Intro to Shell Scripting

ISS Tutorial-1

1. Basic Shell Commands

1.1. Directory Navigation

- `pwd`: prints the current working directory
- `cd`: changes the current directory
  - `cd /path/to/directory`: changes the current directory to the specified directory
  - `cd ..`: moves up one directory
  - `cd ~`: moves to the home directory

1.2. File and Directory Management

- `ls`: lists the files and directories in the current directory
  - `ls -a`: lists all files, including hidden ones
  - `ls -l`: lists files in long format
- `mkdir`: creates a new directory
- `touch`: creates a new file
- `rm`: removes a file or directory
  - `rm -r`: removes a directory and its contents
- `cp`: copies a file or directory
- `mv`: moves a file or directory
- `chmod`: changes the permissions of a file or directory
- `chown`: changes the owner of a file or directory

1.3. Input and Output

- `echo`: prints a message to the screen
- `cat`: prints the contents of a file to the screen
- `less`: allows you to view the contents of a file one page at a time
- `head`: prints the first few lines of a file
- `tail`: prints the last few lines of a file

1.4. Other Useful Commands

- `grep`: it is used for searching for patterns in files. It's a powerful tool that can search for patterns using regular expressions, and can also be used to search recursively in directories.
    - Example: The following command will search for the word "pattern" in `file.txt`.

```
grep "pattern" file.txt
```

- `find`: it is used for searching for files and directories based on various criteria. It can search recursively in directories, and can search based on file name, size, modification time, and many other criteria.
    - Example: The following command will search for all files in the directory `/path/to/directory` with the extension .txt. The -type f option specifies that only files should be returned (not directories).

```
find /path/to/directory -name "*.txt" -type f
```

- `awk`: it is used for processing and manipulating text files. It uses a pattern-action syntax, where the pattern selects which lines to process and the action defines what to do with those lines.
    - Example: The following command will print the first column of `file.txt`.

```
awk '{print $1}' file.txt
```

- `sed`: it is also a tool for processing and manipulating text files, with a focus on modifying the text directly in the file.
    - Example: The following command will replace all instances of "foo" with "bar" in `file.txt`.

```
sed 's/foo/bar/g' file.txt
```

- `cut`: it extracts columns from text files.
    - Example: The following command will extract the first and third columns from `file.csv`, assuming that the columns are separated by commas. The `-d` option specifies the delimiter and the `-f` option specifies the fields to extract.

```
cut -d ',' -f 1,3 file.csv
```

- `ps`: lists the running processes on the system
- `kill`: terminates a process
- `tar`: creates or extracts a tar archive

1.5. `man` command

This is the most important command that you should know how to use. The `man` command in Linux is a utility that is used to display the manual pages for other commands or programs in the terminal. These

man pages provide detailed information about the usage and options available for a particular command or program. So, read the man pages before approaching others for help.

- The basic syntax for the `man` command is:

```
man [option(s)] [command or program name]
```

- Examples:
  - `man ls` displays the man page for the `ls` command.
  - `man man` displays the man page for the `man` command.
  - To search for a specific keyword in the manual pages, you can use the `-k` option followed by the keyword. The following command will display a list of all the manual pages related to the keyword "network".

    ```
    man -k network
    ```

  - To display the manual page for a specific section of the manual, you can use the section number followed by the command or program name. The following command displays the man page for the `printf` function in C. The number `3` refers to the section of the manual that contains information about C library functions.

    ```
    man 3 printf
    ```

2. File Permissions

File permissions are a way to control who can access or modify a file on a computer system.

Every file on a Unix-like system (such as Linux or macOS) has a set of three permission settings for three types of users: the owner of the file, the members of the file's group, and all other users on the system. These permissions are represented by three characters for each user type: read (r), write (w), and execute (x).

The read permission (represented by the letter r) allows a user to view the contents of a file. The write permission (represented by the letter w) allows a user to modify the contents of a file. The execute permission (represented by the letter x) allows a user to execute a file as a program or script.

The three permission settings for each user type can be combined into a three-digit binary number, with each digit representing the permission for each type of user. For example, the permission settings `rw-r--r--` mean that the owner of the file has read and write permissions, while members of the file's group and all other users only have read permissions.

File permissions of directories start with `d`. For example, the permission `drwxr-x---` means its a directory with read, write, and execute permissions for the owner, read and execute permissions for the group, and no permissions for everyone else.

3. Environment Variables

An environment variable is a named value that is stored in the computer's environment and can be accessed by any program that runs in that environment. Some common examples of environment variables include the `PATH` variable, which stores the directories that the system searches when looking for executable files, and the `HOME` variable, which stores the user's home directory.

Environment variables are useful because they allow programs and scripts to share information without having to pass it directly between them. For example, a script might use an environment variable to store the location of a file that is needed by another program. By using an environment variable, the script can easily pass the file location to the other program without having to hard-code it into the script.

The `env` command displays the current environment variables such as `NAME`, `USER`, `PATH`, `LANG`, `PWD`, etc, and the `export` command sets a new environment variable. For example, the following command sets a new environment variable named MYVAR with the value "hello":

```
export MYVAR=hello
```

4. Input-Output Variables

A fundamental aspect of working with Shell scripts is working with input, output, and returning those results. When we execute the script given below, we get to enter project code and we get the value echoed (printed) on screen.

```
#!/bin/bash

# continue after reading 4 characters
read -n 4 -p "Enter Input: " input
echo "Input received is " $input
```

Variables are case sensitive in BASH. To unset a value associated with the variable, use UNSET keyword. By default, all the declared variables with values will be un set as soon as we close the BASH automatically. However, if we wish to set the value permanently made available every time you login to BASH, go to `.bashrc` file, which will be present in your home directory, to declare the variable and set value to it. `.bashrc` is the configuration file for BASH (it is also a shell script) which will be sourced whenever you open BASH.

Few examples for variables:

```
pwd # Current/Previous working directory
echo “Current Usr:” $USER # to print the user logged in

# A way to declare and call the integer type variables.
declare -i intvar
```

```
intvar=345
echo $intvar

# A way to declare and call the string variables
declare -l rovar=”Hyderabad”
echo $rovar

echo ${#rovar} # gives the count of elements in the value of rovar
```

5. Passing and Using Arguments

`$1`, `$2`, and so on are variables that are used to represent the arguments passed when executing the script. For example, when executing the script given below (assume the file name is `args.sh`), values need to be passed as `./args.sh 31 27`.

```
#!/bin/bash
if test "$1" = ""
    then
        echo "No first value supplied"
        exit
fi
if test "$2" - ""
    then
        echo "No second value supplied"
        exit
fi
echo "Sum of values:" $1 + $2 = '$1'+'$2'
```

6. Input-Output Redirection

- `>`: redirects output to a file
- `<`: redirects input from a file
- `>>`: appends output to a file
- `tee`: move the content of one command to another command
- `2>`: redirects error output to a file

The following example explains I/O rediection and piping.

```
#!/bin/bash

# assume that a file `labs.txt` exists

# print the data as output on Standard Output i.e. Screen
```

```
cat labs.txt > /dev/stdout

# print the data in labs.txt in sorted fashion
sort < labs.txt

# find all the strings/elements which has “t”
grep -i “t” < labs.txt

# find files in <foldername> where there are more than 800 characters
find /<foldername>/ -size +800c
```

7. Piping

- `|`: connects the output of one command to the input of another command
- `command1 | command2`: sends the output of `command1` as input to `command2`

Examples:

- `ls | wc -l`: takes the output of first command, passes it to second command and returns result
- `wc -l names.txt | tee -a file2.txt`: file2.txt will contain the output of first command.

8. Formatting Output Variables

The examples below show how output variables can be formatted.

```
echo "Kevin said "Hello World""
echo "Kevin said \"Hello World\""
echo "Stock Price is $500"
echo "Stock Price is \$500"

date
date +'%Y-%m-%d'

numvar=5.5
echo $numvar
printf "%f\n" $numvar
printf "%d\n" $numvar
```

- Output:

```
Kevin said Hello World
Kevin said "Hello World"
Stock Price is 500
Stock Price is $500
```

```
Thu Mar 9 02:57:24 AM IST 2023
2018-04-09
5.5
5.500000
5.500
5
```

9. Math Expressions

In a Shell script, variables are by default treated as strings but not as numbers. This creates challenges for performing the math operation in shell scripts. However, there are few good commands which help us perform arithmetic operations in Shell.

- `let`: It helps us perform arithmetic operation through command line. Example:

```
#!/bin/bash
#Basic Arithmetic using Let command
let a=15+20
echo "a= 15+20 =" $a #35
let "b=29*20"
echo "b= 29*20 =" $a #580
let a++
echo "a++ =" $a #36
let "x=$1+30"
echo "x= $1+30 =" $x #30 + first command line argument
let u=16/4
echo "u= 16/4 =" $u #4
let y=4/16
echo "y =4/16 =" $y #ceil the value to smallest number
```

- `expr`: – It is like let command. However, instead of saving the result to a variable it prints the answer. Unlike let command you don't need to enclose the expression in quotes. You must provide spaces between the items of the expression. It is also common to use expr within command substitution to save the output into a variable. Example:

```
#!/bin/bash
# expr with space does give the value as output
expr 23 + 29 #52
# expr with no spaces just prints the string
expr 23+29 #23+29
# expr with double quotes just prints the string
expr "23+29" #23+29
# expr with escape character (backslash) will give us multiply operator.
# '*' directly does work here.
expr 5 \* $1 # prints based on the argument passed during execution
```

```
# we get syntax error if we give * directly
expr 5 * $1 #expr: syntax error
# modulus operation - remainder when the first item is divided by second
expr 21 % 2 #1
#expr with a substitute in order to display it in a variable
a=$( expr 43 - 5 )
echo $a #38
```

- Double Parenthesis `(( ))`: – It is a way to handle arithmetic operations by passing the expression within the parenthesis. Examples:

```
#!/bin/bash
a=$(( 21 + 26 ))
echo $a #47
c=$(( 49-3))
echo $c #46
b=$(( a * 12))
echo $b #564
x=$(( $a / 2 ))
echo $x #23
(( c++ ))
echo $c #47
#adding 3 values to c
(( c += 3 ))
echo $c #50
```

- `bc` (Basic Calculator): By just running `bc` command, the basic calculator will be opened in interactive mode. We can run the `bc` command in non-interactive mode using | symbol. bc helps us to handle the floatingpoint value-based calculations. Example:

```
echo "5*2" | bc # output will be 10
```

10. If and Else

- The script given below shows how to use `if` and `else` statements.

```
#!/bin/bash
if test $USER != “root”
#to open if. Used $USER env variable to check user
then
    echo “You are not root”
else # to open else
    echo “You are root”
fi #close the if
```

- `test` command: This command checks file type and compare values on command line and returns the success flag based on true or false condition. Examples:

```
test 27 -gt 2 && echo “Yes” # Returns Yes, as 27 is greater than 2
test 27 -lt 2 && echo “Yes” # Returns nothing as we have not defined false condition
test 27 -lt 2 && echo “Yes” || echo “No” # Returns No, as 27 is not less than 2
test 25 -eq 25 && echo “Yes” || echo “No” # Returns Yes, as 25 is equal to 25
test 15 -ne 10 && echo Yes || echo No # Returns Yes, as 15 is not equal to 10

test -f /etc/resolv.conf && echo "File /etc/resolv.conf found." ||
echo "File /etc/resolv.conf not found."
# Returns relevant echo based on the presence of respective file
```

11. Loops

- `for`: loops over a set of values
- `while`: loops while a certain condition is true
- `until`: loops until a certain condition is true

11.1. For loop

- The script given below is an example to demo a `for` loop with conduction using `in` keyword to list of `.sh` files in each sub directory based on the total size of the file. The below script will print the size of all the .sh files in a list and then gives us the sum of the total size.

```
#!/bin/bash

totalsize=0
currentfilesize=0

#currentfile is a temporary variable used for counter
for currentfile in /scripts/*.sh
do
    currentsize=`ls -l $currentfile | tr -s " " | cut -f5 -d " "`
    #Gets file size and cuts it with space delimiter
    #tr command to squeeze the space on the file name spaces
    let totalsize=$totalsize+$currentsize
    #calculate the totalsize
    echo $currentsize
done
```

```
echo “Total space used by Script files is:” $totalsize
```

- Another example for using `for` loop, but this time using arrays:

```
#!/bin/bash
cities=("Chennai", "Hyderabad")
for ((i=0;i<${#cities[@]};i++))
do
    echo ${cities[$i]}
done
```

11.2. While loop

- Example: The script given below waits for the input choice from the user. Until the input is given as 1 the loop is not broken.

```
#!/bin/bash
choice=”0”
while (( $choice != “1”))
do
    echo “Please select and Option”
    echo “1 - Exit”
    read choice
done
```

Note:

- `break` and `continue` can be used to break out of the loops and contiue the loop respectively.

12. Case Statements

- Case statements help us to test a value of something for multiple possible items. Instead of using bunch of if-else, we can use Case statements.
- Example:

```
#!/bin/bash
clear
echo ”City Type”
echo
read -p “Enter your City:” city
case $city in
    “Hyderabad”) city_type =”Tier 2”;;
    # Pipe Symbol is a OR command
    “Bengaluru” | “Mumbai”)city_type=”Tier 1”;;
    “New Delhi”) city_type=”Capital”;;
    # here star is to consider anything else
```

```
    *)clear; echo "Invalid City $city";exit;;
esac #case in reverse to close the case statement
clear
echo "City Type for $city has been set to $city_type"
echo
```

13. Functions

Function is a code snippet which is defined and used to re-run or re-use a logic.

- `function_name() { commands }`: defines a function
- `function_name argument1 argument2`: calls a function with arguments
- `$1`, `$2`, etc.: refer to the first, second, etc. arguments of a function

Below script contains two functions `userinfo()` and `ipinfo()` which are called while running the script.

```
function userinfo()
{
 echo "Current username:" $USER
 echo "User Home directory path:" $HOME
}
function ipinfo()
{
 ip_var=`hostname -I | awk '{ print $1}'`
 echo "IP ADDRESS:" $ip_var
}
userinfo
ipinfo
```