VotaryTech – Shell Scripting

```
$ test 10 -eq 10
$ echo $?
```

```
$ [ 10 -eq 10 ]
$ echo $?
```

```
$ ((10==10))
$ echo $?
```

- true, false

- Arithmetic Operators (+, -, *, /, %)
- Relational Operators (-eq, -ne, -gt, -lt, -ge, -le)
- Boolean or Logical Operators (!, -o, -a)
- String Operators (=, !=, -z, str)
- File Test Operators (-d, -f, -r, -w, etc ….)

Operator	Example
+	`expr $a + $b` will give 30
-	`expr $a - $b` will give -10
*	`expr $a \* $b` will give 200
/	`expr $b / $a` will give 2
%	`expr $b % $a` will give 0

```
$ result=`expr 1 + 2`
$ echo $result

$ result=$((1+2))
$ echo $result
```

- Say, v1 = 10, v2 = 20

Operator	Example
-eq	[$v1 -eq $v2] is false
-ne	[$v1 -ne $v2] is true.
-gt	[$v1 -gt $v2] is false
-lt	[$v1 -lt $v2] is true.
-ge	[$v1 -ge $v2] is false
-le	[$v1 -le $v2] is true.

- Say, v1 = 10, v2 = 20

Operator	Example
!	[! false] is true.
-o	[$v1 -lt 20 -o $v2 -gt 100] is true.
-a	[$v1 -lt 20 -a $v2 -gt 100] is false.

- Say, v1 = “hi” and v2 = “unix”

Operator	Example
=	[$v1 = $v2] is false
!=	[$v1 != $v2] is true
-z	[-z $v1] is false
str	[$v1] is true

```
v1="hi"
v2="UNIX"
v3=""

[ $v1 ]
echo $?
[ $v3 ]
echo $?
[ -z $v1 ]
echo $?
[ -z $v3 ]
echo $?
```

```
v4="hi"
[ $v1 = $v4 ]
echo $?
[ $v1 != $v4 ]
echo $?
```

- Let file="poem.txt" with the following permissions:

```
-rw-rw-r--. 1 guest guest 138 Mar  3 17:49 poem.txt
```

opr	Description	Example
-b	file Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c	file Checks if file is a character special file if yes then condition becomes true.	[-b $file] is false.
-d	file Check if file is a directory if yes then condition becomes true.	[-d $file] is false
-f	file Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.
-p	file Checks if file is a named pipe if yes then condition becomes true.	[-p $file] is false.
-t	file Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.	[-t $file] is false.
-r	file Checks if file is readable if yes then condition becomes true.	[-r $file] is true.
-w	file Check if file is writable if yes then condition becomes true.	[-w $file] is true.

```
file="poem.txt"

[ -f $file ]
echo $?
[ -d $file ]
echo $?
[ -w $file ]
echo $?
[ -x $file ]
echo $?
[ -s $file ]
echo $?
```

```
if [ expression ]
then
  Statement(s) to be executed if expression is true
fi
```

```
echo "Enter hours "
read hrs
echo "Enter minutes"
read mins

mins=`expr $mins + 1`

if [ $mins -gt 59 ]
then
        mins=0
        hrs=`expr $hrs + 1`
fi
```

```
if [ $hrs -gt 23 ]
then
        hrs=0
fi

echo "$hrs:$mins"
```

```
if [ expression ]
then
  Statement(s) to be executed if expression is true
else
  Statement(s) to be executed if expression is not true
fi
```

```
echo "Enter body temperature "
read temp

if [ $temp -gt 99 ]
then
        echo "You got fever"
else
        echo "Normal temperature"
fi
```

```
if [ expression 1 ]
then
  Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
  Statement(s) to be executed if expression 2 is true
else
  Statement(s) to be executed if no expression is true
fi
```

```
echo "Enter 3 numbers "
read num1 num2 num3
if [ $num1 -gt $num2 -a $num1 -gt $num3 ]
then
        biggest=$num1
elif [ $num2 -gt $num3 ]
then
        biggest=$num2
else
        biggest=$num3
fi
echo "Biggest = $biggest"
```

```
case expr in

pattern1)

Statement(s) to be executed if pattern1 matches

;;

pattern2)

Statement(s) to be executed if pattern2 matches

;;

pattern3)

Statement(s) to be executed if pattern3 matches

;;

*)

Default Statement(s)

;;

esac
```

```
echo "Choose service provider [1.Airtel 2.Vodafone 3.AT&T] :"
read opt
case $opt in
1) echo "You chose Airtel"
;;
2) echo "You chose Vodafone"
;;
3) echo "You chose AT&T"
;;
*) echo "Invalid option"
;;
esac
```

```
echo "Enter a file name "
read filename

if [ ! -f $filename ]
then
        echo "Sorry, file does'nt exist"
        exit
fi

echo "Enter cmd [cat/wc/sort/rm/ls] to run on the file : "
read cmd
```

```
case $cmd in
"cat"|"wc"|"sort") $cmd $filename
;;
"rm") $cmd -i $filename
;;
"ls") $cmd -l $filename
;;
*) echo "Error: cmd not in selection list"
;;
esac
```

- Loops enable you to execute a set of commands repeatedly .
  - while
  - for
  - until

```
while expr
do
  Statement(s) to be executed if expr is
  true
done
```

```
num=1
while [ $num -le 10 ]
do
        echo $num
        num=`expr $num + 1`
done
```

```
echo "Enter a number"
read num

fact=1
while [ $num -gt 1 ]
do
        fact=`expr $fact \* $num`
        num=`expr $num - 1`
done

echo "Factorial = $fact"
```

```
for var in word1 word2 ... wordN
do
  Statement(s) to be executed for every word.
done
```

```
for text in "The Pursuit of happiness" Year 2006 Rating 7.9 IMDb
do
        echo $text
done
```

```
dirs=0
files=0
for val in `ls` ;do
        if [ -d $val ]; then
                dirs=`expr $dirs + 1`
        fi
        if [ -f $val ];  then
                files=`expr $files + 1`
        fi
done
echo "Total # of dirs= $dirs”
echo "Total # of regular files= $files”
```

```
#!/bin/bash
# for_1to10.sh
echo "Print numbers from 1 to 10"

for ((num=1;num<=10;num++))
do
        echo "Number: $num"
done

for ((num1=1,num2=20;num1<=num2;num1++,num2--))
do
        echo "Number: $num1 $num2"
done
```

```
#!/bin/bash
#for_loop.sh

for num in 11 2 13 34 5
do
    echo -n $num
    echo -n " "
done
echo

for num in {1..10..3}
do
        echo $num
done
```

```
for num in {10..1..3}
do
        echo $num
Done

echo "Printing odd numbers from 1 to 10"
echo "Number: "
for num in {1..10..-2}
do
        echo -n " $num "
done

echo
```

```
until expr
do
  Statement(s) to be executed until expr is false
done
```

```
num=1
until [ $num -gt 10 ]
do
        echo $num
        num=`expr $num + 1`
done
```

* ? [ ] ' " \ $; & ( ) | ^ < > new-line space tab

Quoting	Description
Single quote	All special characters between these quotes lose their special meaning.
Double quote	Most special characters between these quotes lose their special meaning with these exceptions $ ` \$ \` \" \\
Backslash	Any character immediately following the backslash loses its special meaning.
Backquote	Anything in between back quotes would be treated as a command and would be executed.

- Arrays
- Functions

```
array_name[index]=value
```

```
vehicle[0]="Bus"
vehicle[1]="Car"
vehicle[2]="Bike"
vehicle[3]="Cycle"
vehicle[4]="cart"

echo "This is a  bullock ${vehicle[4]}"
echo "All vehicles: ${vehicle[*]}"
echo "again all vehicles: ${vehicle[@]}"
```

```
# Array Initialization
fruits=(apple mango grapes)
echo All elements are
echo ${fruits[*]}

echo "${fruits[1]} is the king of fruits"
```

```
function_name () {
    list of commands
}
```

```
Hello ()
{
  echo "Hello World"
}
# Invoke your function
Hello
```

```
# function-arg.sh
Hello ()
{
  echo "Hi, $1 $2"
}
# Invoke your function
Hello Its Me!
```

Arguments

```
# Function Returning Value
Sum ()
{
 return `expr $1 + $2`
}
# Invoke your function
Sum 17 6
echo Sum is $?
```

- File Input and Output Operators (<, <<, >, >>)

```
#!/bin/bash
#here_doc.sh
cat <<End-of-message
----------------------------------
This is line 1 of the message.
This is line 2 of the message.
This is line 3 of the message.
This is line 4 of the message.
This is the last line of the message.
----------------------------------
End-of-message
```

```
cat << -ENDOFMESSAGE
        This is line1 of message
        This is line2 of message
        This is line3 of message
-ENDOFMESSAGE

wc -w  << EOF
        This is a test.
        Apple juice.
        100% fruit juice and no added sugar, colour or preservative.
EOF
```

- Shift
  - Shifts the script argument to left on every invoke of shift command

```
#!/bin/bash
# shift_example.sh
echo $*;
echo “Number of args: $#”
shift ; echo “Number of args: $#”
shift ; echo “Number of args: $#”
shift ; echo “First arg is : $1”
```

```
$ ./shift_example.sh welcome to Shell scripting training guys
```

```
#!/bin/bash
#input as
#./shift_test.sh 10 11 12 13 14 15 hi 16 17 28 39
echo "Printing all args: $*"
echo "Args: $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11"
echo "Number of args: $#"
shift
echo "Args: $1 $2 $3 $4 $5 $6 $7 $8 $9 $10"
echo "Number of args: $#"
shift
echo "Args: $1 $2 $3 $4 $5 $6 $7 $8 $9"
echo "Number of args: $#"
shift
```

- This command is used to check valid command line argument are passed to script. Usually used in while loop.

```
getopts {optsring} {variable1}
```

- getopts is used by shell to parse command line argument.
- "optstring contains the option letters to be recognized; if a letter is followed by a colon, the option is expected to have an argument, which should be separated from it by white space.
- Each time it is invoked, getopts places the next option in the shell variable variable1, When an option requires an argument, getopts places that argument into the variable OPTARG. On errors getopts diagnostic messages are printed when illegal options or missing option arguments are encountered. If an illegal option is seen, getopts places ? into variable1."

```
#!/bin/bash
getopts abcl opt
case "$opt" in
        a) echo "You entered a"
           ;;
        b) echo "You entered b"
           ;;
        c) echo "You entered c"
           ;;
        l) echo "You entered l"
           ;;
        *) echo "Invalid choice"
           ;;
esac

$ ./getopts_test1.sh -a
$ ./getopts_test1.sh -a -b
$ ./getopts_test1.sh -x
```

```
#!/bin/bash
while getopts abcl opt
do
        case "$opt" in
                a) echo "You entered a"
                ;;
                b) echo "You entered b"
                ;;
                c) echo "You entered c"
                ;;
                l) echo "You entered l"
                ;;
                *) echo "Invalid choice"
                ;;
        esac
done

$ ./getopts_test2.sh -a
$ ./getopts_test2.sh -a -b -l -x
```

```
#!/bin/bash
while getopts a:b:cl opt
do
        case "$opt" in
                a) echo "You entered a"
                avalue="$OPTARG"
                echo "avalue is $avalue"
                ;;
                b) echo "You entered b"
                bvalue="$OPTARG"
                echo "bvalue is $bvalue"
                ;;
                c) echo "You entered c"
                ;;
                l) echo "You entered l"
                ;;
                *) echo "Invalid choice"
                ;;
        esac
done
```

```
$ ./getopts_test3.sh -b abc -a 123 -c -x -l
```

- Set – useful for large scripts
  - -v, Starts debugging. Shows every statement. Substitutes values into variables
  - -x, Show statement execution status
  - +v, stop debugging.
  - +x, stop debugging.

- Cut - Cuts the character from the lines of file

```
$ cut -c 3-7,15-20 info.txt
```

```
$ cut -c 3-7,13,14,17-20 info.txt
```

```
$ cut -d":" -f1,6 /etc/passwd | tail -5
```

- Cmp
  - Compares two files byte by byte
  - Displays the location of the first mismatch
  - Syntax,

    $ cmp file1.txt file2.txt
- Diff
  - Displays file diffrerences
  - Syntax

    $ diff file1.txt file2.txt
- Uniq
  - Fetches one copy of each line
  - Requires a sorted file as input
  - Syntax,

  $ uniq file1.txt
  - Non repeated lines from a file

  $ uniq –u file1.txt
- Paste
  - Pastes vertically rather than horizontally
  - Displays two or more files adjacently by pasting them
  - Syntax,

    $ paste file1.txt file2.txt
  - Paste uses the tab as the default delimiter, we can specify one or more delimiter with –d option

    $ paste –d file1.txt file2.txt
- Join
  - A join of the two relations specified by the lines of file1 and file2
  - Files are joined on a common key field (column) that should exist in both files.
  - Both files must be sorted on the key field in the same order
  - Paste uses the tab as the default delimiter, we can specify one or more delimiter with –d option

1	Rahim
2	Mohan Kumar
3	Rihan
4	Oliva

1	68.2
2	66.79
3	45.12
4	59

```
$ join -d"\t" file1.txt file2.txt
```

- Translates range of characters.

- *Syntax:*
  tr {pattern-1} {pattern-2}

```
$ tr “a-z’” “A-Z”
hello
HELLO
Hi
HI
^d
```

```
$ echo “Hello world!” | tr “a-z’” “A-Z”
HELLO WORLD!
$
```

- AWK was initially developed in 1977 by Alfred Aho, Peter Weinberger, and Brian Kernighan, from whose initials the language takes its name.
- Used for data manipulation
- *Syntax:*
  awk 'pattern action' {file-name}

```
$ cat cities
hyderabad 040 telangana
bangalore 080 karnataka
bombay 022 maharastra
amravathi 02251 maharastra
tukkuguda 040 telangana
new-delhi 011 delhi
trivendrum 045 kerala
coimbatore 0422 tamilnadu
guntur 0863 andhra
kolkatta 033 bengal
chennai 044 tamilnadu
amravathi 0863 andhra
vijayawada 0866 andhra
$ awk '/andhra/ {print $1,$2}' cities
guntur 0863
amravathi 0863
vijayawada 0866
$
```

- Use awk to handle complex task such as calculation, database handling, report creation etc.
- General Syntax of awk with awk program file:
  Syntax:
  awk -f {awk program file} filename
- awk Program contains are something as follows:
- Pattern {
  action 1
  action 2
  action N
  }
- If pattern is match for each line then given action is taken. Pattern can be regular expressions.

- Math using awk

```
$ cat math.awk
{
        print $1 "+" $2 "=" $1+$2
        print $1 "-" $2 "=" $1-$2
        print $1 "*" $2 "=" $1*$2
        print $1 "/" $2 "=" $1/$2
        print $1 "%" $2 "=" $1%$2
}
```

```
$ awk -f math.awk
3 2
3+2=5
3-2=1
3*2=6
3/2=1.5
3%2=1
8 3
8+3=11
8-3=5
8*3=24
8/3=2.66667
8%3=2
$
```

- Math using awk with pattern file as argument

```
$ cat math_data.txt
3       2
6       4
$ awk -f math.awk math_data.txt
3+2=5
3-2=1
3*2=6
3/2=1.5
3%2=1
6+4=10
6-4=2
6*4=24
6/4=1.5
6%4=2
$
```

- User defined variables.

```
$ cat math2
{
 no1=$1
 no2=$2
 ans=$1+$2
 print no1 " + " no2 " = "ans
}
$ awk -f math2
5 2
5 + 2 = 7
6 66
6 + 66 = 72
```

- User defined variables.

```
BEGIN {
print "========================================="
print "\t\tINVOICE"
print "========================================="
print "Code\tName\tQty\tMRP\tCost"
print "========================================="
}
END {
print "\t\tVISIT AGAIN"
print "========================================="
}

{
 cost=$3*$4
 code=$1
 name=$2
 qty=$3
 price=$4
 print code "\t" name "\t" qty "\t" price "\t Rs. " cost
 print "-----------------------------------------"
}
```

```
p04 proj 1 50000
p02 marked 15 32
p03 board 2 5000
p01 pen 5 12.5
p05 tv 1 28000
```

```
$ awk -f bill invoice
=========================================
                INVOICE
=========================================
Code    Name    Qty     MRP     Cost
=========================================
p04     proj    1       50000    Rs. 50000
-----------------------------------------
p02     marked  15      32       Rs. 480
-----------------------------------------
p03     board   2       5000     Rs. 10000
-----------------------------------------
p01     pen     5       12.5     Rs. 62.5
-----------------------------------------
p05     tv      1       28000    Rs. 28000
-----------------------------------------
                VISIT AGAIN
=========================================
```

- If..else statement

General syntax of if condition is as follows:

Syntax:

```
if ( condition )
{
      Statement 1
      Statement 2
      Statement N
      if condition is TRUE
}
else
{
      Statement 1
      Statement 2
      Statement N
     if condition is FALSE
}
```

```
$ cat math3
{
 op1=$1
 opr=$2
 op2=$3

 if( opr == "+" )
 {
  printf "%d %c %d = %d\n",op1,opr,op2,op1+op2
 }
 else
 {
  printf "Meant for addition only\n"
 }
}
```

Loops

```
{
 n=$1
 rem=0

 while (n>1)
 { rem=n%10
   printf "%d", rem
   n/=10
 }

 printf "\nInput another number or Ctrl+D quit : "
 printf "\n"
}
```

- NR and NF are predefined variables of awk which means Number of input Record, Number of Fields in input record respectively.

- List of awk defined variables

awk Variable	Meaning
FILENAME	Name of current input file
RS	Input record separator character (Default is new line)
OFS	Output field separator string (Blank is default)
ORS	Output record separator string (Default is new line)
NF	Number of input record
NR	Number of fields in input record
OFMT	Output format of number
FS	Field separator character (Blank & tab is default)

- SED is a stream editor.
- A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline). SED works by making only one pass over the input(s), and is consequently more efficient
- sed is used to edit (text transformation) on given stream i.e a file or may be input from a pipeline.
- *Syntax:*
  sed {expression} {file}

```
India's milk is good.
coffee Barista is good.
coffee is better than the tea.
Greetings of the day!
                --Barista
```

```
$ sed '/coffee/s//milk/g' café > milk
```

- Deleting blank lines
  - $ sed '/^$/d' demofile1
  - As you know pattern /^$/, match blank line and d, command deletes the blank line.

```
$ sed '/^$/d' cafe
```

- Unix signals (software interrupts) can be sent as asynchronous events to shell scripts, just as they can to any other program. The default behaviour is to ignore some signals and immediately exit on others.
- Scripts may detect signals and divert control to a handler function or external program.
- This is often used to perform clean-up actions before exiting, or restart certain procedures.
- Execution resumes where it left off, if the signal handler returns.
- Signal traps must be set separately inside of shell functions.
- Signals can be sent to a process with kill.
- Signal Action
  - Default Action
  - Catch the signal
  - Ignore the signal
- USAGE: trap handler signal1,signal2…
  - handler is a command to be read (evaluated first) and executed on receipt of the specified sigs.
  - Signals can be specified by name or number (see kill(1)) e.g. HUP, INT, QUIT, TERM. A Ctrl-C at the terminal generates a INT.
- A handler of - resets the signals to their default values
- A handler of ‘‘(null) ignores the signals

- There are several methods of delivering signals to a program or script. One of the most common is for a user to type CONTROL-C or the INTERRUPT key while a script is executing.

- When you press the Ctrl+C key a SIGINT is sent to the script and as per defined default action script terminates.

```
$ kill ‘ ‘ pid #
```

- The other common method for delivering signals is to use the kill command

```
$ kill -signal pid
```

- EXIT
  - the handler is called when the function exits, or when the whole script exits. The exit signal has value 0.

SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl + C).
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl + D).
SIGFPE	8	Issued if an illegal mathematical operation is attempted
SIGKILL	9	If a process gets this signal it must quit immediately and will not perform any clean-up operations
SIGALRM	14	Alarm Clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default).

```
#!/bin/bash
#signal_test1.sh
trap exithandler       TERM
trap ' '               QUIT
trap inthandler        INT
trap usrhandler        SIGUSR1 SIGUSR2

exithandler()
{
      echo "Received SIGTERM"
      exit 1
}

inthandler()
{
      echo "Received SIGINT"
}
usrhandler()
{
      echo "Received SIGUSR1 or SIGUSR2"
}
```

```
while true ;
do
      echo "Welcome"
      sleep 3
      ps -f
done
```

```
#!/bin/bash
#signal_test2.sh
trap exithandler      TERM
trap ' ‘              ABRT
trap inthandler       INT

exithandler()
{
      echo "Received SIGTERM"
      exit 1
}

inthandler()
{
      echo "Received SIGINT"
      echo "Now making SIGINT to default action"
      trap - INT # Restore the "INT" signal handler to the
default action
}
```

```
ps -f
cntr=1
echo "Welcome to Signal Handler
Program"
while true ;
do
      echo -n $cntr
      sleep 3
      cntr=`expr $cntr + 1`
      echo -n " "
done
```