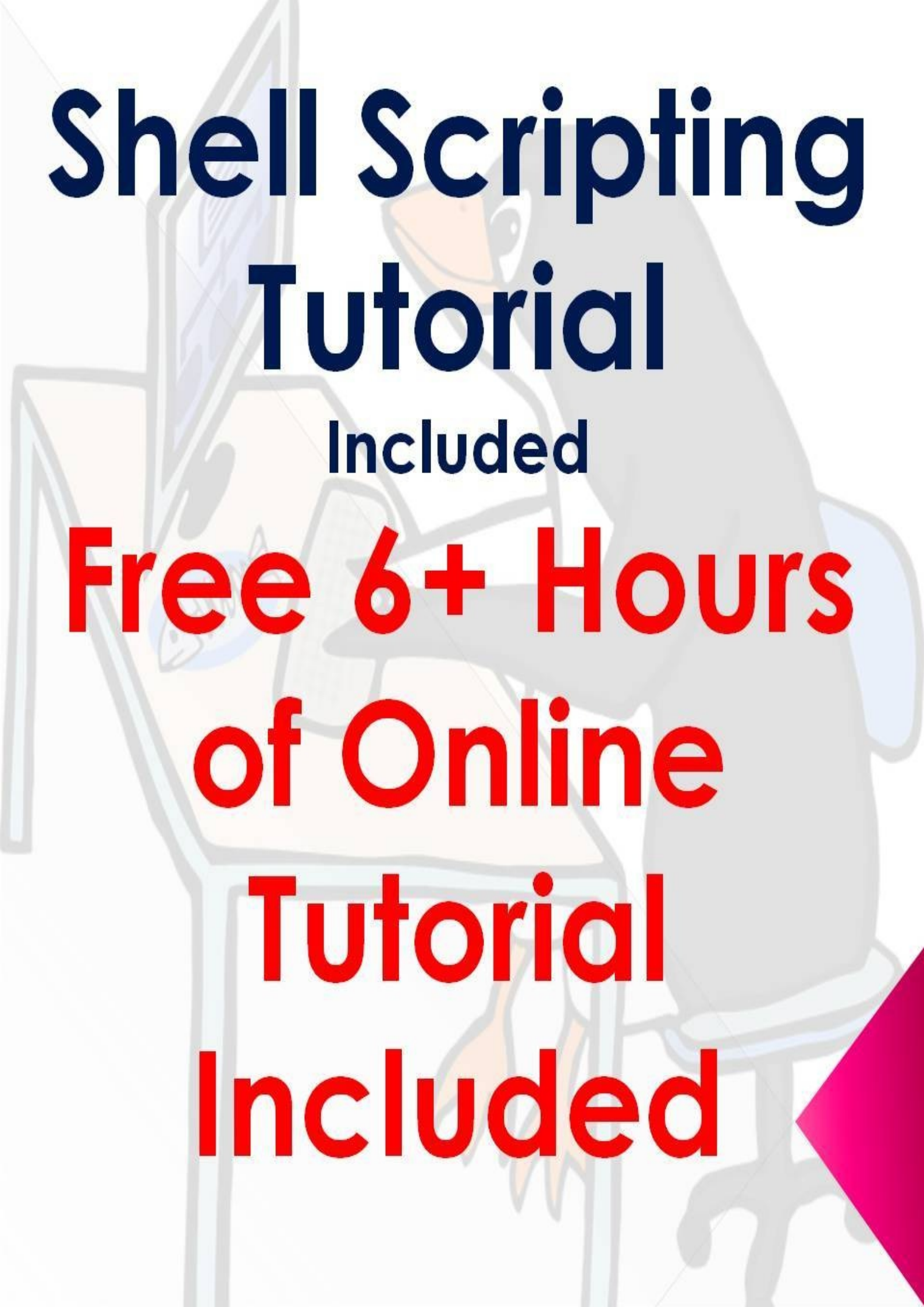
Shell Scripting Tutorial

Included

Free 6+ Hours of Online Tutorial Included



This book comes with Free Access to Online Shell Scripting tutorial (worth of $49). Every concept has been explained with lot of examples so you would know when to apply it and how to apply them in real time world. Also free course includes downloadable ebook containing all concepts and related examples.

- You will become a shell programming expert with complete knowledge of shell programming
- You will learn with more than 60+ programming & real world examples
- You will know How to make use of Mathematical, String and Logical operators in shell script to make decisions
- How to Create functions in shell scripts and improve reusability
- You will learn How to make use of Exit values to determine shell script output status
- How to Accept input from a user and then make decisions on that input.
- You will know How to make use of Expressions in shell scripts
- Dealing with command line arguments and use of it with examples
- You will learn How to make use of Pipe & Process concepts while creating shell scripts
- Use of utilities like cut, paste, join, tr in shell scripts with examples
- Practice exercises with solutions so you can start using what you learn right away.
- Real-world examples of shell scripts, how it is used in corporate world.
- Last but not least, you will get downloadable material containing the scripts and topics contents

PS: Free access to tutorial link is given at end of book, please check index for same.

Thank you!

Contents

- shell or bash script to rename files
- shell or bash script to copy files
- shell or bash script to create directory
- shell or bash script to delete file
- shell or bash script to read file line by line
- shell or bash script to list files in a directory
- shell or bash script to move file to another location
- shell or bash script to get cpu usage of a user

Process, Signals & Traps

- What is a Process and How to view Processes
- Sending signal to Processes
- Terminating Processes
- kill command Examples
- Shell signal values
- The trap statement
- How to clear trap
- trap statements to catch signals and handle errors in a script

Get here - Free Access to Online tutorial

Shell Introduction

What is Linux Shell?

A shell is a special-purpose program designed to read commands typed by a user and execute appropriate programs in response to those commands. Such a program is sometimes known as a command interpreter.

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems).

Important shells in market

A number of important shells have appeared over time:

?? Bourne shell (sh): This is the oldest of the widely used shells, and was written by Steve Bourne. It was the standard shell for Seventh Edition UNIX. The Bourne shell contains many of the features familiar in all shells: I/O redirection, pipelines, filename generation (globbing), variables, manipulation of environment variables, command substitution, background command execution, and functions.

All later UNIX implementations include the Bourne shell in addition to any other shells they might provide.

?? C shell (csh): This shell was written by Bill Joy at the University of California at Berkeley. The name derives from the resemblance of many of the flow-control constructs of this shell to those of the C programming language. The C shell provided several useful interactive features unavailable in the Bourne shell, including command history, command-line editing, job control, and aliases. The C shell was not backward compatible with the Bourne shell. Although the standard interactive shell on BSD was the C shell, shell scripts (described in a moment) were usually written for the Bourne shell, so as to be portable across all UNIX implementations.

?? Korn shell (ksh): This shell was written as the successor to the Bourne shell by David Korn at AT&T Bell Laboratories. While maintaining backward compatibility with the Bourne shell, it also incorporated interactive features similar to those provided by the C shell.

?? Bourne again shell (bash): This shell is the GNU project’s reimplementation of the Bourne shell. It supplies interactive features similar to those available in the C and Korn shells. The principal authors of bash are Brian Fox and Chet Ramey. Bash is probably the most widely used shell on Linux. (On Linux, the Bourne shell, sh, is actually provided by bash emulating sh as closely as possible.)

What is Shell Script ?

Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of ‘n’ number of commands) , the you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands. This is known as ***shell script***.

Shell script defined as:”*Shell Script is* ***series of command*** *written* ***in plain text file****. Shell script is just like batch file is MS-DOS but have more power than the MS-DOS batch file.*”

Why to Write Shell Script?

- Shell script can take input from user, file and output them on screen.
- Useful to create our own commands.
- Save lots of time.
- To automate some task of day today life.
- System Administration part can be also automated.

First “Hello World” Shell script

Saving and executing first “Hello World” shell script

- In this tutorial we are going to get started with shell programming, how to write script, execute them etc. We will get started with writing small shell script, that will print “Hello world” on screen.
- To write shell script you can use any editor supported like vi or mcedit.
- I will use vi editor: To open a new file in vi edit you can use the command vi followed by the script name that you would want to save your file as.
- Note that shell scripts have an extension of .sh

vi hello.sh

My first shell script

echo “Hello World”

- To save this file, press wq and exit
- Before you can set execute the shell script, you need to set execute permission for your script as follows:

syntax:

chmod permission your-script-name

Examples:

$ chmod +x your-script-name

$ chmod 755 your-script-name

Note: This will set read write execute(7) permission for owner, for group and other permission is read and execute only(5).

chmod 755 hello.sh

- There are couple of ways to execute the shell script.

syntax:

bash your-script-name

sh your-script-name

./your-script-name

Examples:

$ bash hello.sh

$ sh hello.sh

$ ./hello.sh

NOTE In the last syntax ./ means current directory, But only . (dot) means execute given command file in current shell without starting the new copy of shell, The syntax for . (dot) command is as follows

Syntax:

. command-name

Example:

$. num_var

- Let us analyses the script we wrote. You can re-open the script vi hello.sh (Start vi editor)
- In the script,

My first shell script

- # followed by any text is considered as comment. Comment gives more information about script, logical explanation about shell script.

  Syntax:

  # comment-text

- Next,

echo “Hello World”

- To print message or value of variables on screen, we use echo command, general form of echo command is as follows

  syntax:

  echo “Message”

Variables

Variables in Shell

- To process our data/information, data must be kept in computers Random Access Memory.
- RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data.
- Programmer can give a unique name to this memory location/address called memory variable or variable
- In Linux (Shell), there are two types of variable:
    - **System variables -** This type of variable defined in CAPITAL LETTERS. These are created and used by the Linux system.
    - **User defined variables (UDV) -** This type of variable defined in lower letters. These are create and used by the user.

How to define User defined variables (UDV)

- To define a variable use the following syntax:

Syntax:

variable_name=value

- ‘**value**‘ is assigned to given ‘**variable_name**‘ and Value must be on right side = sign
- a=10
- b=‘Hello’

System Variable example

- You can see system variables by giving command like **$ set**, some of the important System variables are: Modify this table accordingly.

System Variable	**Meaning**
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/Tom	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our Os type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=Tom	User name who is currently login to this PC

Rules for Naming variable name

- Here are 4 rules you need to consider for naming variable name (Both UDV and System Variable) .

1. Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character. Eg,

HOME, value, num_var, val_num_var

2. Don't put spaces on either side of the equal sign when assigning value to variable.

In following variable declaration there will be no error

$ num_var=10

But there will be problem for any of the following variable declaration:

$ num_var =10

$ num_var= 10

$ num_var = 10

3. Variables are case-sensitive, just like filename in Linux.

num_var, Num_var, NUM_VAR, Num_var are all different variable names.

4. You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition) For e.g.

$ vech=

$ vech=""

Try to print it's value by issuing following command

$ echo $vech

Nothing will be shown because variable has no value i.e. NULL variable.

How to print or access value of User defined variables

- To print or access UDV use following syntax

Syntax:

$variablename

or

echo $variablename

- Define variable vehicle and num as follows:

$ vehicle=Bus

$ num=10

- To print contains of variable ‘vehicle’ type

$ echo $vehicle

- It will print ‘Bus’,To print contains of variable ‘num’ type command as follows

$ echo $num

Note: Do not try **$ echo** vehicle, as it will print vech instead its value ‘Bus’ and **$ echo num**, as it will print n instead its value ‘10’, You must *use $ followed by variable name*.

Usage of Quotes - Concept & Examples

Double, Single and Back Quotes

There are three types of quotes:

Double Quotes - ”” - “Double Quotes” - Anything enclose in double quotes removed meaning of that characters (except \ and $).

Single quotes - ” - ‘Single quotes’ - Enclosed in single quotes remains unchanged.

Back quote - “ - `Back quote` - To execute command

Example:

$ echo “Time is date”

- Can’t print message with time

$ echo “Time is `date`”.

- It will print time as, 17:01:35.99 Can you see that the `date` statement uses back quote?

Exit status value

Exit Status Concept

- By default in Linux if particular command/shell script is executed, it return two type of values which is used to see whether command or shell script executed is successful or not.

1. If return *value is zero* (0), command is successful.
2. If return *value is nonzero*, command is not successful or some sort of error executing command/shell script.

- This value is known as ***Exit Status***.
- But how to find out exit status of command or shell script?
- Simple, to determine this exit Status you can use **$?** special variable of shell.

Exit Status Demo

For e.g. (This example assumes that helloWorld.sh doest not exist on your hard drive)

$ rm helloWorld.sh

- It will show error as follows

rm: cannot remove `helloWorld.sh’: No such file or directory

- and after that if you give command

$ echo $?

- it will print nonzero value to indicate error. Now give command

$ ls

$ echo $?

- It will print 0 to indicate command is successful.

$ date

$ echo $?

$ echo hello

$ echo $?

How to Get Input from user

read Statement to Get Input from user

- If we would like to ask the user for input then we use a command called **read**. This command takes the input and will save it into a variable.

Syntax:

read variable1, variable2,…variableN

- Let’s look at a simple example:

```
#!/bin/bash
# Ask the user for his name
echo Hello, what is your name?
read varname
echo It's nice to meet you $varname
```

- This script would ask the user for his name and greet him.
- Run it as follows:

```
chmod 755 hi.sh
./hi.sh
Hello, what is your name? bashTutor
It’s nice to meet you bashTutor
```

Wild cards

Wild Cards concept

- We will look at 3 wild cards characters,

Wild card /Shorthand	Meaning	Examples	
*	Matches any string or group of characters.	**$ ls ***	will show all files
		$ ls h*	will show all files whose first name is starting with letter ‘h’
		$ ls *.sh	will show all files having extension .sh
		$ ls h*.sh	will show all files having extension .sh but file name must begin with ‘h’.
?	Matches any single character.	**$ ls ?**	will show all files whose names are 1 character long
		$ ls he?	will show all files whose names are 3 character long and file name begin with fo
[…]	Matches any one of the enclosed characters	**$ ls [abc]***	will show all files beginning with letters a,b,c

Note: A pair of characters separated by a minus sign denotes a range.

Wild cards Examples

- To show all files name beginning with letter a,b or c like

$ ls /bin/[a-c]*

/bin/arch /bin/awk /bin/bsh /bin/chmod /bin/cp

/bin/ash /bin/basename /bin/cat /bin/chown /bin/cpio

/bin/ash.static /bin/bash /bin/chgrp /bin/consolechars /bin/csh

- And opposite of that is: If the first character following the [is a ! or a ^ ,then any character not enclosed is matched i.e. do not show us file name that beginning with a,b,c,e…o, like

$ ls /bin/[!a-o]

$ ls /bin/[^a-o]

Redirection of Standard Input/Output

Redirection concept with >,>>,<

Most Unix system commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from a place called standard input, which happens to be your terminal by default. Similarly, a command normally writes its output to standard output, which is also your terminal by default.

For e.g. **$ ls** command gives output to screen; to send output to file of ls command give command.

There are three main redirection symbols **>,>>,<**

(1) > Redirector Symbol

Syntax:

Linux-command > filename

- To output Linux-commands result (output of command or shell script) to file. Note that if file already exist, it will be overwritten else new file is created.
- (2) >> Redirector Symbol

Syntax:

Linux-command >> filename

- To output Linux-commands result (output of command or shell script) to END of file. Note that if file exist , it will be opened and new information/data will be written to END of file, without losing previous information/data, And if file is not exist, then new file is created.
- (3) < Redirector Symbol

Syntax:

Linux-command < filename

- To take input to Linux-command from file instead of key-board.

Redirection examples with >,>>,<

- Let us now look at the redirection
- The > Redirector Symbol
- For e.g. To send output of ls command give

$ ls > myfiles

- Now if '**myfiles**' file exist in your current directory it will be overwritten without any type of warning.
- The >> Redirector Symbol
- For e.g. To send output of date command to already exist file give command

$ date >> myfiles

- The < Redirector Symbol
- For e.g. To take input for cat command give

$ cat < myfiles

- You can also use above redirectors simultaneously as follows
- Create text file as follows

$cat > fruitsnames

Apple

Orange

mango

peas

Press CTRL + D to save.

- Now issue following command.

$ sort < fruitsnames > sorted_fruitsnames

$ cat sorted_fruitsnames

Pipe Concept and Examples

Pipe Concept

- You can connect two commands together so that the output from one program becomes the input of the next program. Two or more commands connected in this way form a pipe.
- To make a pipe, put a vertical bar (|) on the command line between two commands.
- Pipe Defined as:
- *"A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands (Multiple commands) from same command line."*

Syntax:

```
command1 | command2
```

5 Pipe examples used in real time world

Pipe example to print sorted list of users

- The **sort** command arranges lines of text alphabetically or numerically. We will now see an example usage of pipe command to print sorted list of users.

$ who | sort

- Output of who command is given as input to sort command So that it will print sorted list of users

print sorted list of users and output is sent to (redirected) list_of_user file

- For this we can execute,

$ who | sort > list_of_user

- Same as above except output of sort is send to (redirected) list_of_user

print number of user who logon to system

- For this we can execute,

$ who | wc -l

Output of who command is given as input to wc command So that it will number of user who logon to system

print number of files in current directory

- For this we can execute,

$ ls -l | wc -l

- Output of ls command is given as input to wc command So that it will print number of files in current directory.

print if particular user is logon

- For this we can execute,

$ who | grep oshi

- Output of who command is given as input to grep command So that it will print if particular user name if he is logon or nothing is printed (To see particular user is logon or not)

Executing multiple commands in single command line

- There are needs many a time to execute more than one commands on a line the syntax to do this,

Syntax:

command1;command2

- To run two command with one command line.

Examples:

$ time;whoami

- Will print time followed by my username.

Note that You can’t use, it will give an error -

$ time whoami

- for same purpose, you must put semicolon in between time and whoami command.

Many times in our tutorial you will find following in if syntax

if <condition> ; then

commands

fi

since we are giving then on same line where is if condition we give ; before then otherwise same can be writtent without ; as below -

if <condition>

then

commands..

fi

Test or Expression command

Expression Concept & Example

- We use the test command or [expr] to see if an expression is true.
- If it is true it return zero(0), otherwise returns nonzero for false.

Syntax:

test expression OR [expression]

- Example*:* Following script determine whether given argument number is positive.
- when you give expression without test command you just mention it in [expression]
- NOTE -Please make sure space after “[” and before “]”
- if you try without space and it will give an error

Example 1 -

$ cat > check_postive.sh

#!/bin/sh

#

Script to see whether argument is positive

#using mathematical operator gt, will see it in detail later

if test $1 -gt 0

then

echo “$1 number is positive”

```
fi
#same can be written without test as below
#NOTE -Please make sure space after “[” and before “]”
#try without space and it will give an error

if [ $1 -lt 0 ]
then
echo “$1 number is Negative”
fi
```

(press ctrl+c to save and exit script)

- Run it as follows

```
$ chmod 744 check_postive.sh
$ ./check_postive.sh 7
```

- 7 number is positive

```
$ ./check_postive.sh -49
-49 number is Negative

$ check_postive
```

- ./check_postive: test: -gt: unary operator expected

Code walk through

- The line, if test $1 -gt 0 , test to see if first command line argument($1) is greater than 0.
- If it is true(0) then test will return 0 and output will printed as 7 number is positive but for -49 argument other echo statement printed.

- And for last statement we have not supplied any argument hence error ./check_postive: test: -gt: unary operator expected, is generated by shell , to avoid such error we can test whether command line argument is supplied or not.

Expression list options & example

Here is a partial list of the conditions that **test** can evaluate.

Since **test** is a shell command, use " **help test** " to see a complete list.

you can either use it with expression within [ ] Or

with test command as shown in example 2

Expression	**Description**
-d *file*	True if *file* is a directory.
-e *file*	True if *file* exists.
-f *file*	True if *file* exists and is a regular file.
-r *file*	True if *file* is a file readable by you.
-w *file*	True if *file* is a file writable by you.
-x *file*	True if *file* is a file executable by you.
file1 -nt *file2*	True if *file1* is newer than (according to modification time) *file2*
file1 -ot *file2*	True if *file1* is older than *file2*
-z *string*	True if *string* is empty.

Example 2 -

Example with expression to check if file exists or not using expression.

create test file -

cat > mytest.txt

my test file to check

(press ctrl+d or ctrl+c to save and exit)

cat > test_myfile.sh

echo “using expression with if “

if [-f mytest.txt **]; then**
    echo “You have a mytest.txt. Things are fine.”
else
    echo “Yikes! You have no mytest.txt!”
fi

echo “using test command”

if test -f mytest.txt
then
    echo “Awesome! You have a mytest.txt. Things are fine.”
else
    echo “O O! You have no mytest.txt!”
fi

fi

chmod 777 test_myfile.sh

./ test_myfile.sh

rm mytest.txt

run again-

./ test_myfile.sh

similarly you may try all other options with test to verify various things. we will look that with our real time examples in next chapters.

expr for expression evaluation

expr - Arithmetic Operator concept & examples

expr command is very commonly used to perform many arithmetic, logical or string operations in shell script.

You can do -

Value1 + Value2 arithmetic sum of Value1 and Value2.

Value1 - Value2 arithmetic difference of Value1 and Value2.

Value1 * Value2 arithmetic product of Value1 and Value2.

Value1 / Value2 arithmetic quotient of Value1 divided by Value2.

Value1 % Value2 arithmetic remainder of Value1 divided by Value2.

(just run it at command line)

Example 1 - to increment value of variable

```
cnt =0
echo $cnt
cnt= `expr $cnt + 1`
echo $cnt
```

1

Example 2 - sum of numbers

```
expr 1 + 2 + 3 + 4
```

10

Example 3 - Multiplying numbers

```
expr 8 \* 9
```

72

Example 4 -

expr 8 % 5

3

expr - Comparison Operator concept & examples

You can use the following comparision operators with the expr command:

Returns value 1 if condition is true otherwise 0.

Value1 < Value2 : Returns 1 if Value1 is less than Value2. otherwise zero.

Value1 <= Value2 : Returns 1 if Value1 is less than or equal to Value2. otherwise zero.

Value1 > Value2 : Returns 1 if Value1 is greater than Value2. otherwise zero.

Value1 >= Value2 : Returns 1 if Value1 is greater than or equal to Value2. otherwise zero.

Value1 = Value2 : Returns 1 if Value1 is equal to Value2. otherwise zero.

Value1 != Value2 : Returns 1 if Value1 is not equal to Value2. otherwise zero.

Value1 | Value2 : Returns Value1 if Value1 is neither null nor zero. Otherwise Value2.

Value1 & Value2 : Returns Value1 if both Value1 and Value2 is neither null nor zero. Otherwise 0.

Examples - (just run it at command line)

expr 2 \< 3

1

expr 5 \<= 5

1

expr 3 \> 8

0

expr 3 \>= 8

0

expr 9 = 9

1

expr 9 != 81

1

expr 2 \| 6

2

expr 0 \| 7

7

expr 2 \& 7

2

expr 6 \& 4

6

expr 7 \& 0

0

expr 0 \& 4

0

expr - String Operator concept & examples

length string length of string

expr length unix

4

substr string position length

extract a portion of string. Here position is the character position in the string. length is the number of chracters to extract from the main string

expr substr linuxserver 6 6

server

match string pattern

If the chars string is found in the main string, then the index function returns the position of the chars. Otherwise it returns 0.

expr match unixserver uni

3

index string chars

If the chars string is found in the main string, then the index function returns the position of the chars. Otherwise it returns 0.

expr index unix uni

3

Mathematical operators

Mathematical operators concept

- Depending on what type of work you want your scripts to do you may end up using arithmetic a lot or not much at all. It's a reasonable certainty however that you will need to use arithmetic at some point.

Mathematical Operator in Shell Script	**Meaning**	**Normal Arithmetical/ Mathematical Statements**	**But in Shell:**
			For [expr] statement with if command
-eq	is equal to	4 == 7	if [4 -eq 7]
-ne	is not equal to	4 != 7	if [4 -ne 7]
-lt	is less than	4 < 7	if [4 -lt 7]
-le	is less than or equal to	4 <= 7	if [4 -le 7]
-gt	is greater than	4 > 7	if [4 -gt 7]
-ge	is greater than or equal to	4 >= 7	if [4 -ge 7]

NOTE: == is equal, != is not equal.

Mathematical operators example

```
cat > math_opr.sh

#!/bin/bash

# declare integers

NUM1=2

NUM2=2

NUM3=5


if [ $NUM1 -eq $NUM2 ]; then

        echo “Both Values are equal”

else

        echo “Values are NOT equal”

fi


if [ $NUM1 -le $NUM3 ]; then

        echo “NUM1 is Less than NUM3”

else

        echo “NUM3 Greater than NUM1”

fi


press ctrl+c


ls –l

chmod 777 math_opr.sh

./math_opr.sh
```

String Comparison operators

String Comparison operators concept

- There are following string operators supported

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

String Comparison operators example

Please explain code first in detail with concept then output with revisiting code why it showed that output.

cat > string_comparison.sh

```
#!/bin/bash
#declare strings
#script to compare strings
str1=“ABC”
str2=“XYZ”
str3=” “
if [ “$str1” = “XYZ” ]; then
    echo “str1 Matched”
else
    echo “str1 is NOT Matched $str1”
fi;

if [ “$str2” = “XYZ” ]; then
    echo “str2 Matched”
else
    echo “str2 is NOT Matched $str1”
fi;

if [ -n “$str2” ]; then
    echo “str2 NOT NULL”
else
```

```
    echo “str2 is NULL”
fi;

if [ -z “$str3” ]; then
    echo “str3 IS NULL”
else
    echo “str3 is NOT NULL”
fi;

ls –l

chmod 777 string_comparison.sh
./ string_comparison.sh
```

Logical Operators

Logical Operators Concept

Operator	Meaning
! expression	Logical NOT
expression1 -a expression2 / &&	Logical AND
expression1 -o expression2 / ||	Logical OR

Not: This is logical negation. This inverts a true condition into false and vice versa.

AND – you can use –a or && (most common) both conditions need to be satisfied

if [$condition1] && [$condition2]

Same as: if [$condition1 -a $condition2]

Returns true if both condition1 and condition2 hold true…

OR – you may use –o or || (most common) one of condition needs to be true

if [$condition1] || [$condition2]

Same as: if [$condition1 -o $condition2]

Returns true if either condition1 or condition2 holds true

Logical Operators Example

```
cat > logic_opr.sh

str1=“ABC”
str2=“XYZ”
str3=“ABC”

if [ “$str1” = “ABC” ] && [ “$str3” = “ABC” ]; then
    echo “str1 and str3 Matched”
else
    echo “str1 NOT Matched with str3”
fi;

if [ “$str2” = “XYZ” ] || [ “$str3” = “XYZ” ]; then
    echo “str2 or str3 can be XYZ”
else
    echo “str2 or str3 none of them is XYZ”
fi;

if [[ !(“$str1” = “ABC”) ]]; then
echo “str1 is not ABC”
else
    echo “str1 is ABC”
fi;
```

ls –l

chmod 777 logic_opr.sh

./ logic_opr.sh

Command Line Arguments

What is Command Line arguments

- In a shell script, you can pass variables as arguments by entering arguments after the script name, for e.g. ./script.sh arg1 arg2.
- The shell automatically assigns each argument name to a variable.
- To specify an argument that includes spaces, you need to enclose the complete argument in double quotation marks.
- The first argument after the script name is assigned to the variable $1, the second argument to $2, and so on.

Why Command Line arguments required

- We want command line arguments because:
  - Telling the command/utility which option to use.
  - Informing the utility/command which file or group of files to process (reading/writing of files).
- Let’s take rm command:

$ rm {file-name}

- Here rm is command and file-name is file which you would like to remove. We can now tell the rm command which file you would like to remove.
- So we are doing one way communication with our command by specifying file-name.

Command Line argument examples

Example 1

- Lets take ls command

$ ls -a /*

- This command has 2 command line argument -a and /* is another. For shell script,
- Similarly for a script:

$ any_script_name yoyo jar

- Shell Script name i.e. any_script_name
- First command line argument passed to myshell i.e. yoyo
- Second command line argument passed to myshell i.e. jar

Example 2

- Let us write a script to print command line arguments.

$ cat > demo_example.sh

#!/bin/sh

#

echo “Total number of command line argument are $#”

echo “$0 is script name”

echo “$1 is first argument”

echo “$2 is second argument”

echo “Total Number of arguments are $#”

(press ctrl+c to save and exit)

chmod777 demo_example.sh

- Execute the script.

$ ./demo_example Hello World

We will be using command line arguments in many of our examples so please understand it properly.

You may not assign command line arguments in script so following is Invalid

$1 = ABC

$2 = “Name”

echo and read concept & example

- In real time situations, shell script may need to interact with users. You can accomplish this as follows:

Use command line arguments (args) to script when you want interaction i.e. pass command line args to script as :

$ ./script_name.sh fo 4

where fo & 4 are command line args passed to shell script sutil.sh.

OR

Use statement like echo and read to read input into variable from the prompt. For e.g. Write script as:

$ cat > user_input

#

using echo and read command for user interaction

#

echo “Name please :”

read name

echo “Last Name please :”

read l_name

echo “Hello $name $l_name, How are you?”

Save it and run as

$ chmod 755 user_input

$ ./user_input

Conditional statements

if command concept & example

- If statements (and, closely related, case statements) allow us to make decisions in our Bash scripts. If given condition is true then command1 is executed.

Syntax:

if condition or expression

then

command1 if condition is true

…

…

fi

Let us execute commands (assumes you have file called **test.txt**)

$ cat test.txt

$ echo $?

- The cat command return zero(0) i.e. exit status, on successful, this can be used, in if condition as follows, Write shell script as

$ cat > showfile

#!/bin/sh

#

#Script to print file

#

if cat $1

then

echo -e “\n\nFile $1, found and successfully echoed”

fi

- So here is what the script does on execution.
- if cat command finds test.txt file and if its successfully shown on screen, it means our cat command is successful and its exist status is 0 (indicates success), So our if condition is also true and hence statement echo -e “\n\nFile $1, found and successfully echoed” is proceed by shell. Now if cat command is not successful then it returns non-zero value (indicates some sort of failure) and this statement echo -e “\n\nFile $1, found and successfully echoed” is skipped by our shell.

if…else…fi concept & example

5. If given condition is true then command1 is executed otherwise command2 is executed.
6. *Syntax:*

if condition

then

condition is zero (true - 0)

execute all commands up to else statement

else

if condition is not true then

execute all commands up to fi

fi

if…else…fi example

- Let us look at an example to test if a given number is positive or negative,

```
$ vi test_number.sh
#!/bin/sh
#
# Script to see whether argument is positive or negative
# if no arguments are provided
if [ $# -eq 0 ]
then
echo “$0 : You must supply at least one integers”
exit 1
fi
```

```
if test $1 -gt 0
then
echo “$1 number is positive”
else
echo “$1 number is negative”
fi
```

Try it as follows:

$ chmod755 test_number.sh

$ test_number.sh **8**

8 number is positive

$ test_number.sh **-47**

-47 number is negative

$ test_number.sh

./ test_number.sh *: You must supply at least one integers*

$ test_number.sh **0**

0 number is negative

- First script checks whether command line argument is given or not, if not given then it print error message as “*./* test_number.sh *: You must give/supply one integers*“.
- if statement checks whether number of argument ($#) passed to script is not

equal (-eq) to 0, if we passed any argument to script then this if statement is false and if no command line argument is given then this if statement is true.

- The echo command i.e. echo “$0 : You must give/supply one integers”
- $0 will print Name of script
- Rest after $0: will print this error message
- And finally statement exit 1 causes normal program termination with exit status 1 (nonzero means script is not successfully run).
- The last sample run**$** test_number.sh **0** , gives output as “*0 number is negative*“, because given argument is not > 0, hence condition is false and it’s taken as negative number. To avoid this replace second if statement with **if test $1 -ge 0**.

Nested ifs concept & example

- You can write the entire if-else construct within either the body of the if statement of the body of an else statement. This is called the nesting of ifs.

Nested ifs example

- Let us look at an example,

```
$ vi nestedif.sh
osch=0

echo “1. Mac “
echo “2. Linux “
echo -n “Select your os choice [1 or 2]? “
read osch

if [ $osch -eq 1 ] ; then

    echo “You Pick up Mac “

else #### nested if i.e. if within if ######

    if [ $osch -eq 2 ] ; then
    echo “You Pick up Linux “
    else
    echo “What you don’t like Mac/Linux OS.”
    fi
fi
```

- Run the above shell script as follows:

$ chmod +x nestedif.sh

$ ./nestedif.sh

1. Mac

2. Linux

Select you os choice [1 or 2]? 1

You Pick up Mac

$ ./nestedif.sh

1. Mac

2. Linux

Select you os choice [1 or 2]? 2

You Pick up Linux

$ ./nestedif.sh

1. Mac

2. Linux

Select you os choice [1 or 2]? 3

What you don’t like Mac/Linux OS.

- Note that Second if-else construct is nested in the first else statement. If the condition in the first if statement is false the the condition in the second if statement is checked. If it is false as well the final else statement is executed.

Multilevel if-then-else concept & example

- We also have something called as multilevel if-then-else

Syntax:

```
if condition
then
  condition is zero (true - 0)
  execute all commands up to elif statement
elif condition1
then
  condition1 is zero (true - 0)
  execute all commands up to elif statement
elif condition2
then
  condition2 is zero (true - 0)
  execute all commands up to elif statement
else
  None of the above condtion,condtion1,condtion2 are true (i.e.
  all of the above nonzero or false)
  execute all commands up to fi
fi
```

Multilevel if-then-else example

- Let us look at an example

```
$ cat > Multi_if_elif.sh
```

```
#
#!/bin/sh
# Script to test if..elif…else
#
if [ $1 -gt 0 ]; then
  echo “$1 is positive”
elif [ $1 -lt 0 ]
then
  echo “$1 is negative”
elif [ $1 -eq 0 ]
then
  echo “$1 is zero”
else
  echo “Opps! $1 is not a number, give number”
fi
```

Try above script as follows:

```
$ chmod 755 Multi_if_elif.sh
$ ./Multi_if_elif 1
$ ./Multi_if_elif -5
$ ./Multi_if_elif 0
$ ./Multi_if_elif a
```

switch or case command concept

- we may wish to take different paths based upon a variable matching a series of patterns. We could use a series of if and elif statements but that would soon grow to be too long. Solution to this is case statement which can make things cleaner.

Syntax:

```
case  $variable-name  in
pattern1)   command
…
..
command;;
pattern2)   command
…
..
command;;
pattern2)   command
…
..
command;;
patternN)   command
…
..
command;;
*)          command
…
..
command;;
esac
```

- The $variable-name is compared against the patterns until a match is found.

The shell then executes all the statements up to the two semicolons that are next to each other. The default is *) and its executed if no match is found

switch or case command example

- Let us look at an example:

```
$ cat > car_rental
#
# if no vehicle name is given
# i.e. -z $1 is defined and it is NULL
#
# if no command line arg
if [ -z $1 ]
then
  rental=”*** Unknown vehicle ***”
elif [ -n $1 ]
then
# otherwise make first arg as rental
  rental=$1
fi

case $rental in
    “car”) echo “For $rental USD20 per mile”;;
    “van”) echo “For $rental USD10 per mile”;;
    “jeep”) echo “For $rental USD5 per mile”;;
    “cycle”) echo “For $rental 20 cents per mile”;;
    *) echo “Sorry, I cannot get a $rental for you”;;
esac
```

Save it by pressing CTRL+D and run it as follows:

$ chmod +x car_rental

$ car_rental van

$ car_rental car

$ car_rental Ford

$ car_rental cycle

- We will check first, that if $1(first command line argument) is given or not,
- if NOT given set value of rental variable to “*** Unknown vehicle ***”,
- if command line arg is supplied/given set value of rental variable to given value (command line arg).
- The $rental is compared against the patterns until a match is found.
- For first test run its match with van and it will show output “For van USD10 per mile.”
- For second test run it print, “For car USD20 per mile”.
- And for last run, there is no match for Ford, hence default i.e. *) is executed and it prints, “Sorry, I cannot get a Ford for you”.
- Please Note that: esac is always required to indicate end of case statement.

Loops Concept - for & while loop

- Loops allow us to take a series of commands and keep re-running them until a particular situation is reached. They are useful for automating repetitive tasks.
- Loop defined as:
- “Computer can repeat particular instruction again and again, until particular condition satisfies. A group of instruction that is executed repeatedly is called a loop.”
- We will look at for loop and while loop.
- Note that in each and every loop,
    - First, the variable used in loop condition must be initialized, then execution of the loop begins.
    - A test (condition) is made at the beginning of each iteration.
    - The body of loop ends with a statement that modifies the value of the test (condition) variable.

for loop concept

- for loops iterate through a set of values until the list is exhausted:

Syntax:

for { variable name } in { list }

do

execute one for each item in the list until the list is

not finished (And repeat all statement between do and done)

done

- Even you can use following syntax:

Syntax:

for ((expr1; expr2; expr3))

do

…..

…

repeat all statements between do and

done until expr2 is TRUE

Done

for loop example

- Let us look at an example

```
$ cat > iterate.sh
for i in 1 2 3 4 5 6 7 8 9
do
echo “Welcome $i times”
done
```

Run it above script as follows:

```
$ chmod +x iterate.sh
$ ./iterate.sh
```

The for loop first creates i variable and assigned a number to i from the list of number from 1 to 9, The shell execute echo statement for each assignment of i. (This is usually know as iteration) This process will continue until all the items in the list were not finished, because of this it will repeat 9 echo statements.

- To make you idea more clear let us consider one more example:

```
$ cat > multiply_table.sh
#!/bin/sh
#
#Script to test for loop
#
#
if [ $# -eq 0 ]
then
echo “Error - Number missing form command line argument”
```

```
echo “Syntax : $0 number”
echo “Use to print multiplication table for given number”
exit 1
fi
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
do
echo “$n * $i = `expr $i \* $n`”
done
```

Save above script and run it as:

```
$ chmod 755 multiply_table.sh
$ ./multiply_table.sh 9
$ ./multiply_table
```

- For first run, above script print multiplication table of given number where i = 1,2 … 10 is multiply by given n (here command line argument 9) in order to produce multiplication table as

```
9 * 1 = 9
9 * 2 = 18
…
..
9 * 10 = 90
```

- And for second test run, it will print message -

```
Error - Number missing form command line argument
Syntax : ./multiply_table number.sh
```

Use to print multiplication table for given number

- This happened because we have not supplied given number for which we want multiplication table,
- Hence script is showing Error message, Syntax and usage of our script. This is good idea if our program takes some argument, let the user know what is use of the script and how to used the script.
- We can also use for this way:

```
$ cat > for_example.sh
for ((  i = 0 ;  i <= 5;  i++  ))
do
  echo “Welcome $i times”
done
```

Run the above script as follows:

```
$ chmod +x for_example.sh
$ ./for_example.sh
Welcome 0 times
Welcome 1 times
Welcome 2 times
Welcome 3 times
Welcome 4 times
Welcome 5 times
```

- In above example, first expression (i = 0), is used to set the value variable i to zero.
- Second expression is condition i.e. all statements between do and done executed as long as expression 2 (i.e continue as long as the value of variable i is less than or equel to 5) is TRUE.
- Last expression i++ increments the value of i by 1 i.e. it’s equivalent to i = i + 1 statement.

Nested for loop concept & example

- Loop statement can be nested. You can nest the for loop just like nested if.

Nested for loop example

- Let us look at an example

Required output to be produced-

1 1 1 1 1

2 2 2 2 2

3 3 3 3 3

4 4 4 4 4

```
$ vi nest_for.sh
for (( i = 1; i <= 5; i++ ))      ### Outer ###
do

   for (( j = 1 ; j <= 5; j++ )) ### Inner ###
   do
   echo -n “$i “
   done

 echo ”” #### print the new line ###

done
```

Run the above script as follows:

```
$ chmod +x nest_for.sh
$ ./nest_for.sh
```

1 1 1 1 1

2 2 2 2 2

3 3 3 3 3

4 4 4 4 4

5 5 5 5 5

- Here, for each value of i the inner loop is cycled through 5 times, with the variable j taking values from 1 to 5. The inner for loop terminates when the value of j exceeds 5, and the outer loop terminals when the value of i exceeds 5.

while loop concept & example

- while loops can be much more fun! while an expression is true, keep executing these lines of code.

Syntax:

```
while [ condition ]
do
command1
command2
command3
..
….
done
```

while loop example

- Let us look at an example

```
$cat > While_loop
#!/bin/sh
#
#Script to test while statement
#
#
if [ $# -eq 0 ]
then
    echo “Error - Number missing”
    echo “Syntax : $0 number”
    echo ” Use to print multiplication table for given number”
exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
do
  echo “$n * $i = `expr $i \* $n`”
  i=`expr $i + 1`
done
```

Save it and try as

```
$ chmod 755 While_loop
$./While_loop 9
```

n=$1	Set the value of command line argument to variable n.
i=1	Set variable i to 1
while [$i -le 10]	This is our loop condition, here if value of i is less than 10 then, shell execute all statements between do and done
do	Start loop
echo “$n * $i = `expr $i \* $n`”	Print multiplication table as 9 * 1 = 9 9 * 2 = 18 …. 9 * 10 = 90, Here each time value of variable n is multiply be i.
i=`expr $i + 1`	Increment i by 1 and store result to i. (i.e. i=i+1)
done	Loop stops here if i is not less than 10 i.e. condition of loop is not true. Hence loop is terminated.

Part I - Practical Shell Script Examples Explained

Please explain following examples as much details of programming knowledge for logic development of students, you may do more echo or iteration charts to explain and develope programming skills of students.

Print incremental numbers

1 2 3 4 5 6 7

```
k=1
while test $k != 8
do
    echo “$k “
    k=`expr $k + 1`
done
```

Print decremental numbers based on user input

n..8 7 6 5 4 3 2 1

```
echo “Input number”
read k
while test $k != 0
do
    echo “$k “
    k=`expr $k - 1`
done
```

Print Fibbonaci series

what is a Fibbonaci series -

The Fibonacci series looks like following:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

The next number is found by adding up the two numbers before it.

There should be some limit to find series so we will limit - how many numbers you want to see in series by var_num variable in following example.

```
if [ $# -eq 1 ]
then
    var_num=$1
else
    echo -n “Enter a Number :”
    read var_num
fi

f1=0
f2=1

echo “The Fibonacci sequence for the Number $var_num is : “

for (( k=0;k<=var_num;k++ ))
do
    echo -n “$f1 “
    fn=$((f1+f2))
    f1=$f2
    f2=$fn
done
```

Create repeating pattern using for loop

1

22

333

4444

55555

666666

```
for (( k=1; k<=6; k++ ))
do
    for (( j=1; j<=k;  j++ ))
    do
    echo -n “$k”
    done
    echo ””
done
```

Part 2 - Practical Shell Script Examples Explained

Create star pattern using for loop

*

**

***

****

*****

******

```
for (( k=1; k<=6; k++ ))
do
    for (( j=1; j<=k;  j++ ))
    do
    echo -n ” *”
    done
    echo ””
done
```

Drawing Incremental Pattern

1

12

123

1234

12345

123456

```
for (( k=1; k<=6; k++ ))
do
    for (( j=1; j<=i;  j++ ))
    do
    echo -n “$j”
    done
    echo ””
done
```

Reverse the given number by user

For example if user enter 123 as input then 321 is printed as output.

example -

123

first iteration reminder = 3

sd = 3

first iteration digit = 12

second iteration reminder = 2

sd =2

second iteration digit = 1

sd =1

script -

```
echo “Enter number : “
read num

# store single digit
sd=0

# store number in reverse order
rev=””

# store original number
ori_num=$num

# while loop to calculate the sum of all digits
while [ $num -gt 0 ]
do
    sd=$(( $num % 10 )) # to get Remainder
    num=$(( $num / 10 ))  # to get next digit
    # store previous number and current digit in rev variable
    rev=$( echo ${rev}${sd} )
done
```

```
echo  “$ori_num in a reverse order is : $rev”
```

Process Concept & Commands

What is Processes

- When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system.
- Whenever you issue a command in UNIX, it creates, or starts, a new process. When you tried out the **ls** command to list directory contents, you started a process. A process, in simple terms, is an instance of a running program.

$ ls -lR

- **ls** command or a request to list files in a directory and all subdirectory in your current directory - It is a process.
- Process defined as:
- "*A process is program (command given by user) to perform specific Job. In Linux when you start process, it gives a number to process (called PID or process-id), PID starts from 0 to 65535.*"

Why Process required

- As You know Linux is multi-user, multitasking Os. It means you can run more than two process simultaneously if you wish. For e.g. To find how many files do you have on your system you may give command like:

$ ls / -R | wc -l

- This command will take lot of time to search all files on your system. So you can run such command in Background or simultaneously by giving command like

$ ls / -R | wc -l &

- The **ampersand** (**&**) at the end of command tells shells start process (**ls / -R | wc -l**) and run it in background takes next command immediately.
- Process & PID defined as:
- *"An instance of running command is called **process** and the number printed by shell is called **process-id (PID)**, this PID can be use to refer specific running process."*

Linux Command Related with Process

- We will now look at a few commands related to processes.

Command	About the command	Examples
ps	To see currently running process	$ ps
kill {PID}	To stop any process by PID i.e. to kill process	$ kill 1012
killall {Process-name}	To stop processes by name i.e. to kill process	$ killall httpd
ps -ag	To get information about all running process	$ ps -ag
kill 0	To stop all process except your shell	$ kill 0
linux-command &	For background processing (With &, use to put particular command and program in background)	$ ls / -R | wc -l &
ps aux	To display the owner of the processes along with the processes	$ ps aux
ps ax | grep process-U-want-to see	To see if a particular process is running or not. For this purpose you have to use ps command in combination with the grep command	For e.g. you want to see whether Apache web server process is running or not then give command $ ps ax | grep httpd
top	To see currently running processes and other information like memory and CPU usage with real time updates.	$ top Note that to exit from top command press q.

Filter Concept & Example

- When a program takes its input from another program, performs some operation on that input, and writes the result to the standard output, it is referred to as a *filter*.
- For e.g.. Suppose you have file called ‘test.txt’ with 100 lines data, And from ‘test.txt’ you would like to print contents from line number 20 to line number 30 and store this result to file called ‘test_backup.txt’ then give command:

$ tail +20 < test.txt| head -n30 >test_backup.txt

- Here **head** command is filter which takes its input from tail command (tail command start selecting from line number 20 of given file i.e. test.txt) and passes this lines as input to head, whose output is redirected to ‘hlist’ file.
- Consider one more following example

$ sort < sname | uniq > u_sname

- Here uniq is filter which takes its input from sort command and passes this lines as input to uniq; Then uniqs output is redirected to “u_sname” file.

User Defined Functions

What is function?

- Functions in Bash Scripting are a great way to reuse code.
- Think of a function as a small script within a script.
- It’s a small chunk of code which you may call multiple times within your script.
- They are particularly useful if you have certain tasks which need to be performed several times. Instead of writing out the same code over and over you may write it once in a function then call that function every time.
- Function is a series of commands.
- A return statement at end will make end of function. It is a last statement in function.

- To define function use following syntax:

Syntax:

```
function-name ( )
{
command1
command2
…..
…
command n
return
}
```

Need of function?

- Saves lot of time with reusability of code.
- Avoids rewriting of same code again and again
- Program is easier to write due to modularity.
- Program maintenance is very easy since in case of any changes to function, you will just modify function instead of whole program and it will reflect to all places whoever call that function.

Function examples

Example 1

```
#!/bin/bash
# Basic function
print_something () {
echo Hello I am a function
return
}
print_something
```

Function example 2

Passing Arguments

It is often the case that we would like the function to process some data for us. We may send data to the function in a similar way to passing command line arguments to a script. We supply the arguments directly after the function name. Within the function they are

accessible as $1, $2, etc.

arguments_example.sh

```
#!/bin/bash
# Passing arguments to a function
print_something () {
echo Hello $1
return
}
print_something Tim
print_something Kush
```

Sending unwanted output of program

- This is a special Linux file which is used to send any unwanted output from program/command. It discards all data written to it without displaying it on screen.

Syntax:

command > /dev/null

Example:

$ ls -al > /dev/null

- Output of above command is not shown on screen it will send to this special file instead.

When you can use /dev/null-

1. When you want to hide or dicard an output.
2. When you write cron jobs (shell scripts executed by Unix/Linux scheduler automatically)

Usually cron sends an email for every output from the process started with a cronjob.

So by writing the output to /dev/null you prevent being spammed if you have specified your adress in cron.

Conditional command execution with && || operators

Conditional commands execution using &&

7. Logical AND operator can be used with more than 1 conditions. Also known as &&

Syntax:

command1 && command2

command2 is executed if, and only if, command1 runs successfully which means returns an exit status of zero.

Example 1: Create folder, if the folder creation is successful, then only change the directory.

```
mkdir and_dir && cd and_dir
```

Example 2: Change to a folder, if this is success, then list the content.

```
cd and_dir && ls -al
```

Like this, you can use it at various places effectively to save your valuable time.

Conditional commands execution using || operator

- In this section we will look into OR operator. Also known as ||

Syntax:

command1 || command2

command2 is executed if and only if command1 fails which means if command1 returns a non-zero exit status.

Example 1 –

If or_dir doesn’t exists then it will create.

```
[ -d or_dir ] || mkdir or_dir
```

Example 2 –

If or_file doesn’t exists then create it by coping contents from mytest.txt

```
[ -f or_file ] || cat > or_file < mytest.txt
```

Essential Utilities

cut utility concept & example

cut utility concept

- Cut command in unix (or linux) is used to select sections of text from each line of files.

Usage: cut OPTION… [FILE]…

- Print selected parts of lines from each FILE to standard output.
- Here is the man page result for cut,

cut —help that is of interest to us.

Mandatory arguments to long options are mandatory for short options too.

-b, —bytes=LIST select only these bytes

-c, —characters=LIST select only these characters

-d, —delimiter=DELIM use DELIM instead of TAB for field delimiter

-f, —fields=LIST select only these fields;

& many more other option, please check by doing

cut—help

cut utility example

3. Let us now create a file.txt with some random data. Say we want to get the 4th column, or from column 4 to 6.

cat > myfile.txt

unix or linux os

is unix good os

is linux good os

cut -c4 myfile.txt

x

u

l

cut -c4,6 myfile.txt

xo

ui

ln

- Say we want to separate the text by a delimiter.

cut -d’:’ -f1 /etc/passwd

paste utility concept & example

paste utility concept

- Write lines consisting of the sequentially corresponding lines from each FILE, separated by TABs, to standard output.

Usage: paste [OPTION]… [FILE]…

- With no FILE, or when FILE is -, read standard input.
- Here is the man page result for paste,

paste —help that is of interest to us.

Mandatory arguments to long options are mandatory for short options too.

-d, —delimiters=LIST reuse characters from LIST instead of TABs

-s, —serial paste one file at a time instead of in parallel

—help display this help and exit

—version output version information and exit

paste utility example

- Let us create 2 simple files with some random content.

vi hello.txt

hello friend,

Hope to see you again!

vi bye.txt

how are you?

bye

- Now we will use the paste to get the result in 1 line.

paste hello.txt bye.txt

join utility concept & example

join utility concept

- For each pair of input lines with identical join fields, write a line to standard output. The default join field is the first, delimited by whitespace.

Usage: join [OPTION]… FILE1 FILE2

- Here is the man page result for join,

join —help that is of interest to us.

When FILE1 or FILE2 (not both) is -, read standard input.

-a FILENUM print unpairable lines coming from file FILENUM, where FILENUM is 1 or 2, corresponding to FILE1 or FILE2

-e EMPTY replace missing input fields with EMPTY

-i, —ignore-case ignore differences in case when comparing fields

-j FIELD equivalent to `-1 FIELD -2 FIELD’

-t CHAR use CHAR as input and output field separator

-v FILENUM like -a FILENUM, but suppress joined output lines

-1 FIELD join on this FIELD of file 1

-2 FIELD join on this FIELD of file 2

join utility example

- Let us create 2 random file and we will index each line. The idea is to get a common pointer in both files to join them.

vi hello.txt

1 hello

2 How are you?

vi bye.txt

1 hi

2 I am fine

join hello.txt bye.txt

1 Hello bye

tr utility concept & example

tr utility concept

- Translate, squeeze, and/or delete characters from standard input, We generally use it to translate.

Usage: tr[OPTION]… SET1 [SET2]

- Here is the man page result for tr,

tr —help that is of interest to us.

writing to standard output.

-c, -C, —complement use the complement of SET1

-d, —delete delete characters in SET1, do not translate

-s, —squeeze-repeats replace each input sequence of a repeated character that is listed in SET1 with a single occurrence of that character

-t, —truncate-set1 first truncate SET1 to length of SET2

tr utility example

- A quick simple look at an example. Say we want to convert letter from lower case to upper case.

tr a-z A-Z

hello

HELLO

^C

uniq utility concept & example

uniq utility concept

- Filter adjacent matching lines from INPUT (or standard input), writing to OUTPUT (or standard output).

Usage: uniq [OPTION]… [INPUT [OUTPUT]]

- Here is the man page result for uniq,

uniq —help that is of interest to us.

With no options, matching lines are merged to the first occurrence.

Mandatory arguments to long options are mandatory for short options too.

-c, —count prefix lines by the number of occurrences

-d, —repeated only print duplicate lines

-D, —all-repeated[=delimit-method] print all duplicate lines

delimit-method={none(default),prepend,separate}

Delimiting is done with blank lines.

-f, —skip-fields=N avoid comparing the first N fields

-i, —ignore-case ignore differences in case when comparing

-s, —skip-chars=N avoid comparing the first N characters

-u, —unique only print unique lines

-z, —zero-terminated end lines with 0 byte, not newline

-w, —check-chars=N compare no more than N characters in lines

—help display this help and exit

—version output version information and exit

uniq utility example

- uniq command removes duplicate adjacent lines from sorted file while sending one copy of each second file.

Examples

cat > myos.txt

unix

unix

linux

linux

mac

windows

sort myos.txt | uniq -d

will show which lines appear more than once in names file.

Part 3 - Practical Shell Script Examples Explained

shell or bash script to delete old files

ls -t | sed -e ‘1,5d’ | xargs rm

This should handle all characters (except newlines) in a file name.

This will delete all the files except the latest 5 files.

What’s going on here?

ls -t lists all files in the current directory in decreasing order of

modification time. i.e., the most recently modified files are first,

One file name per line.

sed -e ‘1,10d’ deletes the first 10 lines, ie, the 10 newest files.

xargs -d ‘\n’ rm collects each input line (without the terminating

newline) and passes each line as an argument to rm.

shell or bash script to archive files

The shell script will move the old file to tar archive and place the new one in

the current directory.

```
#!/bin/bash

oldfilename=$1

newfilename=$2

month=`date +%B`

year=`date +%Y`

prefix="example"

archivefile=$prefix.$month.$year.tar

# Check for existence of a compressed archive matching the naming convention

if [ -e $archivefile.gz ]

then

    echo "Archive file $archivefile already exists…"

    echo "Adding file '$oldfilename' to existing tar archive…"

    # Uncompress the archive, because you can't add a file to a

    # compressed archive

    gunzip $archivefile.gz

    # Add the file to the archive

    tar -rvf $archivefile $oldfilename

    # Recompress the archive

    gzip $archivefile

# No existing archive - create a new one and add the file

else

    echo "Creating new archive file '$archivefile'…"

    tar -cvf $archivefile $oldfilename
```

```
    gzip $archivefile
fi

# Update the files outside the archive

mv $newfilename $oldfilename
```

shell or bash script to rename files

```
for file in *.png

do

  mv “$file” “${file/_t.png/_tuhina.png}”

done
```

shell or bash script to copy files

```
cd /home/kishs/practice/examples

for f in *.png

do

    cp  $f /home/kishs/practice/${f%.csv}$(date +%m%d%y).csv

done
```

shell or bash script to create directory

```
echo “Enter directory name”

read dirname

if [ ! -d “$dirname” ]

then

echo “File doesn’t exist. Creating now”

mkdir ./$dirname

echo “File created”

else echo “File exists”

fi
```

shell or bash script to delete file

```
#!/bin/bash

echo “Enter filename”

read filename

if [ -f  $filename ]

then

rm filename

fi
```

shell or bash script to read file line by line

```
#!/usr/bin/bash

filename=”$1”

while read -r line

do

name=$line

echo “Name read from file - $name”

done < “$filename”
```

Note: -r Do not treat a backslash character in any special way.

Consider each backslash to be part of the input line.

shell or bash script to list files in a directory

for entry in `ls $search_dir`

do

echo $entry

done

shell or bash script to move file to another location

This script will move the files with extension .png to the specified location.

```
cd /home/kishs/practice/examples

for f in *.png

do

    mv  $f /home/kishs/practice/examples/apple/

done
```

shell or bash script to get cpu usage of a user

echo CPU: `top -b -n1 | grep “CPUs” | awk ‘{print $3}’`

Top Command - top provides an ongoing look at processor activity in real time. It displays a listing of the most CPU-intensive tasks on the system, and can provide an interactive interface for manipulating processes.

grep Command - Used to search the specified string

Awk print – Used to display the certain part of the line

Process, Signals & Traps

What is a Process and How to view Processes

Whenever you issue a command in **UNIX**, it creates, or starts, a new **process**.

When you tried out the ls command to list directory contents, you started

a **process**.

A **process**, in simple terms, is an instance of a running program.

You need to use the ps command, pstree command, and pgrep command

to view a snapshot of the current processes.

To view current process use the ps command:

ps aux | less

ps aux | grep “process-name”

pgrep looks through the currently running processes and lists the process IDs

Which matches the selection criteria to screen.

pgrep -u kish

Sending signal to Processes

Signals are software interrupts that are sent to a program to indicate an important event that has occurred. The events can vary from user requests to illegal memory access errors. Some signals, such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control.

You can send various signals to commands / process and shell scripts using the, pkill command, kill command, and killall command.

Terminating Processes

To terminate foreground process press CTRL+C to send an interrupt signal to

any running command.

To terminate unwanted background process use kill command with -9 signal.

kill -TERM pid

kill -KILL pid

To stop (suspend) a foreground process hit CTRL+Z

To resume the foreground process use the fg command, enter:

fg jobid

TERM is default signal for Kill. To list available signals, enter:

kill –l

kill command Examples

The kill command can send all of the above signals to commands and process.

Below are some signals with their integral values.

SIGHUP (1) - Hangup detected on controlling terminal or death of controlling

process.

SIGINT (2) - Interrupt from keyboard.

SIGKILL (9) - Kill signal i.e. kill running process.

SIGSTOP (19) - Stop process.

SIGCONT (18) - Continue process if stopped.

To send a kill signal to PID # 123 use:

kill -9 123

killall sends a signal to all processes running any of the specified commands .

If no signal name is specified, SIGTERM is sent.

To terminate all process (child and parent), enter:

killall processName

Shell signal values

Whenever user interrupts a signal is send to the command or the script.

Signals force the script to exit. You must know signal and their values while

writing the shell scripts. You cannot use (trap) all available signals.

Some signals can never be caught. For example, the signals SIGKILL (9)

and SIGSTOP (19) cannot be caught, blocked, or ignored.

To view list of all signals, enter:

kill –l

To view numeric number for given signal called SIGTSTP, enter:

kill -l SIGTSTP

You can also view list of signal by visiting /usr/include/linux/signal.h file:

more /usr/include/linux/signal.h

Below is the commonly used signal numbers, description and whether they

can be trapped or not:

Number	Constant	Description	Default action	Trappable (Yes/No)
0	0	Success	Terminate the process.	Yes
1	SIGHUP	Hangup detected on controlling terminal or death of controlling process. Also, used to reload configuration files for many UNIX / Linux daemons.	Terminate the process.	Yes
2	SIGINT	Interrupt from keyboard (Ctrl+C)	Terminate the process.	Yes
3	SIGQUIT	Quit from keyboard (Ctrl-\. or, Ctrl-4 or, on the virtual console, the SysRq key)	Terminate the process and dump core.	Yes

Shell signal values

4	SIGILL	Terminate the process and dump core.	Illegal instruction.	Yes
6	SIGABRT	Abort signal from abort(3) - software generated.	Terminate the process and dump core.	Yes
8	SIGFPE	Floating point exception.	Terminate the process and dump core.	Yes
9	SIGKILL	Kill signal	Terminate the process.	No
15	SIGTERM	Termination signal	Terminate the process.	Yes
20	SIGSTP	Stop typed at tty (CTRL+z)	Stop the process.	Yes

The trap statement

trap defines and activates handlers to be run when the shell receives signals

or other special conditions.

The syntax is as follows

trap arg signal

trap command signal

trap ‘action’ signal1 signal2 signaln

trap ‘action’ SIGINT

trap ‘action’ SIGTERM SIGINT SIGFPE SIGSTP

trap ‘action’ 15 2 8 20

Try the following example at a shell prompt (make sure abc.txt doesn’t exits).

Define a shell variable called $file:

file=abc.txt

Now, try to remove $file, enter:

rm $file

Sample output:

rm: cannot remove `abc.txt’: No such file or directory

Now trap rm command:

trap “rm $file; exit” 0 1 2 3 15

Display list of defined traps, enter:

trap

How to clear trap

To clear a trap use the following syntax:

trap - signal

trap - signal1 signal2

For example, set a trap for rm command:

file=abc.txt

trap ‘rm $file’ 1 2 3 15

trap

To clear SIGINT (2), enter:

trap - SIGINT

trap

To clear all traps, enter:

trap - 1 2 3 15

trap

trap statements to catch signals and handle errors in a script

```
#!/bin/bash

# capture CTRL+C, CTRL+Z and quit singles using the trap

trap ‘echo “CTRL-C disabled.”’ SIGINT

trap ‘echo “Cannot terminate this script.”’ SIGQUIT

trap ‘echo “CTRL-Z disabled.”’ SIGTSTP

a=1

while [ $a -eq 0 ]

do

clear

echo ” OPTIONS AVAILABLE”

echo “————————-”

echo “1. Display date and time.”

echo “2. Display what users are doing.”

echo “3. Exit”

# get and read input from the user

read -p “Enter your choice [ 1 -4 ] ” choice

case $choice in 1)

echo “Today is $(date)”

read -p “Press [Enter] key to continue…” readKey ;;




2) w

read -p “Press [Enter] key to continue…” readKey ;;



3) echo “Bye!” ; a=0 ;;
```

*) echo “Not a valid option”

read -p “Press [Enter] key to continue…” readKey ;;

esac

done

Save and close the file. Run it.

Get here - Free Access to Online tutorial

Click on below link or copy paste in your browser -

https://www.udemy.com/unix-bash-shell-scripting-tutorial-shell-programming-examples-linux/?couponCode=k1

or

CLICK HERE