Loops

In this lesson, you will cover for loops, case statements, and while loops in bash. This lesson will quickly take you through the various forms of looping that you might come across, including 'while' loops, 'case' statements, 'for' loops, and 'C-style' loops.

WE'LL COVER THE FOLLOWING

- How Important is this Lesson?
- `for` Loops
- `while` Loops in Bash
- `case` Statements
- `case` Statements and Command Line Options
- What You Learned
- What Next?
- Exercises

How Important is this Lesson?

For and **while** loops are a basic construct in most programming languages, and bash is no exception.

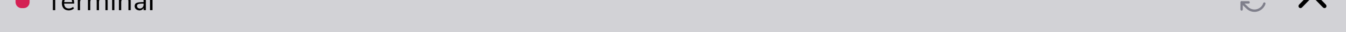
`for` Loops

First you’re going to run a for loop in a ‘traditional’ way:

```
for (( i=0; i < 20; i++ ))
do
  echo $i
  echo $i > file${i}.txt
done
ls
```

Type the above code into the terminal in this lesson.

Terminal



- You just created twenty files, each with a number in them using a `for` loop in the 'C' language style (**line 1**)
- Note there's no `$` sign involved in the variable when it's in the double parentheses!
- **Line 3** `echo`es the value of `i` in each iteration
- **line 4** redirects that value to a file called `file{$i}.txt`, where `${i}` is replaced with its value in that iteration
- **Line 6** shows the listing of files created

```
for f in $(ls *txt)
do
    echo "File $f contains: $(cat $f)"
done
```

Type the above code into the terminal in this lesson.

It's our old friend the command substitution! The command substitution lists all the files we have.

This `for` loop uses the `in` keyword to separate the variable each iteration will assign to `f` and the list to take items from. Here bash evaluates the output of the `ls` command and uses that as the list, but we could have written something like:

```
for f in file1.txt file2.txt file3.txt
do
    echo "File $f contains: $(cat $f)"
done
```

Type the above code into the terminal in this lesson.

with a similar effect.

`while` Loops in Bash

While loops also exist in bash. Try and work out what's going on in this trivial example:

```
n=0
while [[ ! -a newfile ]]
do
    ((n++))
    echo "In iteration $n"
    if [[ $(cat file${n}.txt) == "15" ]]
    then
            touch newfile
    fi
done
```

Type the above code into the terminal in this lesson.

- **Line 1** creates an `n` variable
- **Line 2-3** creates a *while* loop that increments the `n` variable on each iteration (**line 4**)
- **Line 5** output which iteration the loop is on
- If the file with the name `file${n}.txt` (where the `${n}` is replaced by the value of the variable `n` at the time) contains the string `15` (**line 6**), then a `newfile` file is created
- The while loop condition on **line 2** finishes when this `newfile` file exists.

I often use while loops in the following 'infinite loop' form when running quick scripts on the command line:

```
n=0
while true
do
    sleep 1
    ((n++))
    echo $n seconds have passed
    if [[ $n -eq 60 ]]           # When n reaches 60...
    then
        break                    # Break out of the while loop
    fi
done
```

Type the above code into the terminal in this lesson.

The above code is an 'infinite loop' because the while condition is `true`, so will never break out by itself. The `break` statement on **line 11**, if reached, will exit the while loop.

`case` Statements

case Statements

Case statements may also be familiar from other languages. In bash, they're most frequently used when processing command-line arguments within a script.

Before you look at a realistic case statement, type in this trivial one:

```
a=1                                        # Initialize 'a' variable
case "$a" in                               # Start case statement
1) echo 'a is 1'; echo 'ok1';;             # If a == 1, then output ok1
2) echo 'a is 2'; echo 'ok2';;             # If a == 2, then output ok2
*) echo 'a is unmatched'; echo 'failure';; # If nothing was matched, output 'failure'
esac                                       # End case statement
```

Type the above code into the terminal in this lesson.

Try triggering the `a is 2` case, and the `a is unmatched` case.

There are a few of new bits of syntax you may not have seen before.

- The double semi-colons `;;` indicate that the next matching case is coming (rather than just another statement, as indicated by a single semi-colon)
- The `1)` indicates what the `case` value `$a` should match. These values follow the globbing rules (so `*` will match anything)
    - Try adding quotes around the values, or glob values, or matching a longer string with spaces
- The `esac` indicates the `case` statement is finished

`case` Statements and Command Line Options

`case` statements are most often seen in the context of processing command-line options within shell scripts. There is a helper `builtin` just for this purpose: `getopts`.

Now you will write a more realistic example, and more like what is seen in 'real' shell scripts that uses `getopts`.

Create a file `case.sh` to try out a `case` statement with `getopts`:

```
cat > case.sh << 'EOF'
```

```
#!/bin/bash
while getopts "ab:c" opt
do
    case "$opt" in
    a) echo '-a invoked';;
    b) echo "-b invoked with argument: ${OPTARG}";;
    c) echo '-c invoked';;
    esac
done
EOF
chmod +x case.sh
```

Type the above code into the terminal in this lesson.

- **Line 1** creates a file `case.sh` using a *here doc* (see here if you want to learn about these now)
- **Line 2** tells the system that this is a bash script
- **Line 3** Starts the while loop, calling the builtin `getopts`. The builtin returns true while there are more flags to process, and sets the `opt` variable to the flag being processed on this iteration. The string `ab:c` is the specification of the flags, where the `:` indicates that the previous flag mentioned takes an argument
- **Line 5** Starts a `case` statement based on the `opt` value set in this iteration
- **Line 6** deals with the `-a` flag. Note that the `-` sign is removed by `getopts` when passing into `opt`
- **Line 7** deals with the `-b` flag, which expects an argument. The argument is placed in the `OPTARG` variable by `getopts`
- **Line 8** deals with the `-a` flag

Run the above with various combinations and try and understand what’s happening:

```
./case.sh -a
./case.sh -b
./case.sh -b "an argument"
./case.sh -a -b -c
./case.sh
```

Type the above code into the terminal in this lesson.

This is how many bash scripts pick up arguments from the command line and process them.

Loops Quiz

1 How do you get out of a `while` loop while it is running?

COMPLETED 0%

1 of 2 ‹ ›

What You Learned

You’ve now covered the main methods of looping in bash. Nothing about looping in bash should come as a big surprise in future!

What Next?

Next you will learn about **exit codes**, which will power up your ability to write neater bash code and better scripts.

Exercises

1) Find a real program that uses `getopts` to process arguments and figure out what it’s doing.

2) Write a while loop to check where a file exists every few seconds. When it does, break out of the loop with a message.