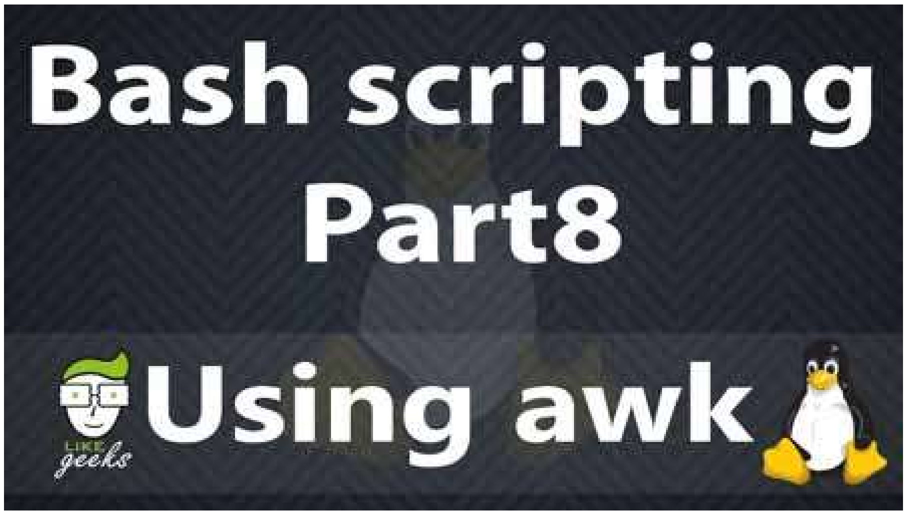
Sunday, March 05, 2017

LIKE GEEKS



LINUX

# 30 Examples For Awk Command In Text Processing

February 21, 2017 admin 3 Comments

On the previous post we've talked about **sed Linux command** and we've seen many examples of using it in text processing and how it is good in this, nobody can deny that sed is very handy tool but it has some limitations, sometimes you need a more advanced tool for manipulating data, one that provides a more programming-like environment giving you more control to modify data in a file more robust. This is where **awk command** comes in.

The awk command or GNU awk specifically because there are many extensions for awk out there takes stream editing one step further than the sed editor by providing a programming language instead of just editor commands. Within the awk **programming language**, you can do the following

- Define variables to store data.
- Use arithmetic and string operators to operate on data.
- Use structured programming concepts and control flow, such as if-then statements and loops, to add logic to your text processing.
- Generate formatted reports

Actually generating formatted reports comes very handy when working with log files contain hundreds or maybe millions of lines and output a readable report that you can benefit from.

**our main points are:**

## awk command options

The awk command has a basic format as follows

```
$ awk options program file
```

And those are some of the **options** for awk command that you will use often:

**-F fs** Specifies a file separator for the fields in a line

**-f file** specifies a file name to read the program from

**-v var=value** Defines a variable and default value used in the awk command

**–**mf **N** specifies the maximum number of fields to process in the data file

**–**mr **N** Specifies the maximum record size in the data file

**-W** keyword Specifies the compatibility mode or warning level for awk

The real power of awk is in the program script. You can write scripts to read the data within a text line and then manipulate and display the data to create any type of output report.

## Reading the program script from the command line

awk program script is defined by opening and closing braces. You must place script commands between the two braces and because the awk command line assumes that the script is a single text string, you must enclose your script in single quotation marks like this

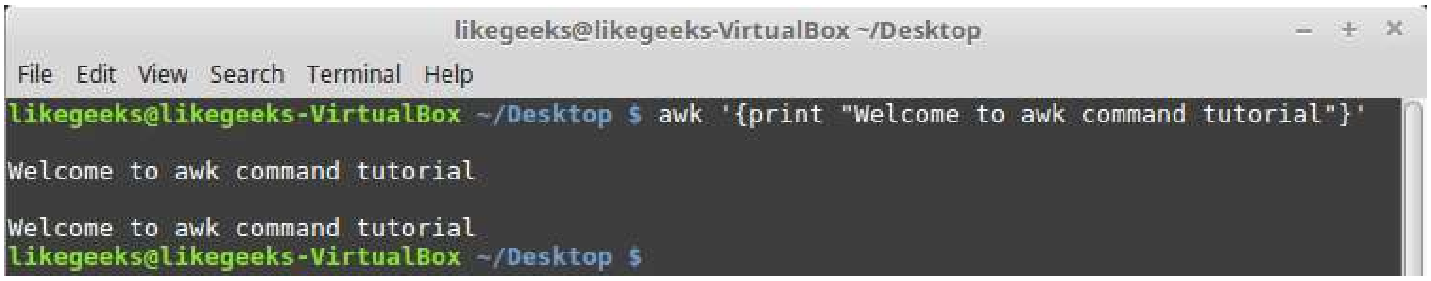
```
$ awk '{print "Welcome to awk command tutorial"}'
```

If you run this command nothing will happen!! And that because no filename was defined in the command line.

The awk command retrieves data from STDIN. When you run the program, it just waits for text to come in via STDIN

If you type a line of text and press the Enter, the awk command runs the text through the program script. Just like the sed editor, the awk command executes the program script on each line of text available in the data stream. Because the program script is set to display a fixed text string, you get the same text output.

```
$ awk '{print "Welcome to awk command tutorial "}'
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{print "Welcome to awk command tutorial"}'

Welcome to awk command tutorial

Welcome to awk command tutorial
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Any strings we typed return the same welcome string we provide.

To terminate the program we have to send End-of-File (EOF) character. The Ctrl+D key combination generates an EOF character in bash. Maybe you disappointed by this example but wait for the awesomeness.

## Using data field variables

One of the primary features of awk is its ability to manipulate data in the text file. It does this by automatically assigning a variable to each data element in a line. By default, awk assigns the following variables to each data field it detects in the line of text

- $0 represents the entire line of text.
- $1 represents the first data field in the line of text.
- $2 represents the second data field in the line of text.

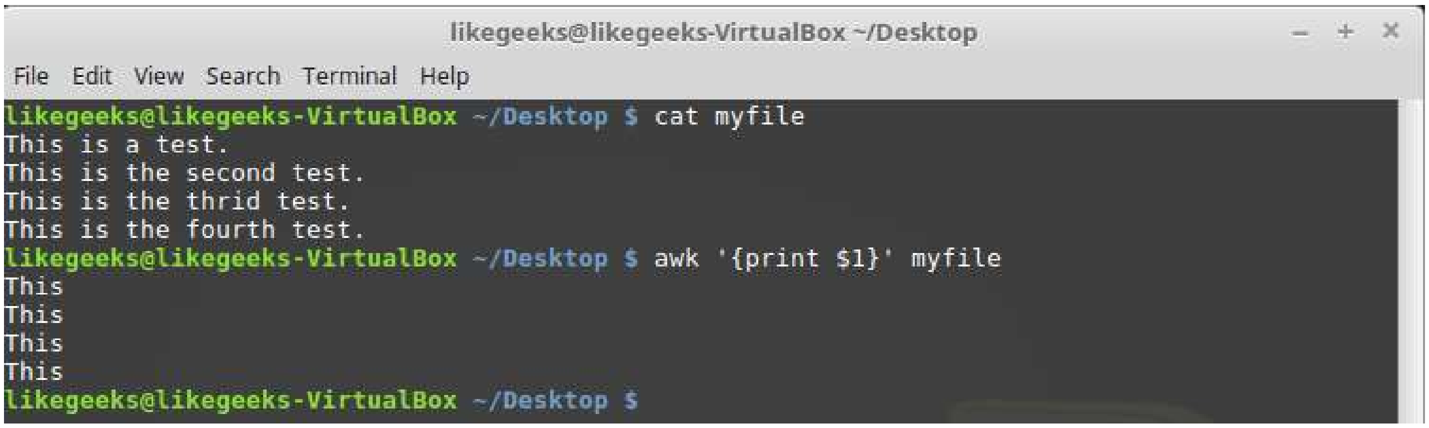
- $n represents the *n*th data field in the line of text.

Each data field is determined in a text line by a field separation character.

The default field separation character in awk is any whitespace character like tab or space

Look at the following file and see how awk deal with it

```
$ awk '{print $1}' myfile
```

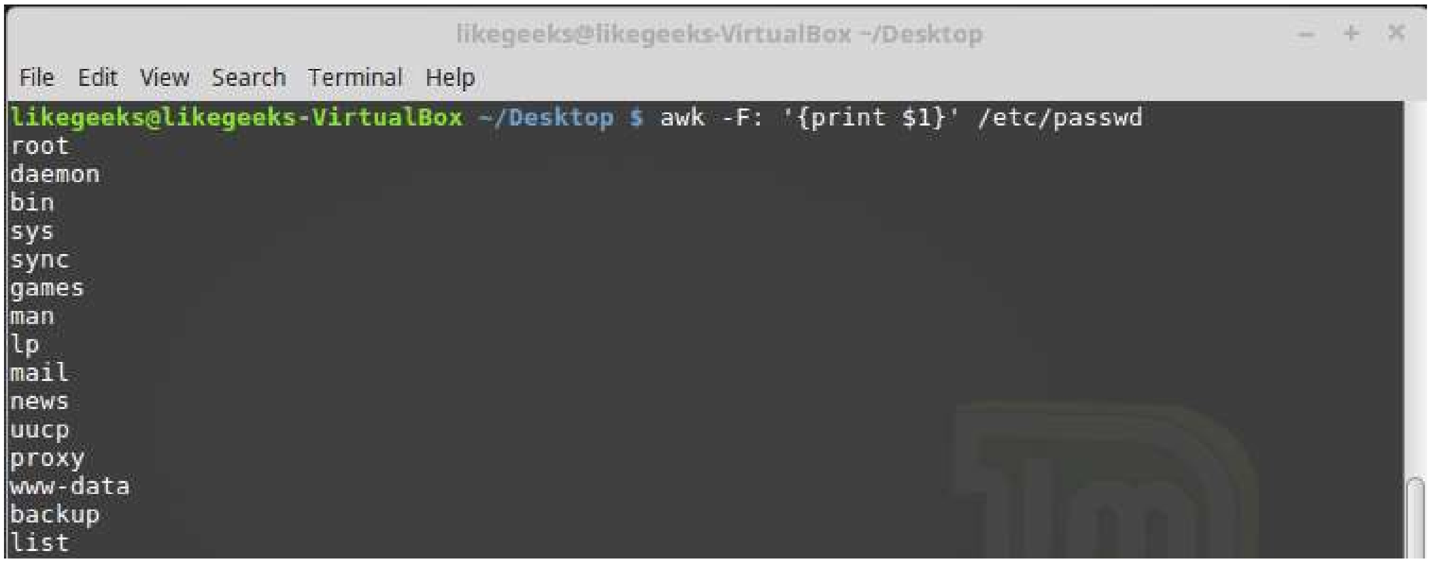

likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
This is a test.
This is the second test.
This is the thrid test.
This is the fourth test.
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{print $1}' myfile
This
This
This
This
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

This command uses the $1 field variable to display only the first data field for each line of text.

Sometimes the separator in some files is not space or a tab but something else. You can specify it using –F option

```
$ awk -F: '{print $1}' /etc/passwd
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk -F: '{print $1}' /etc/passwd
root
daemon
bin
sys
sync
games
man
lp
mail
news
uucp
proxy
www-data
backup
list
```

This command displays the first data field in the passwd file. Because the /etc/passwd file uses a colon to separate the data fields.

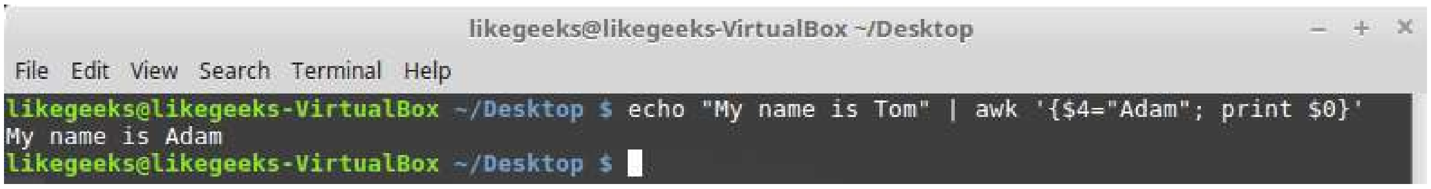
## Using multiple commands

Any programming language wouldn't be very useful if you could only execute one command.

The awk programming language allows you to combine commands into a normal program.

To use multiple commands on the command line, just place a semicolon between each command

```
$ echo "My name is Tom" | awk '{$4="Adam"; print $0}'
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "My name is Tom" | awk '{$4="Adam"; print $0}'
My name is Adam
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The first command assigns a value to the $4 field variable. The second command then prints the entire line.

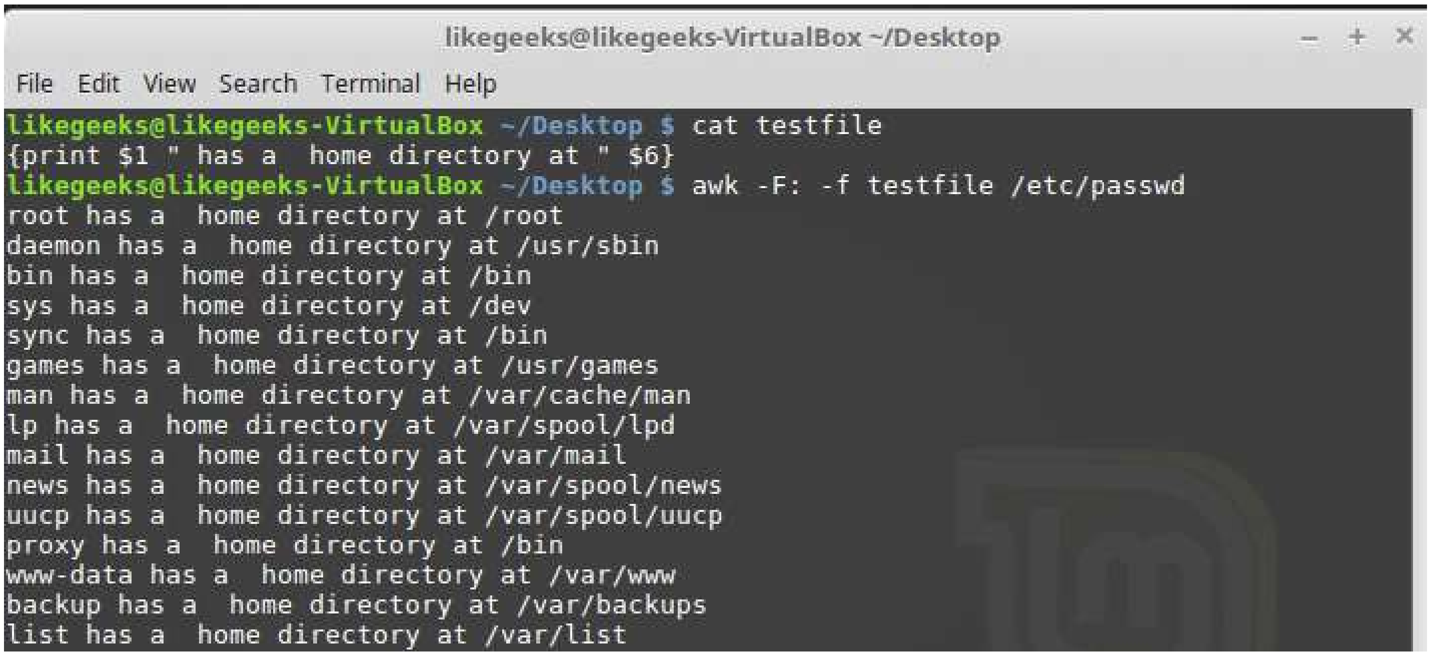
## Reading the program from a file

As with sed command, the awk command allows you to store your scripts in a file and refer to them in the command line with the –f option

Our file contains this script

```
{print $1 " has a  home directory at " $6}
```

```
$ awk -F: -f testfile /etc/passwd
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat testfile
{print $1 " has a  home directory at " $6}
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk -F: -f testfile /etc/passwd
root has a  home directory at /root
daemon has a  home directory at /usr/sbin
bin has a  home directory at /bin
sys has a  home directory at /dev
sync has a  home directory at /bin
games has a  home directory at /usr/games
man has a  home directory at /var/cache/man
lp has a  home directory at /var/spool/lpd
mail has a  home directory at /var/mail
news has a  home directory at /var/spool/news
uucp has a  home directory at /var/spool/uucp
proxy has a  home directory at /bin
www-data has a  home directory at /var/www
backup has a  home directory at /var/backups
list has a  home directory at /var/list
```

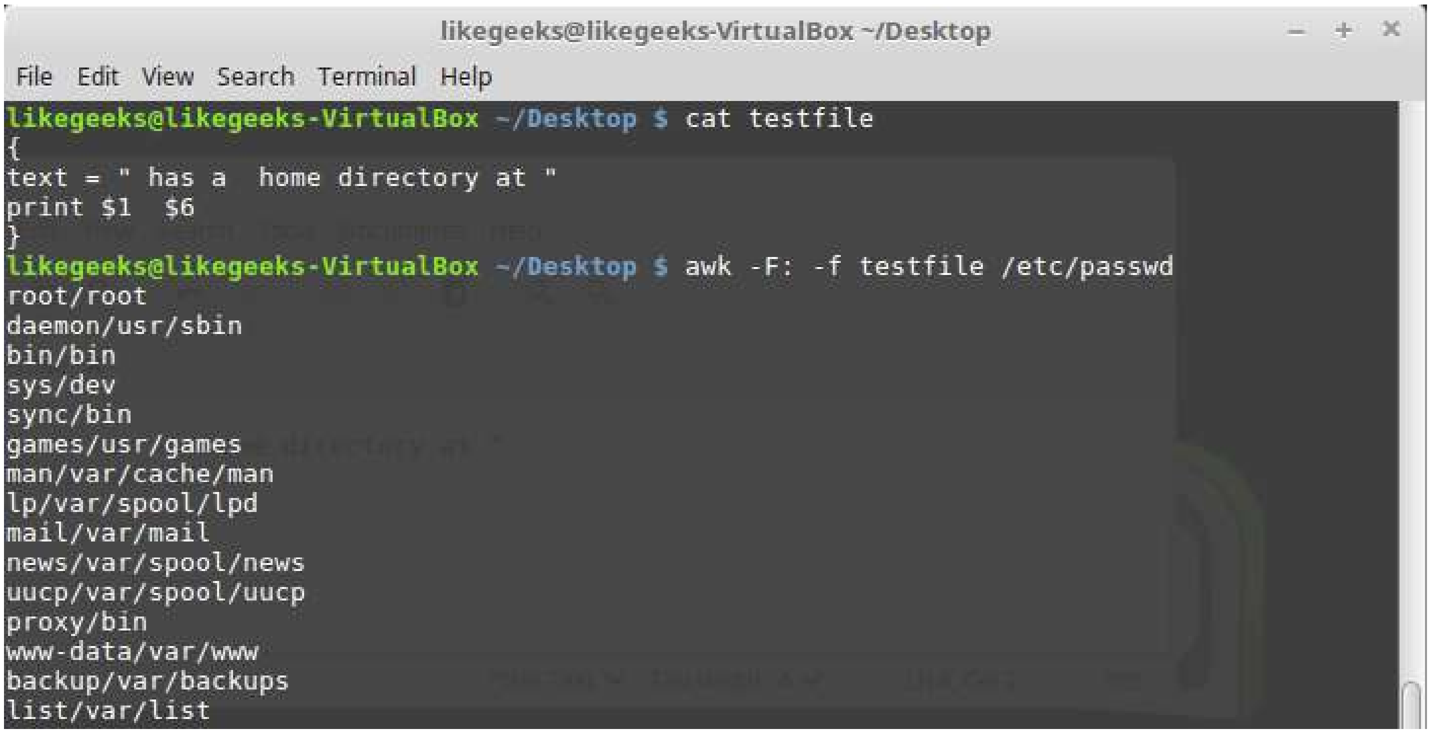
Here we print the username which is the first field $1 and the home path which is the sixth field $6 from /etc/passwd and we specify the file that contains that script which is called myscipt with -f option and surely the separator is specified with capital -F which is the colon

You can specify multiple commands in the script file. just place each command on a separate line. You don't need to use semicolons

This is our file

```
{
text = " has a  home directory at "
print $1 text $6
}
```

```
$ awk -F: -f testfile /etc/passwd
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat testfile
{
text = " has a  home directory at "
print $1  $6
}
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk -F: -f testfile /etc/passwd
root/root
daemon/usr/sbin
bin/bin
sys/dev
sync/bin
games/usr/games
man/var/cache/man
lp/var/spool/lpd
mail/var/mail
news/var/spool/news
uucp/var/spool/uucp
proxy/bin
www-data/var/www
backup/var/backups
list/var/list
```

Here we define a variable that holds a text string used in the print command.

## Running scripts before processing data

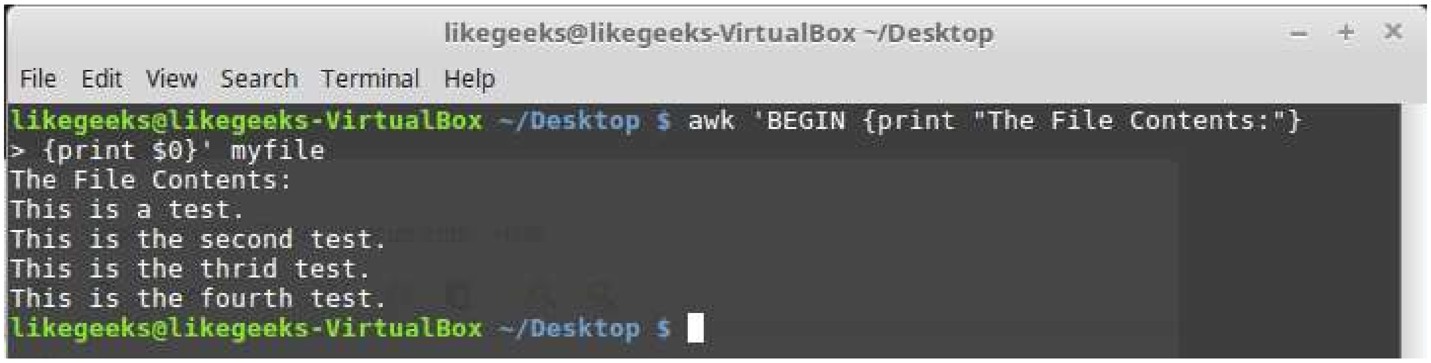
Sometimes, you may need to run a script before processing data, such as to create a header section for a report or something similar.

The BEGIN keyword is used to accomplish this. It forces awk to execute the script specified after the BEGIN keyword and before awk reads the data

```
$ awk 'BEGIN {print "Hello World!"}'
```

Let's apply it to something we can see the result

```
$ awk 'BEGIN {print "The File Contents:"}
{print $0}' myfile
```



File Edit View Search Terminal Help

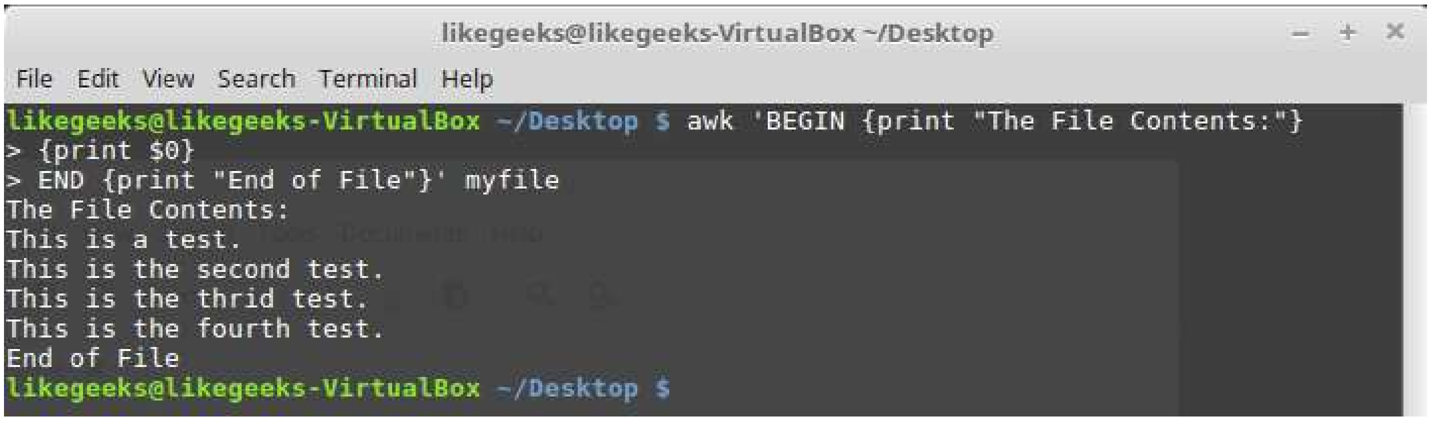
```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN {print "The File Contents:"}
> {print $0}' myfile
The File Contents:
This is a test.
This is the second test.
This is the thrid test.
This is the fourth test.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Now after awk command executes the BEGIN script, it uses the second script to process any file data. Be careful when doing this; both of the scripts are still considered one text string on the awk command line. You need to place your single quotation marks accordingly.

## Running scripts after processing data

The END keyword allows you to specify a program script that awk command executes after reading the data

```
$ awk 'BEGIN {print "The File Contents:"}
{print $0}
END {print "End of File"}' myfile
```



File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN {print "The File Contents:"}
> {print $0}
> END {print "End of File"}' myfile
The File Contents:
This is a test.
This is the second test.
This is the thrid test.
This is the fourth test.
End of File
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

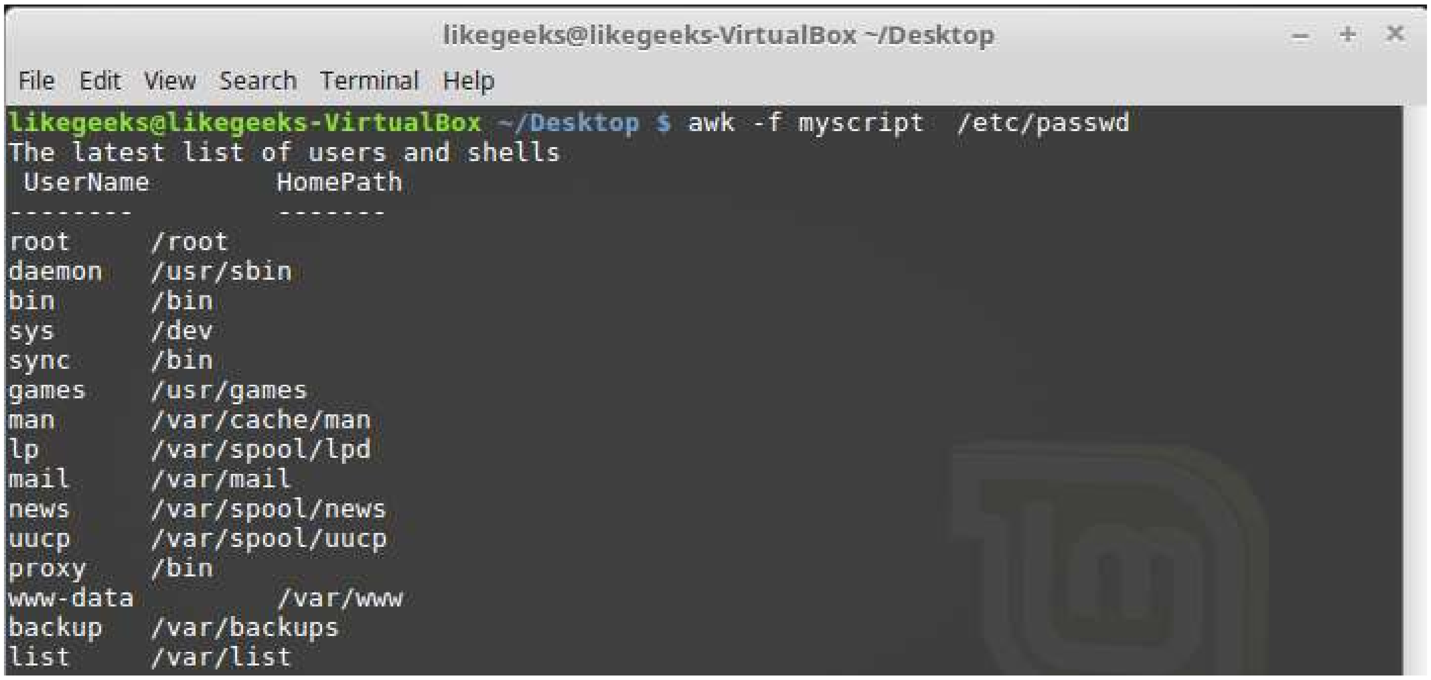
When the awk command is finished printing the file contents, it executes the commands in the END script. This is useful to use to add the footer as an example.

We can put all these elements together into a nice little script file

```
BEGIN {
print "The latest list of users and shells"
print " UserName \t HomePath"
print "-------- \t -------"
FS=":"
}
{
print $1 " \t " $6
}
END {
print "The end"
}
```

This script uses the BEGIN script to create a header section for the report. It also defines the file separator FS and prints the footer at the end. Then we use the file

```
$ awk -f myscript  /etc/passwd
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk -f myscript  /etc/passwd
The latest list of users and shells
 UserName        HomePath
--------         -------
root     /root
daemon   /usr/sbin
bin      /bin
sys      /dev
sync     /bin
games    /usr/games
man      /var/cache/man
lp       /var/spool/lpd
mail     /var/mail
news     /var/spool/news
uucp     /var/spool/uucp
proxy    /bin
www-data         /var/www
backup   /var/backups
list     /var/list
```

This gives you a small taste of the power available when you use simple awk scripts.

## Built-in variables

The awk command uses built-in variables to reference specific features within the program data

We've seen the data field variables $1, $2 and so on to extract data fields, we also deal with the field separator FS which is by default is a whitespace character, such as space or a tab.

But those are not the only variables, there are more built-in variables

The following list is some of the built-in variables that awk command use:

**FIELDWIDTHS** A space-separated list of numbers defining the exact width (in spaces) of each data field

**FS** Input field separator character

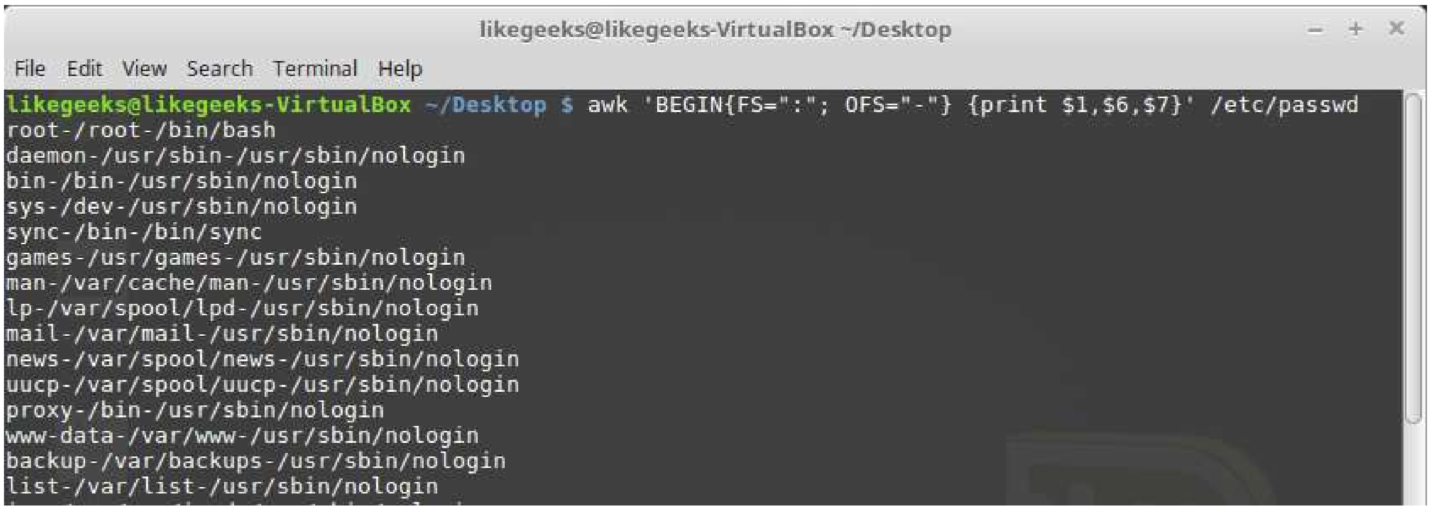
**RS** Input record separator character

**OFS** Output field separator character

**ORS** Output record separator character

By default, awk sets the OFS variable to a space, By setting the OFS variable, you can use any string to separate data fields in the output

```
$ awk 'BEGIN{FS=":"; OFS="-"} {print $1,$6,$7}' /etc/passwd
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{FS=":"; OFS="-"} {print $1,$6,$7}' /etc/passwd
root-/root-/bin/bash
daemon-/usr/sbin-/usr/sbin/nologin
bin-/bin-/usr/sbin/nologin
sys-/dev-/usr/sbin/nologin
sync-/bin-/bin/sync
games-/usr/games-/usr/sbin/nologin
man-/var/cache/man-/usr/sbin/nologin
lp-/var/spool/lpd-/usr/sbin/nologin
mail-/var/mail-/usr/sbin/nologin
news-/var/spool/news-/usr/sbin/nologin
uucp-/var/spool/uucp-/usr/sbin/nologin
proxy-/bin-/usr/sbin/nologin
www-data-/var/www-/usr/sbin/nologin
backup-/var/backups-/usr/sbin/nologin
list-/var/list-/usr/sbin/nologin
```

The FIELDWIDTHS variable allows you to read records without using a field separator character.

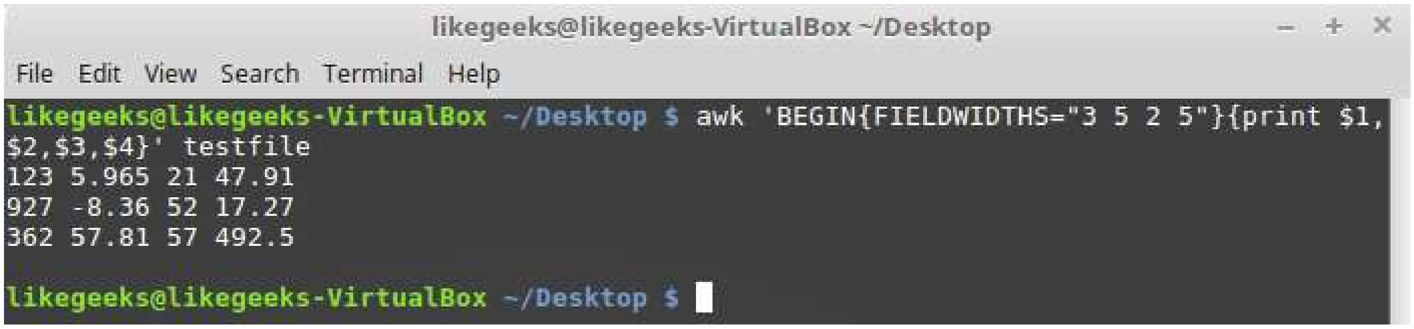
In some situations, instead of using a field separator, data is placed in specific columns within the record. In these instances, you must set the FIELDWIDTHS variable to match the layout of the data in the records

After you set the FIELDWIDTHS variable, awk ignores the FS and calculates data fields based on the provided field width sizes

Suppose we have this content

```
1235.9652147.91

927-8.365217.27

36257.8157492.5
```

```
$ awk 'BEGIN{FIELDWIDTHS="3 5 2 5"}{print $1,$2,$3,$4}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{FIELDWIDTHS="3 5 2 5"}{print $1,
$2,$3,$4}' testfile
123 5.965 21 47.91
927 -8.36 52 17.27
362 57.81 57 492.5

likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Look at the output. The FIELDWIDTHS variable defines four data fields, and awk command parses the data record accordingly. The string of numbers in each record is split based on the defined field width values

The RS and ORS variables define how your awk command handles records in the data. By default, awk sets the RS and ORS variables to the newline character which means that each new line of text in the input data stream is a new record

Sometimes, you run into situations where data fields are spread across multiple lines in the data stream

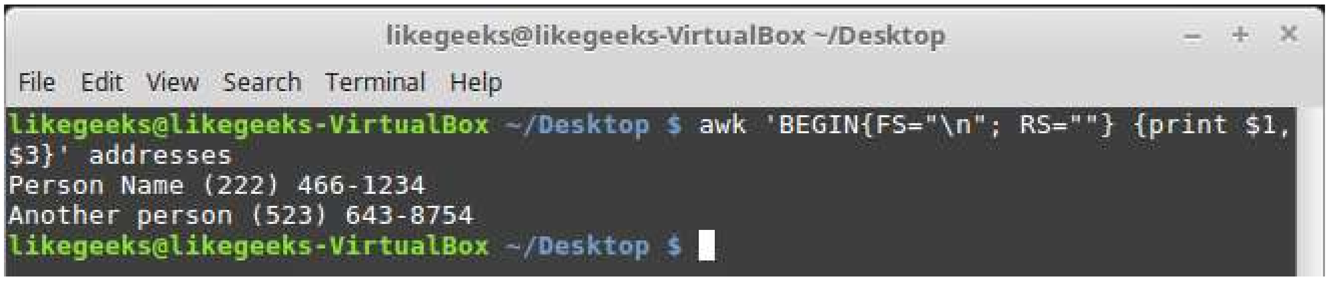
Like an address and phone number, each on a separate line

```
Person Name

123 High Street

(222) 466-1234


Another person

487 High Street

(523) 643-8754
```

If you try to read this data using the default FS and RS variable values, awk reads each line as a separate record and interprets each space in the record as a field separator. This is not what you want

To solve this problem, you need to set the FS variable to the newline character. This indicates that each line in the data is a separate field and all the data on a line belongs to the data field and set the RS variable to an empty string. The awk command interprets each blank line as a record separator

```
$ awk 'BEGIN{FS="\n"; RS=""} {print $1,$3}' addresses
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{FS="\n"; RS=""} {print $1,
$3}' addresses
Person Name (222) 466-1234
Another person (523) 643-8754
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Awesome! The awk command interpreted each line in the file as a data field and the blank lines as record separators.

## Data variables

Besides the built-in variables we've seen, awk provides some other built-in variables to help you know what's going on with your data and extract information from the shell environment

**ARGC** The number of command line parameters present

**ARGIND** The index in ARGV of the current file being processed

**ARGV** An array of command line parameters

**ENVIRON** An associative array of the current shell environment variables and their values

**ERRNO** The system error if an error occurs when reading or closing input files

**FILENAME** The filename of the data file used for input to the awk command

**FNR** The current record number in the data file

**IGNORECASE** If set to a non-zero value ignores the case of characters in strings used in the awk command

**NF** The total number of data fields in the data file

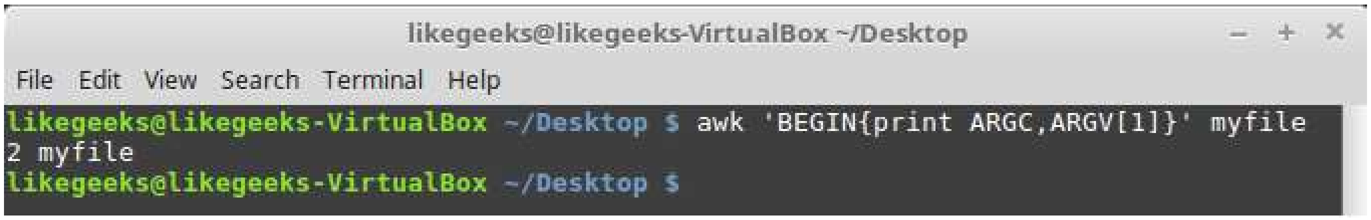
**NR** The number of input records processed

You should recognize a few of these variables from previous posts about **shell scripting** series

The ARGC and ARGV variables allow you to retrieve the number of command line parameters

This can be a little tricky because awk command doesn't count the script as part of the command line parameters

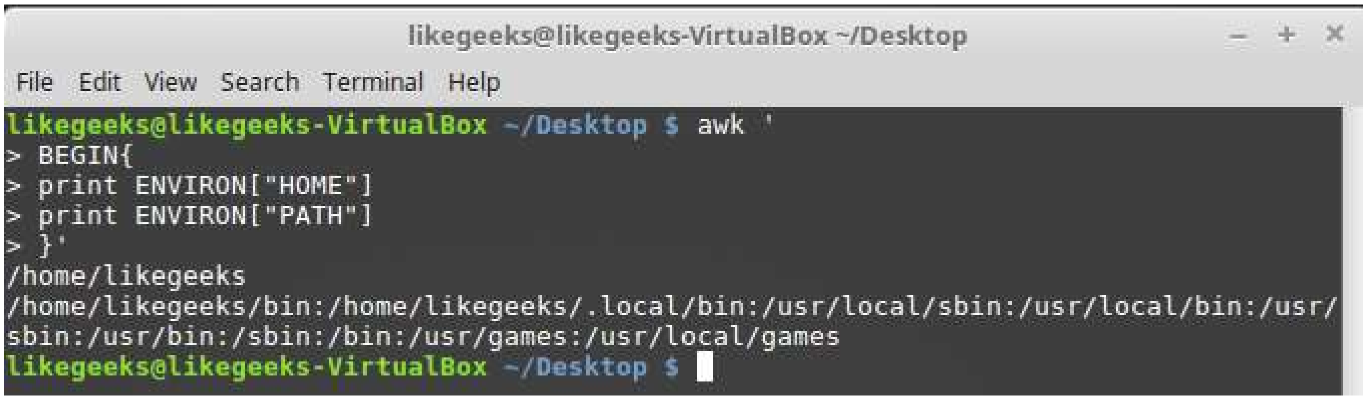
```
$ awk 'BEGIN{print ARGC,ARGV[1]}' myfile
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{print ARGC,ARGV[1]}' myfile
2 myfile
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The ENVIRON variable uses an associative array to retrieve shell environment variables like this.

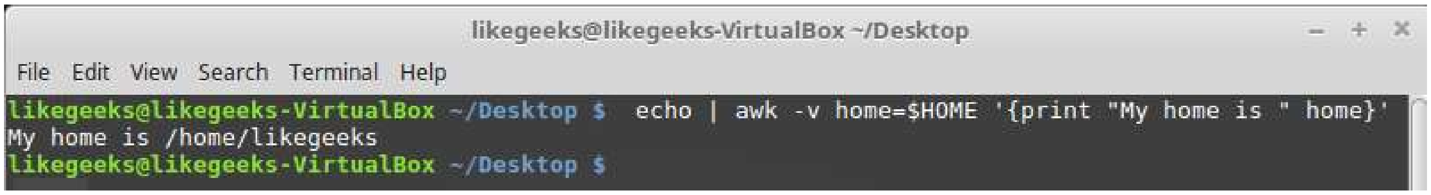
```
$ awk '
BEGIN{
print ENVIRON["HOME"]
print ENVIRON["PATH"]
}'
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '
> BEGIN{
> print ENVIRON["HOME"]
> print ENVIRON["PATH"]
> }'
/home/likegeeks
/home/likegeeks/bin:/home/likegeeks/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

You can use shell variables without ENVIRON variables like this

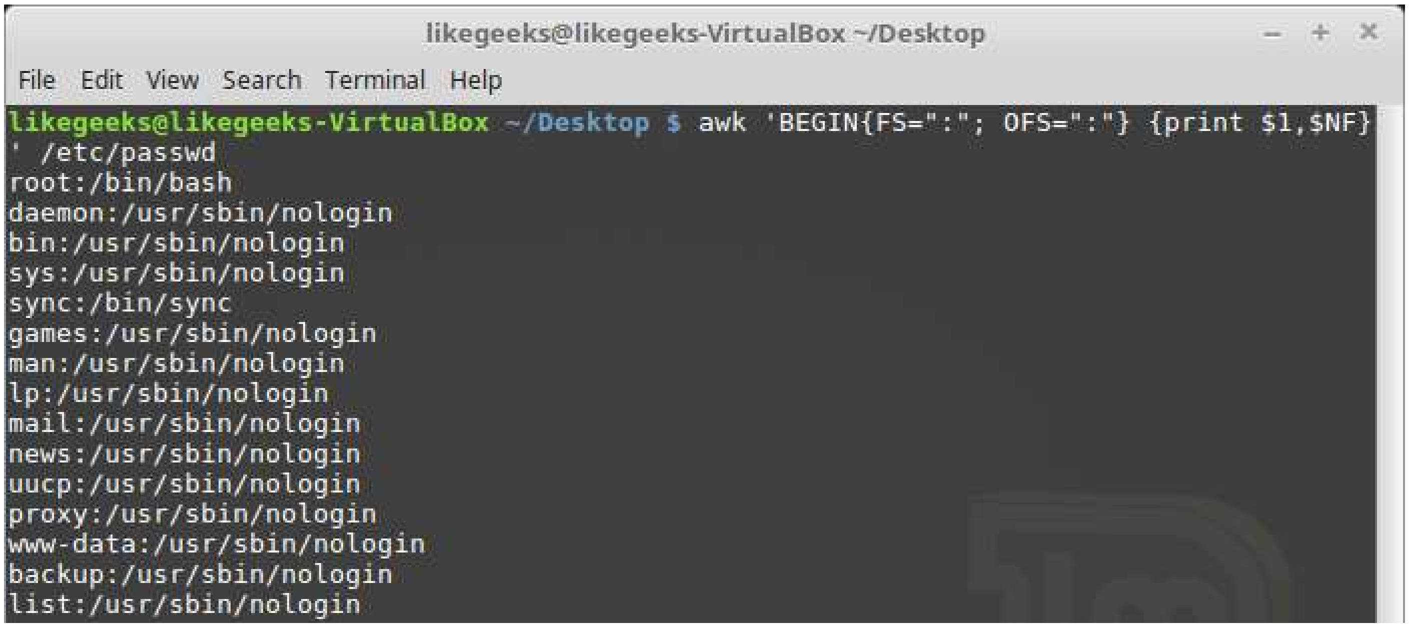
```
$  echo | awk -v home=$HOME '{print "My home is " home}'
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $  echo | awk -v home=$HOME '{print "My home is " home}'
My home is /home/likegeeks
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The NF variable allows you to specify the last data field in the record without having to know its position

```
$ awk 'BEGIN{FS=":"; OFS=":"} {print $1,$NF}' /etc/passwd
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

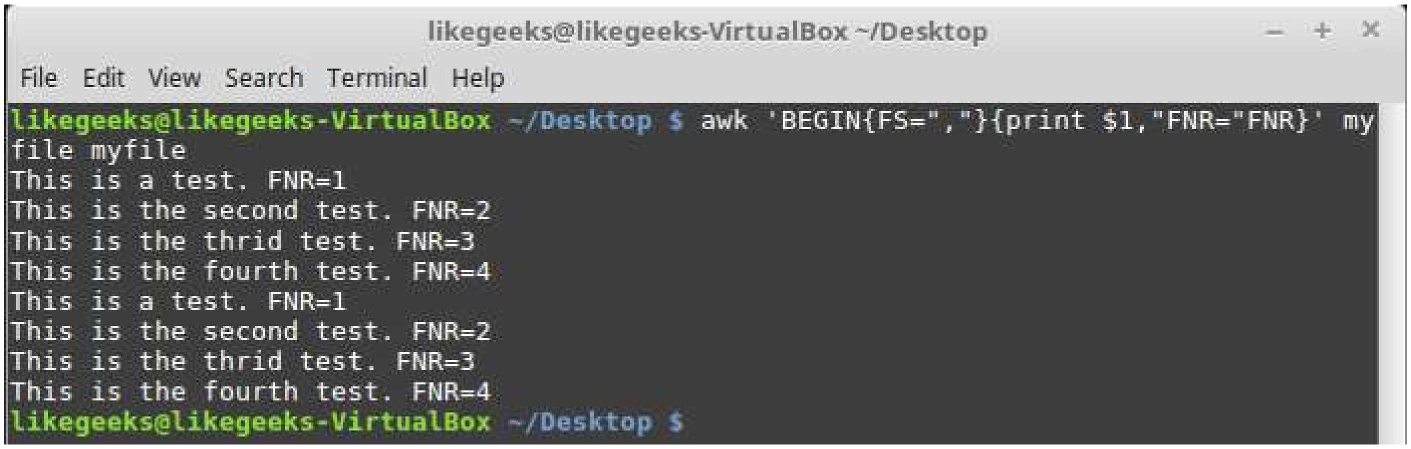
```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{FS=":"; OFS=":"} {print $1,$NF}
' /etc/passwd
root:/bin/bash
daemon:/usr/sbin/nologin
bin:/usr/sbin/nologin
sys:/usr/sbin/nologin
sync:/bin/sync
games:/usr/sbin/nologin
man:/usr/sbin/nologin
lp:/usr/sbin/nologin
mail:/usr/sbin/nologin
news:/usr/sbin/nologin
uucp:/usr/sbin/nologin
proxy:/usr/sbin/nologin
www-data:/usr/sbin/nologin
backup:/usr/sbin/nologin
list:/usr/sbin/nologin
```

The NF variable contains the numerical value of the last data field in the data file. You can then use it as a data field variable by placing a dollar sign in front of it

The FNR and NR variables are similar to each other but slightly different. The FNR variable contains the number of records processed in the current data file. The NR variable contains the total number of records processed.

Let's take a look at those two examples to illustrate the difference

```
$ awk 'BEGIN{FS=","}{print $1,"FNR="FNR}' myfile myfile
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

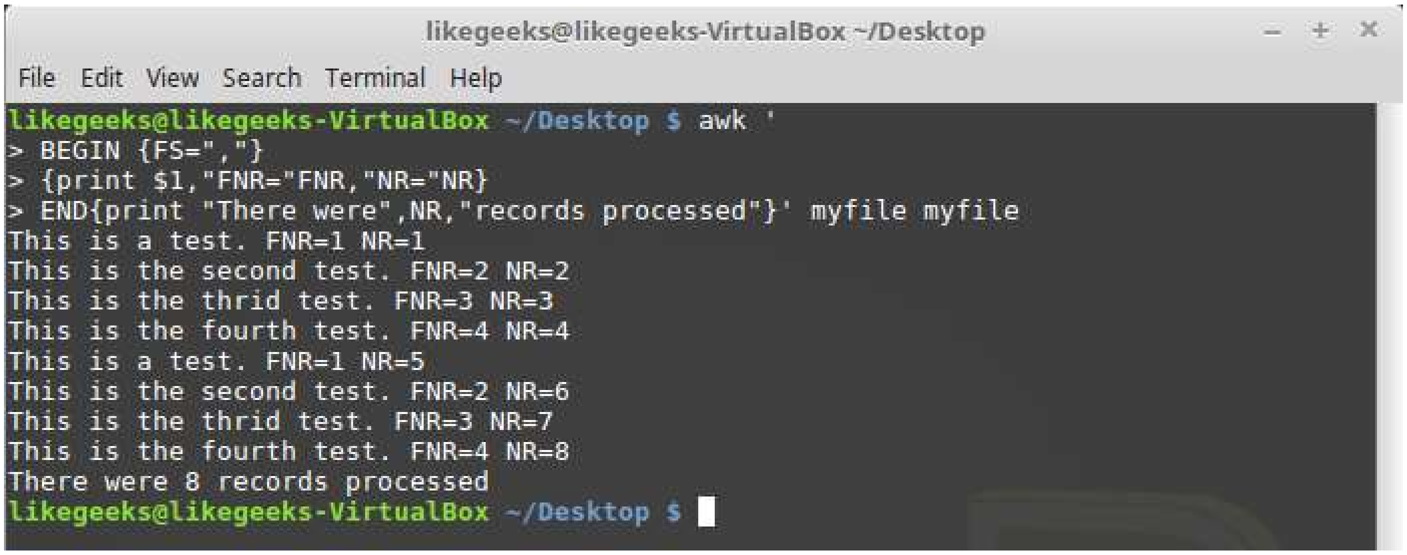
```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{FS=","}{print $1,"FNR="FNR}' my
file myfile
This is a test. FNR=1
This is the second test. FNR=2
This is the thrid test. FNR=3
This is the fourth test. FNR=4
This is a test. FNR=1
This is the second test. FNR=2
This is the thrid test. FNR=3
This is the fourth test. FNR=4
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

In this example, the awk command defines two input files. (It specifies the same input file twice.) The script prints the first data field value and the current value of the FNR variable.

Notice that the FNR value was reset to 1 when the awk command processed the second data file

Now, let's add the NR variable and see the difference

```
$ awk '
BEGIN {FS=","}
{print $1,"FNR="FNR,"NR="NR}
END{print "There were",NR,"records processed"}' myfile myfile
```



likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '
> BEGIN {FS=","}
> {print $1,"FNR="FNR,"NR="NR}
> END{print "There were",NR,"records processed"}' myfile myfile
This is a test. FNR=1 NR=1
This is the second test. FNR=2 NR=2
This is the thrid test. FNR=3 NR=3
This is the fourth test. FNR=4 NR=4
This is a test. FNR=1 NR=5
This is the second test. FNR=2 NR=6
This is the thrid test. FNR=3 NR=7
This is the fourth test. FNR=4 NR=8
There were 8 records processed
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The FNR variable value was reset when awk processed the second data file, but the NR variable maintained its count into the second data file.

## User defined variables

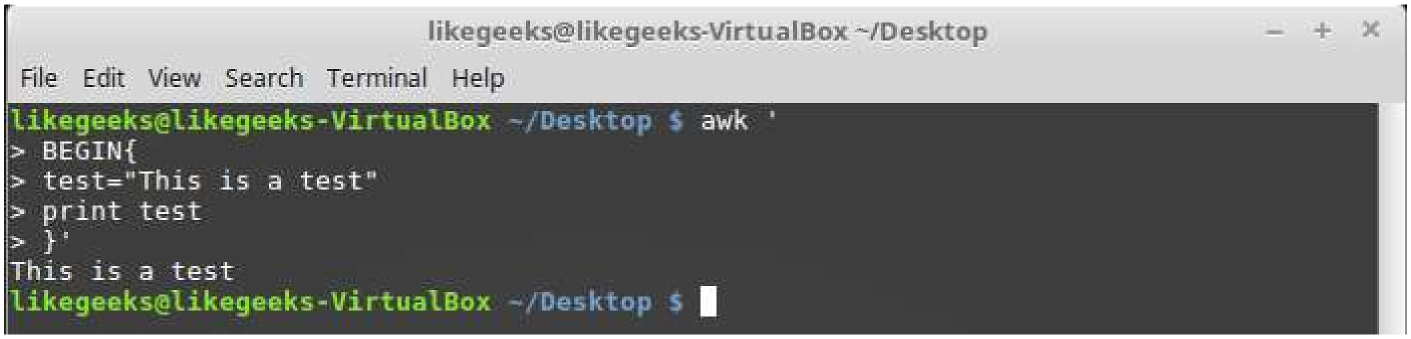
Like any other programming language, awk allows you to define your own variables for use within the script.

awk user-defined variable name can be any number of letters, digits, and underscores, but it **can't begin with a digit.**

You can assign a variable as in shell scripting like this

```
$ awk '
```

```
BEGIN{
test="This is a test"
print test
}'
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '
> BEGIN{
> test="This is a test"
> print test
> }'
This is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

## Structured Commands

The awk programming language supports the standard if-then-else format of the if statement. You must define a condition for the if statement to evaluate, enclosed in parentheses.

Here is an example

testfile contains the following

10

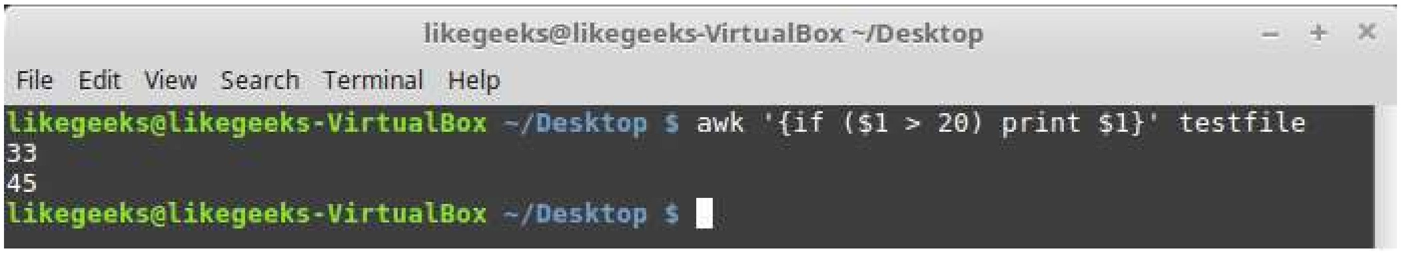
15

6

33

45

```
$ awk '{if ($1 > 20) print $1}' testfile
```

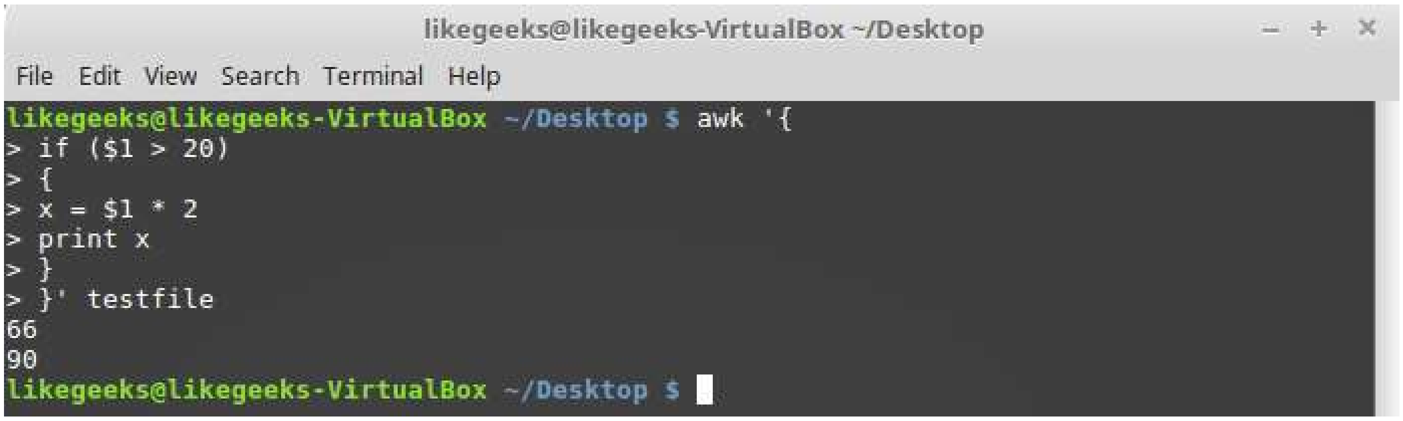

likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{if ($1 > 20) print $1}' testfile
33
45
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Just that simple

You can execute multiple statements in the if statement, you must enclose them with braces

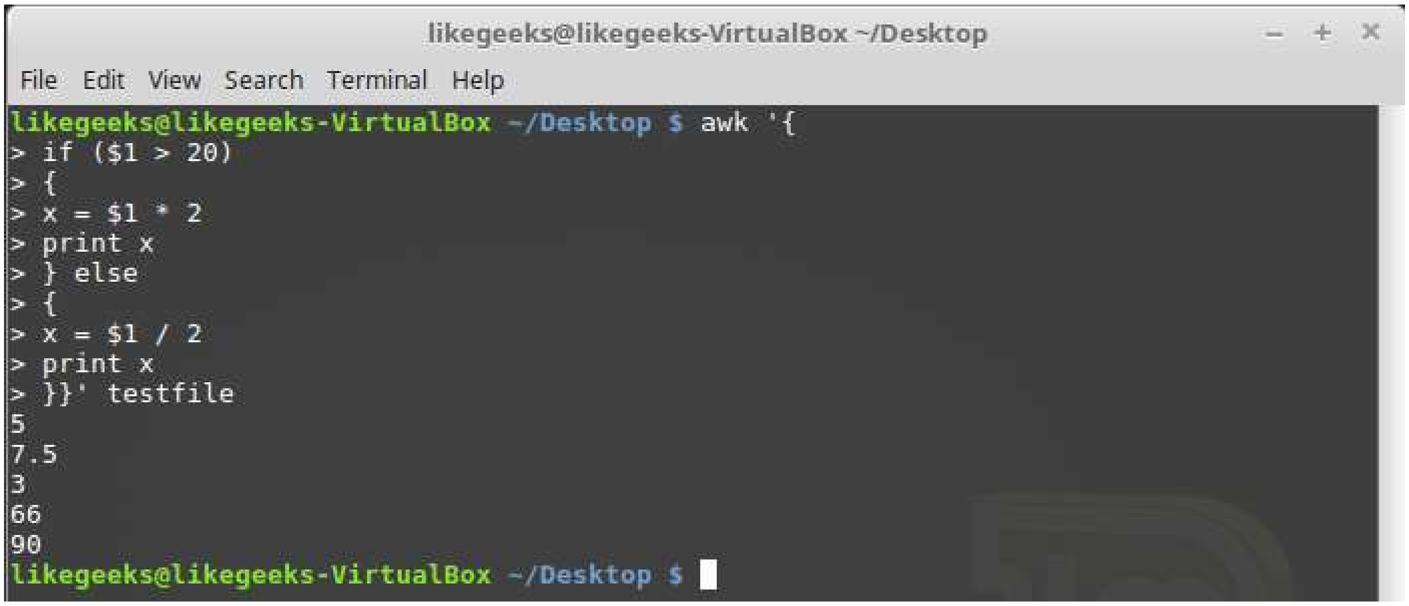
```
$ awk '{
if ($1 > 20)
{
x = $1 * 2
print x
}
}' testfile
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{
> if ($1 > 20)
> {
> x = $1 * 2
> print x
> }
> }' testfile
66
90
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

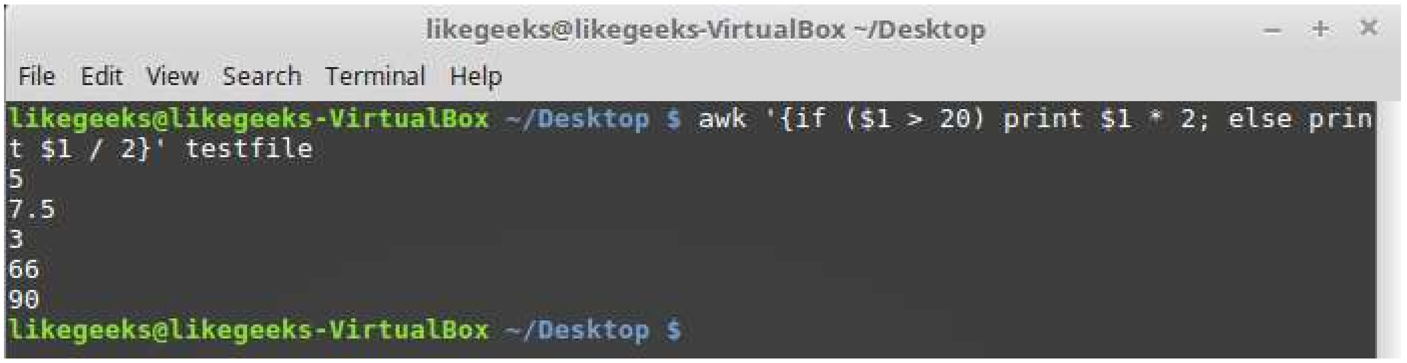
The awk if statement also supports the else clause like this

```
$ awk '{
if ($1 > 20)
{
x = $1 * 2
print x
} else
{
x = $1 / 2
print x
}}' testfile
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{
> if ($1 > 20)
> {
> x = $1 * 2
> print x
> } else
> {
> x = $1 / 2
> print x
> }}' testfile
5
7.5
3
66
90
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

You can use the else clause on a single line, but you must use a semicolon after the if statement


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{if ($1 > 20) print $1 * 2; else prin
t $1 / 2}' testfile
5
7.5
3
66
90
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

```
$ awk '{if ($1 > 20) print $1 * 2; else print $1 / 2}' testfile
```

## While loop

The while loop allows you to iterate over a set of data, checking a condition that stops the iteration

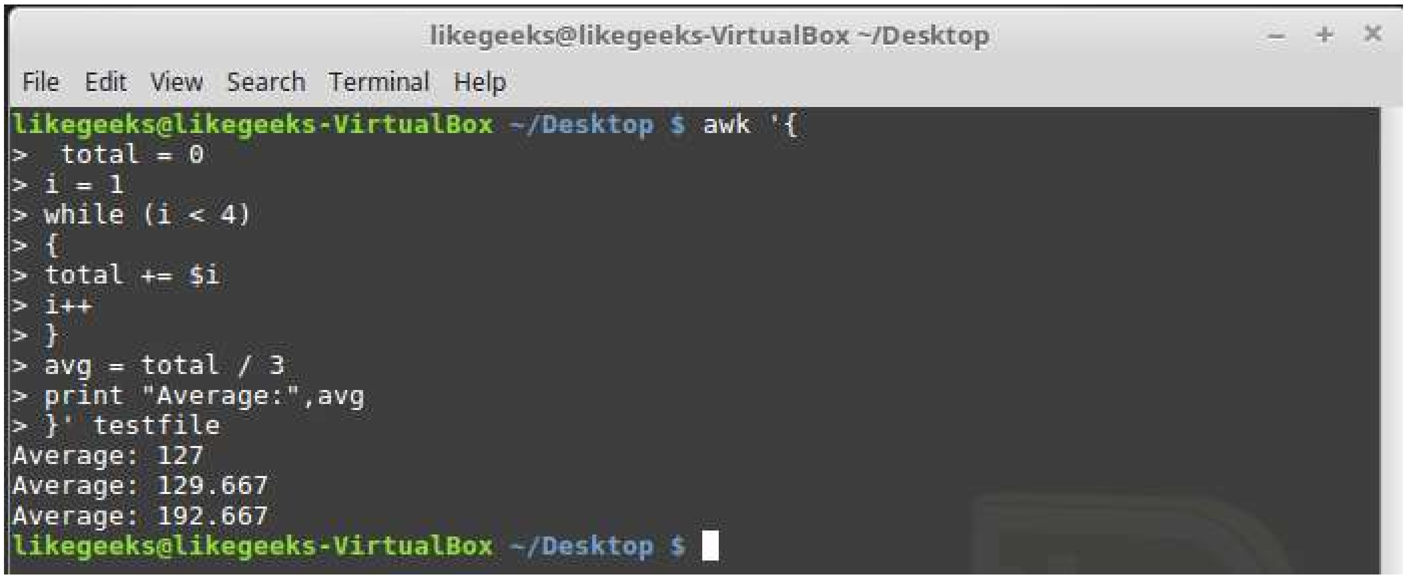
```
cat myfile
```

124 127 130

112 142 135

175 158 245

```
$ awk '{
 total = 0
i = 1
while (i < 4)
{
total += $i
i++
}
avg = total / 3
print "Average:",avg
}' testfile
```



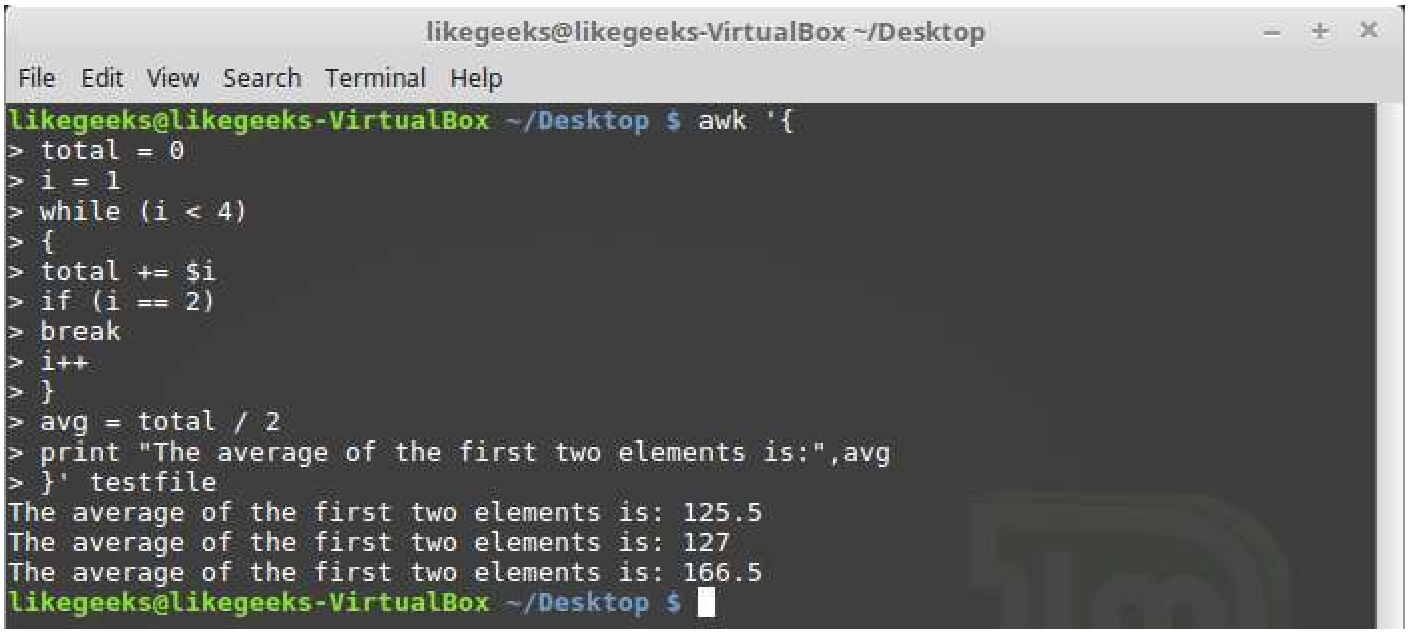
```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{
>  total = 0
> i = 1
> while (i < 4)
> {
> total += $i
> i++
> }
> avg = total / 3
> print "Average:",avg
> }' testfile
Average: 127
Average: 129.667
Average: 192.667
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The while statement iterates through the data fields in the record, adding each value to the total variable and incrementing the counter variable i.

When the counter value is equal to 4, the while condition becomes FALSE, and the loop terminates, dropping through to the next statement in the script. That statement calculates the average and prints the average.

The awk programming language supports using the break and continue statements in while loops, allowing you to jump out of the middle of the loop

```
 $ awk '{
total = 0
i = 1
while (i < 4)
{
total += $i
if (i == 2)
break
i++
}
avg = total / 2
print "The average of the first two elements is:",avg
}' testfile
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

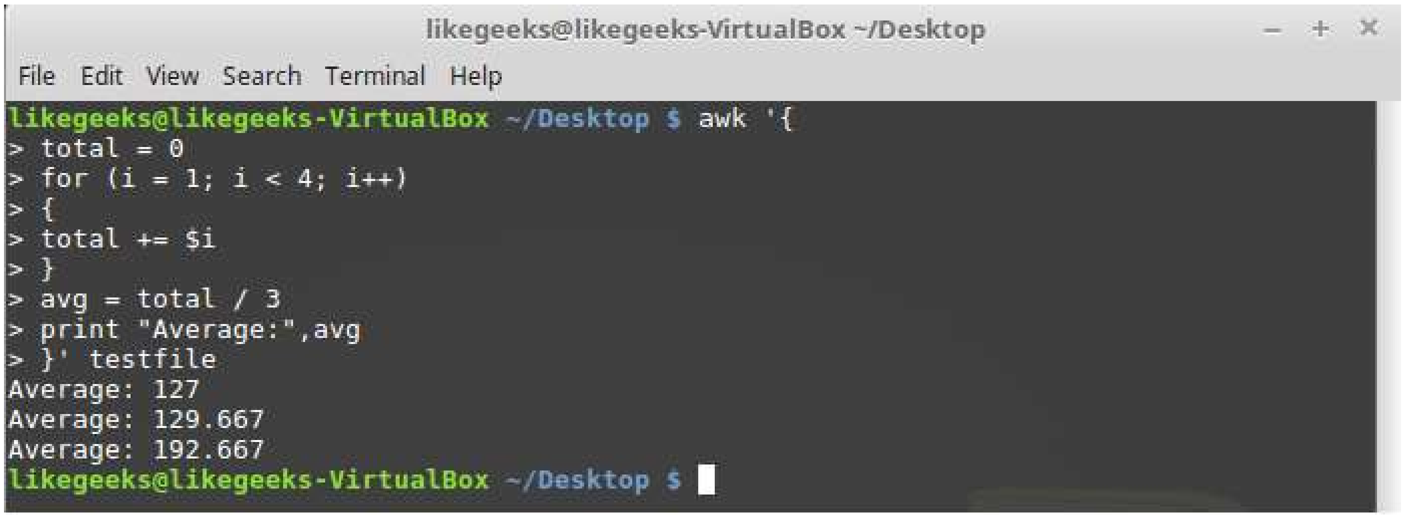
```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{
> total = 0
> i = 1
> while (i < 4)
> {
> total += $i
> if (i == 2)
> break
> i++
> }
> avg = total / 2
> print "The average of the first two elements is:",avg
> }' testfile
The average of the first two elements is: 125.5
The average of the first two elements is: 127
The average of the first two elements is: 166.5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

## The for loop

The for loop is a common method used in many programming languages for looping.

The awk programming language supports for loops

```
$ awk '{
total = 0
for (i = 1; i < 4; i++)
{
total += $i
}
avg = total / 3
print "Average:",avg
}' testfile
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
> total += $i
> }
> avg = total / 3
> print "Average:",avg
> }' testfile
Average: 127
Average: 129.667
Average: 192.667
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

By defining the iteration counter in the for loop, you don't have to worry about incrementing it yourself as you did when using the while statement.

## Formatted Printing

The printf command in awk allows you to specify detailed instructions on how to display data.

It specifies exactly how the formatted output should appear, using both text elements and format specifiers. A format specifier is a special code that indicates what type of variable is displayed and how to display it. The awk command uses each format specifier as a placeholder for each variable listed in the command. The first format specifier matches the first variable listed; the second matches the second variable, and so on.

The format specifiers use the following format

%[modifier]control-letter

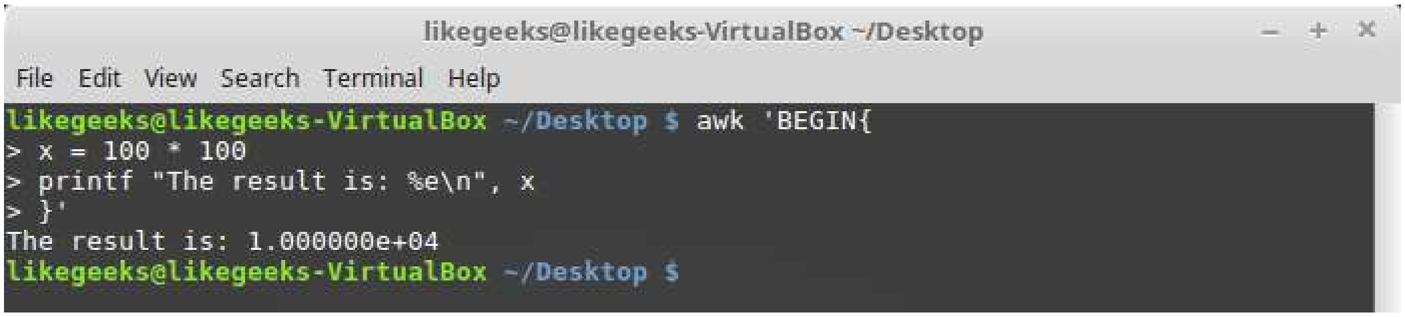
This list is the format specifiers you can use with printf:

| | |
|---|---|
| c | Displays a number as an ASCII character |
| d | Displays an integer value |
| i | Displays an integer value (same as d) |

| | |
|---|---|
| e | Displays a number in scientific notation |
| f | Displays a floating-point value |
| g | Displays either scientific notation or floating point, whichever is shorter |
| o | Displays an octal value |
| s | Displays a text string |

Here we use printf to format our output

```
$ awk 'BEGIN{
x = 100 * 100
printf "The result is: %e\n", x
}'
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{
> x = 100 * 100
> printf "The result is: %e\n", x
> }'
The result is: 1.000000e+04
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Here as an example, we display a large value using scientific notation %e.

We are not going to try every format specifier. You know the concept

## Built-In Functions

The awk programming language provides quite a few built-in functions that perform mathematical, string, and time functions. You can utilize these functions in your awk scripts

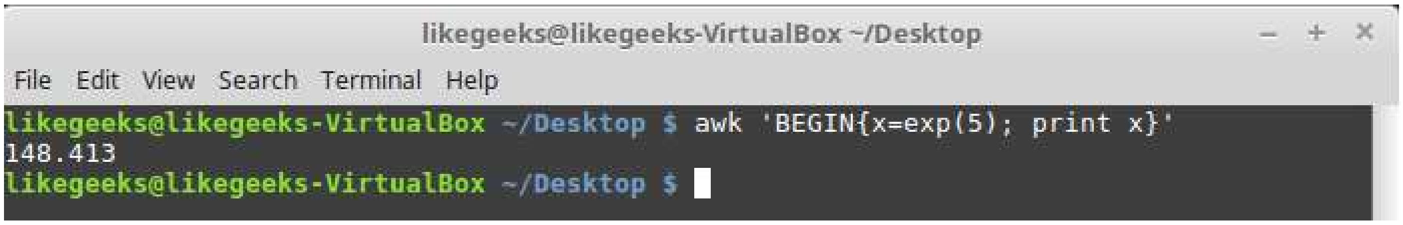
### Mathematical functions

If you love math, those are some of the mathematical functions you can use with awk

|  |  |
|---|---|
| **cos(x)** | The cosine of x, with x specified in radians |
| **exp(x)** | The exponential of x |
| **int(x)** | The integer part of x, truncated toward 0 |
| **log(x)** | The natural logarithm of x |
| **rand()** | A random floating point value larger than 0 and less than 1 |
| **sin(x)** | The sine of x, with x specified in radians |
| **sqrt(x)** | The square root of x |

and they can be used normally

```
$ awk 'BEGIN{x=exp(5); print x}'
```

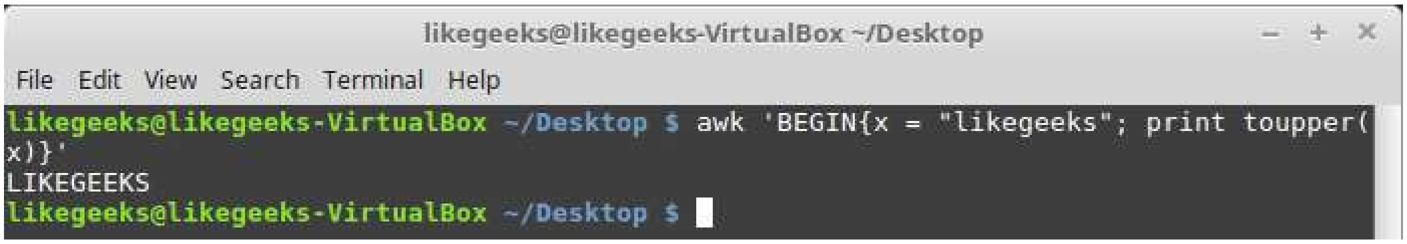

likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{x=exp(5); print x}'
148.413
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

## String functions

There are many string functions you can check the list but we will examine one of them as an example and the rest is the same

```
$ awk 'BEGIN{x = "likegeeks"; print toupper(x)}'
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

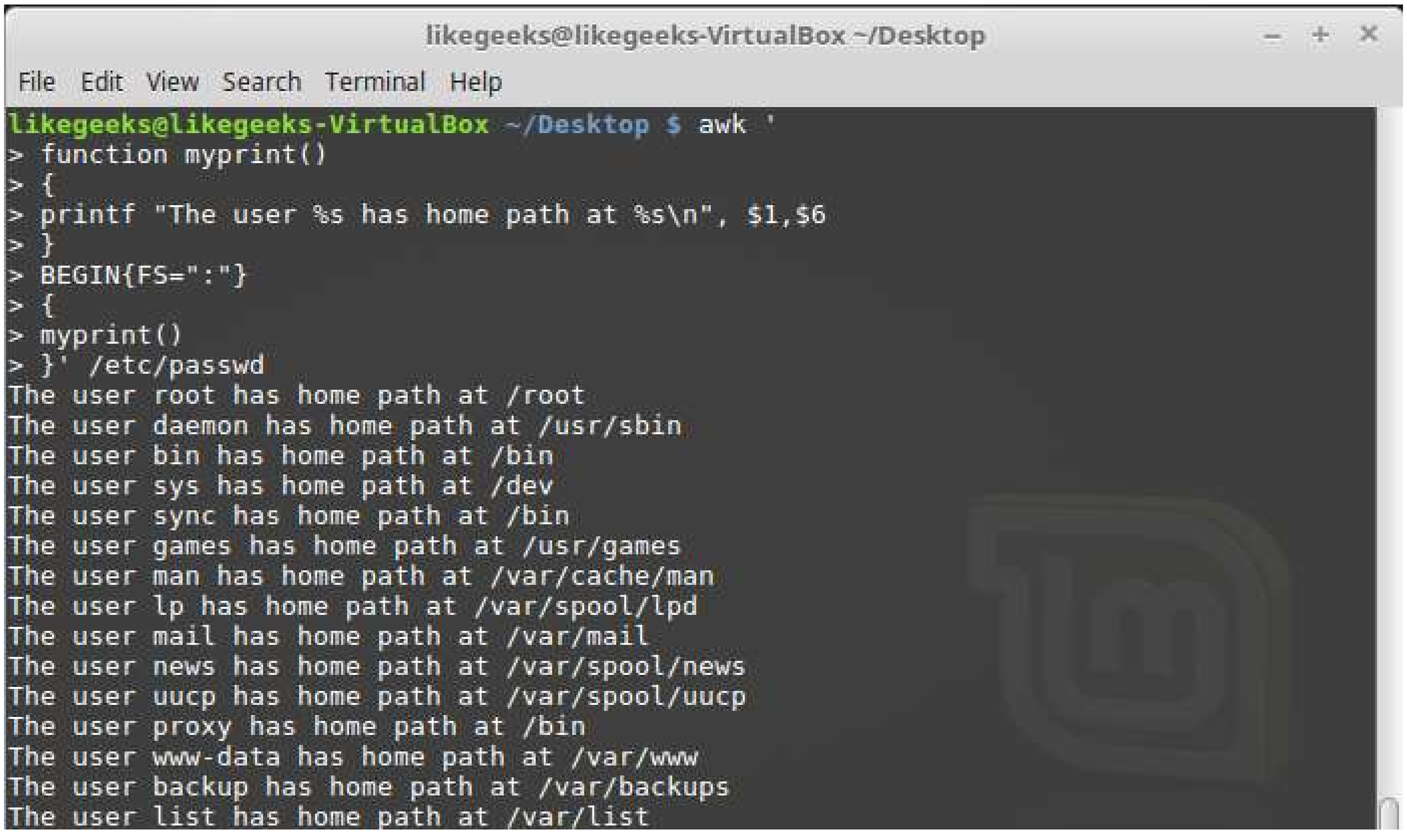
```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{x = "likegeeks"; print toupper(
x)}'
LIKEGEEKS
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The function toupper convert case to upper case for the string passed.

## User Defined Functions

You can create your own functions for use in awk scripts, just define them and use them

```
$ awk '
function myprint()
{
printf "The user %s has home path at %s\n", $1,$6
}
BEGIN{FS=":"}
{
myprint()
}' /etc/passwd
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '
> function myprint()
> {
> printf "The user %s has home path at %s\n", $1,$6
> }
> BEGIN{FS=":"}
> {
> myprint()
> }' /etc/passwd
The user root has home path at /root
The user daemon has home path at /usr/sbin
The user bin has home path at /bin
The user sys has home path at /dev
The user sync has home path at /bin
The user games has home path at /usr/games
The user man has home path at /var/cache/man
The user lp has home path at /var/spool/lpd
The user mail has home path at /var/mail
The user news has home path at /var/spool/news
The user uucp has home path at /var/spool/uucp
The user proxy has home path at /bin
The user www-data has home path at /var/www
The user backup has home path at /var/backups
The user list has home path at /var/list
```

Here we define a function called myprint then we use it in out script to print output using printf built-in function.

With this example, I finish my post today hope you like it.

Thank you

**21**

**Admin**

https://likegeeks.com

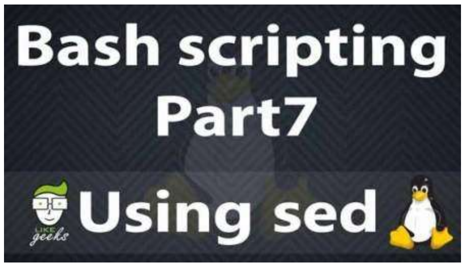
## RELATED ARTICLES

LINUX

**Shell scripting the awesome guide part4**

February 13, 2017 admin

On the previous post we've talked about parameters and options in detail and today we will talk about something is very important in shell scripting which is input & output & redirection. So far, you've seen two methods for displaying the output from your shell scripts Displaying



LINUX

**31+ Examples for sed Linux command in text manipulation**

February 19, 2017 admin

On the previous post we've talked about bash functions and how to use it from the command line and we've seen some other cool stuff I recommend you to review it, Today we will talk about a very useful tool for string manipulation called sed, sed

LINUX

**Bash scripting the awesome guide part6 Bash functions**

February 17, 2017 admin

Before we talk about bash functions let's discuss this situation. When writing bash scripts, you'll find yourself that you are using the same code in multiple places. If you get tired writing the same blocks of code over and over in your bash script. It would be nice to just write the block of code […]

output on the screen
Redirecting output to a […]

Linux command is one of the
most common tools […]

46

9

0

◂ 31+ Examples for sed Linux command in text manipulation

Regex tutorial for Linux ▸

3 Comments likegeeks

1 Login

Recommend 1 Share

Sort by Best

Join the discussion…

clfapujc • 12 days ago
I can't get the output shown on the first terminal screen shot. Is the command correct?

edit. I get it now. Sorry for disturbing in the comments :D
1 • Reply • Share ›

sigzero ➦ clfapujc • 12 days ago
You have to hit ENTER. Awk won't print that per the instructions.
1 • Reply • Share ›

likegeeks Mod ➦ clfapujc • 11 days ago
No problem you are welcome all time.
Regards
• Reply • Share ›

Subscribe Add Disqus to your site Add Disqus Add Privacy

## SEARCH

Search …

Search