**Shell functions:**

Functions enable to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual tasks when needed.

Using functions to perform repetitive tasks is an excellent way to create **code reuse**.

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

### Example

Following example shows the use of function

#!/bin/sh

# Define function here

Hello () {

echo "Hello World"

}

# Invoke your function

Hello

## Pass Parameters to a Function

We can define a function that will accept parameters while calling the function. These parameters would be represented by **$1**, **$2** and so on.

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

#!/bin/sh

# Define function here

Hello () {

echo "Hello World $1 $2"

}

# Invoke your function

Hello Zara Ali

## Returning Values from Functions

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

return code

### Example

Following function returns a value 10

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World $1 $2"

return 10

}

# Invoke your function

Hello Zara Ali

# Capture value returnd by last command

ret=$?

echo "Return value is $ret"

## Nested Functions

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

#!/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

### Return Values

We use the keyword **return** to indicate a return status.

* Remember that the variable [$?](https://ryanstutorials.net/bash-scripting-tutorial/bash-variables.php#others) contains the return status of the previously command or function.

1. ./return\_status\_example.sh
2. Hello Mars
3. Hello Jupiter
4. The previous function has a return value of 5

#### return\_hack.sh

#!/bin/bash

*# Setting a return value to a function*

lines\_in\_file () {

cat $1 | wc -l

}

num\_lines=$( lines\_in\_file $1 )

echo The file $1 has $num\_lines lines in it.

Let's break it down:

* This command will print the number of lines in the file referred to by $1.
* We use command substitution to take what would normally be printed to the screen and assign it to the variable **num\_lines**

./lab01 myfile.txt

cat myfile.txt

Tomato

Lettuce

Capsicum

The file myfile.txt has 3 lines in it.

## Variable Scope

Scope refers to which parts of a script can see which variables. By default a variable is **global**. This means that it is visible everywhere in the script. We may also create a variable as a **local** variable. When we create a local variable within a function, it is only visible within that function. To do that we use the keyword **local** in front of the variable the first time we set it's value.

**local var\_name=<var\_value>**

It is generally considered good practice to use local variables within functions so as to keep everything within the function contained. This way variable are safer from being inadvertently modified by another part of the script which happens to have a variable with the same name (or vice versa).

#### local\_variables.sh

1. #!/bin/bash
2. *# Experimenting with variable scope*
3. var\_change () {
4. local var1='local 1'
5. echo Inside function: var1 is $var1 : var2 is $var2
6. var1='changed again'
7. var2='2 changed again'
8. }
9. var1='global 1'
10. var2='global 2'
11. echo Before function call: var1 is $var1 : var2 is $var2
12. var\_change
13. echo After function call: var1 is $var1 : var2 is $var2
14. ./local\_variables.sh
15. Before function call: var1 is global 1 : var2 is global 2
16. Inside function: var1 is local 1 : var2 is global 2
17. After function call: var1 is global 1 : var2 is 2 changed again

Scope of Variables

Programmers used to other languages may be surprised at the scope rules for shell functions. Basically, there is no scoping, other than the parameters (**$1**, **$2**, **$@**, etc).

#!/bin/sh

myfunc()

{

echo "I was called as : $@"

x=2

}

### Main script starts here

echo "Script was called with $@"

x=1

echo "x is $x"

myfunc 1 2 3

echo "x is $x"

The script, when called as **./lab45 a b c**, gives the following output:

Script was called with a b c

x is 1

I was called as : 1 2 3

x is 2

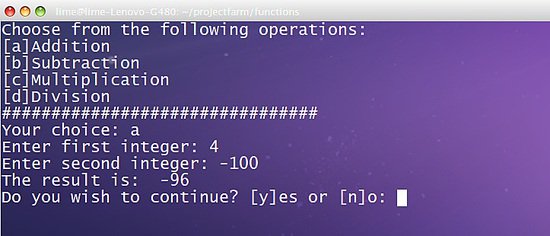
The variable **x**, however, is effectively a global variable **myfunc** changed it, and that change is still effective when control returns to the main script.

..

**Write a Shell program to find the sum of two numbers using function programming.**

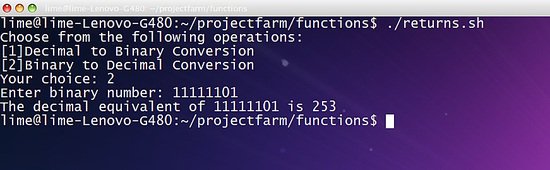
For example, consider the code below:

*#!/bin/bash  
while(true)  
do  
    clear  
    printf "Choose from the following operations: \n"  
    printf "[a]ddition\n[b]Subtraction\n[c]Multiplication\n[d]Division\n"  
    printf "################################\n"  
    read "Your choice: " choice  
    case $choice in  
    [aA])  
        read "Enter first integer: " int1  
        read "Enter second integer: " int2  
        res=$((int1+int2))  
   
    ;;  
    [bB])  
        read "Enter first integer: " int1  
        read "Enter second integer: " int2  
    res=$((int1-int2))  
   
    ;;  
    [cC])  
        read "Enter first integer: " int1  
        read "Enter second integer: " int2  
        res=$((int1\*int2))  
   
    ;;  
    [dD])  
        read "Enter first integer: " int1  
        read "Enter second integer: " int2  
        res=$((int1/int2))  
   
    ;;  
    \*)  
        res=0  
        echo "wrong choice!"  
    esac  
  
    echo "The result is: " $res  
    read "Do you wish to continue? [y]es or [n]o: " ans  
    if [ $ans == 'n' ]  
        then  
         echo "Exiting the script. Have a nice day!"  
        break  
    else  
        continue  
    fi  
  
done*

[](https://www.howtoforge.com/images/bash_scripting_part5/big/img03.jpg)

Let's have another example that uses functions, passes parameters to it and returns value.

*#!/bin/bash  
  
   
clear(){  
    clear  
}  
  
bin(){  
    bin1=$(echo "obase=2;$1"|bc)  
    echo $bin1  
}  
  
dec(){  
    dec1=$(echo "ibase=2;$1"|bc)  
    return $dec1  
}  
  
########Main#########  
    printf "Choose from the following operations:\n[1]Decimal to Binary Conversion\n"  
    printf "[2]Binary to Decimal Conversion\n"  
    read "Your choice: " op  
    case $op in  
  
    1)  
        read "Enter integer number: " int  
        bin $int  
    ;;  
  
    2)  
        read "Enter binary number: " int  
        dec $int  
        echo "The decimal equivalent of $int is $?"  
    ;;  
  
    \*)  
        echo "Wrong Choice!"  
    esac*

[](https://www.howtoforge.com/images/bash_scripting_part5/big/img09.jpg)

[](https://www.howtoforge.com/images/bash_scripting_part5/big/img10.jpg)

The given example converts a given input to both binary or decimal value using obase and ibase command. The line $(echo "obase=2;$1"|bc) converts a given decimal value to binary digit and store it to bin1 variable. Next we displayed the value of $bin1 by using echo command.