1.Shell scripting interview questions

Shell is an environment in which we can run our commands, programs, and shell scripts.shell is a group of commands, variables , loops , conditions, that can be execute together to form a shell script.

2.What is $ in command prompt?

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time −

$date

Thu Jun 25 08:30:19 MST 2009

3.differnt types of shells?

Shell Types

In Unix, there are two major types of shells −

* **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
* **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

* Bourne shell (sh)
* Korn shell (ksh)
* Bourne Again shell (bash)
* POSIX shell (sh)

The different C-type shells follow −

* C shell (csh)
* TENEX/TOPS C shell (tcsh)

4. which shell types you used

Bash shell - If you are using a Bourne-type shell, the **$** character is the default prompt

5.extension of shell script?

.sh extension

6.how to execute shell script?

Filename.sh

./filename.sh

How to debug and executeshell script?

sh -x filename.sh

here x means debug

write a sample shell script ?

#!/bin/bash

pwd

ls

what is read command?

**read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

Example:

#!/bin/bash

# Author : Zara Ali

# Copyright (c) Tutorialspoint.com

# Script follows here:

echo "What is your name?"

read PERSON

echo "Hello, $PERSON"

what is she bang in shell script ?

#!/bin/bash

What is variable?

Variable is a name that it can stored the value or name

a=10;

b=”sunni”

how to declare and assign the variable in shell script?

a=10;

echo $a

what is readonly variable?

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME −

[Live Demo](http://tpcg.io/tawT1C)

#!/bin/sh

NAME="Zara Ali"

readonly NAME

NAME="Qadiri"

The above script will generate the following result −

/bin/sh: NAME: This variable is read only.

What is unset variable?

## Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command −

unset variable\_name

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works −

#!/bin/sh

NAME="Zara Ali"

unset NAME

echo $NAME

The above example does not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

What are special variables ?

The following table shows a number of special variables that you can use in your shell scripts −

|  |  |
| --- | --- |
| **Sr.No.** | **Variable & Description** |
| 1 | **$0**  The filename of the current script. |
| 2 | **$n**  These variables correspond to the arguments with which a script was invoked. Here **n** is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| 3 | **$#**  The number of arguments supplied to a script. |
| 4 | **$\***  All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. |
| 5 | **$@**  All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| 6 | **$?**  The exit status of the last command executed. |
| 7 | **$$**  The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| 8 | **$!**  The process number of the last background command |

What is array ?

Array is a group of elements and can hold the multiple values at the same time

Syntax: array\_name[index]=value

How to declare the array?

A=(“srinu” “hanuma” “venky” “harsh” “ganga Bhaskar” “prasad anna” “chinnu bro”)

How to acess the array using quotes with individual

B=(“sunni” “Sunitha” “sirigineedi”)

echo ${B[\*]}

how to acess the array using single qotes

c={“sunni Sunitha”}

echo ${c[@]}

what are the various operators in shell script

the following operators −

* Arithmetic Operators
* Relational Operators
* Boolean Operators
* String Operators
* File Test Operators

Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/unix/unix-arithmetic-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / (Division) | Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| = (Assignment) | Assigns right operand in left operand | a = $b would assign value of b into a |
| == (Equality) | Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != (Not Equality) | Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[ $a == $b ]** is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

Relational Operators

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable **a** holds 10 and variable **b** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/unix/unix-relational-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-eq** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a -eq $b ] is not true. |
| **-ne** | Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true. | [ $a -ne $b ] is true. |
| **-gt** | Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true. | [ $a -gt $b ] is not true. |
| **-lt** | Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true. | [ $a -lt $b ] is true. |
| **-ge** | Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true. | [ $a -ge $b ] is not true. |
| **-le** | Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true. | [ $a -le $b ] is true. |

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[ $a <= $b ]** is correct whereas, **[$a <= $b]** is incorrect.

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/unix/unix-boolean-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **!** | This is logical negation. This inverts a true condition into false and vice versa. | [ ! false ] is true. |
| **-o** | This is logical **OR**. If one of the operands is true, then the condition becomes true. | [ $a -lt 20 -o $b -gt 100 ] is true. |
| **-a** | This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false. | [ $a -lt 20 -a $b -gt 100 ] is false. |

String Operators

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then −

[Show Examples](https://www.tutorialspoint.com/unix/unix-string-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **=** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a = $b ] is not true. |
| **!=** | Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true. | [ $a != $b ] is true. |
| **-z** | Checks if the given string operand size is zero; if it is zero length, then it returns true. | [ -z $a ] is not true. |
| **-n** | Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true. | [ -n $a ] is not false. |
| **str** | Checks if **str** is not the empty string; if it is empty, then it returns false. | [ $a ] is not false. |

File Test Operators

We have a few operators that can be used to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on −

[Show Examples](https://www.tutorialspoint.com/unix/unix-file-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-b file** | Checks if file is a block special file; if yes, then the condition becomes true. | [ -b $file ] is false. |
| **-c file** | Checks if file is a character special file; if yes, then the condition becomes true. | [ -c $file ] is false. |
| **-d file** | Checks if file is a directory; if yes, then the condition becomes true. | [ -d $file ] is not true. |
| **-f file** | Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true. | [ -f $file ] is true. |
| **-g file** | Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true. | [ -g $file ] is false. |
| **-k file** | Checks if file has its sticky bit set; if yes, then the condition becomes true. | [ -k $file ] is false. |
| **-p file** | Checks if file is a named pipe; if yes, then the condition becomes true. | [ -p $file ] is false. |
| **-t file** | Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true. | [ -t $file ] is false. |
| **-u file** | Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true. | [ -u $file ] is false. |
| **-r file** | Checks if file is readable; if yes, then the condition becomes true. | [ -r $file ] is true. |
| **-w file** | Checks if file is writable; if yes, then the condition becomes true. | [ -w $file ] is true. |
| **-x file** | Checks if file is executable; if yes, then the condition becomes true. | [ -x $file ] is true. |
| **-s file** | Checks if file has size greater than 0; if yes, then condition becomes true. | [ -s $file ] is true. |
| **-e file** | Checks if file exists; is true even if file is a directory but exists. | [ -e $file ] is true |

What are the types of shell decision making ?

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here −

* The **if...else** statement
* The **case...esac** statement

What is if..fi statement

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

## Syntax

if [ expression ]

then

Statement(s) to be executed if expression is true

fi

example

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

fi

if [ $a != $b ]

then

echo "a is not equal to b"

fi

what is if…else...fi statement

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

Syntax

if [ expression ]

then

Statement(s) to be executed if expression is true

else

Statement(s) to be executed if expression is not true

fi

The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

Example

The above example can also be written using the *if...else* statement as follows

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

else

echo "a is not equal to b"

fi

what is if…elif…if?

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax

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

elif [ expression 3 ]

then

Statement(s) to be executed if expression 3 is true

else

Statement(s) to be executed if no expression is true

fi

This code is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then *else* block is executed.

Example

[Live Demo](http://tpcg.io/i5Hw31)

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

elif [ $a -gt $b ]

then

echo "a is greater than b"

elif [ $a -lt $b ]

then

echo "a is less than b"

else

echo "None of the condition met"

fi

what is **case...esac** ?

Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

Syntax

The basic syntax of the **case...esac** statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

case word 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 condition to be executed

;;

esac

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command ;; indicates that the program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

Example

[Live Demo](http://tpcg.io/fuk3nK)

#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in

"apple") echo "Apple pie is quite tasty."

;;

"banana") echo "I like banana nut bread."

;;

"kiwi") echo "New Zealand is famous for kiwi."

;;

esac