Chapter 8

Shell Functions

School of Computer Science,
MIT-World Peace University,
Pune, Maharashtra, India.

Amol D. Vibhute (PhD)
Assistant Professor

Email ID:- amol.vibhute@mitwpu.edu.in

Roadmap of Chapter

- Creating Functions
- Pass Parameters to a Function
- Returning Values from Functions
- Nested Functions
- Function Call from Prompt.

Introduction:

- 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**.
- 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 is executed.
- Shell functions are executed in the current shell context; no new process is created to interpret them.
- Functions enable us 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.
- This is an important part of modern object-oriented programming principles.
- Shell functions are like 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 your function is **function_name**, and that's what we will use to call it from elsewhere in our 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 your function here
  - Demo ()
  - {
  - echo "Demo of Shell Functions“
  - }
  - # Invoke your function
  - Demo

Output: Demo of Shell Functions

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 "**Shell**" and "**Functions**" and then we capture and print these parameters in the function.

```
#!/bin/sh
# Define your function here
Demo ()
{
echo "Demo of Shell Function in $1 $2"   }
# Invoke your function
Demo Unix OS
```

- Output: Demo of Shell Function in Unix OS.

Returning Values from Functions:

- If we 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 we want to just terminate execution of the function, then there is way to come out of a defined function.
- Based on the situation we can return any value from our function using the return command whose syntax is as follows − return code
- Here **code** can be anything we choose here, but obviously we should choose something that is meaningful or useful in the context of script as a whole.

Cont.…

- Example
  - Following function returns a value 10 −
    - #!/bin/sh
    - # Define your function here
    - Hello () {
    - echo "Hello World $1 $2"
    - return 10
    - }
    - # Invoke your function
    - Hello Returning Values
    - # Capture value returned by last command
    - ret=$?
    - echo "Return value is $ret“
  - Upon execution, we will receive the following result −
    - Hello World Returning Values
    - Return value is 10

Cont.…

- The bash shell uses the return command to exit a function with a specific exit status. The return command allows you to specify a single integer value to define the function exit status, providing an easy way for you to programmatically set the exit status of your function:

```
#!/bin/bash
# using the return command in
function
function dbl {
 read -p "Enter a value: " value
 echo "doubling the value"
 return $[ $value * 2 ]
}
dbl
echo "The new value is $?"
$
```

There is a limitation for using this return value technique. Because an exit status must be less than 256, the result of your function must produce an integer value
less than 256. Any value over that returns an error value:

```
$ bash return1.sh
Enter a value: 200
doubling the value
The new value is 144
$
```

Cont.…

Using function output:

Just as you can 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:

result=$(functionname par1,par2…)

```
#!/bin/bash
# using the echo to return a value
function dbl {
 read -p "Enter a value: " value
 echo $[ $value * 2 ]
}
result=$(dbl)
echo "The new value is $result"
```

```
$ bash return2.sh
Enter a value: 200
The new value is 400
```

Cont.…

- **Handling variables in a function:**
  - One thing that causes problems for shell script programmers is the scope of a variable.
  - The scope is where the variable is visible. Variables defined in functions can have a different scope than regular variables. That is, they can be hidden from the rest of the script.
  - Functions use two types of variables:
    - Global
    - Local
- The following sections describe how to use both types of variables in your functions.

Cont.…

```
#!/bin/bash
# demonstrating the local keyword
function func1 {
# here temp is local variable
 local temp=$[ $value + 5 ]
 result=$[ $temp * 2 ]
}
temp=4
value=6
func1
echo "The result is $result“
# here value of global temp is available
if [ $temp -gt $value ]
then
 echo "temp is larger"
else
 echo "temp is smaller"
fi
```

1. Without local keyword for temp

Output:
The result is 22
temp is larger

2. If we use local keyword for temp

Output:
The result is 22
temp is smaller

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**.

- Following example demonstrates nesting of two functions −
  - #!/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
- Upon execution, we will receive the following result −
  - This is the first function speaking...
  - This is now the second function speaking...

Function Call from Prompt:

- We can put definitions for commonly used functions inside our ***.profile***. These definitions will be available whenever we log in and we can use them at the command prompt.

- Alternatively, we can group the definitions in a file, say ***test.sh***, and then execute the file in the current shell by typing −

  – $. test.sh

- This has the effect of causing functions defined inside ***test.sh*** to be read and defined to the current shell as follows—
  – $ number_one
  – This is the first function speaking...
  – This is now the second function speaking...
  – $
- To remove the definition of a function from the shell, use the unset command with the **.f** option. This command is also used to remove the definition of a variable to the shell.

  – $ unset -f function_name

References:

- Unix Shell Programming: Yashwant Kanitkar, BPB Publications, New Delhi.

Thank You !!!