SHELL PROGRAMMING PART 5

Looping Statements | Shell Script

Looping Statements in Shell Scripting: There are total 2 looping statements which can be used in bash programming

1. while statement
2. for statement

To alter the flow of loop statements, two commands are used they are,

1. break
2. continue

Their descriptions and syntax are as follows:

- **while statement**
  Here command is evaluated and based on the result loop will executed, if command raise to false then loop will be terminated
  Syntax

```
while command

do

   Statement to be executed

done
```

- **for statement**
  The for loop operate on lists of items. It repeats a set of commands for every item in a list.
  Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.
  Syntax

```
for var in word1 word2 ...wordn

do
```

```
    Statement to be executed
done
```

Example Programs

`for` loops iterate through a set of values until the list is exhausted:

---

for.sh

```
#!/bin/sh
for i in 1 2 3 4 5
do
  echo "Looping ... number $i"
done
```

Try this code and see what it does. Note that the values can be anything at all:

---

for2.sh

```
#!/bin/sh
for i in hello 1 * 2 goodbye
do
```

```
  echo "Looping ... i is set to $i"
done
```

This is well worth trying. Make sure that you understand what is happening here. Try it without the `*` and grasp the idea, then re-read the Wildcards section and try it again with the `*` in place. Try it also in different directories, and with the `*` surrounded by double quotes, and try it preceded by a backslash (`*`)
In case you don't have access to a shell at the moment (it is very useful to have a shell to hand whilst reading this tutorial), the results of the above two scripts are:

```
Looping .... number 1
Looping .... number 2
Looping .... number 3
Looping .... number 4
Looping .... number 5
```

and, for the second example:

```
Looping ... i is set to hello
Looping ... i is set to 1
Looping ... i is set to (name of first file in current directory)
    ... etc ...
Looping ... i is set to (name of last file in current directory)
Looping ... i is set to 2
Looping ... i is set to goodbye
```

So, as you can see, `for` simply loops through whatever input it is given, until it runs out of input.

While Loops

`while` loops can be much more fun! (depending on your idea of fun, and how often you get out of the house...)

---

while.sh

```
#!/bin/sh
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do
  echo "Please type something in (bye to quit)"
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

What happens here, is that the echo and read statements will run indefinitely until you type "bye" when prompted. Review Variables - Part I to see why we set `INPUT_STRING=hello` before testing it. This makes it a repeat loop, not a traditional while loop.

The colon (`:`) always evaluates to true; whilst using this can be necessary sometimes, it is often preferable to use a real exit condition. Compare quitting the above loop with the one below; see which is the more elegant. Also think of some situations in which each one would be more useful than the other:

---

while2.sh

```
#!/bin/sh
while :
do
  echo "Please type something in (^C to quit)"
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

Another useful trick is the `while read f` loop. This example uses the case statement, which we'll cover later. It reads from the file `myfile`, and for each line, tells you what language it thinks is being used. Each line must end with a LF (newline) - if `cat myfile` doesn't end with a blank line, that final line will not be processed.

---

while3a.sh

```
#!/bin/sh
while read f
do
  case $f in
```

```
      hello)           echo English     ;;
      howdy)           echo American    ;;
      gday)       echo Australian  ;;
      bonjour)    echo French;;
      "guten tag")     echo German;;
      *)          echo Unknown Language: $f
            ;;
   esac
done < myfile
```

On many Unix systems, this can also be done as:

---

while3b.sh

```
#!/bin/sh
while f=`line`
do
  .. process f ..
done < myfile
```

But since the `while read f` works with any *nix, and doesn't depend on the external program `line`, the former is preferable. See External Programs to see why this method uses the backtick (`).
Had I referred to `$i` (not `$f`) in the default ("Unknown Language") case above, there would have been no warnings or errors, even though `$i` has not been declared or defined. For example:

```
$ i=THIS_IS_A_BUG
$ export i
$ ./while3.sh something
Unknown Language: THIS_IS_A_BUG
$
```

So make sure that you avoid typos. This is also another good reason for using `${x}` and not just `$x` - if `x="A"` and you want to say "A1", you need `echo ${x}1`, as `echo $x1` will try to use the variable `x1`, which may not exist, or may be set to "`B2`," or anything else unexpected.
I recently found an old thread on Usenet which I had been involved in, where I actually learned more ... Google has it here..

A handy Bash (but not Bourne Shell) tip I learned recently from the Linux From Scratch project is:

```
mkdir rc{0,1,2,3,4,5,6,S}.d
```

instead of the more cumbersome:

```
for runlevel in 0 1 2 3 4 5 6 S
do
  mkdir rc${runlevel}.d
done
```

And this can be done recursively, too:

```
$ cd /
$ ls -ld {,usr,usr/local}/{bin,sbin,lib}
drwxr-xr-x    2 root     root         4096 Oct 26 01:00 /bin
drwxr-xr-x    6 root     root         4096 Jan 16 17:09 /lib
drwxr-xr-x    2 root     root         4096 Oct 27 00:02 /sbin
drwxr-xr-x    2 root     root        40960 Jan 16 19:35 usr/bin
drwxr-xr-x   83 root     root        49152 Jan 16 17:23 usr/lib
drwxr-xr-x    2 root     root         4096 Jan 16 22:22 usr/local/bin
drwxr-xr-x    3 root     root         4096 Jan 16 19:17 usr/local/lib
drwxr-xr-x    2 root     root         4096 Dec 28 00:44 usr/local/sbin
drwxr-xr-x    2 root     root         8192 Dec 27 02:10 usr/sbin
```

We will use while loops further in the Test and Case sections.

Example 1:
Implementing for loop with break statement

filter_none

brightness_4

```
#Start of for loop

for a in 1 2 3 4 5 6 7 8 9 10

do
```

```
    # if a is equal to 5 break the loop
    if [ $a == 5 ]
    then
        break
    fi
    # Print the value
    echo "Iteration no $a"
done
```

Output

```
$bash -f main.sh
Iteration no 1
Iteration no 2
Iteration no 3
Iteration no 4
```

Example 2:

Implementing for loop with continue statement

filter_none

brightness_4

```
for a in 1 2 3 4 5 6 7 8 9 10
do
    # if a = 5 then continue the loop and
    # don't move to line 8
```

```
    if [ $a == 5 ]
    then
        continue
    fi
    echo "Iteration no $a"
done
```

Output

```
$bash -f main.sh
Iteration no 1
Iteration no 2
Iteration no 3
Iteration no 4
Iteration no 6
Iteration no 7
Iteration no 8
Iteration no 9
Iteration no 10
```

Example 3:
Implementing while loop

filter_none

brightness_4

```
a=0
```

```
# -lt is less than operator


#Iterate the loop until a less than 10
while [ $a -lt 10 ]
do
    # Print the values
    echo $a

    # increment the value
    a=`expr $a + 1`
done
```

Output:

```
$bash -f main.sh
0
1
2
3
4
5
6
7
8
```

```
9
```

Note: Shell scripting is a case-sensitive language, which means proper syntax has to be followed while writing the scripts.

Example

Here is a simple example of loop nesting. Let's add another countdown loop inside the loop that you used to count to nine −

```
#!/bin/sh


a=0

while [ "$a" -lt 10 ]    # this is loop1

do

   b="$a"

   while [ "$b" -ge 0 ]  # this is loop2

   do

      echo -n "$b "

      b=`expr $b - 1`

   done

   echo

   a=`expr $a + 1`
```

```
done
```

This will produce the following result. It is important to note how **echo -n**works here. Here **-n** option lets echo avoid printing a new line character.

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

Example

Here is a simple example that uses the **while** loop to display the numbers zero to nine −

```
#!/bin/sh


a=10


until [ $a -lt 10 ]

do

   echo $a
```

```
   a=`expr $a + 1`

done
```

This loop continues forever because **a** is always **greater than** or **equal to 10**and it is never less than 10.

The break Statement

The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

Syntax

The following **break** statement is used to come out of a loop −

```
break
```

The break command can also be used to exit from a nested loop using this format −

```
break n
```

Here **n** specifies the **nth** enclosing loop to the exit from.

Example

Here is a simple example which shows that loop terminates as soon as **a**becomes 5 −

```
#!/bin/sh


a=0


while [ $a -lt 10 ]
```

```
do
   echo $a
   if [ $a -eq 5 ]
   then
      break
   fi
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result –

```
0
1
2
3
4
5
```

Here is a simple example of nested for loop. This script breaks out of both loops if **var1 equals 2** and **var2 equals 0** –

```
#!/bin/sh


for var1 in 1 2 3
do
   for var2 in 0 5
```

```
   do
      if [ $var1 -eq 2 -a $var2 -eq 0 ]
      then
         break 2
      else
         echo "$var1 $var2"
      fi
   done
done
```

Upon execution, you will receive the following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from the inner loop as well.

```
1 0
1 5
```

The continue statement

The **continue** statement is similar to the **break** command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

Syntax

```
continue
```

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

```
continue n
```

Here **n** specifies the **nth** enclosing loop to continue from.

Example

The following loop makes use of the **continue** statement which returns from the continue statement and starts processing the next statement –

```
#!/bin/sh


NUMS="1 2 3 4 5 6 7"


for NUM in $NUMS
do
   Q=`expr $NUM % 2`
   if [ $Q -eq 0 ]
   then
      echo "Number is an even number!!"
      continue
   fi
```

```
   echo "Found odd number"

done
```

Upon execution, you will receive the following result –

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

he shell performs substitution when it encounters an expression that contains one or more special characters.

Example

Here, the printing value of the variable is substituted by its value. Same time, **"\n"** is substituted by a new line –

```
#!/bin/sh


a=10

echo -e "Value of a is $a \n"
```

You will receive the following result. Here the **-e** option enables the interpretation of backslash escapes.

```
Value of a is 10
```

Following is the result without **-e** option −

```
Value of a is 10\n
```

The following escape sequences which can be used in echo command −

Sr.No.	Escape & Description
1	**** backslash
2	**\a** alert (BEL)
3	**\b** backspace
4	**\c** suppress trailing newline
5	**\f** form feed

6	**\n** new line
7	**\r** carriage return
8	**\t** horizontal tab
9	**\v** vertical tab

You can use the **-E** option to disable the interpretation of the backslash escapes (default).

You can use the **-n** option to disable the insertion of a new line.