

Bash Functions

In this topic, we have demonstrated the basics of bash functions and how they work in bash shell scripting.

Functions in bash scripting are a great option to reuse code. A Bash function can be defined as a set of commands which can be called several times within bash script. The purpose of function in bash is to help you make your scripts more readable and avoid writing the same code again and again. It also allows the developers to break a complicated and lengthy code to small parts which can be called whenever required. Functions can be called anytime and repeatedly, which will enable us to reuse, optimize, and minimize the code.

Following are some key points about bash functions:

- A function has to be declared in the shell script before we can use it.
 - Arguments can be passed to the functions and accessed inside the function as \$1, \$2, etc.
 - Local variables can be assigned within the function, and the scope of such variables will only be that particular function.
 - Built-in commands of Bash shell can be overridden using functions.
-

Syntax

The syntax for declaring a bash function can be defined in two formats:

1. The first method starts with the function name, followed by parentheses. It is the most preferred and commonly used method:

1. `function_name () {`
2. `commands`
3. `}`

Single line version can be mentioned as below:

1. `function_name () { commands; }`

2. The second method starts with the function reserved word, followed by the function name:

```
function function_name {  
  commands  
}
```

Single line version can be mentioned as below:

```
1. function function_name { commands; }
```

Compared to most of the programming languages, Bash functions are somewhat limited. Let's understand the concept with the help of some examples:

Example: Method 1

```
#!/bin/bash  
  
JTP () {  
  echo 'Welcome to Javatpoint.'  
}  
  
JTP
```

Output

Welcome to Javatpoint.

Example: Method 2

```
#!/bin/bash  
  
function JTP {  
  echo 'Welcome to Javatpoint.'  
}  
  
JTP
```

Output

Welcome to Javatpoint.

Passing Arguments

Like most of the programming languages, we can also pass the arguments and process the data in bash functions. We can insert the data to the function in a similar way as passing-command line arguments to a bash script.

To pass any number of arguments to the bash function, we are required to insert them just after the function's name. We must apply spaces between function name and arguments. It will also be a great choice to use double quotes around the arguments to prevent misparsing of the arguments with spaces in it.

Following are some key points about passing arguments to the bash functions:

- The given arguments are accessed as \$1, \$2, \$3 ... \$n, corresponding to the position of the arguments after the function's name.
 - The \$0 variable is kept reserved for the function's name.
 - The \$# variable is used to hold the number of positional argument/ parameter given to the function.
 - The \$* and @\$ variables are used to hold all the arguments/ parameters given to the function.
-
- When \$* is used with double quotes (i.e., "\$*"), it expands to a single string separated by the space. For example, "\$1 \$2 \$n etc".
 - When @\$ is used with double quotes (i.e., "\$@"), it expands to the separate string. For example, "\$1" "\$2" "\$n" etc.
 - When \$* and \$# are not used with the double quotes, they both are the same.

Following is the code that illustrates the procedure on how to pass arguments to functions, and access the arguments inside the function.

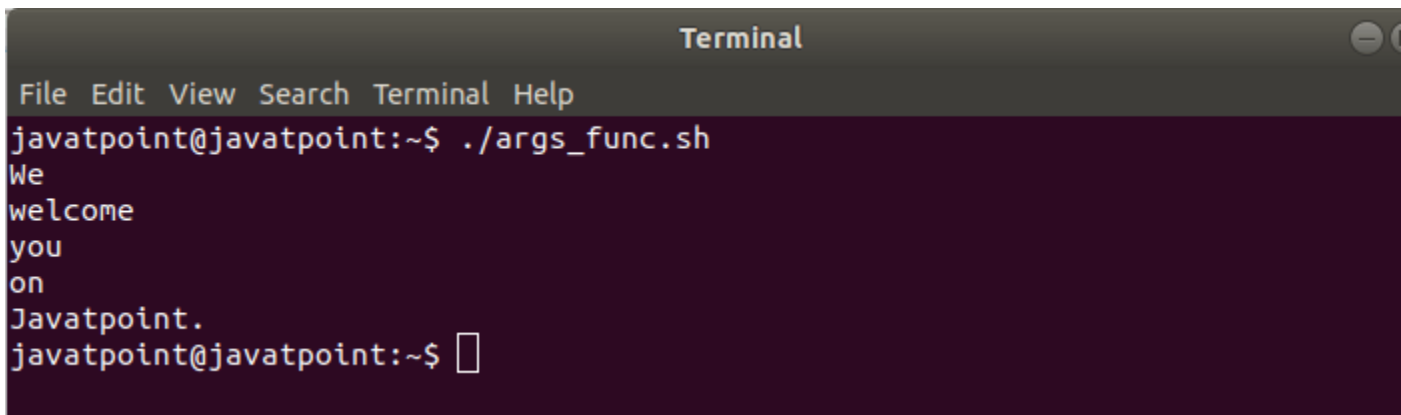
Bash Script

```
#!/bin/bash
#Script to pass and access arguments

function_arguments()
{
    echo $1
    echo $2
    echo $3
    echo $4
    echo $5
}

#Calling function_arguments
function_arguments "We""welcome""you""on""Javatpoint."
```

Output

A screenshot of a macOS Terminal window. The title bar is dark gray with the word "Terminal" in white. Below the title bar is a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help" in light gray. The main area has a dark purple background. The prompt "javatpoint@javatpoint:~\$" is followed by the command "./args_func.sh". The output consists of five lines: "We", "welcome", "you", "on", and "Javatpoint.". The prompt "javatpoint@javatpoint:~\$" is followed by a white cursor block.

```
Terminal
File Edit View Search Terminal Help
javatpoint@javatpoint:~$ ./args_func.sh
We
welcome
you
on
Javatpoint.
javatpoint@javatpoint:~$
```

In this script, we have added the values "We", "welcome", "you", "on" and "Javatpoint" after we have called the `function_arguments`. Those values are passed to the **function_arguments** as parameters and stored in a local variable. However, unlike other languages, the interpreter stores the passed values into predefined variables, which are then named according to the sequence of passing parameters.

For example,

"We" word is stored to the variable 1.
"welcome" word is stored to the variable 2.
"you" word is stored to the variable 3.
"on" word is stored to the variable 4.
"Javatpoint" word is stored to the variable 5.

Variable Scope

Global variables are defined as the variables which can be accessed anywhere within the script regardless of the scope. By default, all the variables are defined as global variables, even if they are declared inside the function. We can also create variables as a local variable. Local variables can be declared within the function body with the `local` keyword when they are assigned for first time. They are only accessible inside that function. We can create local variables with the same name in different functions. To add a local variable, we can use the following syntax:

1. local `var_name=<var_value>`

To better understand how variables scope works in Bash Scripting, check out the following example:

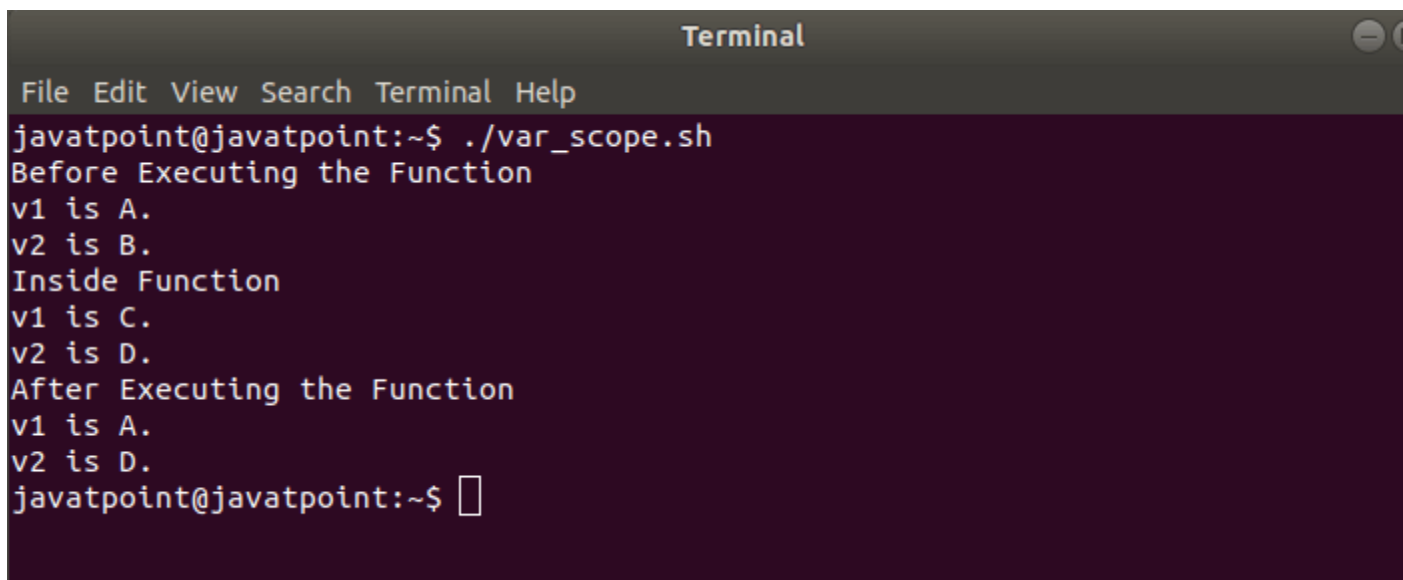
Bash Script

```
#!/bin/bash

v1='A'
v2='B'
my_var () {
  local v1='C'
  v2='D'
  echo "Inside Function"
  echo "v1 is $v1."
  echo "v2 is $v2."
}
echo "Before Executing the Function"
echo "v1 is $v1."
echo "v2 is $v2."
my_var
```

```
echo "After Executing the Function"  
echo "v1 is $v1."  
echo "v2 is $v2."
```

Output



```
Terminal  
File Edit View Search Terminal Help  
javatpoint@javatpoint:~$ ./var_scope.sh  
Before Executing the Function  
v1 is A.  
v2 is B.  
Inside Function  
v1 is C.  
v2 is D.  
After Executing the Function  
v1 is A.  
v2 is D.  
javatpoint@javatpoint:~$
```

As per output, if we set a local variable within the function body with the same name as an existing global variable, then it will have precedence over the global variable. Global variables can be modified within the function.

Return Values

Most of the programming languages have the concept of returning a value for the functions. It means that the function has to send the data back to the original calling location. Unlike functions in 'real' programming languages, Bash function doesn't provide support to return a value when it is called. However, they allow us to set a return status which is similar to how a program or command exits with an exit status. When a bash function completes, its return value is the status of the last executed statement in the function. It returns 0 for the success status and non-zero decimal number in the 1-255 range for failure.

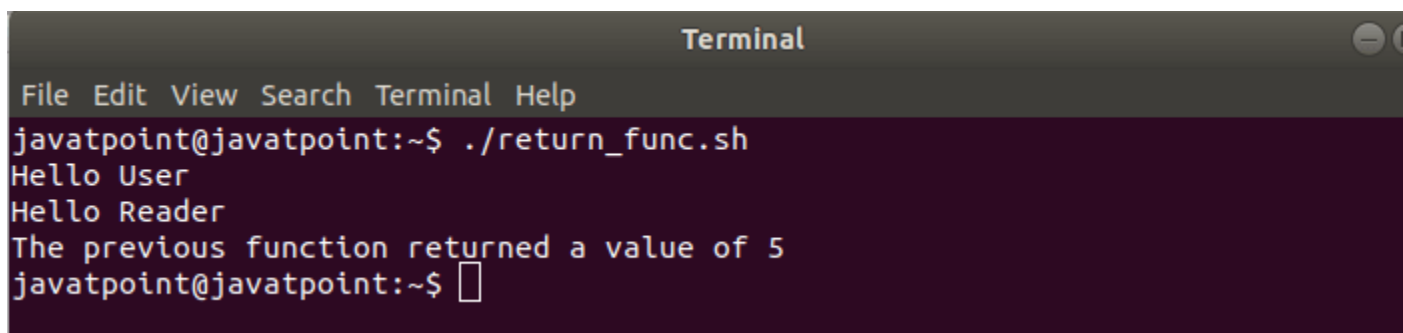
The return status can be indicated by using the 'return' keyword, and it is assigned to the variable `?`. The return statement terminates the function and works as the function's exit status.

For example, consider the following code:

Bash Script

1. `#!/bin/bash`
2. `#Setting up a return status for a function`
- 3.
4. `print_it () {`
5. `echo Hello $1`
6. `return 5`
7. `}`
- 8.
9. `print_it User`
10. `print_it Reader`
11. `echo The previous function returned a value of $?`

Output

A screenshot of a macOS Terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal content shows a user at the "javatpoint@javatpoint:~" prompt running the command `./return_func.sh`. The script outputs three lines: "Hello User", "Hello Reader", and "The previous function returned a value of 5". The prompt returns to `javatpoint@javatpoint:~$` with a cursor.

```
Terminal
File Edit View Search Terminal Help
javatpoint@javatpoint:~$ ./return_func.sh
Hello User
Hello Reader
The previous function returned a value of 5
javatpoint@javatpoint:~$
```

Another better option to return a value from a function is to send the value to **stdout** using **echo** or **printf** commands, as shown below:

Bash Script

1. `#!/bin/bash`
- 2.
3. `print_it () {`
4. `local my_greet="Welcome to Javatpoint."`
5. `echo "$my_greet"`
6. `}`
- 7.
8. `my_greet="$(print_it)"`
9. `echo $my_greet`

Output

Welcome to Javatpoint.

Overriding Commands

We have an option to override the bash commands by creating a function with the same name as the command that we are going to override. For example, if we want to override the 'echo' command, then we have to create a function with the name 'echo'.

This concept of overriding the bash commands may be helpful in some scenarios like when we want to use a command with specific options. Also, when we do not like to provide the whole command with options for several times within the script. In such cases, we can override the in-built bash command for command with options. Now, let's understand the concept of overriding the commands in Bash Shell Scripting with the help of some examples:

Example

In this example, we have overridden the 'echo' command and added the time stamp in the form of the argument to the 'echo' command.

Bash Script

1. `#!/bin/bash`
2. `#Script to override command using function`
- 3.
4. `echo () {`
5. `builtin echo -n `date +"[%m-%d %H:%M:%S]"` ": "`
6. `builtin echo $1`
7. `}`
- 8.
9. `echo "Welcome to Javatpoint."`

Output

```
Terminal
File Edit View Search Terminal Help
javatpoint@javatpoint:~$ ./override.sh
[09-24 17:58:28] : Welcome to Javatpoint.
javatpoint@javatpoint:~$
```