The uptime command output on the server: 100.26.187.33"
sshpass -f pass ssh -o StrictHostKeyChecking=No automation@100.26.187.33 "uptime"
echo "------------------------------------------------------"
echo "The free -m command output on the server: 100.26.187.33"
sshpass -f pass ssh -o StrictHostKeyChecking=No automation@100.26.187.33 "free -m"
echo "------------------------------------------------------"
my_com


for each_ser in $(cat remote_servers.txt)
do
  echo "Executing cmds on $each_ser"
  echo "============================"
  for each_cmd in date uptime "free -m"
  do
    echo "The $each_md command output on the server: $each_ser"
    #sshpass -f pass ssh -o StrictHostKeyChecking=No automation@$each_ser "$each_cmd"
    ssh -o StrictHostKeyChecking=No automation@$each_ser "$each_cmd"
    echo "------------------------------------------------------"
  done

done
```

diff cmd on diff ser

Monday, January 1, 2024 8:02 PM

servers_info.txt

```
----------
18.212.27.210  automation automation@123 date
18.212.185.2   tomcat tomcat123 whoami
```

execute_different_cmds_on_different_servers_with_differnt_users_and_passwords.sh

```
---------------------------------------------------------------------------
#!/bin/bash
while read ser user pass cmd1 cmd2
do
  echo "Executing $cmd1 on $ser with user as $user and password $pass"
  sshpass -p $pass ssh -n -o StrictHostKeyChecking=No $user@$ser "$cmd1"
  echo "Executing $cmd2 on $ser with user as $user and password $pass"
  sshpass -p $pass ssh -n -o StrictHostKeyChecking=No $user@$ser "$cmd2"
  echo "------------------------------------------------------"
done < servers_info.txt
```

Functions

Monday, January 1, 2024 9:06 PM

A function is a block of code that performs a specific task and which is reusable
Functions concept reduces the code length
Two was to define a function:

```
function function_name
{
        command/statements
}

function_name()
{
        command/statements
}
```

```
#!/bin/bash


read_inputs()
{
 read -p "Enter first num: " num1
 read -p "Enter second num: " num2
}

addition()
{
 sum=$((num1+num2))
 echo "The addition of $num1 and $num2 is: $sum"
}

subtraction()
{
 sub=$((num1-num2))
 echo "The sub of $Num1 and $num2 is: $sub"
}


read_inputs
addition
subtraction
```

Passing value to a function

Wednesday, January 3, 2024 8:25 AM

```
addition()
{
m=$1
n=$2
result=$((m+n))
echo "the addition of $m and $n is: $result"
}

x=6
y=2
addition $x $y --> this will be taken as command line argument

p=3
q=7
addition $p $q

addition 4 9 --> this will be taken as command line argument which will be assigned to $1 and $2
```

printf command

Wednesday, January 3, 2024 8:31 AM

- both echo and printf commands are used to display string or value of a variable
- the differeence is that echo sends a newline at the end of its output, there is no "send" an EOF in printf command.

- The advantage of printf command:

We can format the output
Useful in awk command/scripting as well

- Syntax:

printf “format\n” “arguments”

Syntax:
printf “format_with_modifiers\n” “arguments”
Note: format/format_with_modifiers is an optional and we can omit it.
printf “format\n” “arguments”

- Different types of formats are:

Format	Description
%d	For signed decimal numbers
%i	For signed decimal numbers
%u	For unsigned decimal numbers
%o	For unsigned octal numbers
%x	For unsigned hexadecimal numbers with lower case letters (a-f)
%X	For unsigned hexadecimal numbers with upper case letters (A-F)
%f	For floating point numbers
%s	For string
%%	For percent % symbol

printf command: format with modifiers

syntax:
printf "format wit h modifiers\n" "arguments"
differnet tyoes of modifiers are:

Format	Description
`N`	This specifies the width of the field for output.
`*`	This is the placeholder for the width.
`-`	To left align output in the field. (Default: Right align)
`0`	Pad result with leading 0s.
`+`	To put + sign before positive numbers and - sign for negative numbers.

awk command

Wednesday, January 3, 2024 9:21 AM

- the awk command is programing language, which requires no compiling and allows the iuser to use variables, numeric functions, string functions and logical operators.
- the awk command in unix is just like a scripting language which is used for text processing.
- awk is used like
    - a command
    - a scripting language

- awk command in unix is used to find and replace the text.

```
systemctl status httpd | awk 'NR==3 {print $2 }'
```

awk can take following options:

- -F to specify a field separator
- -v to declare a variable

```
cat /etc/passwd | awk -F : '{print $1}'
awk '{ print $1 }' demo.txt
awk '{ print $1, $2 }' demo.txt


awk 'BEGIN { print "==============begin block======="} /root/ { print $0 } END { print
"============end block============"}' /etc/passwd
```

awk script

Wednesday, January 3, 2024 10:45 AM

```
#!/bin/awk -f
BEGIN {
        print "hello world !!"
}
```

awk -f abc.awk

sed command

Sunday, January 7, 2024 9:06 AM

- sed command stands for stream editor
- sed command perform lots of function on file like
  - viewing file content
  - searching
  - find and replace
  - insertion or deletion
- sed command also supports regular expression which allows it perform complex pattern matching.
- Advantage of sed over vi editor
  - edit files without opening it.
- syntax
  - sed <options> commands <file>

sed -n 'p' abc.txt
sed -n '3p' abc.txt
sed -n '$p' abc.txt --> -n = suppress the default print output
$ = to print the last line
sed -n '3,10p' abc.txt
sed -n '1`3p' abc.txt
sed '49d' abc.txt --> print except 49th line
sed '4,$d' abc.txt --> delete 4 th to last line
sed '10,+12d' abc.txt --> start from 10th line and after 12 lines delete remaining lines.
sed -n '/echo/p' abc.txt --> print lines having echo word
sed -n -e '/echo/p' -e '/bin/p' abc.txt --> -e is used for extra option (execute extra commands)

x="echo"
se_w= "echo"
sed -n "/$se_w/p" abc.txt -->use "" when you want to execute variable value.

sed '10,$!d' abc.txt
sed -i.back '/echo/!d' abc.txt

sed 's/root/udemy/' abc.txt --> replace first word
sed 's/root/udemy/2' abc.txt --> replace 2nd occurred word
sed 's/root/udemy/g' abc.txt --> replace all words
sed -i.back's/root/udemy/g' abc.txt --> replace all words permanently

sed '/devops/s/bash/ksh/' abc.txt

- **Insertion and deletion using sed command**
  - -a to add after the line
  - -i to add before the line
  - -d to delete

cat abc.txt
sed '1i new line added' abc.txt
sed -i '1i new line added' abc.txt

sed -a '1i new line added' abc.txt
sed -i '3d' abc.txt

- **Regex command**
    - \s\t. * \+ \? and \

\s --> matches for space
sed -n '/\/p' abc.txt
sed -n '/\srohan/p' abc.txt -->rohan starting with space

\ --> escape character
sed -n '/\\s/p' abc.txt

\t --> matches for tab
sed -n '/\t/p' abc.txt

. --> matches any character, excluding newlline
sed -n '/roh.n/p' abc.txt --> print words having rohaXn

* --> matches a sequence of previous character
sed -n '/this*/p' abc.txt

\+ --> as *, but matches one or more

\? --> as * but matches zero or one time only

- **^ and $ characters**

^ charac
sed -n '/^put/p' abc.txt --> matches put word at the starting of the line

$
sed -n '/pot$/' abc.txt

sed '/^$/d' abc.txt --> delete empty line (start with nothing end with nothing)

- **special characters [] {} and ()**

[] matches any single character in list
{} matches for required number of repetitions
eg \{i\}
\{I,j\}
\{I,\}
() search for zero or more whole sequence.

sed -n '/this\{3\}/p' abc.txt --> has s 3 times