Sunday, March 05, 2017

LIKE GEEKS

LINUX

Bash Scripting The Awesome Guide Part6 Bash Functions

February 17, 2017 *admin* 2 Comments

Before we talk about bash functions let's discuss this situation. When writing bash scripts, you'll find yourself that you are using the same code in multiple places.

If you get tired writing the same blocks of code over and over in your bash script. It would be nice to just write the block of code once and refer to that block of code anywhere in your bash script without having to rewrite it.

The **bash shell** provides a feature allowing you to do just that called Functions.

Bash functions are blocks of script code that you assign a name to and reuse anywhere in your code. Anytime you need to use that block of code in your script, you simply use the function name you assigned it.

We are going to talk about how to create your own bash functions and how to use them in other shell scripts.

Our main points are:

Creating a function

Using functions

Using the return command

Using function output

Passing parameters to a function

Handling variables in bash functions

Passing arrays to functions

Recursive function

Creating libraries

Use bash functions from command line

Creating a function

You can create a function like this

```
functionName {

}
```

Or like this

functionName() {

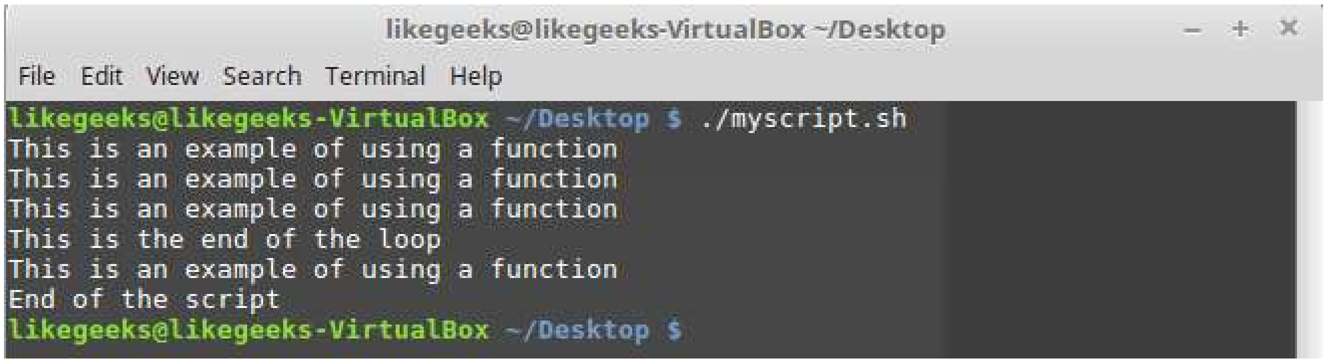
}

The parenthesis on the second way is used to pass values to the function from outside of it so these values can be used inside the function.

Using functions

```
#!/bin/bash
function myfunc {
echo "This is an example of using a function"
}
count=1
while [ $count -le 3 ]
do
myfunc
count=$(( $count + 1 ))
done
echo "This is the end of the loop"
myfunc
echo "End of the script"
```

Here we've created a function called myfunc and in order to call it, we just type it's name.

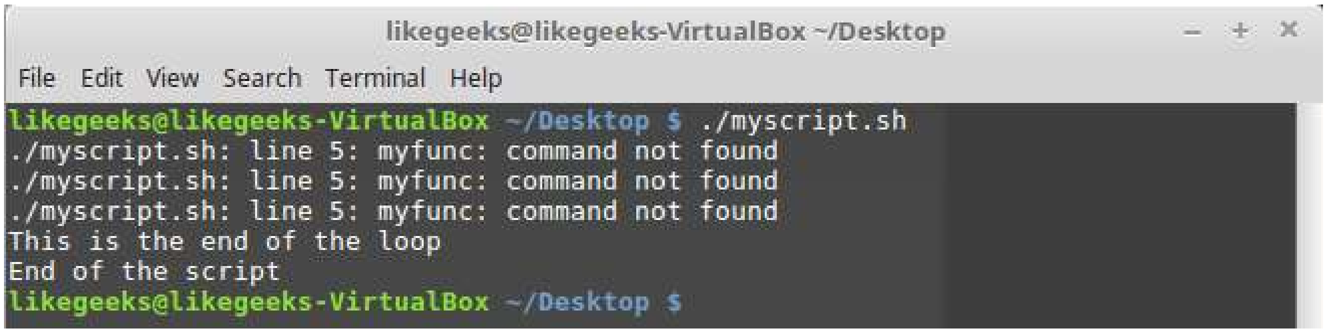

likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
This is an example of using a function
This is an example of using a function
This is an example of using a function
This is the end of the loop
This is an example of using a function
End of the script
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The function can be called many times as you want.

Notice: If you attempt to use a function before it's defined, you'll get an error message

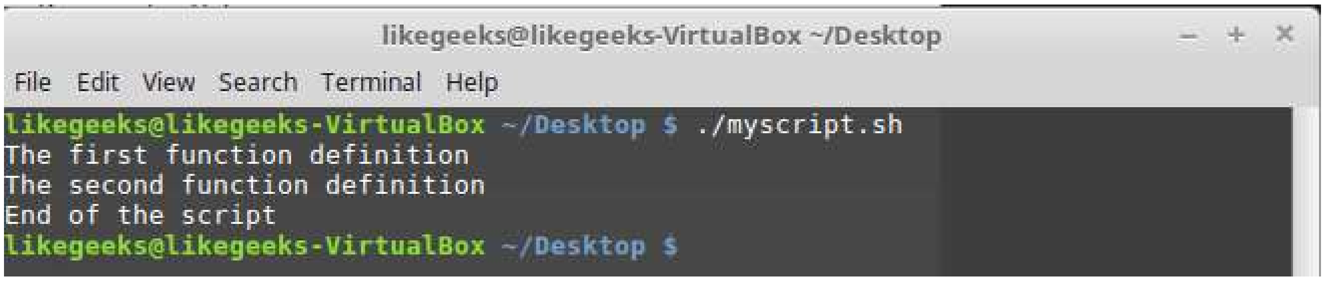
```
#!/bin/bash
count=1
while [ $count -le 3 ]
do
myfunc
count=$(( $count + 1 ))
done
echo "This is the end of the loop"
function myfunc {
echo "This is an example of using a function"
}
echo "End of the script"
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
./myscript.sh: line 5: myfunc: command not found
./myscript.sh: line 5: myfunc: command not found
./myscript.sh: line 5: myfunc: command not found
This is the end of the loop
End of the script
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Another notice: bash function name must be unique, or you'll have a problem. If you redefine a function, the new definition overrides the original function definition without any errors

```
#!/bin/bash

function myfunc {

echo "The first function definition"

}

myfunc

function myfunc {

echo "The second function definition"

}

myfunc

echo "End of the script"
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The first function definition
The second function definition
End of the script
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

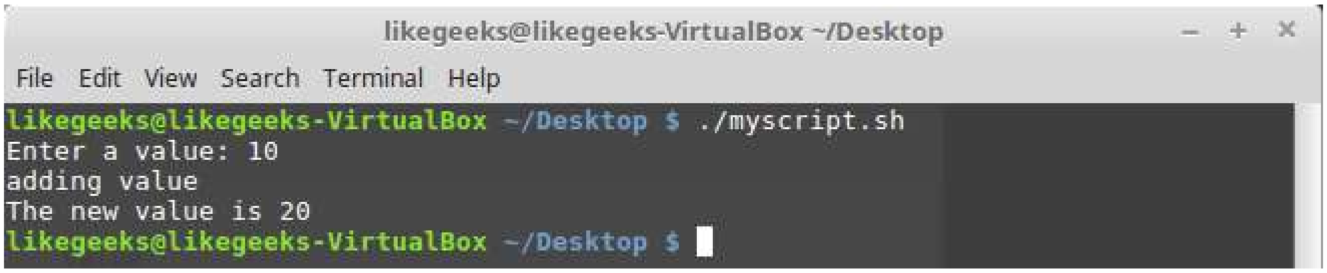
As you see the second function definition takes control from the first one without any error so take care when defining functions.

Using the return command

The return command allows you to specify a single integer value to define the function exit status.

There are two ways of using return command; the first way is like this

```
#!/bin/bash

function myfunc {

read -p "Enter a value: " value

echo "adding value"

return $(( $value + 10 ))
}
myfunc
echo "The new value is $?"
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter a value: 10
adding value
The new value is 20
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The myfunc function adds 10 to the value contained in the $value variable provided by the user input. It then returns the result using the return command, which the script displays using the $? Variable.

If you execute any other commands before retrieving the value of the function, using the $? variable, the return value from the function is lost. Remember that the $? Variable returns the exit status of the last executed command.

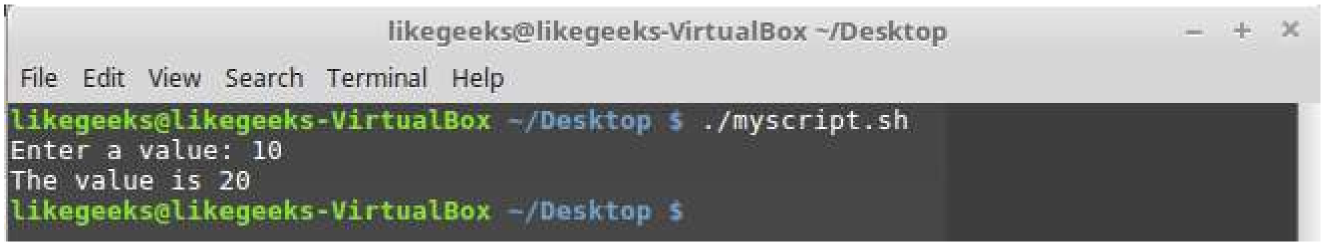
You cannot use this return value technique if you need to return either larger integer values or a string value.

Using function output

The second way of returning a value from a bash function is to capture the output of a command to a shell variable; you can also capture the output of a function to a shell variable. You can use this technique to retrieve any type of output from a function to assign to a variable

```
#!/bin/bash
function myfunc {
read -p "Enter a value: " value
echo $(( $value + 10 ))
}
result=$( myfunc)
```

```
echo "The value is $result"
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter a value: 10
The value is 20
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Passing parameters to a function

We can deal with bash functions like small snippets that we can reuse and that's ok but we need to make the function like an engine, we give it something and it returns another thing based on what we gave.

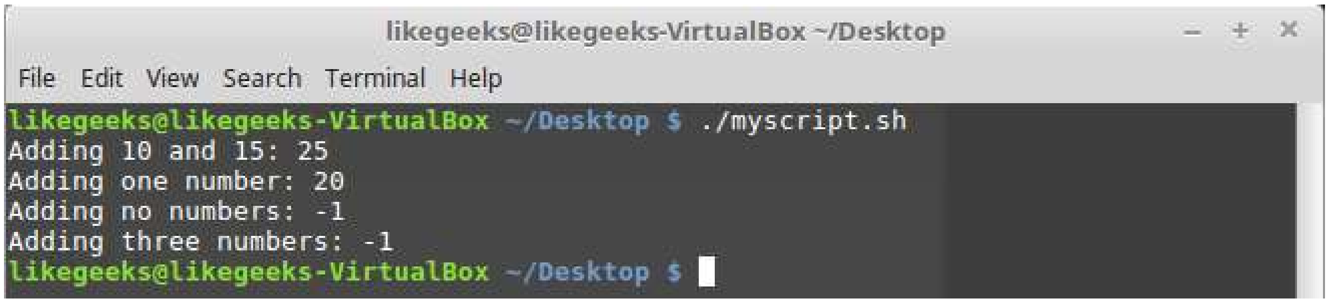
Functions can use the standard parameter environment variables to represent any parameters passed to the function on the command line. For example, the name of the function is defined as *$0* variable, and any other parameters passed on the function command line are defined using the variables *$1*, *$2*, and so on. You can also use the special variable $# to determine the number of parameters passed to the function. I recommend you to review the previous posts to empower your knowledge about them on **Linux bash scripting**.

We pass parameters to functions on the same command line as the function, like this

```
myfunc $val1 10 20
```

The following example shows you how to retrieve the parameter values using the parameter environment variables

```
#!/bin/bash
function addnum {
if [ $# -eq 0 ] || [ $# -gt 2 ]
then
echo -1
elif [ $# -eq 1 ]
then
echo $(( $1 + $1 ))
else
echo $(( $1 + $2 ))
fi
}
echo -n "Adding 10 and 15: "
value=$(addnum 10 15)
echo $value
echo -n "Adding one number: "
value=$(addnum 10)
echo $value
echo -n "Adding no numbers: "
value=$(addnum)
echo $value
echo -n "Adding three numbers: "
value=$(addnum 10 15 20)
echo $value
```

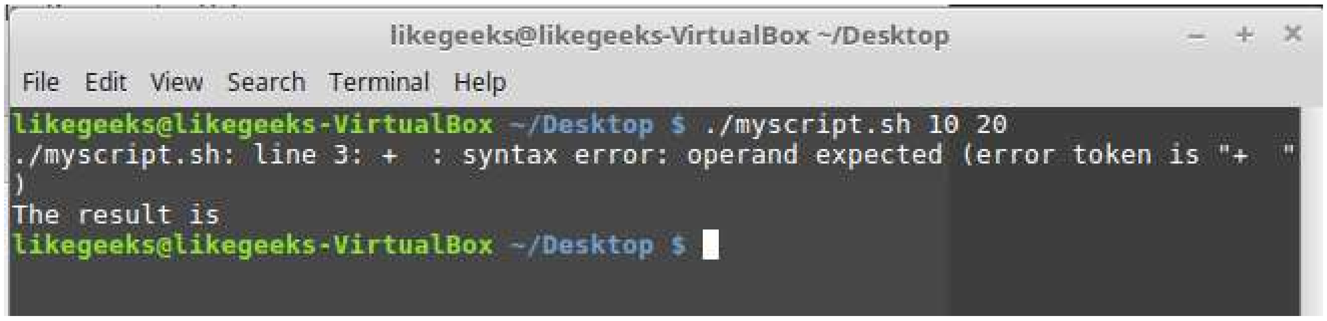

likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Adding 10 and 15: 25
Adding one number: 20
Adding no numbers: -1
Adding three numbers: -1
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The addnum function checks the number of parameters passed to it by the script. If there are no parameters, or if there are more than two parameters, addnum returns a value of -1. If there's one parameter, addnum adds the parameter to itself for the result. If there are two parameters, addnum adds them together and if you try to add three parameters it will return -1 as the bash function imply.

You can't directly access the script parameter values from the command line of the script. The following example fails

```
#!/bin/bash
function myfunc {
echo $(( $1 + $2 ))
}
if [ $# -eq 2 ]
then
value=$( myfunc)
echo "The result is $value"
else
echo "Usage: myfunc  a b"
fi
```

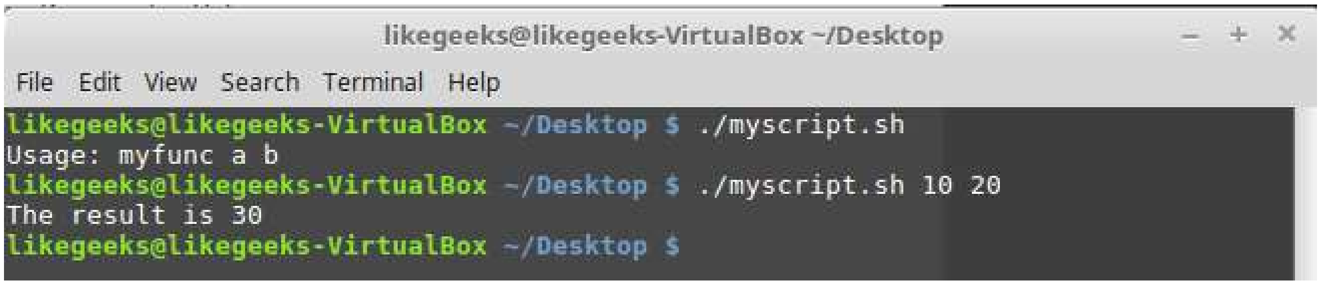


```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh 10 20
./myscript.sh: line 3: +  : syntax error: operand expected (error token is "+  "
)
The result is
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Instead, if you want to use those values in your bash function, you have to manually pass them when you call the function like this

```
#!/bin/bash
function myfunc {
echo $(( $1 + $2 ))
}
if [ $# -eq 2 ]
then
value=$(myfunc $1 $2)
```

```
echo "The result is $value"
else
echo "Usage: myfunc a b"
fi
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Usage: myfunc a b
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh 10 20
The result is 30
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Now they are available for the function to use, just like any other parameter

Handling variables in bash functions

Every variable we use has a scope, the scope is where the variable is visible

Variables defined inside functions can have a different scope than regular variables.

They can be hidden from the rest of the script.

There are two types of variables:

- Global
- Local

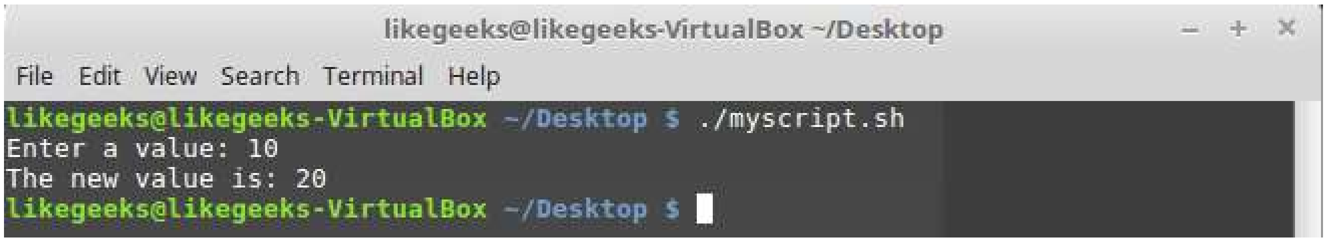
Global variables

Global variables are variables that are visible and valid anywhere in the bash script. If you define a global variable in the main section of a script, you can retrieve its value inside a function.

The same, if you define a global variable inside a function, you can retrieve its value in the main section of the script.

By default, any variables you define in the script are global variables. Variables defined outside of a function can be accessed inside the function without problems

```
#!/bin/bash
function myfunc {
value=$(( $value + 10 ))
}
read -p "Enter a value: " value
myfunc
echo "The new value is: $value"
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter a value: 10
The new value is: 20
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

When the variable is assigned a new value inside the function, that new value is still valid when the script references the variable as the above example the variable $value is changed inside the function.

So how to overcome something like this; Use local variables

Local variables

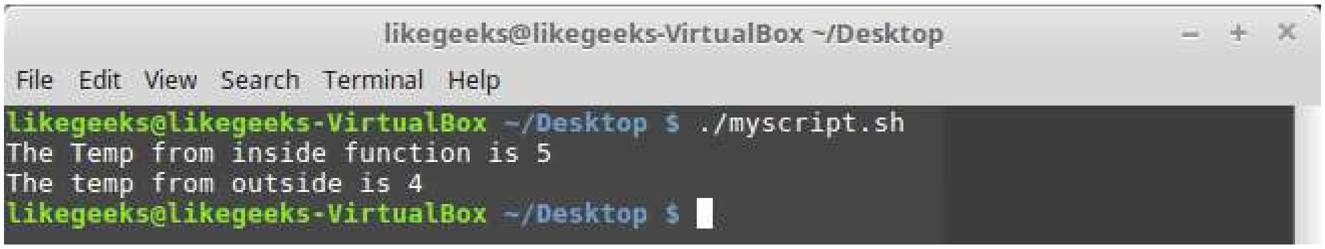
Any variables that the bash function uses internally can be declared as local variables. To do that, just use the local keyword in front of the variable like this

```
local temp=$(( $value + 5 ))
```

If a variable with the same name appears outside the function in the script, the shell keeps the two variable values separate. Now you can easily keep your function variables separate from your script variables

```
#!/bin/bash
```

```
function myfunc {
local temp=$[ $value + 5 ]
echo "The Temp from inside function is $temp"
}
temp=4
myfunc
echo "The temp from outside is $temp"
```



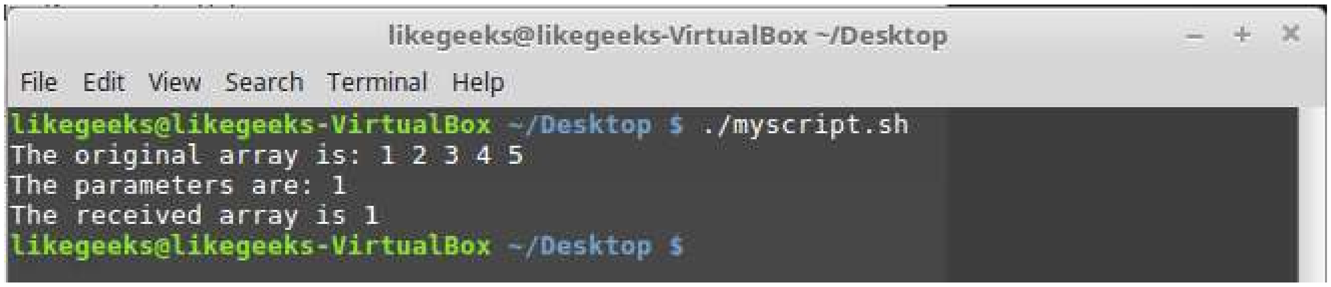
```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The Temp from inside function is 5
The temp from outside is 4
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Now when you use the $temp variable inside the myfunc function, it doesn't affect the value assigned to the $temp variable in the main script.

Passing arrays to functions

The art of passing an array variable to a bash function can be confusing. If you try to pass the array variable as a single parameter, it doesn't work

```
#!/bin/bash
function myfunc {
echo "The parameters are: $@"
arr=$1
echo "The received array is ${arr[*]}"
}
myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
myfunc $myarray
```

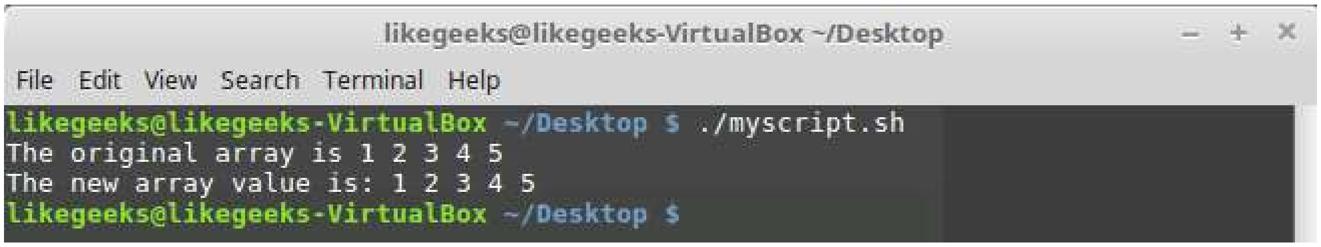


```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The original array is: 1 2 3 4 5
The parameters are: 1
The received array is 1
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

If you try using the array variable as a function parameter, the function only picks up the first value of the array variable

To solve this problem, you must disassemble the array variable into its individual values and use the values as function parameters. Inside the function, you can reassemble all the parameters into a new array variable like this.

```
#!/bin/bash
function myfunc {
local newarray
newarray=("$@")
echo "The new array value is: ${newarray[*]}"
}
myarray=(1 2 3 4 5)
echo "The original array is ${myarray[*]}"
myfunc ${myarray[*]}
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The original array is 1 2 3 4 5
The new array value is: 1 2 3 4 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The function rebuilds the array variable from the command line parameters

Recursive function

This feature enables the function to call itself from within the function itself

The classic example of a recursive function is calculating factorials. A factorial of a number is the value of the preceding numbers multiplied with the number. Thus, to find the factorial of 5, you'd perform the following equation. Thus the factorial of 5 is:

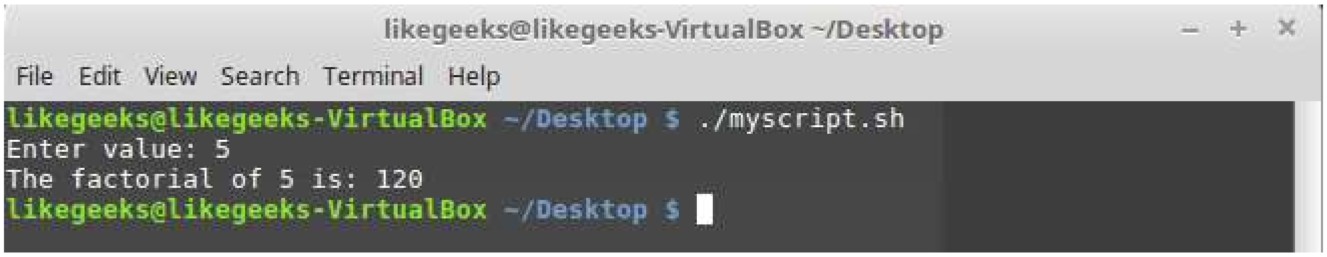
5! = 1 * 2 * 3 * 4 * 5

Using recursion, the equation is reduced down to the following format

```
x! = x * (x-1)!
```

So to write factorial function using bash scripting it will be like this

```
#!/bin/bash
function factorial {
if [ $1 -eq 1 ]
then
echo 1
else
local temp=$(( $1 - 1 ))
local result=$(factorial $temp)
echo $(( $result * $1 ))
fi
}
read -p "Enter value: " value
result=$(factorial $value)
echo "The factorial of $value is: $result"
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter value: 5
The factorial of 5 is: 120
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Using recursive bash functions is so easy!

Creating libraries

Now we know how to write functions and how to call them but what if you want to **use these bash functions or blocks of code on different bash script files** without copying and pasting it over your files.

The bash shell allows you to create a library file for your functions and then reference that single library file in as many scripts as you need to.

The key to using function libraries is the source command. You use the source command to run the library file script inside of your shell script. This makes the functions available to the script, without it, the function will not be visible in the scope of the bash script

The source command has a shortcut alias, called the dot operator*.* To source a file in a shell script, you just need to add the following line:

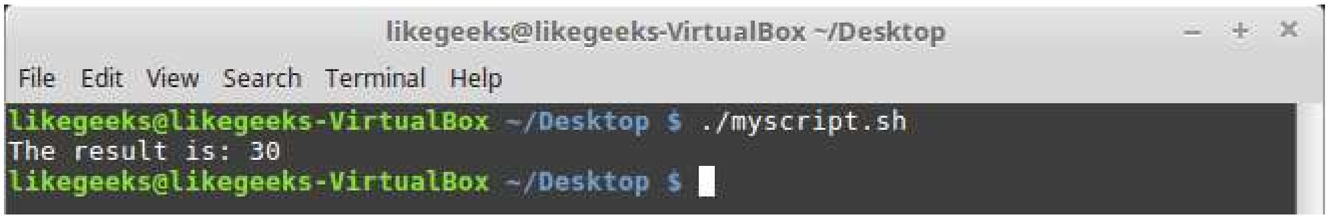
```
. ./myscript
```

Let's assume that we have a file called myfuncs and contains the following

```
function addnum {
echo $(( $1 + $2 ))
}
```

Now we will use it inside another bash script file like this

```
#!/bin/bash
. ./myfuncs
result=$(addnum 10 20)
echo "The result is: $result"
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The result is: 30
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Awesome!! We've used the bash functions inside myfuncs file inside our bash script file but what if we want to use those functions from our bash shell directly?

Use bash functions from command line

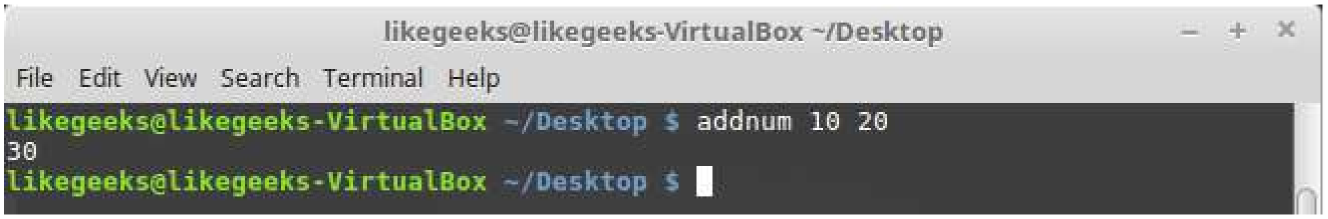
Well, that is easy if you read the previous post which was about **signals and jobs** you will have idea that we can source our functions file in .bashrc file and hence we can use the functions directly from the bash shell. Cool

Edit .bashrc file and add this line

```
. /home/likegeeks/Desktop/myfuncs
```

Just make sure you type the correct path and now from the shell when we type the following

```
$ addnum 10 20
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ addnum 10 20
30
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Even better, the shell also passes any defined bash functions to child shell processes so your functions are automatically available for any bash scripts without sourcing wow!! That really cool right

Note: you may need to logout and login to use the bash functions from the shell

Another note: if you make your function name like the any of the built-in commands you will overwrite the default command so you should take care of that.