Linux Plus for AWS and DevOps

Session - 7

CLARUSWAY
WAY TO REINVENT YOURSELF

Table of Contents

Loops!!!

When writing programs in shell, in some cases it is not enough to execute the block of code only once. The loops are used to repeat (iterate) the execution of a block of code.

Loops allow us to take a series of commands and keep re-running them until a particular situation is reached. They are useful for automating repetitive tasks.

There are 3 basic loop structures in Bash scripting which we'll look at below. There are also a few statements which we can use to control the loops operation.

1. **while loops** - One of the easiest loops to work with is while loops. They say, while an expression is true, keep executing these lines of code.

2. **until loops** - The until loop is fairly similar to the while loop. The difference is that it will execute the commands within it until the test becomes true.

3. **for loops** - The for loop is a little bit different to the previous two loops. What it does is say for each of the items in a given list, perform the given set of commands.

While loops

While loops have a boolean logic, similar to if statements. This loop will execute the commands until the test becomes **false**

As long as the result of the condition returns True, the code block under while loop runs.

When the condition returns to False, the loop execution is terminated and the program control moves further to the next operation

```
while [[ <some test> ]]
do
  <commands>
done
```

```
#!/bin/bash

number=1

while [[ $number -le 10  ]]
do
 echo $number
  ((number++))
done
echo "Now, number is $number"
```

```
$./while-loops.sh
1
2
3
4
5
6
7
8
9
10
Now, number is 11
```

Until loops

The until loop is identical to the while loop, except that it will execute the commands within it until the test becomes **true.**

As long as the result of the condition returns false, the code block under while loop runs.

```
until [[ <some test> ]]
do
  <commands>
done
```

```
#!/bin/bash

number=1

until [[ $number -ge 10  ]]
do
 echo $number
 ((number++))
done
echo "Now, number is $number"
```

```
$./until.sh
1
2
3
4
5
6
7
8
9
Now, number is 10
```

CLARUSWAY©
WAY TO REINVENT YOURSELF

For loops

Sometimes we want to iterate a block of code for each of the items in a given list.
For this we use `**for loop**`.

```
for item in [list]
do
  commands
done
```

```
#!/bin/bash

echo "Numbers:"

for number in 0 1 2 3 4 5 6 7 8 9
do
  echo $number
done
```

Output:

```
$./for-loop.sh
Numbers:
0
1
2
3
4
5
6
7
8
9
```

Continue and Break Statement

- A loop will continue forever unless the necessary condition is not met.
- A loop that runs endlessly without terminating can run for an infinite number of times.
- For this reason, these loops are named infinite loops.

Infinite loop

```
#!/bin/bash

number=1

until [[ $number -lt 1  ]]
do
 echo $number
 ((number++))
done
echo "Now, number is $number"
```

Continue and Break Statement

The break statement tells Bash to leave the loop straight away. It may be that there is a normal situation that should cause the loop to end but there are also exceptional situations in which it should end as well.

For instance, maybe we are copying files but if the free disk space get's below a certain level we should stop copying.

Break Statement

```
#!/bin/bash

number=1

until  [[ $number -lt 1  ]]
do
 echo $number
 ((number++))
 if [[ $number -eq 10 ]]
 then
   break
 fi
done
```

Output:

```
./infinite-loop.sh
1
2
3
4
5
6
7
8
9
```

CLARUSWAY©
WAY TO REINVENT YOURSELF

Continue and Break Statement

Continue Statement

The Continue statement is similar to the Break command, except it causes the current iteration of the loop to exit, instead of the whole loop.

```
#!/bin/bash
for i in {1..10}
do
        if [[ $i == '9' ]]
        then
                echo "Number $i!"
                continue
        fi
        echo "$i"
done
echo "Done!"
```

Output:

```
$./continue.sh
1
2
3
4
5
6
7
8
Number 9!
10
Done!
```

CLARUSWAY©
WAY TO REINVENT YOURSELF

Select Loops

- The Select Loop generates a numbered menu from which users can select options.
- It's helpful when you need to ask the user to select one or more items from a list of options.
- The select loop uses the PS3 variable to prompt the user.
- The user selection is read from the standard input.
- If the user selection is empty, the menu and the PS3 prompt is shown again.

```
#!/bin/bash

read -p "Input first number: " first_number
read -p "Input second number: " second_number

PS3="Select the operation: "

select operation in addition subtraction exit
do
  case $operation in
    addition)
      echo "result= $(( $first_number + $second_number))"
    ;;
    subtraction)
       echo "result= $(( $first_number - $second_number))"
    ;;
    exit)
       break
    ;;
    *)
       echo "Wrong choice..."
    ;;
  esac
done
```

CLARUSWAY©
WAY TO REINVENT YOURSELF

Homework Exercise 1

1. Calculate sum of the numbers between 1 to 100.
2. Print result.

Homework Exercise 2

1. Ask user to input multiple names in a single line
2. Print “Hello” message for each name in separate lines.

Functions

A bash function is a method used in shell scripts to group reusable code blocks.

Bash function is a piece of code that only runs when it is called. Functions enable you to reuse code. We can call the functions numerous times.

```
function function_name () {
  commands
}
```

```
#!/bin/bash

function Welcome () {
   echo "Welcome to Linux Lessons"
}

Welcome
```

CLARUSWAY©
WAY TO REINVENT YOURSELF

Passing Arguments to Functions

We can pass any number of arguments to the bash function in a similar way to passing command line arguments to a script. We simply supply them right after the function’s name, separated by a space. These parameters would be represented by $1, $2 and so on, corresponding to the position of the parameter after the function’s name.

```
#!/bin/bash

function Welcome () {
   echo "Welcome to Linux Lessons
$1 $2 $3"
}

Welcome Joe Matt Timothy
```

Output:

```
$./functions.sh
Welcome to Linux Lessons Joe Matt Timothy
```

Nested Functions

One of the useful features of functions is that they can call themselves and other functions.

```
#!/bin/bash

function_one () {
  echo "This is from the first
function"
  function_two
}

function_two () {
  echo "This is from the second
function"
}

function_one
```

Output:

```
$./nested.function.sh
This is from the first function
This is from the second function
```

Variables Scope

Global variables are variables that can be accessed from anywhere in the script regardless of the scope.
In Bash, by default all variables are defined as global, even if declared inside the function.
Local variables can be declared within the function body with the local keyword and can be used only inside that function.

```
#!/bin/bash

var1='global 1'
var2='global 2'

function var_scope () {
 local var1='function 1'
 local var2='function 2'
 echo -e "Inside function:\nvar1: $var1\nvar2: $var2"
}

echo -e "Before calling function:\nvar1: $var1\nvar2: $var2"

var_scope

echo -e "After calling function:\nvar1: $var1\nvar2: $var2"
```

local variable_name=value

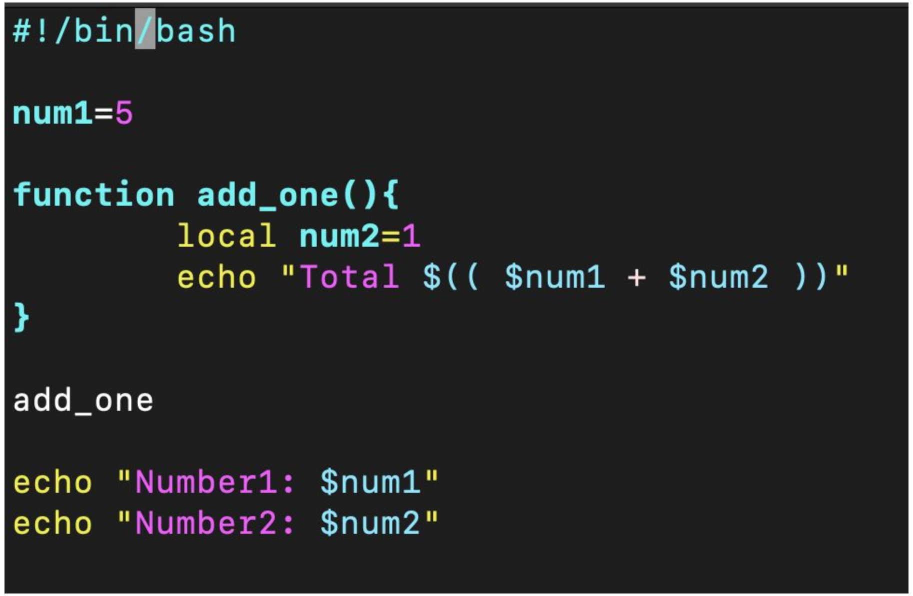
Output:

```
Before calling function:
var1: global 1
var2: global 2
Inside function:
var1: function 1
var2: function 2
After calling function:
var1: global 1
var2: global 2
```

Functions

Local variable

local variable_name=value



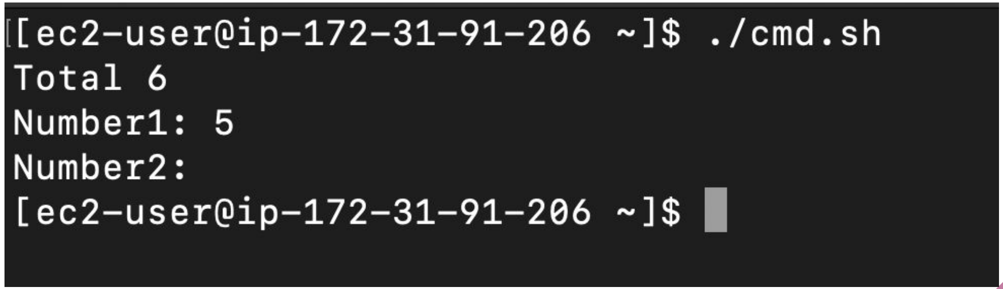
```
#!/bin/bash

num1=5

function add_one(){
        local num2=1
        echo "Total $(( $num1 + $num2 ))"
}

add_one

echo "Number1: $num1"
echo "Number2: $num2"
```



```
[ec2-user@ip-172-31-91-206 ~]$ ./cmd.sh
Total 6
Number1: 5
Number2:
[ec2-user@ip-172-31-91-206 ~]$
```

Homework Exercise 3

1. Create a function named **print_age** that accepts one argument

    Ask user to input his/her year of birth and store it to **local** **birth_year** variable

    Calculate **age** using current year value from the first argument

    Print **age** with a message

2. Call **print_age** function with **2023**

THANKS!

Any questions?

You can find me at:

- @sumod
- sumod@clarusway.com