Bash recap

What is bash script?

The **bash script** is a **shell programming language**. Generally, we run many types of shell commands from the terminal by typing each command separately that require time and efforts. If we need to run the same commands again then we have to execute all the commands from the terminal again. But using a bash script, **we can store many shell command statements in a single bash file and execute the file any time by a single command.** Many system administration related tasks, program installation, disk backup, evaluating logs, etc. can be done by using proper bash script.

What are the advantages of using bash scripts?

Bash script has many advantages which are described below:

- It is **easy** to use and learn.
- Many **manual tasks** that need to run frequently can be done **automatically** by writing a bash script.
- The **sequence of multiple shell commands** can be executed by a single command.
- Bash script written in one Linux operating system can easily execute in other Linux operating system. So, it is **portable**.
- Debugging in bash is easier than other programming languages.
- Command-line syntax and commands that are used in the terminal are similar to the commands and syntax used in bash script.
- Bash script can be used to link with other script files.

Mention the disadvantages of bash scripts

Some disadvantages of bash script are mentioned below:

- It works **slower than other languages**.
- The improper script can damage the entire process and generate a **complicated error**.
- It is **not suitable** for developing a **large** and **complex** application.
- It **contains less data structure** compare to other standard programming languages.

What types of variables are used in bash?

Two types of variables can be used in bash script. These are:

System variables (a.k.a **environment** variables)
The variables which are pre-defined and maintained by the Linux operating system are called system variables. These type of variables are always used **by an uppercase letter**. The **default** values of these variables can be **changed** based on requirements.

`set`, `env` and `printenv` commands can be used to print the list of system variables.

Example:

```
#!/bin/bash
 # Printing System Variables

 #Print Bash shell name
 echo $BASH

 # Print Bash shell Version
 echo $BASH_VERSION

 # Print Home directory name
 echo $HOME
```

User-defined variable (a.k.a local variables)

The variables which are created and maintained by users are called **user-defined variables**. They are also called local variables. These types of variables can be **declared by using lowercase or uppercase** or both uppercase and lowercase letters.

- But it is better to avoid using all uppercase letter to differentiate the variables from system variables.

Example:

```
#!/bin/bash

 num=100
 echo $num
```

- `export` command is useful to make local variables accessible in other sub-shells.

How to declare and delete variables in bash?

The variable can be declared in bash **by data type or without data type**. If any bash variable is declared without `declare` command, then the **variable will be treated as a string**. Bash variable is declared with **declare** command to define the data type of the variable at the time declaration.

–r**, -i, -a, -A, -l, -u, -t** and **–x** options can be used with **declare** command to declare a variable with different data types.

Example:

```
#!/bin/bash

 #Declare variable without any type
 num=10

 #Values will be combined but not added
 result=$num+20
 echo $result

 #Declare variable with integer type
 declare -i num=10

 #Values will be added
 declare -i result=num+20
 echo $result
```

`unset` command is used to remove any bash variable. The variable will be inaccessible or undefined after using **unset** command.

- Bash is considered weakly-typed language.
- Options to define data types are limited and don't support all types of data. For example, **float** data type can't be declared by using **declare** command.

Example:

```
#!/bin/bash

 str="Linux Hint"
 echo $str
 unset $str
 echo $str
```

How to add comments in a bash script?

Single line and multi-line comments can be used in bash script. '**#**' symbol is used for single-line comment. **'<<'** symbol with a delimiter are used for adding multi-line comment. The latter is known as HereDoc trick.

Example:

```
#!/bin/bash
 #Print the text [Single line comment]
 echo "Bash Programming"
 <<-'TOKEN'
 Calculate the sum
 Of two numbers [multiline comment]
 TOKEN
 num=25+35
 echo $num
```

How can you combine strings in a bash script?

String values can be combined in bash in different ways. Normally, the string values are combined by placing together but there are other ways in bash to combine string data.

Example:

```
#!/bin/bash
#Initialize the variables
 str1="PHP"
 str2="Bash"
 str3="Perl"

 # Print string together with space
 echo $str1 $str2 $str3

 #Combine all variables and store in another variable
 str="$str1, $str2 and $str3"

 #Combine other string data with the existing value of the string
 str+=" are scripting languages"
 #Print the string
 echo $str
```

Which commands are used to print output in bash?

`echo` and `printf` commands can be used to print output in bash. `echo` command is used to print the simple output and `printf` command is used to print the formatted output.

Example:

```
#!/bin/bash

 #Print the text
 echo "Welcome to LinuxHint"
 site="linuxhint.com"
 #Print the formatted text
 printf "%s is a popular blog site\n" $site
```

How to take input from the terminal in bash?

`read` command is used in a bash script to take input from the terminal.

```
Example:**

#!/bin/bash
#Print message
echo "Enter your name"
#Take input from the user
read name
# Print the value of $name with other string
echo "Your name is $name"
```

How to use command-line arguments in bash?

Command-line arguments are read by `$1`, `$2`, `$3` ... `$n` variables. Command-line argument values are provided in the terminal when executing the bash script. **$1** is used to read the first argument, **$2** is used to read the second argument and so on.

Example:

```
#!/bin/bash
#Check any argument is provided or not
 if [[ $# -eq 0 ]]; then
   echo "No argument is given."
   exit 0
 fi
#Store the first argument value
 color=$1
# Print the argument with other string
 printf "You favorite color is %s\n" $color
 echo $?
 echo $0
 echo $$
```

How to read the second word or column from each line of a file?

The second word or column of a file can be read in a bash script by using different bash commands easily, such as `awk`, `sed` etc. Here, the use of **`awk`** is shown in the following example.
Example: Suppose, course.txt file contains the following content and we have printed only the second word of each line of this file.

```
CSE201  Java Programming
CSE303  Data Structure
CSE408  Unix Programming

#!/bin/bash
# The following script will print the second word of each line from course.txt
file.
# the output of cat command will pass to awk command that will read the second word
# of each line.
echo `cat course.txt | awk '{print $2}'`
```

How to declare and access an array variable in bash?

Both **numeric** and **associative arrays** are supported by a bash script. An array variable can be declared with and without declare command. **–a** option is used with declare command to define a numeric array and **–A** option is used with declare statement to define an associative array in bash. Without declare command, **the numeric array can be defined only in bash**.

Example:

```
#!/bin/bash

# Declare a simple numeric array
 arr1=( CodeIgniter Laravel ReactJS )

# Print the first element value of $arr1
 echo ${arr1[0]}

# Declare a numeric array using declare command
 declare -a arr2=( HTML CSS JavaScript )

# Print the second element value of $arr2
 echo ${arr2[1]}

# Declare an associative array using declare statement
 declare -A arr3=( [framework]=Laravel [CMS]=Wordpress [Library]=JQuery )

# Print the third element value of $arr3
 echo ${arr3[Library]}

All elements of an array can be accessed by using any loop or ‘*’ symbol as an
array index.
```

How can conditional statements be used in bash?

The most common conditional statement in most programming languages is **the if-elseif-else** statement. The syntax of **if-elseif-else** statement in bash is a little bit different from other programming languages. **‘If’** statement can be declared in two ways in a bash script and every type of **‘if’** block must be closed with **‘fi’**. **‘if’** statement can be defined by third brackets or first brackets like other programming languages.

Syntax:

```
 A.

if [ condition ]; then
 statements
 fi

B.

if [ condition ]; then
 statements 1
 else
 statement 2
 fi

C.

if [ condition ]; then
 statement 1
 elif [ condition ]; then
 statement 2
 ….
 else
 statement n
 fi
```

Example:

```
#!/bin/bash

# Assign a value to $n
 n=30
# Check $n is greater than 100 or not
 if [ $n -lt 100 ]; then
   echo "$n is less than 100"
# Check $n id greater than 50 or not
 elif [ $n -lt 50 ]; then
   echo "$n is less than 50"
 else
   echo "$n is less than 50"
 fi
```

How to compare values in bash?

Six types of comparison operators can be used in bash to compare values. There are two ways to use these operators in bash depending on the data type. These are mentioned below.

String Comparison	Integer Comarison	Description
==	-eq	It is used to check equality
!=	-ne	It is used to check inequality
<	-lt	It is used check the first value is less than the second value or not
>	-gt	It is used check the first value is greater than the second value or not
<=	-le	It is used check the first value is less than or equal to the second value or not
>=	-ge	It is used check the first value is greater than or equal to the second value or not

Example:

```
#!/bin/bash
# Initialize $n
 n=130
 o="even"
# Check $n is greater than or equal to 100 or not using '–ge'.
 if [ $n -ge 100 ]; then
    echo "$n is greater than or equal to 100"
 else
    echo "$n is less than 100"
 fi
# Check $n is even or odd using '==' operator
 if (( $o == "even" )); then
    echo "The number is even"
 else
    echo "The number is odd"
 fi
```

Which conditional statement can be used as an alternative to if-elseif-else statements in bash?

`case` statement can be used as an alternative tp **if-elseif-if** statement. `case` block is closed by `esac` statement in bash. No `break` statement is used inside '**case**' block to terminate from the block.

Syntax:

```
case in
 Match pattern 1) commands;;
 Match pattern 2) commands;;
 ……
 Match pattern n) commands;;
 esac
```

What different types of loops can be used in bash?

Three types of loops are supported by a bash script. These are **while, for** and **until** loops. Loops in bash check the condition at the start of the loop. **While** loop works until the condition remains true and **until** loop works until the condition remains false. There are two ways to use **for** loop. One is general **for** loop that contains three parts and another is **for-in** loop. The uses of these three loops are shown in the following example.

Example:

```bash
#!/bin/bash
# Initialize $n
 n=5
# Calculate the square of 5-1 using while loop
 while [ $n -gt 0 ]
 do
   sqr=$((n*n))
   echo "The square of $n is $sqr"
   ((n--))
 done

# Calculate the square of 5-1 using for loop
 for (( i=5; i>0; i-- ))
 do
   sqr=$((i*i))
   echo "The square of $i is $sqr"
 done
```

How to cut and print some part of a string data in bash?

Example:

```bash
#!/bin/bash
# Initialize a string value into $string
 string="Python Scripting Language"
# Cut the string value from the position 7 to the end of the string
 echo ${string:7}
# Cut the string value of 9 characters from the position 7
 echo ${string:7:9}
# Cut the string value from 17 to 20
 echo ${string:17:-4}
```

Mention some ways to perform arithmetic operations in bash?

Arithmetic operations can be done in multiple ways in bash. **'let', 'expr', 'bc'** and **double brackets** are the most common ways to perform arithmetic operations in bash. The uses of these commands are shown in the following example.

Example:

```
!/bin/bash
# Calculating the subtraction by using expr and parameter expansion
 var1=$( expr 120 - 100 )
# print the result
 echo $var1
# Calculate the addition by using let command
 let var2=200+300
# Print the rsult
 echo $var2
# Calculate and print the value of division using ‘bc’ to get the result
# with fractional value
 echo "scale=2; 44/7" | bc
# Calculate the value of multiplication using double brackets
 var3=$(( 5*3 ))
# Print the result
 echo $var3
```

How to check a directory exists or not using bash?

Bash has many test commands to check if a file or directory exists or not and the type of the file. **'-d'** option is used with a directory path as a conditional statement to check if the directory exists or not in bash. If the directory exists, then it will return true otherwise it will return false.

Example:

```
!/bin/bash
# Assign the directory with path in the variable, $path
 path="/home/ubuntu/temp"
# Check the directory exists or not
 if [[ -d "$path" ]]; then
# Print message if the directory exists
   echo "Directory exists"
 else
 # Print message if the directory doesn’t exist
   echo "Directory not exists"
 fi
```

- Another option is to use `test`

  ```
  if test -d dirA; then echo "true"; fi
  test -f file1 && echo "true" || echo "false"
  ```

  Many options are available in bash to test file. Some options are mentioned below.

Option	Description
-f	It is used to test the file exists and it is a regular file.
-e	It is used to test the file exists only.
-r	It is used to test the file exists and it has read permission.
-w	It is used to test the file exists and it has to write permission.
-x	It is used to test the file exists and it has execution permission.
-d	It is used to test the directory exists.
-L	It is used to test the file exists and It is a symbolic link.
-S	It is used to test the file exists and It is a socket.
-b	It is used to test the file is a block device.
-s	It is used to check the file is not zero sizes.
-nt	It used to check the content of the first file is newer than the second file. For example, file1 -nt file2 indicates that file1 is newer than file2.
-ot	It used to check the content of the first file is older than the second file. For example, file1 -ot file2 indicates that file1 is older than file2.
-ef	It is used to check that two hard links refer to the same file. For example, flink1 -ef flink2 indicates that flink1 and flink2 are hard links and both refer to the same file.

How can a bash script be terminated without executing all statements?

Using **'exit'** command, a bash script can be terminated without executing all statements. The following script will check if a particular file exists or not. If the file exists, then it will print the total characters of the file and if the file does not exist then it will terminate the script by showing a message.

Example:

```
#!/bin/bash

 # Initialize the filename to the variable, $filename
 filename="course.txt"
 # Check the file exists or not by using -f option
 if [ -f "$filename" ]; then
   # Print message if the file exists
   echo "$filename exists"
 else
   # Print message if the file doesn't exist
   echo "$filename not exists"
   # Terminate the script
   exit 1
 fi

 # Count the length of the file if the file exists
 length=`wc -c $filename`

 # Print the length of the file
 echo "Total characters - $length"
```

What are the uses of break and continue statements in bash?

break statement is used to terminate from a loop without completing the full iteration based on a condition and **continue** statement is used in a loop to omit some statements based on a condition. The uses of **break** and **continue** statements are explained in the following example.

Example:

```
#!/bin/bash
 # Initialize the variable $i to 0 to start the loop
 i=0
 # the loop will iterate fot 10 times
 while [ $i -le 10 ]
 do
   # Increment the value $i by 1
   (( i++ ))
   # If the value of $i equal to 8 then terminate the loop by using 'break'
statement
   if [ $i -eq 8 ]; then
      break;
   fi
   # If the value of $i is greater than 6 then omit the last statement of the loop
   #  by using continue statement
   if [ $i -ge 6 ]; then
```

```
      continue;
   fi
   echo "the current value of i = $i"
 done

 # Print the value of $i after terminating from the loop
 echo "Now the value of i = $i"
```

How to make a bash file executable?

Executable bash files can be made by using **'chmod'** command. Executable permission can be set by using **'+x'** in **chmod** command with the script filename. Bash files can be executed without the explicit **'bash'** command after setting the execution bit for that file.

Example:

```
# Set the execution bit
 $ chmod +x filename.sh

 # Run the executable file
 $ ./filename.sh
```

How can you print a particular line of a file in bash?

There are many ways to print a particular line in bash. How the **'awk', 'sed'** and **'tail'** commands can be used to print a particular line of a file in bash is shown in the following example.

Example:

```
#!/bin/bash

 # Read and store the first line from the file by using `awk` command with NR
variable
 line1=`awk '{if(NR==1) print $0}' course.txt`
```

```
# Print the line
echo $line1

# Read the second line from the file by using `sed` command with -n option
line2=`sed -n 2p course.txt`
# Print the line
echo $line2

# Read the last line from the file by using `tail` command with -n option
line3=`tail -n 1 course.txt`
# Print the file
echo $line3
```

What is IFS?

IFS is a special shell variable. The full form of **IFS** is Internal Field Separator,
 it acts as delimiter to separate the word from the line of text. It is mainly used for splitting a string, reading a command, replacing text etc.

Example:

```
#!/bin/bash
# Declare ':' as delimiter for splitting the text
 IFS=":"
# Assign text data with ':' to $text
 text="Red:Green:Blue"
# for loop will read each word after splitting the text based on IFS
 for val in $text; do
   # Print the word
   echo $val
 done
```

How to find out the length of a string data?

`expr` ** `wc` ** and `awk` commands can be used to find out the length of a string data in bash. **'expr'** and **'awk'** commands use **length** option, **'wc'** command uses **'–c'** option to count the length of the string.

Example:

The uses of the above commands are shown in the following script.

```
#!/bin/bash
 # Count length using `expr` length option
 echo `expr length "I like PHP"`
 # Count length using `wc` command
 echo "I like Bash" | wc -c
 # Count length using `awk` command
 echo "I like Python" | awk '{print length}'
```

How to run multiple bash script in parallel?

Multiple bash scripts can be executed in parallel by using **nohup** command. How multiple bash files can be executed in parallel from a folder is shown in the following example.

Example:

```

 # for loop will read each file from the directory and execute in parallel
 for script in dir/*.sh
 do
   nohup bash "$script" &
 done
```

Reference: https://linuxhint.com/bash_scripting_interview_questions/