Operating System

**LAB-5**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

UNIX/LINUX Shell programming

Decision Making, Loop, Shell Function

**Objective: Understand and implement Decision making, Loop, and Shell function in Linux Shell.**

Linux Shell supports conditional statements which are used to perform different actions based on different conditions. Here we will explain following two decision making statements −

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

## **The if...else statements:**

If else statements are useful decision making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of if..else statement −

[**if...fi statement**](https://www.tutorialspoint.com/unix/if-fi-statement.htm)**:**

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

Here Shell *expression* is evaluated. If the resulting value is *true*, given*statement(s)* are executed. If *expression* is *false* then no statement would be not executed. Most of the times you will use comparison operators while making decisions.

Give you attention on the spaces between braces and expression. This space is mandatory otherwise you would get syntax error.

If **expression** is a shell command then it would be assumed true if it return 0 after its execution. If it is a boolean expression then it would be true if it returns true.

## **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

This will produce following result −

a is not equal to b

[**if...else...fi statement**](https://www.tutorialspoint.com/unix/if-else-statement.htm)**:**

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

## **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

Here Shell *expression* is evaluated. If the resulting value is *true*, given*statement(s)* are executed. If *expression* is *false* then no statement would be not executed.

## **Example**

If we take above example then it can be written in better way using *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

This will produce following result −

a is not equal to b

[**if...elif...else...fi statement**](https://www.tutorialspoint.com/unix/if-elif-statement.htm)**:**

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

There is nothing special about this code. It 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 non of the condition is true then *else* block is executed.

## **Example**

#!/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

This will produce following result −

a is less than b

**Loops**:

Loops are a powerful programming tool that enable you to execute a set of commands repeatedly. In this tutorial, you would examine the following types of loops available to shell programmers −

[**The while loop**](https://www.tutorialspoint.com/unix/while-loop.htm)

The while loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

## **Syntax**

while command

do

Statement(s) to be executed if command is true

done

Here Shell *command* is evaluated. If the resulting value is *true*, given*statement(s)* are executed. If *command* is *false* then no statement would be not executed and program would jump to the next line after done statement.

## **Example**

Here is a simple example that uses the while loop to display the numbers zero to nine −

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

This will produce following result −

0 1 2 3 4 5 6 7 8 9

[**The for loop**](https://www.tutorialspoint.com/unix/for-loop.htm)

The for loop operate on lists of items. It repeats a set of commands for every item in a list.

## **Syntax**

for var in word1 word2 ... wordN

do

Statement(s) to be executed for every word.

done

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

## **Example**

Here is a simple example that uses for loop to span through the given list of numbers −

#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9

do

echo $var

done

This will produce following result −

0 1 2 3 4 5 6 7 8 9

Following is the example to display all the files starting with **.bash** and available in your home. I'm executing this script from my root −

#!/bin/sh

for FILE in $HOME/.bash\*

do

echo $FILE

done

This will produce following result −

/root/.bash\_history

/root/.bash\_logout

/root/.bash\_profile

/root/.bashrc

[**The until loop**](https://www.tutorialspoint.com/unix/until-loop.htm)

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

## **Syntax**

until command

do

Statement(s) to be executed until command is true

done

Here Shell *command* is evaluated. If the resulting value is *false*, given*statement(s)* are executed. If *command* is *true* then no statement would be not executed and program would jump to the next line after done statement.

## **Example**

Here is a simple example that uses the until loop to display the numbers zero to nine −

#!/bin/sh

a=0

until [ ! $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

This will produce following result −

0 1 2 3 4 5 6 7 8 9

[**The select loop**](https://www.tutorialspoint.com/unix/select-loop.htm)

You would use different loops based on dfferent situation. For example while loop would execute given commands until given condition remains true where as until loop would execute until a given condition becomes true.

The *select* loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices.

## **Syntax**

select var in word1 word2 ... wordN

do

Statement(s) to be executed for every word.

done

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

For every selection a set of commands would be executed with-in the loop. This loop was introduced in ksh and has been adapted into bash. It is not available in sh.

## **Example**

Here is a simple example to let the user select a drink of choice −

#!/bin/ksh

select DRINK in tea cofee water juice appe all none

do

case $DRINK in

tea|cofee|water|all)

echo "Go to canteen"

;;

juice|appe)

echo "Available at home"

;;

none)

break

;;

\*) echo "ERROR: InvUniversityd selection"

;;

esac

done

The menu presented by the select loop looks like the following −

$./test.sh

1) tea

2) cofee

3) water

4) juice

5) appe

6) all

7) none

#? juice

Available at home

#? none

$

So far you have looked at creating loops and working with loops to accomplish different tasks. Sometimes you need to stop a loop or skip iterations of the loop.

In this tutorial you will learn following two statements used to control shell loops −

* The **break** statement
* The **continue** statement

## **The infinite Loop**

All the loops have a limited life and they come out once the condition is false or true depending on the loop.

A loop may continue forever due to required condition is not met. A loop that executes forever without terminating executes an infinite number of times. For this reason, such loops are called infinite loops.

## **Example**

Here is a simple example that uses the while loop to display the numbers zero to nine −

#!/bin/sh

a=10

until [ $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

This loop would continue forever because a is alway greater than or equal to 10 and it would never become less than 10.

## **The break statement**

The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

## **Syntax**

The following **break** statement would be used to come out of a loop −

break

The break command can also be used to exit from a nested loop using this format −

break n

Here **n** specifies the nth enclosing loop to exit from.

## **Example**

Here is a simple example which shows that loop would terminate as soon as a becomes 5:

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

if [ $a -eq 5 ]

then

break

fi

a=`expr $a + 1`

done

This will produce following result −

0 1 2 3 4 5

Here is a simple example of nested for loop. This script breaks out of both loops if var1 equals 2 and var2 equals 0 –

#!/bin/sh

for var1 in 1 2 3

do

for var2 in 0 5

do

if [ $var1 -eq 2 -a $var2 -eq 0 ]

then

break 2

else

echo "$var1 $var2"

fi

done

done

This will produce following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from inner loop as well.

1 0

1 5

## **The continue statement**

The **continue** statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

## **Syntax**

continue

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

continue n

Here n specifies the nth enclosing loop to continue from.

## **Example**

The following loop makes use of continue statement which returns from the continue statement and start processing next statement −

#!/bin/sh

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS

do

Q=`expr $NUM % 2`

if [ $Q -eq 0 ]

then

echo "Number is an even number!!"

continue

fi

echo "Found odd number"

done

This will produce following result −

Found odd number

Number is an even number!!

Found odd number

Number is an even number!!

Found odd number

Number is an even number!!

Found odd number

**Functions**

Functions enable you to break down the overall functionUniversityty of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

## **Creating Functions**

To declare a function, simply use the following syntax −

function\_name () {

list of commands

}

The name of your function is function\_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.

## **Example**

Following is the simple example of using function −

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World"

}

# Invoke your function

Hello

When you would execute above script it would produce following result −

$./test.sh

Hello World

$

## **Pass Parameters to a Function**

You can define a function which would accept parameters while calling those function. These parameters would be represented by $1, $2 and so on.

Following is an example where we pass two parameters *Iqra* and *University* and then we capture and print these parameters in the function.

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World $1 $2"

}

# Invoke your function

Hello Iqra University

This would produce following result −

$./test.sh

Hello World Iqra University

$

## **Returning Values from Functions**

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the**return** command whose syntax is as follows −

return code

Here *code* can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

## **Example**

Following function returns a value 1 −

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World $1 $2"

return 10

}

# Invoke your function

Hello Iqra University

# Capture value returnd by last command

ret=$?

echo "Return value is $ret"

This would produce following result −

$./test.sh

Hello World Iqra University

Return value is 10

$

## **Nested Functions**

One of the more interesting features of functions is that they can call themselves as well as call other functions. A function that calls itself is known as a *recursive function*.

Following simple example demonstrates a nesting of two functions −

#!/bin/sh

# Calling one function from another

number\_one () {

echo "This is the first function speaking..."

number\_two

}

number\_two () {

echo "This is now the second function speaking..."

}

# Calling function one.

number\_one

This would produce following result −

This is the first function speaking...

This is now the second function speaking...

**Task:**

1. Sort integer Array in ascending order.
2. Find shortest value in integer array.
3. Make shell function which can find out that input number is prime or not
4. Make shell function which can eliminate all odd numbers from integer array.
5. Make shell function which can find the last three prime number lesser than 100.