Shell Script

17 February 2025 10:37 AM

Functions In Shell Script

Shell Functions are used to specify the blocks of commands that may be repeatedly invoked at different stages of execution.
The main advantages of using unix Shell Functions are to reuse the code and to test the code in a modular way.

The purpose of the function

- Don't repeat yourself.
- Write once, use many times.
- Reduce the script length.
- A single place to edit and troubleshoot.
- Easier to maintain.
- If you are repeating yourself, use a function.
- Reusable code.
- Must define before use.
- Has parameters supports.
- The best practice is to put all the functions on top of the script.

How to Create Function in Shell Script

In Shell script, we can write functions in a variety of different ways.

```
# type one.
function function_name(){
   # code goes here
}
# type two
function_name(){
   # code goes here
}
# method three
function function_name {
  # function code here.
}
```

Simply use the function name as a command to run a function.

```
# invode the function
function_name
```

example:

```
#!/bin/bash
# funtions
function install(){
  #### installations code.
  echo "installationscode1"
}
configuration(){
  # configurations code
  echo "configcode1"
```

```
}
function deploy {
  # deploy code.
  echo "deploy code 1"
}
configuration
install
deploy
```

let's run the above code and see the output.

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./functions.sh
configcode1
installationscode1
deploy code 1
```

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.

Functions can be Recursive.

A function may return a value in one of four different ways:

- Change the state of a variable or variables
- Use the**return**command to end the function, and return the supplied value to the calling section of the shell script
- echo output to stdout, which will be caught by the caller just as `expr $a + $b` is caught
- we can get the function name using **FUNCNAME** variable.

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

```
#!/bin/bash
function install(){
  echo "executing ${FUNCNAME} - start"
  echo "installing ${1}"
  echo "executing ${FUNCNAME} - end"
}
function configuration(){
  echo "config ${1}"
```

```
  echo "${FUNCNAME}"
}
function deploy() {
  echo "deploying ${1}"
  echo "${FUNCNAME}"
}
install "nginx"
configuration "nginx"
deploy "webapplication"
```

Output:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./functions-args.sh
executing install - start
installing nginx
executing install - end
config nginx
configuration
deploying webapplication
deploy
```

Create Local Variable In Shell Script

- All variables are global by default.
- Modifying a variable in a function changes it in the whole script. This could lead to issues.
- **local** command can only be used within a function.
- It makes the variable name have a **visible scope restricted to that function** and its children only.
- All function variables are local. This is a good programming practice.

example

```
#!/bin/bash
packageName="nginx"
function install(){
    local myname="gaurav"
    echo "installing ${1}"
}
function configuration(){
    packageName="tomcat"
    echo "config ${1}"
}
echo "first ${packageName}"
echo "myname = ${myname}"
install "${packageName}"
echo "myname = ${myname}"
echo "second ${packageName}"
configuration "${packageName}"
echo "third ${packageName}"
```

output:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./variables-in-functions.sh
first nginx
myname =
installing nginx
myname =
second nginx
config nginx
third tomcat
```

Return Status ($?) and Test Command

Exit Status

- Every command returns an **exit status **(sometimes referred to as a *return status *or *exit code*).
- A successful command returns a ***0***, while an unsuccessful one returns a non-zero value that usually can be interpreted as an *error code*.
- Well-behaved UNIX commands, programs, and utilities return a 0 exit code upon successful completion, though there are some exceptions.
- When a script ends with an **exit** that has no parameter, the exit status of the script is the exit status of the last command executed in the script (previous to the **exit**).

```
┌──(gaurav㉿learning-ocean)-[~]
└─$ echo "Learning-Ocean"
Learning-Ocean
┌──(gaurav㉿learning-ocean)-[~]
└─$ echo "echo: Command Not Found"
echo: Command Not Found
┌──(gaurav㉿learning-ocean)-[~]
└─$ echo $?
0 # getting exit status is zero means last command executed successfully.
┌──(gaurav㉿learning-ocean)-[~]
└─$ asdasd
asdasd: command not found
┌──(gaurav㉿learning-ocean)-[~]
└─$ echo $?
127 # getting exit status as non zero means last command not exexuted successfully.
```

Test Command

- A test command is a command that is used to test the validity of a command.
- It checks whether the command/expression is true or false.
- It is used to check a number, string and file expression
- It is used to check the type of file and the permissions related to a file.
- Test command returns 0 as a successful exit status if the

command/expression is true, and returns 1 if the command/expression is false.

- Test is used by virtually every shell script written. It may not seem that way, because **test** is not often called directly. **test** is more frequently called as **[**. **[** is a symbolic link to **test**, just to make shell programs more readable. It is also normally a shell builtin.

```
┌──(gaurav㉿learning-ocean)-[~]
└─$ a=5 # assign a value(5) to variable a.
┌──(gaurav㉿learning-ocean)-[~]
└─$ test $a -eq 5 # checking that a = 5 or not
┌──(gaurav㉿learning-ocean)-[~]
└─$ echo $?
0 # in last command our expression is true so its return a successfull status means zero.
# now let's try with a==4
┌──(gaurav㉿learning-ocean)-[~]
└─$ test $a -eq 4
# above express if not true so return non zero (1) exit status
┌──(gaurav㉿learning-ocean)-[~]
└─$ echo $?
1
```

If With Command

This block will process if the exit status of **COMMAND** is zero(or a command executed successfully).

```
if COMMAND
then
# if Block
# Your code here.
fi
```

Example: let's create a file with the name if-condition.sh with the below content.

```
#!/bin/bash
if grep -i localhost /etc/hosts>/dev/null
then
  echo "Grep Command Executed successfully"
fi
echo "I am Here"
```

let's execute that file and see the output.

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./if-condition.sh
Grep Command Executed successfully
I am Here
```

Now let's do some changes in the if-condition.sh file and try to search for another word that is not present in the /etc/hosts file.

```
#!/bin/bash
if grep -i gauravyt /etc/hosts>/dev/null
then
  echo "Grep Command Executed successfully"
fi
echo "I am Here"
```

now let's check the output

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./if-condition.sh
I am Here
```

Now you can see that our if block does not execute because the grep command does not return a zero status.

IF With Number

We can use IF with the test command.

```
#!/bin/bash
number=5
if test $number -eq 5
then
  echo "number is euqal to five"
fi
```

instead of test command, we can use its alias that is **[**.

- eq use for equals operations.
- lt is used for less than.
- gt is used for greater then
- le is used for less than or equal.
- ge is used for greater than or equal.
- ne is used for not equal.

```
#!/bin/bash
number=15
# eq is for equal, if number is equal to 5 then the below condition will become true.
if [ $number -eq 5 ]
then
  echo "number is eq 5"
fi
# lt is for less then, if number is less than 11 then the below condition will become true.
if [ $number -lt 10 ]
then
  echo "number is less than 10"
fi
# gt is for greater then, if number is greater than 4 then the below condition will become true.
if [ $number -gt 4 ]
```

```
then
  echo "number is greater then 4"
fi
# ge is for greater then or equal, if number is greater than or equal to 5 then the below condition will become true.
if [ $number -ge 5 ]
then
  echo "number is grater than or equal to 5"
fi
# le is for less then or equal, if number is less than or equal to 5 then the below condition will become true.
if [ $number -le 5 ]
then
  echo "number is less than or equal to 5"
fi
# ne is for not equal, is number is not euqal to 5 then below condition will become true.
if [ $number -ne 5 ]
then
  echo "number is not equal to five."
fi
```

If with Files

Suppose we want to check that file is a regular file or directory, the file have read, write or execute permission then again we have use test command or ([[or [) with if condition.
here is an example.

```
#!/bin/bash
file_full_path="/home/kali/abc.txt"
# check file is a directory.
if [[ -d $file_full_path ]]
then
   echo  "${file_full_path} is a dir"
fi
# -b means file is block device.
if [[ -b $file_full_path ]]
then
   echo  "${file_full_path} is a block device"
fi
#check, file is a char device.
if [[ -c $file_full_path ]]
then
   echo  "${file_full_path} is a char device"
fi
#check, file exists.
if [[ -e $file_full_path ]]
then
   echo  "${file_full_path} is a exist device"
fi
#check, file have read permission.
```

```
if [[ -r $file_full_path ]]
then
  echo  "${file_full_path} have read permission"
fi
# check, file have write permission
if [[ -w $file_full_path ]]
then
  echo  "${file_full_path} have write permission"
fi
# check file have execute permission.
if [[ -x $file_full_path ]]
then
  echo  "${file_full_path} have execute permission"
fi
```

Not Operator in ShellScript

The NOT logical operator reverses the true/false outcome of the expression that immediately follows.
For example, if a file does not exist, then display an error on the screen.

```
#!/bin/bash
name="saurav sharma"
othername="gaurav sharma"
if [[ ! ${othername} == ${name} ]]
then
   echo "both are same"
fi
```

now let's run the above the program

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./if-not.sh
both are same
```