SHREE UTTAR GUJARAT BCA COLLEGE

Unit 3: Shell Scripting in Linux

3.1 Creating and Executing Shell Scripts (nano, vi, ./script.sh)

3.2 Shell Metacharacters and Operators

3.2.1 Filename Expansion (wildcards: *,?, [])

3.2.2 Input/output Redirection (>, >>, <)

3.2.3 Pipes (|)

3.2.4 Command Substitution ($(...), ...)

3.3 Control Flow Structures (if-else, case, for, while, until)

3.4 Logical Operators (&&, ||, !)

3.5 test and [ ] command for Condition Testing (file, numeric, string)

3.6 Arithmetic Operations (expr, $(( )))

3.1 Creating and Executing Shell Scripts (nano, vi, ./script.sh)

What is a Shell Script?

A **shell script** is a file containing a list of commands that are executed by the **shell** (such as Bash). It's used to automate tasks.

Steps: How to Create and Run Shell Scripts

STEP 1: Open Terminal

Press Ctrl + Alt + T in Linux to open the terminal.

STEP 2: Create a New File Using an Editor

A. Using nano editor (Easy)

1. Type the following command:

   nano myscript.sh

2. Write this script inside nano:

   echo "Hello from Nano Script!"

3. Save the file:
   - Press Ctrl + O → then Enter to save
   - Press Ctrl + X to exit nano

B. Using vi editor (Advanced)

1. Type the following command:

   Vi myscript.sh

2. Press i to enter **Insert Mode**
3. Write this script:

   echo "Hello from VI Script!"

4. Save and Exit:
   - Press Esc to exit insert mode
   - Type :wq and press Enter (write and quit)

STEP 3: Make the Script Executable

Before running a script, give it **execute permission**:

chmod +x myscript.sh

This allows the script to be executed like a program.

STEP 4: Run the Script

Now you can run your shell script using:

./myscript.sh

Output will be:

Hello from Nano Script!

Example Shell Script

```
echo "Welcome to Shell Scripting"
date
whoami
```

Output:

```
Welcome to Shell Scripting
Tue Jul 29 10:00:00 IST 2025
your_username
```

Editor	Shortcut to Save	Shortcut to Exit
nano	Ctrl + O	Ctrl + X
vi	:w	:q
vi	:wq	Save + Exit
vi	i	Insert mode

Summary

Command	Purpose
nano file.sh	Create/Edit using Nano
vi file.sh	Create/Edit using Vi
chmod +x file.sh	Make script executable
./file.sh	Run the shell script

3.2.1 Filename Expansion (wildcards: *, ?, [])

Metacharacter	Function	Example	Description
*	Wildcard	ls *.txt	Matches any number of characters; lists all .txt files.
?	Wildcard	ls file?.txt	Matches any single character; lists files like file1.txt, fileA.txt.
[ ]	Character class	ls file[1-3].txt	Matches any one character within the brackets; lists file1.txt, file2.txt, file3.txt.
{ }	Brace expansion	echo file{1,2,3}.txt	Expands to file1.txt file2.txt file3.txt.
~	Home directory	cd ~	Represents the current user's home directory.
$	Variable substitution	echo $HOME	Expands to the value of the HOME environment variable.
;	Command separator	cd /; ls	Executes multiple commands sequentially.
&&	Logical AND	make && make install	Executes the second command only if the first succeeds.
`		`	Logical OR
&	Background execution	sleep 60 &	Runs the command in the background.
`	`	Pipe	`ls
>	Output redirection	ls > files.txt	Redirects standard output to a file, overwriting it.
>>	Output append	echo "Hello" >> file.txt	Appends output to the end of a file.
<	Input redirection	sort < file.txt	Takes input from a file instead of standard input.
2>	Error redirection	command 2> error.log	Redirects standard error to a file.
&>	Output and error	command &>	Redirects both standard output

Metacharacter	Function	Example	Description
	redirection	all.log	and error to a file.
()	Subshell grouping	(cd /tmp; ls)	Executes commands in a subshell.
{}	Command grouping	{ cd /tmp; ls; }	Groups commands in the current shell.
`command`	Command substitution	echo `date`	Replaces with the output of the command.
"	Double quotes	echo "Hello $USER"	Preserves spaces and allows variable expansion.
'	Single quotes	echo 'Hello $USER'	Preserves literal value; no variable expansion.
\	Escape character	echo \$HOME	Escapes the special meaning of the next character.

3.2.1 Filename Expansion (wildcards: *, ?, [])

Filename expansion, also called **globbing**, allows users to match multiple filenames using *wildcards*. It is very useful when working with multiple files in the Linux shell.

1. Asterisk (*)

- Matches **zero or more characters**.
- Example:

```
ls *.txt
```

This lists all files ending with .txt (like file.txt, notes.txt, a.txt).

2. Question Mark (?)

- Matches **exactly one character**.

- Example:

ls file?.txt

This matches files like file1.txt, fileA.txt, but **not**file10.txt.

3. Square Brackets ([ ])

- Matches **one character** from a set or range.
- Examples:
  - ls file[123].txt — matches file1.txt, file2.txt, file3.txt.
  - ls file[a-c].txt — matches filea.txt, fileb.txt, filec.txt.
  - ls file[!a-z].txt — matches files **not** containing lowercase letters in that position.

Examples of Usage

```
ls *.sh          # All shell scripts
ls data??.csv    # Files like data01.csv, data99.csv
ls report[1-3].pdf # report1.pdf, report2.pdf, report3.pdf
```

3.2.2 Input/output Redirection (>, >>, <)

Input/output redirection in Linux allows you to change the standard input/output behavior of commands. By default:

- **Standard Input (stdin)** comes from the keyboard.
- **Standard Output (stdout)** goes to the screen.

Redirection lets you send or receive data from files instead.

1. Output Redirection: >

- Sends output **to a file**, **overwriting** it if it already exists.
- **Syntax:**

  command> filename

- **Example:**

echo "Hello" > message.txt

This creates (or overwrites) message.txt with Hello.

2. Append Output: >>

- Appends output **to the end** of an existing file.
- **Syntax:**

  command>> filename

- **Example:**

  echo "World" >> message.txt

  Adds World to the end of message.txt without deleting the original content.

3. Input Redirection: <

- Takes input **from a file** instead of the keyboard.
- **Syntax:**

  command< filename

- **Example:**

  cat< message.txt

  Displays the contents of message.txt.

Examples Summary

Command	Meaning
ls > files.txt	Save directory list to a file
echo "Hi" >> greet.txt	Add text to the end of a file
wc< file.txt	Count lines/words/chars from a file

3.2.3 Pipes (|)

The **pipe (|)** is a powerful feature in Linux that lets you **connect the output of one command directly as input to another command** — creating a chain or "pipeline".

Syntax:

command1 | command2

- command1 generates output.
- command2 takes that output as its input.

How It Works:

- The **stdout (standard output)** of the first command becomes the **stdin (standard input)** of the next.

Examples of Pipes

1. **View long file content page by page:**

   cat filename.txt | less

   Same as:

   less filename.txt

2. **Count files in a directory:**

   ls | wc -l

   - ls: lists files
   - wc -l: counts lines (each file is one line)

3. **Find a specific word in a file:**

cat file.txt | grep "hello"

- grep filters lines that contain "hello".

4. **Sort and remove duplicates:**

cat names.txt | sort | uniq

3.2.4 Command Substitution ($(...), ...)

Command Substitution allows the output of one command to be used as an argument in another command.
It's very useful when you want to **capture the result of a command** and pass it as input to another.

1. Modern Syntax: $(...)

This is the **preferred and more readable** form of command substitution.

Example:

echo "Today is: $(date)"

- date runs first, and its output replaces $(date).

example:

files=$(ls *.txt)
echo "Text files are: $files"

2. Older Syntax: `...` (Backticks)

Same as $(...), but harder to read and **not recommended** for nested commands.

Example:

echo "Today is: `date`"

Nested Commands (use $(...))

```
echo "Line count: $(wc -l < $(find . -name 'file1.txt'))"
```

Backticks don’t support nesting easily:

```
# Hard to read:
echo "Line count: `wc -l < \`find . -name file1.txt\``"
```

3.3 Control Flow Structures (if-else, case, for, while, until)

Control flow structures help you write **decision-making** and **repetitive** logic in shell scripts. Below are the commonly used structures:

1. if–else Statement

Used for conditional execution of commands.

Syntax:

```
if [ condition ]
then
commands
elif [ another_condition ]
then
commands
else
commands
fi
```

Example:

```
num=5
if [ $num -gt 0 ]
then
echo"Positive number"
else
echo"Zero or negative"
fi
```

2. case Statement

Used as a multi-way decision structure (like switch-case in other languages).

Syntax:

```
case$variablein
  pattern1)
commands ;;
  pattern2)
commands ;;
  *)
default commands ;;
esac
```

Example:

```
echo "Enter a number (1-7):"

read day


case $day in

  1)

echo "Monday"

    ;;

  2)

echo "Tuesday"

    ;;

  3)

echo "Wednesday"

    ;;
```

```
  4)
echo "Thursday"
    ;;
  5)
echo "Friday"
    ;;
  6)
echo "Saturday"
    ;;
  7)
echo "Sunday"
    ;;
  *)
echo "Invalid input! Please enter a number between 1 and 7."
    ;;
esac
```

3. for Loop

Repeats a block of code for each item in a list.

Syntax:

```
forvarin list
do
commands
done
```

Example:

```
for i in 1 2 3 4 5

do

echo "Number: $i"

done
```

4. while Loop

Runs as long as the condition is **true**.

Syntax:

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

Example:

```
i=1
while [ $i -le 5 ]
do
echo"Number: $i"
  i=$((i+1))
done
```

5. until Loop

Runs as long as the condition is **false**.

Syntax:

until [condition]
do
commands
done

Example:

```
i=1

while [ $i -le 5 ]

do

echo "Number: $i"

  i=$((i + 1))

done
```


3.4 Logical Operators (&&, ||,!)

Logical operators are used to control the **execution flow** based on the **success or failure** of commands or conditions in shell scripting.

1. AND Operator: &&

- Executes the second command **only if** the first command **succeeds (exit status 0)**.

Syntax:

command1 && command2

Example:

Mkdir mydir&&cd mydir

- cdmydir runs **only if**mkdirmydir is successful.

2. OR Operator: ||

- Executes the second command **only if** the first command **fails (exit status non-zero)**.

Syntax:

command1 || command2

Example:

rm myfile.txt || echo"File not found"

- If rm fails (file doesn't exist), it prints the message.

3. NOT Operator: !

- **Negates** the result of a condition or command.
- Useful in [ ] tests and control structures.

Syntax:

if![-f myfile.txt]; then
echo"File does not exist"
fi

Operator	Description	Example
&&	Executes next if previous command succeeds	mkdirdir&& cd dir
!	Negates a condition or command	if ! [-f file]; then ...

3.5 test and [ ] command for Condition Testing (file, numeric, string)

The test command (or its shorthand [ ]) is used to evaluate **conditions** in shell scripts — especially inside if, while, and other control structures.

Basic Syntax

You can use either:

test condition

Both are functionally the same, but [ ] is more commonly used.

Always put spaces around brackets: [$a -eq 5] is correct, [$a- eq5] is wrong.

1. File Tests

Expression	Description
-e file	File exists
-f file	Regular file
-d file	Directory
-r file	Readable
-w file	Writable
-x file	Executable

Example:

```
if [ -d /home/user ]
then
echo "Directory exists"
fi
```

2. Numeric Comparisons

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-lt	Less than
-ge	Greater or equal

Operator	Meaning
-le	Less or equal

Example:

```
a=10
if [ $a -gt 5 ]
then
echo "a is greater than 5"
fi
```

3. String Comparisons

Operator	Meaning
= or ==	Equal
!=	Not equal
-z	String is empty
-n	String is not empty

Example:

```
name="Alice"
if [ "$name" = "Alice" ]
then
echo "Welcome Alice"
fi
```

Logical Operators Inside [ ]

You can use:

- && (and)
- || (or)
- ! (not)

Example:

```
if [ $age -gt 18 ] && [ $age -lt 60 ]
then
echo "Working age"
fi
```

Type	Test Example	Meaning
File	[-f file.txt]	file exists and is a file
Numeric	[$a -le 100]	a ≤ 100
String	["$str" != ""]	string is not empty

3.6 Arithmetic Operations (expr, $(( )))

1. Using expr(Older Method)

The expr command evaluates expressions **with spaces between operators and operands**.

Syntax:

expr operand operator operand

Example:

```
a=10
b=5
sum=$(expr$a + $b)
echo"Sum is: $sum"
```

Note: You **must have spaces** around operators.
Wrong: expr $a+$b
Correct: expr $a + $b

2. Using $(( )) (Modern and Preferred)

$(( )) is shell arithmetic expansion — simpler, faster, and more readable.

Syntax:

result=$((expression))

Example:

```
a=10
b=5
sum=$((a + b))
echo"Sum is: $sum"
```

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)
**	Power (in some shells)

Method	Syntax	Notes
expr	expr $a + $b	Requires spacing
$(( ))	$((a + b))	Recommended