2. How to define and use functions in Linux shell script?

→ • Function is a reusable block of code. often we put repeated code in a function and call the function from various places.

• Library is a collection of functions. We can define commonly used function in a library and other scripts can use them without duplicating code.

→ Defining the function:

• Function can be defined in two ways –

ⓐ Eg: function name follow the Parentheses

```
Simple_function () {
    for ((i=0; i<5; ++i)) do
        echo -n " " $i " ";
    done
}
Simple_function
```

→ We call our function by merely invoking its name. but we must define it before executing it.

- the above example will Print just some numbers.

0 1 2 3 4.

ⓑ Another way to ~~de~~ define function by the function keyword and omit the Parenthesis.

Eg:

```
function simple_function {
    for ((i=0; i<3; ++i)) do
        echo -n " " $i " ";
    done
```

Output of above example is 0, 1, 2, 3.

- Since we used that the body could be any compound command. We can even omit the curly braces.

eg:

```
function simple_for_loop()
    for ((i=0; i<5; ++i)) do
        echo -n " " "$i" " ";
    done.
```

→ In the above example, the output is the same as before.

→ However, this only works when we're executing instructions inside the for loop. That's because the looping construct acts as a compound command.

→ Passing Input Arguments:

eg:

```
function simple_inputs() {
    echo "This is the first argument [$1]"
    echo "This is the second argument [$2]"
    echo "Calling function with $# arguments"
}
simple_inputs one 'two three'
```

- So, in above example, we print the two inputs from positional parameters.
- After that, we also print the total number of arguments using a special parameter.

O/P →

```
This is the first argument [one]
This is the second argument [two three]
Calling function with 2 arguments.
```

→ If we do not return an exit code, then Bash will return the exit status of the last command in our function.

- For Calculating Sum of 2 numbers:

```
Sum = 0
function simple_outputs () {
    Sum = $(($1 + $2))
}

simple_outputs 1 2
echo "Sum is $Sum"
```

→ In this above example we used global variable to store the actual result.

O/P → Sum is 3

→ Using Argument References:

As of Bash 4.3, we can pass an input argument by reference and then modify its state inside the function.

Eg:

```
function ref_outputs () {
    declare -n Sum_ref = $3
    Sum_ref = $(($1 + $2))
}
```

- Let's drill down into this example to understand it better.
  First, we declare a nameref variable that stores the third argument's name.
- As a second step, we use this variable as a left-hand-side operand on the assignment operation.
- We can do this because this variable is a reference to the third argument.
- Finally, call our function by specifying as positional

argument is both the inputs and output.

eg: ref_outputs 1 2 Sum
echo "Sum is $Sum"

And we'll see the same result.
Sum is 3.

→ Parameterised functions:

```
$ function add    # use add as function name
{
a=$1              # assigning 1st. value & display
b=$2              # assigning 2nd value & displa
add=$((a+b))      # Resulted value stored in add
echo $add         # Display add value.
}
```

O/P→ add 3 4
7

→ help function: It displays help information.

$ help function

O/P→ function: function name { commands ; } or name()
{ commands ; }
Define shell function.
!
!
Returns Success unless NAME is readly