

Contents

What is Shell Script in linux.	3
Why do we use shell script	3
What is a Variable?	3
Types of Variables in Shell Script	3
1 User Defined Variable	3
2 System Defined Variable	4
3 Built-in Variable	4
Naming Convention in Shell Script	5
Rules for Variable Naming	5
What is an Array?	6
Types of Array in Linux (Shell Script)	6
1Indexed Array	6
Example:- 1	6
Next Program:-- Indexed Array Example – 2	7
Next Program :- Indexed Array Example – 3	7
2 :-- Associative Array	8
Example 1	8
Next Program - Associative Array Example – 2	8
Difference between Indexed Array and Associative Array	9
Explanation (Easy in Hindi):	9
Operator in Linux	10
Types of Operators in Linux:.....	10
1 Arithmetic Operators (Linux)	10
PROGRAM 1 :----	11
Program 2 :- Arithmetic Operators (By User)	11
2 Relational Operators (Linux)	12
4 String Operators (Linux)	16
Function in Shell Script	19
1 No Argument – No Return	19

2 With Argument – No Return.....	19
3 No Argument – With Return.....	20
4 With Argument – With Return.....	21

What is Shell Script in linux.

A shell script is a set of commands written in a file.

These commands are read and executed by the Bash shell.

The program executes the commands line by line.

By storing commands in a shell script, it becomes easy to execute them again and again.

Why do we use shell script

Shell scripting is used to automate repetitive tasks in Linux.

It helps in tasks such as creating backups, monitoring system resources, managing user accounts, and running multiple commands automatically.

Shell scripting saves time, reduces manual effort, and makes system administration easy.

What is a Variable?

A variable is a named memory location used to store data in a shell script.

The value of a variable can change during program execution.

&

Variables are used to store numbers, strings, or command output so that they can be used again and again in a script.

Types of Variables in Shell Script

There are three types of variables in shell scripting:

1 User Defined Variable

User defined variables are variables created by the user to store values.

Example:

```
name=Akhilesh  
age=20  
echo $name  
echo $age
```

Explanation:

Here name and age are user defined variables.

2 System Defined Variable

System defined variables are created and maintained by the Linux system. They store system related information.

Examples:

- HOME
- PATH
- USER
- SHELL

Example:

```
echo $USER
```

```
echo $HOME
```

3 Built-in Variable

Built-in variables are special variables provided by the shell. They give information about script execution.

Examples:

- \$0 → Script name
- \$1 → First argument
- \$# → Number of arguments
- \$? → Exit status

Example:

```
echo $0
```

```
echo $#
```

Naming Convention in Shell Script

Naming convention means rules for naming variables in shell scripting.

Rules for Variable Naming

1 Variable name must start with a letter (a-z / A-Z) or underscore _

✓ Correct:
name
_total
userName

✗ Wrong:
1name
-total

2 Variable name can contain letters, numbers, and underscore only

✓ Correct:
user1
file_name
totalMarks

✗ Wrong:
user-name
file@name

3 No spaces are allowed in variable names

✓ Correct:
myvar=10

✗ Wrong:
my var=10

4 Variable names are case-sensitive

Example:

```
name=Akhilesh  
NAME=Pal  
(name and NAME are different)
```

5 Do not use special characters

 Wrong:

name\$

total#

6 Variable assignment must be without spaces

 Correct:

age=20

 Wrong:

age = 20

What is an Array?

An array is a variable that is used to store multiple values in a single variable name.

Each value in an array is stored at a specific index or key.

Example:

```
fruits=(apple banana mango)
```

```
echo ${fruits[1]}
```

Output:

```
banana
```

Types of Array in Linux (Shell Script)

There are two types of arrays in Linux shell scripting:

1Indexed Array

An indexed array stores values using numeric index numbers (0, 1, 2, ...).

Syntax:-- array_name=(value1 value2 value3)

Example:- 1

```
numbers=(10 20 30 40)
```

```
echo ${numbers[2]}
```

Output:

```
30
```

Explanation:

Each element is accessed by its **index number**.
index numbers (0, 1, 2, ...).

Next Program:-- Indexed Array Example – 2

```
#!/bin/bash

#== Creating indexed array==

subjects=(Math English Science Computer)

echo "First subject : ${subjects[0]}"

echo "Second subject : ${subjects[1]}"

echo "Third subject : ${subjects[2]}"

echo "Fourth subject: ${subjects[3]}"

echo "All subjects : ${subjects[@]}"

echo "Total subjects : ${#subjects[@]}
```

First subject : Math
Second subject : English
Third subject : Science
Fourth subject : Computer
All subjects : Math English Science Computer
Total subjects : 4

Explanation (Easy):

- ❖ subjects → array name
- ❖ [0] [1] [2] [3] → index numbers
- ❖ Index starts from **0**
- ❖ \${#subjects[@]} → number of elements

Next Program :- Indexed Array Example – 3

```
#!/bin/bash

# Creating indexed array

marks=(65 72 80 90 88)

echo "Marks of students:"

echo "Student 1 : ${marks[0]}"

echo "Student 2 : ${marks[1]}"

echo "Student 3 : ${marks[2]}"

echo "Student 4 : ${marks[3]}"

echo "Student 5 : ${marks[4]}"

echo "All Marks : ${marks[@]}"

echo "Total Students : ${#marks[@]}
```

OUTPUT :-

Marks of students:
Student 1 : 65
Student 2 : 72
Student 3 : 80
Student 4 : 90
Student 5 : 88
All Marks : 65 72 80 90 88

Explanation (Simple):

- marks → indexed array
- Index starts from **0**
- \${marks[i]} → access element
- \${marks[@]} → all elements
- \${#marks[@]} → array length

2 :-- Associative Array

An associative array stores values in the form of key-value pairs instead of numbers.

Syntax:- declare -A array_name

Example 1 :-

```
declare -A student
student[name]=Akhilesh
student[course]=BCA
echo ${student[course]}
```

Output:- BCA

Explanation:- Values are accessed using meaningful keys.

Next Program - Associative Array Example – 2

```
#!/bin/bash

declare -A product

# Store key-value pairs

product[id]=P101
product[name]="Laptop"
product[brand]="Dell"
product[series]="Latitude 5320"
product[price]=28000
product[stock]="Available"

# Display product details
echo "Product ID : ${product[id]}"
echo "Product Name : ${product[name]}"
echo "Brand      : ${product[brand]}"
echo "Series     : ${product[series]}"
echo "Price      : ${product[price]}"
echo "Stock Status : ${product[stock]}"

# Display all keys and values
echo "All Keys  : ${!product[@]}"
echo "All Values : ${product[@]}
```

OUTPUT :-

```
[root@clint shellscript]# sh array.sh
Product ID      : P101
Product Name    : Laptop
Brand           : Dell
Series          : Latitude 5320
Price           : 28000
Stock Status   : Available
All Keys        : price brand id series name stock
All Values      : 28000 Dell P101 Latitude 5320 Laptop Available
[root@clint shellscript]#
```

Explanation (Easy):

1. declare -A product → associative array declare karta hai
2. id, name, brand, series, price, stock → keys hote hain
3. Values ko keys ke through access kiya jata hai
4. \${product[key]} → particular key ki value show karta hai
5. \${!product[@]} → all keys show karta hai
6. \${product[@]} → all values show

Difference between Indexed Array and Associative Array

Indexed Array	Associative Array
Uses numeric index numbers (0, 1, 2, ...)	Uses key-value pairs
Index starts from 0	Uses string keys
declare -a required	declare -A required
Stores elements in sequence/order	Order of elements is not fixed
Suitable for simple lists	Suitable for complex records
Uses less memory	Uses more memory
Access is faster using index	Access depends on key lookup
Easy to learn for beginners	Slightly complex
Example: arr[1]=50	Example: arr[name]="Linux"
Bash version independent	Needs Bash 4.0 or above

Explanation (Easy in Hindi):

- Indexed Array → values access hote hain index number se
- Associative Array → values access hote hain keys se
- Indexed array simple data ke liye use hota hai
- Associative array structured data ke liye use hota hai

Operator in Linux

An operator in Linux is a symbol or keyword used to perform operations such as calculation, comparison, and condition checking in shell scripting.

Explanation:

Operators are used to compare values, check conditions, and perform actions in Linux commands and shell scripts.

Types of Operators in Linux:

1. Arithmetic Operators
2. Relational Operators
3. File operators
4. String Operators
5. Boolean Operators

1 Arithmetic Operators (Linux)

Arithmetic operators are used to perform mathematical calculations in Linux shell scripting.

List of Arithmetic Operators

Operator	Meaning
+	Addition
-	Subtraction

Operator	Meaning
*	Multiplication
/	Division
%	Remainder (Modulus)

PROGRAM 1 :----

```
#!/bin/bash
a=20
b=10
echo "Addition = $((a+b))"
echo "Subtraction = $((a-b))"
echo "Multiplication = $((a*b))"
echo "Division = $((a/b))"
echo "Remainder = $((a%b))"
```

OUTPUT:-

Addition = 30
Subtraction = 10
Multiplication = 200
Division = 2
Remainder = 0

Program 2 :- Arithmetic Operators (By User)

```
#!/bin/bash
read -p "Enter first number:" a
read -p "Enter second number:" b
echo "Addition = $((a+b))"
echo "Subtraction = $((a-b))"
echo "Multiplication = $((a*b))"
echo "Division = $((a/b))"
echo "Remainder = $((a%b))"
```

OUTPUT:-

Enter first number:10
Enter second number:3
Addition = 13
Subtraction = 7
Multiplication = 30
Division = 3
Remainder = 1

2 Relational Operators (Linux)

Relational operators are used to compare two values and check conditions in shell scripting.

List of Relational Operators

Operator	Meaning
-eq	equal to
-ne	not equal
-gt	greater than
-lt	less than
-ge	greater than or equal
-le	less than or equal

Program 1 : Relational Operators (By User)

```
#!/bin/bash

read -p "Enter first number:" a

read -p "Enter second number:" b

if [ $a -eq $b ];then

    echo "Both numbers are equal"

fi

if [ $a -gt $b ];then

    echo "First number is greater"

fi

if [ $a -lt $b ]

    then

        echo "First number is smaller"

fi
```

OUTPUT:-

```
Enter first number:20

Enter second number:10

First number is greater
```

```
[root@clint shellscrip]# sh array
Enter first number:20
Enter second number:10
First number is greater
[root@clint shellscrip]#
```

Program 2 : Relational Operators

```
#!/bin/bash  
  
a=15  
  
b=10  
  
if [ $a -eq $b ]  
then  
echo "Both values are  
equal"  
  
fi  
  
if [ $a -gt $b ]  
then  
echo "A is greater than B"  
  
fi  
  
if [ $a -lt $b ]  
then  
echo "A is less than B"  
  
fi
```

OUTPUT:-

A is greater than B

```
[root@clint shellscript]# sh  
A is greater than B  
[root@clint shellscript]#
```

Explanation (Short):

- ❖ a=15 and b=10
- ❖ 15 -eq 10 → false
- ❖ 15 -gt 10 → true
- ❖ 15 -lt 10 → false

3 File Test Operators (Linux)

File test operators are used to check file or directory properties in Linux shell scripting.

List of File Test Operators

Operator	Meaning
-e	File or directory exists
-f	Exists and is a regular file
-d	Exists and is a directory
-s	File exists and is not empty
-r	File is readable
-w	File is writable
-x	File is executable
-L	File is a symbolic link
-h	File is a symbolic link
-b	File is a block special file
-c	File is a character special file
-p	File is a named pipe (FIFO)
-S	File is a socket
-O	File is owned by current user
-G	File group ID matches current user

Operator	Meaning
-u	Set-user-ID (SUID) bit is set
-g	Set-group-ID (SGID) bit is set
-k	Sticky bit is set
-nt	File is newer than another file
-ot	File is older than another file
-ef	Both files refer to same file

Program: File Operators

```
#!/bin/bash
file="/etc/passwd"
if [ -f $file ]
then
echo "File exists"
fi

if [ -r $file ]
then
echo "File is readable"
fi

if [ -w $file ]
then
echo "File is writable"
fi
```

OUTPUT:-

File exists
File is readable
File is writable

```
[root@clint shellscript]# sh
File exists
File is readable
File is writable
[root@clint shellscript]#
```

Explanation

- -f checks whether file exists
- -r checks read permission
- -w checks write permission
- File path already defined in variable

4 String Operators (Linux)

String operators are used to compare strings and check string conditions in Linux shell scripting.

List of String Operators

Operator	Meaning
=	strings are equal
!=	strings are not equal
-z	string is empty
-n	string is not empty

Program: String Operators

```
#!/bin/bash
name="Ipst"
user="Admin"
if [ $name = "Ipst" ];then
echo "Name is Ipst"
fi
if [ $name != $user ];then
echo "Strings are not equal"
fi
if [ -n $name ];then
echo "String is not empty"
fi
```

OUTPUT:-

Name is Ipst
Strings are not equal
String is not empty

```
[root@clint shellscript]# sh
Name is Ipst
Strings are not equal
String is not empty
[root@clint shellscript]#
```

Explanation (short):

- = compares two strings
- != checks strings are different
- -n checks string is not empty
- Values are already defined

5 Boolean Operators (Linux)

Boolean operators are used to combine conditions and return true or false in Linux shell scripting.

List of Boolean Operators

Operator	Meaning
-a	AND
-o	OR
!	NOT

Program: Boolean Operators (Value Defined)

```
#!/bin/bash

x=25

y=15

# AND operator

if [ $x -gt 20 -a $y -gt 10 ];then
echo "Both conditions are TRUE"
fi

# OR operator

if [ $x -lt 20 -o $y -gt 10 ];then
echo "At least one condition is TRUE"
fi

# NOT operator

if [ ! $x -lt 10 ];then
echo "NOT condition is TRUE"
fi
```

OUTPUT:-

Both conditions are TRUE

At least one condition is TRUE

NOT condition is TRUE

```
[root@clint function2]# sh 4.sh
Both conditions are TRUE
At least one condition is TRUE
NOT condition is TRUE
[root@clint function2]#
```

```
#!/bin/bash  
  
a=10  
  
b=5  
  
if [ $a -gt 5 -a $b -lt 10 ];then  
  
echo "Both conditions are true"  
  
fi  
  
if [ $a -lt 5 -o $b -lt 10 ];then  
  
echo "At least one condition is true"  
  
fi  
  
if [ ! $a -lt 5 ];then  
  
echo "Condition is false"  
  
fi
```

OUTPUT:-

Both conditions are true

At least one condition is true

Condition is false

```
[root@clint shellscrip]# sh op  
Both conditions are true  
At least one condition is true  
Condition is false  
[root@clint shellscrip]# ]
```

Explanation

- -a (AND) → both conditions must be true
- -o (OR) → any one condition true
- ! (NOT) → reverses the condition

Function in Shell Script

A function is defined with a name, a set of parameters, and a block of code that performs a particular operation.

A function is a reusable block of code that performs a specific task.

It helps to reuse code, provides better readability, and is easy to maintain.

1 No Argument – No Return

This type of function does not take any argument and does not return any value.

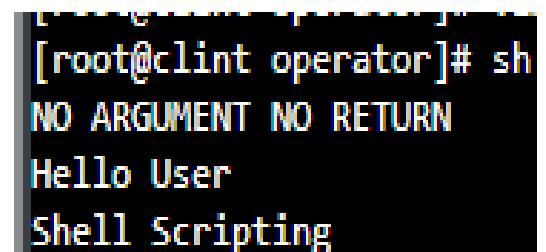
It is used only to display output or perform a fixed task.

Example:

```
#!/bin/bash
echo "NO ARGUMENT NO RETURN "
function fun1() {
    echo "Hello User"
    echo "Shell Scripting"
}
fun1
```

OUTPUT:-

NO ARGUMENT NO RETURN
Hello User
Shell Scripting



```
[root@clint operator]# sh
NO ARGUMENT NO RETURN
Hello User
Shell Scripting
```

2 With Argument – No Return

This type of function takes arguments but does not return any value.

It uses the arguments to perform an operation and show the result.

```
#!/bin/bash
echo " WITH ARGUMENT
NO RETURN "
function fun2() {
    echo "Course : $1"
    echo "Duration : $2"
}
fun2 Linux 2Years
```

OUTPUT:-

WITH ARGUMENT NO RETURN
Course : Linux
Duration : 2Years

```
[root@clint operator]# sh new.sh
 WITH ARGUMENT NO RETURN
Course : Linux
Duration : 2Years
```

3 No Argument – With Return

This type of function does not take any argument but returns a value after calculation.

Example:

```
#!/bin/bash
fun3() {
    num=4
    multi=$((num * num))
    return $multi
}
fun3
echo " Multi..value is : " $? 
```

OUTPUT:-

Multi..value is : 16

num=4 → variable num ko value **4** assign karta hai
multi=\$((num * num)) → num ka square calculate karta hai (**4 × 4 = 16**)
return \$multi → function se calculated value **16** return karta ha

4 With Argument – With Return

This type of function takes arguments and returns a value after processing them.

Example:

```
#!/bin/bash

add() {
    sum=$(( $1 + $2 ))
    echo $sum
}

result=$(add 10 20)
echo "Result = $result"
```

OUTPUT:-

Result= 30

Explanation:-

- add() → function name
- \$1 \$2 → arguments
- echo \$sum → return value
- result=\$(add 10 20) → returned value store ki