**Functions**

When your scripts start to become very large, you may tend to notice that you are repeating code more often in your scripts. You have the ability to create functions inside of your script to help with code reuse. Writing the same code in multiple sections of your script can lead to severe maintenance problems. When you fix a bug in a section of code you need to be sure that all sections of code that are repeated will also have those fixes. A function is a block of code that can be called from other parts of your script. It can have parameters passed to it as if it were a separate script itself. As an example, we will create a function called **logit**, which will take two parameters, a level and a message. The level will be a number between 1 and 3 that will indicate the severity of the log message. The level of messages that you want to view will be passed in the command line of the script.

#!/bin/sh

# logit function declaration.

logit()

{

MSG\_LEVEL=$1

# Shifts the position of the parameters over one place.

shift

if [ "$MSG\_LEVEL" -ge 1 ] && [ "$MSG\_LEVEL" -le 3 ]; then

if [ "$LEVEL" -eq 1 ] && [ "$MSG\_LEVEL" -ge 1 ]; then

echo "Msg Level $MSG\_LEVEL: $@"

elif [ "$LEVEL" -eq 2 ] && [ "$MSG\_LEVEL" -ge 2 ]; then

echo "Msg Level $MSG\_LEVEL: $@"

elif [ "$LEVEL" -eq 3 ] && [ "$MSG\_LEVEL" -ge 3 ]; then

echo "Msg Level $MSG\_LEVEL: $@"

fi

fi

}

#Load the log level from the command line...

LEVEL=$1

# Call the function a couple of times.

logit 1 Logit Test one

logit 2 Logit Test two

logit 3 Logit Test three

logit 4 Logit Test four

Let's save this file and run it several times with different input levels...

$ **test.sh 1**

Msg Level 1: Logit Test one

Msg Level 2: Logit Test two

Msg Level 3: Logit Test three

$ **test.sh 2**

Msg Level 2: Logit Test two

Msg Level 3: Logit Test three

$ **test.sh 3**

Msg Level 3: Logit Test three

$ **test.sh 4**

$

In this script we use the unix command **shift** to shift the position of the **$@** pointer to the right. Remember that **$@** points to the entire command line. In the case of functions, the command line parameters are localized to the function, so for the first call to logit, the following parameters are passed "1 Logit Test one". Inside the logit function, the $@ variable contains the data "1 Logit Test one". We already read the first parameter ("1") into our MSG\_LEVEL variable, so we want to use the other parameters are the text message. access them all together without the first parameter, we call **shift**, which moves the $@ variable one to the right so that it now contains the text "Logit Test one".

Another interesting thing in this script is that for the $LEVEL variable, the scope does not matter. Remember that in a unix script, a varible is just an environment variable and is accessible from anywhere.

You can write recursive functions in a shell script as well. Consider the factoral:

#!/bin/sh

fac()

{

if [ "$1" -gt 1 ]; then

NEXT=`expr $1 - 1`

REC=`fac $NEXT`

PROD=`expr $1 \\* $REC`

echo $PROD

else

echo 1

fi

}

echo "Enter a number: "

read NUM

echo "$NUM! = `fac $NUM`"

A recursive function is a function that calls itself. Notice at line 7 in the script, it calls the fac function inside itself. Recursive functions have the possibility of going in an endless loop and crashing if you do not code them right, so only use a recursive function if you really understand what you are trying to do.

# **Define a bash shell function**

Shell functions are a way to group commands for later execution using a single name for the group. They are executed just like a regular command. When the name of a shell function is used as a simple command name, the list of commands associated with that function name will be executed.

Functions are declared using this syntax:

[ function ] *name* () { *command-list*; }

This defines a shell function named *name*. The reserved word function is optional.

If a function called *name* already exists it will be overwritten. If you already have an [alias](https://ss64.com/bash/alias.html) called *name*, this alias should be removed with unalias before creating the function.

## function body

The *body* of the function is the *command-list* between { and }. This list is executed whenever *name* is specified as the name of a command.

Note that for historical reasons, the curly braces that surround the body of the function must be separated from the body by blanks or newlines. This is because the braces are reserved words and are only recognized as such when they are separated by whitespace. Also, the *command-list* must be terminated with a semicolon or a newline.

If the function reserved word is supplied, the parentheses { } are optional.   
The curly parentheses { } indicate a compound command, it is possible to write a [one line function](http://www.catonmat.net/blog/bash-functions/) without these, or to use normal ( ) parenthesis which will spawn the function in a subshell rather than execute it in the current environment.

## Executing a function

Shell functions are executed in the current shell context; no new process is created to interpret them.

When a function is executed, the arguments to the function become the positional parameters during its execution.  
The special parameter `#' that expands to the number of positional parameters is updated to reflect the change.   
Positional parameter 0 is unchanged.   
The FUNCNAME variable is set to the name of the function while the function is executing.

If the builtin command return is executed in a function, the function completes and execution resumes with the next command after the function call. When a function completes, the values of the positional parameters and the special parameter `#' are restored to the values they had prior to the function's execution. If a numeric argument is given to return, that is the function's return status; otherwise the functions's return status is the exit status of the last command executed before the return.

The exit status of a function is the exit status of the last command executed in the body.

To view a bash function's definition use the [type](https://ss64.com/bash/type.html) command: type myfunctionname

Variables local to the function can be declared with the local builtin. These variables are visible only to the function and the commands it invokes.

Functions can be recursive. No limit is placed on the number of recursive calls.

## Introduction

A function, also known as a subroutine in programming languages is a set of instructions that performs a specific task for a main routine [1]. It allows programmers to break a complicated and lengthy code to small sections which can be called whenever needed. Each function needs to be called by a main routine in order to run, thus, it is isolated with other parts of your code and this creates an easy way of code testing. Additionally, functions can be called anytime and repeatedly, this allows you reuse, optimize and minimize your codes. Like most programming languages, the bash shell also supports functions.

### General Syntax:

1. Syntax 1:

*function function\_name  
{  
    ##### set of commands  
}*

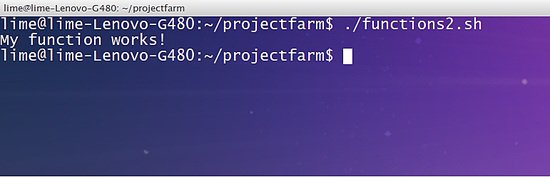
1. Syntax 2:

*function\_name()  
{  
    #### set of commands  
}*

## Creating Functions

The bash supports two structures for functions. In using the first syntax, you have to use the keyword function, followed by your function name and open and close parentheses and curly braces to separate the contents of your functions to your main routine. You will find this syntax familiar if you have a background in PHP because functions in PHP are declared in the same way. The other syntax only consists of a function name, open and close parentheses and curly braces.

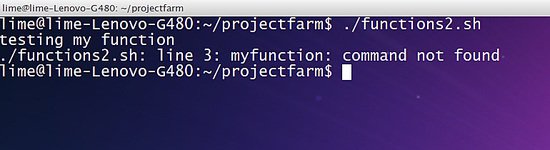
*#!/bin/bash  
myfunction(){  
    echo "My function works!"  
}  
myfunction*

[](https://www.howtoforge.com/images/bash_scripting_part5/big/img01.jpg)

I have used the second syntax in our example. After creating the function *myfunction*, it was then invoked by calling its function name to our main routine. The main routine will be anywhere in our script that was not defined as part of our function.

Now let's rearrange our code to test whether functions can be declared anywhere in our script. Consider the code below:

*#!/bin/bash  
echo "testing my function"  
myfunction  
  
myfunction(){  
    echo "My function works!"  
}*

[](https://www.howtoforge.com/images/bash_scripting_part5/big/img02.jpg)

The line 3 in the above code returns a command not found error. This only means that:

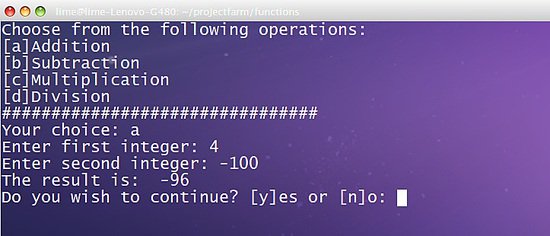
*The function only works if it is declared before your main routine. The interpreter will return an error if you have declared your function after your main routine.*

## Restructuring codes using functions

One of the best features of functions is being able to reuse codes. When a procedure requires repeatedly executing commands but could not be structured using looping statements then a function can be a solution.

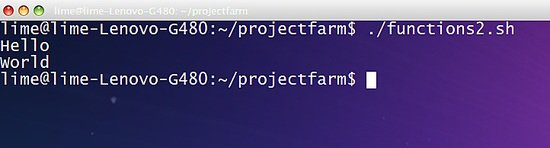
For example, consider the code below:

*#!/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 -p "Your choice: " choice  
    case $choice in  
    [aA])  
        read -p "Enter first integer: " int1  
        read -p "Enter second integer: " int2  
        res=$((int1+int2))  
   
    ;;  
    [bB])  
        read -p "Enter first integer: " int1  
        read -p "Enter second integer: " int2  
    res=$((int1-int2))  
   
    ;;  
    [cC])  
        read -p "Enter first integer: " int1  
        read -p "Enter second integer: " int2  
        res=$((int1\*int2))  
   
    ;;  
    [dD])  
        read -p "Enter first integer: " int1  
        read -p "Enter second integer: " int2  
        res=$((int1/int2))  
   
    ;;  
    \*)  
        res=0  
        echo "wrong choice!"  
    esac  
  
    echo "The result is: " $res  
    read -p "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)

The script is running well, however, notice that the lines for accepting inputs are repeatedly done in each pattern in our switch statement.

*#!/bin/bash  
inputs(){  
     read -p "Enter first integer: " int1  
    read -p "Enter second integer: " int2  
}  
  
exitPrompt(){  
    read -p "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  
}  
  
while(true)  
    do  
    clear  
    printf "Choose from the following operations: \n"  
    printf "[a]Addition\n[b]Subtraction\n[c]Multiplication\n[d]Division\n"  
    printf "################################\n"  
    read -p "Your choice: " choice  
  
    case $choice in  
    [aA])  
        inputs  
        res=$((int1+int2))  
    ;;  
  
    [bB])  
        inputs  
        res=$((int1-int2))  
    ;;  
  
    [cC])  
        inputs  
        res=$((int1\*int2))  
    ;;  
  
    [dD])  
        inputs  
        res=$((int1/int2))  
    ;;  
  
    \*)  
        res=0  
        echo "wrong choice!"  
    esac  
  
    echo "The result is: " $res  
    exitPrompt  
done*

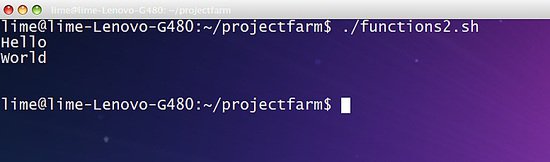
[](https://www.howtoforge.com/images/bash_scripting_part5/big/img04.jpg)

We improved our code by creating subsections *inputs* and *exitPrompt*. It works exactly the same with our previous code, however, our current code is easier to troubleshoot because it is structured properly.

## Passing parameters on functions

Like most of the programming languages, you can pass parameters and process those data in functions in bash. The code below shows the procedure on how to pass values in shell scripting:

*#!/bin/bash  
myfunction(){  
    echo $1  
    echo $2  
}  
  
myfunction "Hello" "World"*

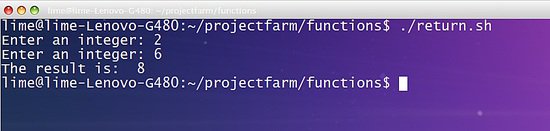
[](https://www.howtoforge.com/images/bash_scripting_part5/big/img05.jpg)

Notice in our example, we added the values "Hello" and "World" after we called the *myfunction*. Those values are passed to the *myfunction* as parameters and stored in a local variable. However, the unlike other languages, the interpreter stores the passed values into predefined variables, which is named according to the sequence of passing the parameters, 1 as the starting name up to the order of passing. Notice that the "Hello" word is stored to the variable *1* and value "World" is stored in variable *2*.

Note: The *1* and *2* in our example are local variables and thus, are not accessible to other parts of the script aside from the function where the parameters are being passed.

For instance,

*#!/bin/bash  
myfunction(){  
    echo $1  
    echo $2  
}  
  
myfunction "Hello" "World"  
echo $1  
echo $2*

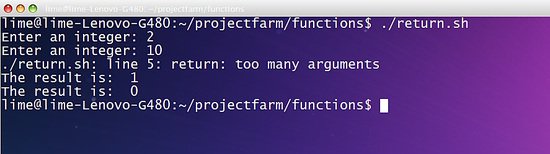
[](https://www.howtoforge.com/images/bash_scripting_part5/big/img06.jpg)

The echo *$1* and echo *$2* in the last two lines of our script have no display since the interpreter does not recognize both variables because they are both local to the *myfunction*.

## Returning Values from Functions

Aside from creating functions and passing parameters to it, bash functions can pass the values of a function's local variable to the main routine by using the keyword *return*. The returned values are then stored to the default variable *$?* For instance, consider the following code:

*#!/bin/bash  
add(){  
    sum=$(($1+$2))  
    return $sum  
}  
  
read -p "Enter an integer: " int1  
read -p "Enter an integer: " int2  
add $int1 $int2  
echo "The result is: " $?*

[](https://www.howtoforge.com/images/bash_scripting_part5/big/img07.jpg)

In the example, we pass the parameters *int1* and *int2* to the add function. Next the add function processes it through the line *sum=$(($1+$2))*. Then the value of the sum variable is passed to the main routine through the line return *$sum*. By default, the values of *$sum* will be stored to the default variable *$?*Finally, the line *echo "The result is: " $?* prints the result.

*Note: Shell scripts can only return a single value.*

Unlike other programming languages, shell scripts cannot return multiple values from a function. Let's take a look in this example:

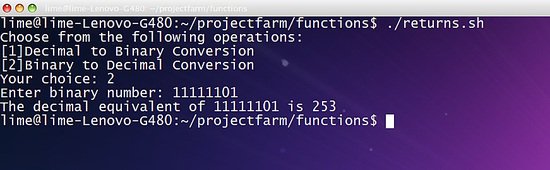
*#!/bin/bash  
add(){  
    sum=$(($1+$2))  
    dif=$(($1-$2))  
    return $sum   
}  
  
read -p "Enter an integer: " int1  
read -p "Enter an integer: " int2  
add $int1 $int2  
echo "The result is: " $?  
echo "The result is: " $?*

[](https://www.howtoforge.com/images/bash_scripting_part5/big/img08.jpg)

## To sum it up

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

*#!/bin/bash  
#####################  
#Author: HowtoForge #  
#####################  
   
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 -p "Your choice: " op  
    case $op in  
  
    1)  
        read -p "Enter integer number: " int  
        bin $int  
    ;;  
  
    2)  
        read -p "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.

*Note: It's better to use echo directly when converting from decimal to binary because when you return command to pass a binary value, the bash converts the binary value to decimal before returning it.*

Additionally, we have converted the binary value to decimal using the command *$(echo "ibase=2;$1"|bc)*.

You must also remember that the interpreter is only capable of accepting 8-bit binary digit. You will input digit that exceeds the 8-bit limit, it will generate an overflow and the most significant bit of the digit will be discarded.

The 10-bits binary digit 1000001010 returns 10 since following the 8-bit rule, the remaining 2 bits in the right side (most significant bit) will be omitted, thus, 1000001010 will become equal to 00001010 which is equal to 10. If you want an operation that accepts binary digits exceeding 8-bits then you have to create the code manually.