Sed & Awk 101 Hacks

Enhance Your UNIX / Linux Life with Sed and Awk

Ramesh Natarajan
www.thegeekstuff.com

Table of Contents

Introduction

> *"Enhance Your UNIX and Linux Life with Sed and Awk"*

If you are a developer, or system administrator, or database administrator, or IT manager, or just someone who spends a significant amount of time on UNIX / Linux, you should become proficient in Sed and Awk.

Sed and Awk are two great utilities that can solve a lot of complex tasks quickly with only a few lines of code--in most cases, with just a single line of code.

This book explains the following:

- Chapters 1 – 7 cover sed. Chapters 8 – 13 cover awk.
- Chapters 1 - 5 explain various sed commands, including the powerful sed substitute command, regular expressions, and different methods to execute sed commands.
- Chapters 6 and 7 describe sed hold space and pattern space, sed multi-line commands, and sed loops. Clear examples are included.
- Chapters 8 – 11 cover various awk programming language components, with examples and built-in variables.
- Chapters 12 and 13 explain the powerful awk associative array, plus additional built-in awk functions and commands with clear examples.

A note on the examples: Most examples are identified in the following way.

Example Description

```
Lines of code for you to type, with the result you will
see on screen.
```

Any additional clarification or discussion will appear below the code section in plain text.

Also please note that commands should be typed on one line. If you copy and paste, be sure that command is pasted as a single line.

About the Author

I'm Ramesh Natarajan, author of The Geek Stuff blog thegeekstuff.com and numerous ebooks including this one.

I have done extensive programming in several languages and C is my favorite. I have done a lot of work on the infrastructure side including Linux system administration, DBA, Networking, Hardware and Storage (EMC).

I also developed passworddragon.com — a free, easy and secure password manager that runs on Windows, Linux and Mac.

I wrote the following books:

- Linux 101 Hacks eBook – linux.101hacks.com (free)
- Vim 101 Hacks ebook – vim.101hacks.com
- Nagios Core 3 – thegeekstuff.com/nagios-core-ebook

If you have any feedback about this eBook, please use this contact form - thegeekstuff.com/contact – to get in touch with me.

Copyright & Disclaimer

The information provided in this book is provided "as is" with no implied warranties or guarantees.

Version

Version	Date	Revisions
1.0	06. Apr. 2011	First Edition

Chapter 1: Sed Syntax and Basic Commands

For all sed examples, we'll be using the following employee.txt file. Please create this text file to try out the commands given in this book.

```
$ vi employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

The above employee database contains the following fields for every record:

- Employee Id
- Employee Name
- Title

Sed stands for Stream Editor. It is very powerful tool to manipulate, filter, and transform text. Sed can take input from a file, or from a pipe. You might even have several sed one line commands in your bash startup file that you use for various scenarios without exactly understanding the sed scripts.

For beginners, sed script might look cryptic. Once you understand the sed commands in detail, you'll be able to solve a lot of complex text manipulation problems by writing a quick sed script.

In this book, I've explained all sed commands and provided easy-to-understand examples.

1. Sed Command Syntax

The purpose of this section is to get you familiarized with sed syntax and command structure. This is not meant to explain the individual sed commands, which are covered in detail later.

Basic sed syntax:

```
sed [options] {sed-commands} {input-file}
```

Sed reads one line at a time from the {input-file} and executes the {sed-commands} on that particular line.

It reads the 1st line from the {input-file} and executes the {sed-commands} on the 1st line. Then it reads the 2nd line from the {input-file} and executes the {sed-commands} on the 2nd line. Sed repeats this process until it reaches the end of the {input-file}.

There are also a few optional command line options that can be passed to sed as indicated by [options].

The following example demonstrates the basic sed syntax. This simple sed example prints all the lines from the /etc/passwd file.

```
sed -n 'p' /etc/passwd
```

The main focus here is on the {sed-commands}, which can be either a single sed command or multiple sed commands. You can also combine multiple sed-commands in a file and call the sed script file using the -f option as shown below.

Basic sed syntax for use with sed-command file:

```
sed [options] -f {sed-commands-in-a-file} {input-file}
```

The following example demonstrates the basic syntax. This example prints lines beginning with root and nobody from the /etc/passwd file.

```
$ vi test-script.sed
/^root/ p
/^nobody/ p

$ sed -n -f test-script.sed /etc/passwd
```

While executing multiple sed commands, you can also directly specify them in the command line using -e as shown below.

Basic sed syntax using -e:

```
sed [options] -e {sed-command-1} -e {sed-command-2}
{input-file}
```

The following example demonstrates the basic syntax. This prints lines beginning with root and nobody from /etc/passwd file:

```
sed -n -e '/^root/ p' -e '/^nobody/ p' /etc/passwd
```

If you are executing a lot of commands in a single line using several -e arguments, you can split them into multiple lines using a back slash as shown below.

```
sed -n \
-e '/^root/ p' \
-e '/^nobody/ p' \
/etc/passwd
```

You can also execute multiple sed commands in the command line by grouping them together using { }:

Basic sed syntax using { }:

```
sed [options] '{
sed-command-1
sed-command-2
```

```
}' input-file
```

The following example demonstrates this version of the basic syntax. This also prints lines beginning with root and nobody from /etc/passwd file.

```
sed -n '{
/^root/ p
/^nobody/ p
}' /etc/passwd
```

Note: Sed never modifies the original file. It always prints the output to stdout. If you want to save the changes, you should redirect the output to a file by explicitly specifying > filename.txt.

2. Sed Scripting Flow

Sed scripting follows the easily remembered sequence Read, Execute, Print, Repeat. Use the simple REPR acronym to remember sed execution flow.

We look at the steps in this sequence. Sed will:

- **Read** a line into the pattern space (an internal temporary sed buffer, where it places the line it reads from the input file).
- **Execute** the sed command on the line in the sed pattern space. If there are more than one sed commands available, either via a sed script, -e options, or { }, it executes all the sed commands one by one in sequence on the line that is currently in the pattern space.
- **Print** the line from the pattern space. After printing this line, the sed pattern space will be empty.
- **Repeat** this again until the end of the input file is reached.

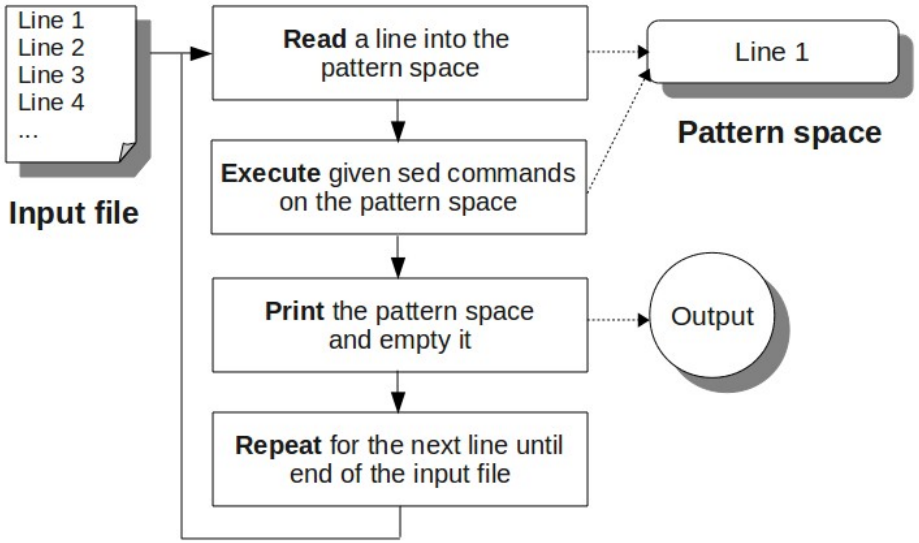

Fig: Illustration of SED execution flow

3. Print Pattern Space (p command)

Using the sed p command, you can print the current pattern space.

You may wonder why you would need the p command, since by default sed prints the pattern buffer after executing its commands.

There are reasons, as you will see; the command allows you to specifically control what is printed to stdout. Usually when p is used you will use the -n option to suppress the the default printing that happens as part of the standard sed flow. Otherwise, when execute p (print) as one of the commands, the line will be printed twice.

The following example prints every line of employee.txt twice:

```
$ sed 'p' employee.txt
101,John Doe,CEO
101,John Doe,CEO
102,Jason Smith,IT Manager
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
```

```
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
105,Jane Miller,Sales Manager
```

Print each line once (functionally the same as 'cat employee.txt'):

```
$ sed -n 'p' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Specifying an Address Range

If you don't specify an address range before the sed command, by default it matches all the lines. The following are some examples of specifying an address range before the sed command.

Print only the 2nd line:

```
$ sed -n '2 p' employee.txt
102,Jason Smith,IT Manager
```

Print from line 1 through line 4:

```
$ sed -n '1,4 p' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

Print from line 2 through the last line ($ represents the last line):

```
$ sed -n '2,$ p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Modify Address Range

You can modify address range using comma, plus, and tilde.

In the examples above, we saw the use of the comma (,) as part of an address range specification. Its meaning is clear: n,m indicates n through m.

The plus (+) may be used in conjunction with the comma, to specify a number of lines instead of an absolute line number. For example, n, +m means the m lines starting with n.

The tilde (~) may also be used in an address range. Its special meaning is to skip lines between commands. For example, address range n~m indicates that sed should start at the nth line and pick up every mth line from there.

- 1~2 matches 1,3,5,7, etc.
- 2~2 matches 2,4,6,8, etc.
- 1~3 matches 1,4,7,10, etc.
- 2~3 matches 2,5,8,11, etc.

Print only odd numbered lines:

```
$ sed -n '1~2 p' employee.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
105,Jane Miller,Sales Manager
```

Pattern Matching

Just as you can specify a numbered address (or address range), you can also specify a pattern (or pattern range) to match, as shown in the next few examples.

Print lines matching the pattern “Jane”:

```
$ sed -n '/Jane/ p' employee.txt
105,Jane Miller,Sales Manager
```

Print lines starting from the 1st match of "Jason" until the 4th line:

```
$ sed -n '/Jason/,4 p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

Note: If there were no matches for "Jason" in the 1st 4 lines, this command would print the lines that match "Jason" after the 4th line.

Print lines starting from the 1st match of "Raj" until the last line:

```
$ sed -n '/Raj/,$ p' employee.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Print lines starting from the line matching "Raj" until the line matching "Jane":

```
$ sed -n '/Raj/,/Jane/ p' employee.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Print the line matching "Jason" and 2 lines immediately after that:

```
$ sed -n '/Jason/,+2 p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

4. Delete Lines (d command)

Using the sed d command, you can delete lines. Please note that the lines are only deleted from the output stream. Just like any other sed command, the d command doesn't modify the content of the original input file.

By default if you don't specify any address range before the sed command, it matches all the lines. So, the following example will not print anything, as it matches all the lines in the employee.txt and deletes them.

```
sed 'd' employee.txt
```

It's more useful to specify an address range to be deleted. The following are some examples:

Delete only the 2nd line:

```
$ sed '2 d' employee.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Delete from line 1 through 4:

```
$ sed '1,4 d' employee.txt
105,Jane Miller,Sales Manager
```

Delete from line 2 through the last line:

```
$ sed '2,$ d' employee.txt
101,John Doe,CEO
```

Delete only odd number of lines:

```
$ sed '1~2 d' employee.txt
102,Jason Smith,IT Manager
104,Anand Ram,Developer
```

Delete lines matching the pattern "Manager":

```
$ sed '/Manager/ d' employee.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

Delete lines starting from the 1st match of "Jason" until the 4th line:

```
$ sed '/Jason/,4 d' employee.txt
101,John Doe,CEO
105,Jane Miller,Sales Manager
```

If there are no matches for "Jason" in the 1st 4 lines, this command deletes only the lines that match "Jason" after the 4th line.

Delete lines starting from the 1st match of "Raj" until the last line:

```
$ sed '/Raj/,$ d' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
```

Delete lines starting from the line matching "Raj" until the line matching "Jane":

```
$ sed '/Raj/,/Jane/ d' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
```

Delete lines starting from the line matching "Jason" and 2 lines immediately after that:

```
$ sed '/Jason/,+2 d' employee.txt
101,John Doe,CEO
105,Jane Miller,Sales Manager
```

Useful Delete Examples

The following examples are very helpful in actual day-to-day operations.

Delete all the empty lines from a file:

```
sed '/^$/ d' employee.txt
```

Delete all comment lines (assuming the comment starts with #):

```
sed '/^#/ d' employee.txt
```

Note: When you have multiple sed commands, the moment sed encounters the 'd' command, the whole line matching the pattern will be deleted, and no further commands will be executed on the deleted line.

5. Write Pattern Space to File (w command)

Using the sed w command, you can write the current pattern space to a file. By default as per the sed standard flow, the pattern space will be printed to stdout, so if you want output to file but not screen you should also use the sed option -n.

The following are some examples.

Write the content of employee.txt file to file output.txt (and display on screen):

```
$ sed 'w output.txt' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
```

```
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager

$ cat output.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Write the content of employee.txt file to output.txt file but not to screen:

```
$ sed -n 'w output.txt' employee.txt

$ cat output.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Write only the 2nd line:

```
$ sed -n '2 w output.txt' employee.txt

$ cat output.txt
102,Jason Smith,IT Manager
```

Write lines 1 through 4:

```
$ sed -n '1,4 w output.txt' employee.txt

$ cat output.txt
101,John Doe,CEO
```

```
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

Write from line 2 through the last line:

```
$ sed -n '2,$ w output.txt' employee.txt

$ cat output.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Write only odd numbered lines:

```
$ sed -n '1~2 w output.txt' employee.txt

$ cat output.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
105,Jane Miller,Sales Manager
```

Write lines matching the pattern "Jane":

```
$ sed -n '/Jane/ w output.txt' employee.txt

$ cat output.txt
105,Jane Miller,Sales Manager
```

Write lines starting from the 1st match of "Jason" until the 4th line:

```
$ sed -n '/Jason/,4 w output.txt' employee.txt

$ cat output.txt
102,Jason Smith,IT Manager
```

```
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

If there are no matches for "Jason" in the 1st 4 lines, this command writes only the lines that match "Jason" after the 4th line.

Write lines starting from the 1st match of "Raj" until the last line:

```
$ sed -n '/Raj/,$ w output.txt' employee.txt

$ cat output.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Write lines starting from the line matching "Raj" until the line matching "Jane":

```
$ sed -n '/Raj/,/Jane/ w output.txt' employee.txt

$ cat output.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Write the line matching "Jason" and the next 2 lines immediately after that:

```
$ sed -n '/Jason/,+2 w output.txt' employee.txt

$ cat output.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

Note: You might not use the w command frequently. Most people use UNIX output redirection, instead, to store the output of sed to a file. For example: sed 'p' employee.txt > output.txt

Chapter 2. Sed Substitute Command

The most powerful command in the stream editor is **s**ubstitute. It has such power and so many options that we give it a whole chapter.

6. Sed Substitute Command Syntax

```
sed '[address-range|pattern-range] s/original-
string/replacement-string/[substitute-flags]' inputfile
```

In the above sed substitute command syntax:

- address-range or pattern-range is optional. If you don't specify one, sed will execute the substitute command on all lines.
- s – tells Sed to execute the substitute command
- original-string – this is the string to be searched for in the input file. The original-string can also be a regular expression.
- replacement-string – Sed will replace original-string with this string.
- substitute-flags are optional. More on this in the next section.

Remember that *the original file is not changed;* the substitution takes place in the pattern space buffer which is then printed to stdout.

The following are couple of simple sed substitute examples (changes shown in **bold**).

Replace all occurrences of Manager with Director:

```
$ sed 's/Manager/Director/' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Director
103,Raj Reddy,Sysadmin
```

```
104,Anand Ram,Developer
105,Jane Miller,Sales Director
```

Replace Manager with Director only on lines that contain the keyword 'Sales':

```
$ sed '/Sales/s/Manager/Director/' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Director
```

Note that the use of the address range caused just one change rather than the two shown in the previous example.

7. Global Flag (g flag)

Sed substitute flag g stands for global. By default sed substitute command will replace only the 1st occurrence of the {original-string} on each line. If you want to change all the occurrences of the {original-string} in the line to the {replacement-string}, you should use the global flag g.

Replace the 1st occurrence of lower case a with upper case A:

```
$ sed 's/a/A/' employee.txt
101,John Doe,CEO
102,JAson Smith,IT Manager
103,RAj Reddy,Sysadmin
104,AnAnd Ram,Developer
105,JAne Miller,Sales Manager
```

Replace all occurrences of lower case a with upper case A:

```
$ sed 's/a/A/g' employee.txt
101,John Doe,CEO
102,JAson Smith,IT MAnAger
103,RAj Reddy,SysAdmin
```

```
104,AnAnd RAm,Developer
105,JAne Miller,SAles MAnAger
```

Note: these examples were applied to the entire file because no address range was specified.

8. Number Flag (1,2,3.. flag)

Use the number flag to specify a specific occurrence of the original-string. Only the n-th instance of original-string will trigger the substitution. Counting starts over on each line, and n can be anything from 1 to 512.

For example, /11 will replace only the 11th occurrence of the original-string in a line.

The following are few examples.

Replace the 2nd occurrence of lower case a to upper case A:

```
$ sed 's/a/A/2' employee.txt
101,John Doe,CEO
102,Jason Smith,IT MAnager
103,Raj Reddy,SysAdmin
104,Anand RAm,Developer
105,Jane Miller,SAles Manager
```

For this example, create the following file with three lines:

```
$ vi substitute-locate.txt
locate command is used to locate files
locate command uses database to locate files
locate command can also use regex for searching
```

In the file you just created, change only the 2nd occurrence of locate to find:

```
$ sed 's/locate/find/2' substitute-locate.txt
locate command is used to find files
locate command uses database to find files
```

```
locate command can also use regex for searching
```

Note: On line 3 in the above example, there is only one "locate" in the original substitute-locate.txt file. So, nothing is changed on line 3.

9. Print Flag (p flag)

The sed substitute flag p stands for print. When the substitution is successful, it prints the changed line. Like most print commands in sed, it is most useful when combined with the -n option to suppress default printing of all lines.

Print only the line that was changed by the substitute command:

```
$ sed -n 's/John/Johnny/p' employee.txt
101,Johnny Doe,CEO
```

In our number flag example, we used /2 to change the 2nd occurrence of "locate" to "find". On line 3 of locate.txt there is no 2nd occurrence and substitution never happened on that line. Adding the p flag to the command we used before will print the two lines that did change.

Change the 2nd instance of “locate” to “find” and print the result:

```
$ sed -n 's/locate/find/2p' substitute-locate.txt
locate command is used to find files
locate command uses database to find files
```

10. Write Flag (w flag)

The sed substitute flag w stands for write. When the substitution is successful, it writes the changed line to a file. Most people use the p flag instead, and redirect the output to a file. We include this command for completeness.

Write only the line that was changed by the substitute command to output.txt:

```
$ sed -n 's/John/Johnny/w output.txt' employee.txt


$ cat output.txt
101,Johnny Doe,CEO
```

Just as we showed for the p command, adding w to our example with the substitute-locate.txt file will send the two lines that were changed to the output file.

Change the 2nd instance of “locate” to “find”,write the result to a file, print all lines:

```
$ sed 's/locate/find/2w output.txt' substitute-
locate.txt
locate command is used to find files
locate command uses database to find files
locate command can also use regex for searching


$ cat output.txt
locate command is used to find files
locate command uses database to find files
```

11. Ignore Case Flag (i flag)

The sed substitute flag i stands for ignore case. You can use the i flag to match the original-string in a case-insensitive manner. This is available only in GNU Sed.

In the following example Sed will not replace "John" with "Johnny", as the original-string was given in lower case "john".

Replace “john” with Johnny:

```
$ sed 's/john/Johnny/' employee.txt
101,John Doe,CEO
```

```
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Replace “john” or “John” with Johnny:

```
$ sed 's/john/Johnny/i' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

12. Execute Flag (e flag)

The sed substitute flag e stands for execute. Using the sed e flag, you can execute whatever is available in the pattern space as a shell command, and the output will be returned to the pattern space. This is available only in the GNU sed.

The following are few examples.

For these examples create the following files.txt that contains a list of filenames with their full path:

```
$ cat files.txt
/etc/passwd
/etc/group
```

Add the text "ls -l " in front of every line in the files.txt and print the output:

```
$ sed 's/^/ls -l /' files.txt
ls -l /etc/passwd
ls -l /etc/group
```

Add the text "ls -l " in front of every line in the files.txt and execute the output:

```
$ sed 's/^/ls -l /e' files.txt
-rw-r--r-- 1 root root 1547 Oct 27 08:11 /etc/passwd
-rw-r--r-- 1 root root 651 Oct 27 08:11 /etc/group
```

13. Combine Sed Substitution Flags

You can combine one or more substitute flags as required.

The following example will replace all occurrences of "Manager" or "manager" to "Director". This will also print only the line that was changed by the substitute command to the screen, and write the same information to the output.txt file.

Combine g,i,p and w flags:

```
$ sed -n 's/Manager/Director/gipw output.txt'
employee.txt
102,Jason Smith,IT Director
105,Jane Miller,Sales Director

$ cat output.txt
102,Jason Smith,IT Director
105,Jane Miller,Sales Director
```

14. Sed Substitution Delimiter

In all the above sed examples, we used the default sed delimiter /. i.e. s/original-string/replacement-string/ When there is a slash / in the original-string or the replacement-string, we need to escape it using \. For this example create a path.txt file which contains a directory path as shown below.

```
$ vi path.txt
reading /usr/local/bin directory
```

Now, let us change /usr/local/bin to /usr/bin using the sed substitute command. In this sed substitution example, the delimiter path delimiter '/' was escaped using back slash '\' in the original-string and the replacement-string.

```
$ sed 's/\/usr\/local\/bin/\/usr\/bin/' path.txt
reading /usr/bin directory
```

Ugly isn't it? When you are trying to replace a long path name, it might be very confusing to use all those escape characters '\'. Fortunately, you can use any character as substitution delimiter. For example, | or ^ or @ or !.

All of the following are valid and easy to read. I have not shown the output of the commands since they all produce exactly the same result. I prefer to use @ (or !) symbol when replacing a directory path but it is your personal choice.

```
sed 's|/usr/local/bin|/usr/bin|' path.txt
sed 's^/usr/local/bin^/usr/bin^' path.txt
sed 's@/usr/local/bin@/usr/bin@' path.txt
sed 's!/usr/local/bin!/usr/bin!' path.txt
```

15. Multiple Substitute Commands Affecting the Same Line

As we discussed earlier, the sed execution flow is Read, Execute, Print, Repeat. The Execute portion, as we mentioned, may consist of multiple sed commands, which sed will execute one-by-one.

For example, if you have two sed commands, sed will execute command-1 on the pattern space, then execute command-2 on the pattern space. If command-1 changed something in the pattern space, command-2 will be executed on the newly changed pattern space (and not the original line that was Read).

The following example demonstrates the execution of two sed substitute commands on the pattern space.

Change Developer to IT Manager, then change Manager to Director:

```
$ sed '{
s/Developer/IT Manager/
s/Manager/Director/
}' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Director
103,Raj Reddy,Sysadmin
104,Anand Ram,IT Director
105,Jane Miller,Sales Director
```

Let us analyze the sed execution flow for line 4 in the example.

1. Read: At this stage, Sed reads the line and puts it in the pattern space. So, the following is the content of the pattern space.

```
104,Anand Ram,Developer
```

2. Execute: Sed executes the 1st sed command on the pattern space, which is s/Developer/IT Manager/. So, after this command, the following is the content of the pattern space.

```
104,Anand Ram,IT Manager
```

Now, sed executes the 2nd sed command on the pattern space, which is s/Manager/Director/. After this command, the following is the content of the pattern space.

```
104,Anand Ram,IT Director
```

Remember: Sed executes the 2nd command *on the content of the current pattern space after execution of the first command.*

3. Print: It prints the content of the current pattern space, which is the following.

```
104,Anand Ram,IT Director
```

4. Repeat: It moves on to the next line and repeats from step#1.

16. Power of & - Get Matched Pattern

When & is used in the replacement-string, it replaces it with whatever text matched the original-string or the regular-expression. This is very powerful and useful.

The following are few examples.

Enclose the employee id (the 1st three numbers) between [and], i.e. 101 becomes [101], 102 becomes [102], etc.

```
$ sed 's/^[0-9][0-9][0-9]/[&]/g' employee.txt
[101],John Doe,CEO
[102],Jason Smith,IT Manager
[103],Raj Reddy,Sysadmin
[104],Anand Ram,Developer
[105],Jane Miller,Sales Manager
```

Enclose the whole input line between < and >

```
$ sed 's/^.*/<&>/' employee.txt
<101,John Doe,CEO>
<102,Jason Smith,IT Manager>
<103,Raj Reddy,Sysadmin>
<104,Anand Ram,Developer>
<105,Jane Miller,Sales Manager>
```

17. Substitution Grouping (Single Group)

Grouping can be used in sed just like in a normal regular expression. A group is opened with “\(” and closed with “\)”. Grouping can be used in combination with back-referencing.

A back-reference is the re-use of a part of a regular expression selected by grouping. Back-references in sed can be used in both a regular expression and in the replacement part of the substitute command.

Single grouping:

```
$ sed 's/\([^,]*\).*/\1/g' employee.txt
101
102
103
104
105
```

In the above example:

- Regular expression \([^,]*\) matches the string up to the 1st comma.
- \1 in the replacement-string replaces the first matched group.
- g is the global substitute flag.

This sed example displays only the first field from the /etc/passwd file, i.e. it displays only the username:

```
sed 's/\([^:]*\).*/\1/' /etc/passwd
```

The following example encloses the 1st letter in every word inside (), if the 1st character is upper case.

```
$ echo "The Geek Stuff" | sed 's/\(\b[A-Z]\)/\(\1\)/g'
(T)he (G)eek (S)tuff
```

For the next example create a numbers.txt sample file as shown below.

```
$ vi numbers.txt
1
12
123
```

```
1234
12345
123456
```

Commify numbers, i.e. insert commas to make them more readable:

```
$ sed 's/\(^\|[^0-9.]\)\([0-9]\+\)\([0-9]\
{3\}\)/\1\2,\3/g' numbers.txt
1
12
123
1,234
12,345
123,456
```

The above command should be executed in a single line as shown below.

```
sed 's/\(^\|[^0-9.]\)\([0-9]\+\)\([0-9]\{3\}\)/\1\2,\3/g' numbers.txt
```

18. Substitution Grouping (Multiple Group)

In multi grouping, you can have multiple groups enclosed in multiple "\(" and "\)". When you have multiple groups in the substitute regular expression, you can use \n to specify the nth group in the sed replacement string. An example is shown below.

Get only the 1st column (employee id) and the 3rd column (title):

```
$ sed 's/\([^,]*\),\([^,]*\),\([^,]*\).*/\1,\3/g'
employee.txt
101,CEO
102,IT Manager
103,Sysadmin
104,Developer
105,Sales Manager
```

The above command should be executed in a single line as shown below.

```
sed 's/\([^,]*\),\([^,]*\),\([^,]*\).*/\1,\3/g' employee.txt
```

In the above example, you can see three groups mentioned in the original-string (reg-ex). These three groups are separated by commas.

- ([^,]*\) is group 1 that matches the employee id
- , is the field separator after group 1
- ([^,]*\) is group 2 that matches the employee name
- , is the field separator after group 2
- ([^,]*\) is group 3 that matches the employee title
- , is the field separator after group 3 The replacement-string section of the above example indicates how these groups should be used.
- \1 is to print group 1 (employee id)
- , is to print a comma after printing group 1
- \3 is to print group 1 (title)

Note: Sed can hold a maximum of 9 groups referenced using \1 through \9

Swap field 1 (employee id) with field 2 (employee name); print the employee.txt file:

```
$ sed 's/\([^,]*\),\([^,]*\),\(.*\).*/\2,\1,\3/g'
employee.txt
John Doe,101,CEO
Jason Smith,102,IT Manager
Raj Reddy,103,Sysadmin
Anand Ram,104,Developer
Jane Miller,105,Sales Manager
```

The above command should be executed in a single line as shown below.

```
sed 's/\([^,]*\),\([^,]*\),\(.*\).*/\2,\1,\3/g' employee.txt
```

19. Gnu Sed Only Replacement String Flags

These flags are available only in GNU version of sed. They can be used in the replacement-string part of the sed substitute command.

\l replacement string flag

When you specify \l in the replacement-string part, it treats the character that immediately follows \l as lower case. You already know the following simple example will change John to JOHNNY.

```
sed 's/John/JOHNNY/' employee.txt
```

The following example contains \l before H in the replacement-string (i.e. JO\lHNNY). This will change only the character h in JOHNNY to lower case.

Change John to JOhNNY:

```
$ sed -n 's/John/JO\lHNNY/p' employee.txt
101,JOhNNY Doe,CEO
```

\L replacement string flag

When you specify \L in the replacement-string part, it treats the rest of the characters as lower case.

The following example contains \L before H in the replacement-string (i.e. JO\lHNNY). This will change the rest of the characters from h to lower case.

Change Johnny to JOhnny:

```
$ sed -n 's/John/JO\LHNNY/p' employee.txt
101,JOhnny Doe,CEO
```

\u replacement string flag

Just like \l, but for upper case. When you specify \l in the replacement-string part, it treats the character that immediately

follows \u as upper case. The following example contains \u before h in the replacement-string (i.e. jo\uhnny). This will change only the character h in johnny to upper case.

Change John to joHnny:

```
$ sed -n 's/John/jo\uhnny/p' employee.txt
101,joHnny Doe,CEO
```

\U replacement string flag

When you specify \U in the replacement-string part, it treats the rest of the characters as upper case. The following example contains \U before h in the replacement-string (i.e. jo\Uhnny). This will change the rest of the characters from h in johnny to upper case.

Change John to joHNNY:

```
$ sed -n 's/John/jo\Uhnny/p' employee.txt
101,joHNNY Doe,CEO
```

\E replacement string flag

This should be used in conjunction with either \L or \U. This stops the conversion initiated by either \L or \U. The following example prints the whole replacement string "Johnny Boy" in upper case, as we have \U at the beginning of the replacement-string.

Change John to JOHNNY BOY:

```
$ sed -n 's/John/\UJohnny Boy/p' employee.txt
101,JOHNNY BOY Doe,CEO
```

Change John to JOHNNY Boy:

```
$ sed -n 's/John/\UJohnny\E Boy/p' employee.txt
101,JOHNNY Boy Doe,CEO
```

The above example prints only "Johnny" in the upper case, as we have \E immediately after "Johnny" in the replacement-string.

Replacement String Flag Usages

The above static examples are shown only to understand how these switches works. However, the flags don't have much value when used with static values, as you can just type the static values in the exact case needed.

The flags are quite useful when combined with grouping. In the previous example we learned how to swap field 1 with field 3 using grouping. You can convert a whole grouping to either upper or lower case using these switches.

Employee name in all upper case, and title in all lower case:

```
$ sed 's/\([^,]*\),\([^,]*\),\(.*\).*/\U\2\E,\1,\L\3/g'
employee.txt
JOHN DOE,101,ceo
JASON SMITH,102,it manager
RAJ REDDY,103,sysadmin
ANAND RAM,104,developer
JANE MILLER,105,sales manager
```

The above command should be executed in a single line as shown below.

```
sed 's/\([^,]*\),\([^,]*\),\(.*\).*/\U\2\E,\1,\L\3/g' employee.txt
```

In the above example, in the replacement-string, we have the following:

- \U\2\E - This indicates that this field, which is the 2nd group (employee name), should be converted to upper case. \U start the upper case conversion, and \E stops it.
- \L\3 - This indicates that this field, which is 3rd group (title), should be converted to lower case. \L starts the lower case conversion for rest of the characters.

Chapter 3. Regular Expressions

20. Regular Expression Fundamentals

Regular expressions (or regex) are used in many *nix commands, including sed.

Beginning of line (^)

The Caret Symbol ^ matches at the start of a line.

Display lines which start with 103:

```
$ sed -n '/^103/ p' employee.txt
103,Raj Reddy,Sysadmin
```

Note that ^ matches the expression at the beginning of a line, only if it is the first character in a regular expression. In this example, ^N matches all the lines that begins with N.

End of line ($)

The dollar symbol $ matches the end of a line.

Display lines which end with the letter r:

```
$ sed -n '/r$/ p' employee.txt
102,Jason Smith,IT Manager
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Single Character (.)

The special meta-character “.” (dot) matches any character except the end of the line character.

- . matches single character
- .. matches two characters

- ... matches three characters
- etc.

In the following example, the pattern "J followed by three characters and a space" will be replaced with "Jason followed by a space".

So, "J... " matches both "John " and "Jane " from employee.txt, and these two lines are replaced accordingly as shown below.

```
$ sed -n 's/J... /Jason /p' employee.txt
101,Jason Doe,CEO
105,Jason Miller,Sales Manager
```

Zero or more Occurrences (*)

The special character “*” (star) matches zero or more occurrences of the previous character. For example, the pattern ’1*’ matches zero or more ’1'.

For this example create the following log.txt file:

```
$ vi log.txt
log: Input Validated
log:
log:  testing resumed
log:
log:output created
```

Suppose you would like to view only the lines that contain "log:" followed by a message. The message might immediately follow the log: or might have some spaces. You don't want to view the lines that contain "log:" without anything.

Display all the lines that contain "log:" followed by one or more spaces followed by a character:

```
$ sed -n '/log: *./ p' log.txt
log: Input Validated
log:  testing resumed
```

```
log:output created
```

Note: In the above example the dot at the end is necessary. If not included, sed will also print all the lines containing "log:" only.

One or more Occurrence (\+)

The special character “\+” matches one or more occurrence of the previous character. For example, a space before “\+”, i.e ” \+” matches at least one or more space character.

Let us use the same log.txt as an example file.

Display all the lines that contain "log:" followed by one or more spaces:

```
$ sed -n '/log: \+/ p' log.txt
log: Input Validated
log:  testing resumed
```

Note: In addition to not matching the "log:" only lines, the above example also didn't match the line "log:output created", as there is no space after "log:" in this line.

Zero or one Occurrence (\?)

The special character “?” matches zero or one occurrences of the previous character as shown below.

```
$ sed -n '/log: \?/ p' log.txt
log: Input Validated
log:
log:  testing resumed
log:
log:output created
```

Escaping the Special Character (\)

If you want to search for special characters (for example: * , dot) in the content you have to escape the special character in the regular expression.

```
$ sed -n '/127\.0\.0\.1/ p' /etc/hosts
127.0.0.1       localhost.localdomain localhost
```

Character Class ([0-9])

The character class is nothing but a list of characters mentioned within a square bracket; this is used to match only one out of several characters.

Match any line that contains 2 or 3 or 4:

```
$ sed -n '/[234]/ p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

Within the square bracket, you can use a hyphen you can specify a range of characters. For example, [0123456789] can be represented by [0-9], and alphabetic ranges can be specified such as [a-z],[A-Z] etc.

Match any line that contains 2 or 3 or 4 (alternate form):

```
$ sed -n '/[2-4]/ p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

21. Additional Regular Expressions

OR Operation (|)

The pipe character (|) is used to specify that either of two whole subexpressions could occur in a position. “subexpression1|subexpression2” matches either subexpression1 or subexpression2.

Print lines containing either 101 or 102:

```
$ sed -n '/101\|102/ p' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
```

Please note that the | symbol is escaped with a /.

Print lines that contain a character from 2 to 3 or that contain the string 105:

```
$ sed -n '/[2-3]\|105/ p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
105,Jane Miller,Sales Manager
```

Exactly M Occurrences ({m})

A Regular expression followed by {m} matches exactly m occurrences of the preceding expression.

For this example create the following numbers.txt file.

```
$ vi numbers.txt
1
12
123
1234
12345
123456
```

Print lines that contain any digit (will print all lines):

```
$ sed -n '/[0-9]/ p' numbers.txt
1
12
123
1234
12345
```

```
123456
```

Print lines consisting of exactly 5 digits:

```
$ sed -n '/^[0-9]\{5\}$/ p' numbers.txt
12345
```

M to N Occurrences ({m,n})

A regular expression followed by {m,n} indicates that the preceding item must match at least m times, but not more than n times. The values of m and n must be non-negative and smaller than 255.

Print lines consisting of at least 3 but not more than 5 digits:

```
$ sed -n '/^[0-9]\{3,5\}$/ p' numbers.txt
123
1234
12345
```

A Regular expression followed by {m,} is a special case that matches m or more occurrences of the preceding expression.

Word Boundary (\b)

\b is used to match a word boundary. \b matches any character(s) at the beginning (\bxx) and/or end (xx\b) of a word, thus \bthe\b will find the but not they. \bthe will find the or they.

Create the following sample file for testing.

```
$ cat words.txt
word matching using: the
word matching using: thethe
word matching using: they
```

Match lines containing the whole word "the":

```
$ sed -n '/\bthe\b/ p' words.txt
word matching using: the
```

Please note that if you don't specify the \b at the end, it will match all lines.

Match lines containing words that start with “the”:

```
$ sed -n '/\bthe/ p' words.txt
word matching using: the
word matching using: thethe
word matching using: they
```

Back References (\n)

Back references let you group expressions for further use.

Match only the line that has the word "the" repeated twice:

```
$ sed -n '/\(the\)\1/ p' words.txt
```

Using the same logic, the regular expression "\([0-9]\)\1" matches two digit number in which both the digits are same number—like 11,22,33 ...

22. Sed Substitution Using Regular Expression

The following are few sed substitution examples that uses regular expressions.

Replace the last two characters in every line of employee.txt with ",Not Defined":

```
$ sed 's/..$/,Not Defined/' employee.txt
101,John Doe,C,Not Defined
102,Jason Smith,IT Manag,Not Defined
103,Raj Reddy,Sysadm,Not Defined
104,Anand Ram,Develop,Not Defined
105,Jane Miller,Sales Manag,Not Defined
```

Delete the rest of the line starting from “Manager”:

```
$ sed 's/Manager.*//' employee.txt
101,John Doe,CEO
102,Jason Smith,IT
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales
```

Delete all lines that start with "#" :

```
sed -e 's/#.*// ; /^$/ d' employee.txt
```

Create the following test.html for the next example:

```
$ vi test.html
<html><body><h1>Hello World!</h1></body></html>
```

Strip all html tags from test.html:

```
$ sed -e 's/<[^>]*>//g' test.html
Hello World!
```

Remove all comments and blank lines:

```
sed -e 's/#.*//' -e '/^$/ d' /etc/profile
```

Remove only the comments. Leave the blank lines:

```
sed -e '/^#.*/ d' /etc/profile
```

You can convert DOS newlines (CR/LF) to Unix format Using sed. When you copy the DOS file to Unix, you could find \r\n in the end of each line.

Convert the DOS file format to Unix file format using sed:

```
sed 's/.$//' filename
```

Chapter 4. Sed Execution

23. Multiple Sed Commands in Command Line

As we showed in Chapter 1, there are several methods to execute multiple sed commands from the command line.

1. Use multiple -e option in the command line

Use multiple sed commands using -e sed command line option as shown below:

```
sed -e 'command1' -e 'command2' -e 'command3'
```

Search for root, or nobody, or mail in the /etc/passwd file:

```
sed -n -e '/^root/ p' -e '/^nobody/ p' -e '/^mail/
p' /etc/passwd
```

The above command should be executed in a single line as shown below.

```
sed -n -e '/^root/ p' -e '/^nobody/ p' -e '/^mail/ p' /etc/passwd
```

2. Break-up several sed commands using \

When you have a very long command, such as when executing several sed commands in the command line using -e, you can break it up using \

```
sed -n -e '/^root/ p' \
-e '/^nobody/ p' \
-e '/^mail/ p' \
/etc/passwd
```

3. Group multiple commands using { }

When you have a lot of sed commands to be executed, you can group them together using { } as shown below.

```
sed -n '{
/^root/ p
/^nobody/ p
/^mail/ p
}' /etc/passwd
```

24. Sed Script Files

If you want to reuse a set of sed commands, create a sed script file with all the sed commands and execute it using -f command line option as shown below.

First, create a file that contains all the sed commands as shown below. You already know what these individual sed commands do, as we explained it in the previous sections.

```
$ vi mycommands.sed
s/\([^,]*\),\([^,]*\),\(.*\).*/\2,\1,\3/g
s/^.*/<&>/
s/Developer/IT Manager/
s/Manager/Director/
```

Next, execute this sed command file on the input file.

```
$ sed -f mycommands.sed employee.txt
<John Doe,101,CEO>
<Jason Smith,102,IT Director>
<Raj Reddy,103,Sysadmin>
<Anand Ram,104,IT Director>
<Jane Miller,105,Sales Director>
```

25. Sed Comments

Sed comments start with a #. We all understand that sed uses very cryptic language. The sed commands that you write today might look unfamiliar if you view them after a long time. So, it is recommended to document what you mean inside the sed script file using sed comments, as shown below.

```
$ vi mycommands.sed
# Swap field 1 (employee id) with field 2 (employee
name)
s/\([^,]*\),\([^,]*\),\(.*\).*/\2,\1,\3/g
# Enclose the whole line within < and >
s/^.*/<&>/
# Replace Developer with IT Manager
s/Developer/IT Manager/
# Replace Manager with Director
s/Manager/Director/
```

Note: If the 1st 2 characters of the 1st line in the *.sed script are #n, sed will automatically use the -n (don't print the pattern buffer) option.

26. Sed as an Interpreter

Just as you write shell scripts and execute them from the command line just by calling the file name, you can set up sed scripts for execution from the command line, i.e. Sed can be involved as an interpreter. To do this, add "#!/bin/sed -f" as the 1st line to your sed-script.sh file as shown below.

```
$ vi myscript.sed
#!/bin/sed -f
# Swap field 1 (employee id) with field 2 (employee
name)
s/\([^,]*\),\([^,]*\),\(.*\).*/\2,\1,\3/g
# Enclose the whole line within < and >
s/^.*/<&>/
# Replace Developer with IT Manager
```

```
s/Developer/IT Manager/
# Replace Manager with Director
s/Manager/Director/
```

Now, execute the sed script directly by invoking it from the command line.

```
chmod u+x myscript.sed

./myscript.sed employee.txt
```

You can also specify -n in the 1st line of the sed script to suppress output.

```
$ vi testscript.sed
#!/bin/sed -nf
/root/ p
/nobody/ p
```

Now, execute the above test script as shown below.

```
chmod u+x testscript.sed

./testscript.sed /etc/passwd
```

Just for testing purposes, remove the -n from the 1st line of testscript.sed and execute it again to see how it works.

Important note: you must use -nf (and not -fn). If you specify -fn, you'll get the following error message when you execute the sed script.

```
$ ./testscript.sed /etc/passwd
/bin/sed: couldn't open file n: No such file or
directory
```

27. Modifying the Input File Directly

As you know already, sed doesn't modify the input files by default. Sed writes the output to standard output. When you want to store that in a file, you redirect it to a file (or use the w command.

Before we continue with this example, take a backup of employee.txt:

```
cp employee.txt employee.txt.orig
```

To make a modification directly on the input-file, you typically redirect the output to a temporary file, and then rename the temporary file to a new file.

```
sed 's/John/Johnny/' employee.txt > new-employee.txt
mv new-employee.txt employee.txt
```

Instead, you can use the sed command line option -i, which lets sed directly modify the input file.

Replace John with Johnny in the original employee.txt file itself:

```
$ sed -i 's/John/Johnny/' employee.txt

$ cat employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Again, please pay attention that **-i modifies the input-file**. Probably you will want to do this sometimes, but be very careful. One thing you can do to protect yourself is to add a file extension whenever you use -i. Sed will make a backup of the original file before writing the new content.

Replace John with Johnny in the original employee.txt file but save a backup copy:

```
$ sed -ibak 's/John/Johnny/' employee.txt
```

This takes the backup of the original file as shown below.

```
$ cat employee.txtbak
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

The original input file was modified by the above sed command.

```
$ cat employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Instead of -i, you can also use the longer form, --in-place. Both of the following commands are the same.

```
sed -ibak 's/John/Johnny/' employee.txt
sed --in-place=bak 's/John/Johnny/' employee.txt
```

Finally, restore the original employee.txt file, as we need that for the rest of our examples:

```
cp employee.txt.orig employee.txt
```

Chapter 5. Additional Sed Commands

28. Append Line After (a command)

You can insert a new line after a specific location by using the sed append command (a).

Syntax:

```
$ sed '[address] a the-line-to-append' input-file
```

Add a new record to the employee.txt file after line number:

```
$ sed '2 a 203,Jack Johnson,Engineer' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
203,Jack Johnson,Engineer
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Add a new record to the end of the employee.txt file:

```
$ sed '$ a 106,Jack Johnson,Engineer' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Jack Johnson,Engineer
```

You can also append multiple lines using the sed a command.

Add two lines after the line that matches 'Jason':

```
$ sed '/Jason/a\
203,Jack Johnson,Engineer\
204,Mark Smith,Sales Engineer' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
203,Jack Johnson,Engineer
204,Mark Smith,Sales Engineer
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

29. Insert Line Before (i command)

The sed insert command (i) works just like the append command except that it inserts a line **before** a specific location instead of after the location.

Syntax:

```
$ sed '[address] i the-line-to-insert' input-file
```

Insert a new record before line number 2 of the employee.txt file:

```
$ sed '2 i 203,Jack Johnson,Engineer' employee.txt
101,John Doe,CEO
203,Jack Johnson,Engineer
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Insert a new record before the last line of the employee.txt file:

```
$ sed '$ i 108,Jack Johnson,Engineer' employee.txt
101,John Doe,CEO
```

```
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
108,Jack Johnson,Engineer
105,Jane Miller,Sales Manager
```

You can also insert multiple lines using the sed i command.

Insert two lines before the line that matches 'Jason':

```
$ sed '/Jason/i\
203,Jack Johnson,Engineer\
204,Mark Smith,Sales Engineer' employee.txt
101,John Doe,CEO
203,Jack Johnson,Engineer
204,Mark Smith,Sales Engineer
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

30. Change Line (c command)

The sed change command (c) lets you replace an existing line with new text.

Syntax:

```
$ sed '[address] c the-line-to-insert' input-file
```

Delete the record at line number 2 and replace it with a new record:

```
$ sed '2 c 202,Jack Johnson,Engineer' employee.txt
101,John Doe,CEO
202,Jack Johnson,Engineer
103,Raj Reddy,Sysadmin
```

```
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

You can also replace a single line with multiple lines.

Delete the line that matches 'Raj' and replaces it with two new lines:

```
$ sed '/Raj/c\
203,Jack Johnson,Engineer\
204,Mark Smith,Sales Engineer' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
203,Jack Johnson,Engineer
204,Mark Smith,Sales Engineer
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

31. Combine a, i, and c Commands

You can also combine the a, i, and c commands. the following sed example does all these three things:

- a - Append 'Jack Johnson' after 'Jason'
- i - Insert 'Mark Smith' before 'Jason'
- c - Change 'Jason' to 'Joe Mason'

```
$ sed '/Jason/ {
a\
204,Jack Johnson,Engineer
i\
202,Mark Smith,Sales Engineer
c\
203,Joe Mason,Sysadmin
```

```
}' employee.txt
101,John Doe,CEO
202,Mark Smith,Sales Engineer
203,Joe Mason,Sysadmin
204,Jack Johnson,Engineer
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

32. Print Hidden Characters (l command)

The sed l command prints the hidden characters, for example, \t for tab, and $ for end of the line.

For testing, create a test file with the following content. Make sure to use the tab key between the fields in this file.

```
$ cat tabfile.txt
fname     First Name
lname     Last Name
mname     Middle Name
```

Executing the sed l command will display \t for tab, and $ for EOL:

```
$ sed -n l tabfile.txt
fname\tFirst Name$
lname\tLast Name$
mname\tMiddle Name$
```

When you specify a number followed by the l command, the output line is wrapped at the nth number using a non printable character as shown in the example below. This works only on GNU sed.

```
$ sed -n 'l 20' employee.txt
101,John Doe,CEO$
```

```
102,Jason Smith,IT \
Manager$
103,Raj Reddy,Sysad\
min$
104,Anand Ram,Devel\
oper$
105,Jane Miller,Sal\
es Manager$
```

33. Print Line Numbers (= command)

The sed = command prints line numbers followed by the line content from the input-file.

Print all line numbers:

```
$ sed = employee.txt
1
101,John Doe,CEO
2
102,Jason Smith,IT Manager
3
103,Raj Reddy,Sysadmin
4
104,Anand Ram,Developer
5
105,Jane Miller,Sales Manager
```

Note: You can print the line number and the line content in the same line by combining = command with N command (more on this later).

Print line numbers only for lines 1,2 and 3:

```
$ sed '1,3 =' employee.txt
1
101,John Doe,CEO
2
```

```
102,Jason Smith,IT Manager
3
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Print the line number only for those lines that contain the keyword Jane. This still prints the original line content from the intput-file:

```
$ sed '/Jane/ =' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
5
105,Jane Miller,Sales Manager
```

If you want to know only the line numbers of lines that contains the keyword (i.e. without printing the original lines from the file), use -n option along with = as shown below.

```
$ sed -n '/Raj/ =' employee.txt
3
```

Print the total number of lines in a file:

```
$ sed -n '$ =' employee.txt
5
```

34. Change Case (using the y 'transform' command)

The sed y command transforms characters by position. A convenient use for this is to convert upper case to lower case and vice versa.

In this example character "a" will be transformed to A, b to B, c to C, etc.:

```
$ sed 'y/abcde/ABCDE/' employee.txt
101,John DoE,CEO
102,JAson Smith,IT MAnAgEr
103,RAj REDDy,SysADmin
104,AnAnD RAm,DEvElopEr
105,JAnE MillEr,SAlEs MAnAgEr
```

Transform all lower-case letters to upper-case:

```
$ sed
'y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXY
Z/' employee.txt
101,JOHN DOE,CEO
102,JASON SMITH,IT MANAGER
103,RAJ REDDY,SYSADMIN
104,ANAND RAM,DEVELOPER
105,JANE MILLER,SALES MANAGER
```

The above command should be executed in a single line as shown below.

```
sed 'y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/' employee.txt
```

35. Multiple Files in Command Line

In all our previous sed examples, we passed only one input file. You can also pass multiple input files as shown below.

The following example searches for root in the /etc/passwd file and prints it:

```
$ sed -n '/root/ p' /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

The following example searches for root in the /etc/group and prints it:

```
$ sed -n '/root/ p' /etc/group
root:x:0:
```

Search for root in both the /etc/passwd and /etc/group file:

```
$ sed -n '/root/ p' /etc/passwd /etc/group
root:x:0:0:root:/root:/bin/bash
root:x:0:
```

36. Quit Sed (q command)

The sed q command causes sed to quit executing commands.

As we discussed earlier, the normal sed execution flow is Read, Execute, Print, Repeat.

When sed executes the q command, it simply quits without executing the rest of the sed commands, and without repeating the rest of the lines from the input-file.

Quit after printing the 1st line:

```
$ sed 'q' employee.txt
101,John Doe,CEO
```

Quit after the 5th line. So, this prints the 1st 5 lines:

```
$ sed '5 q' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

Print all the lines until the 1st line that contains the keyword 'Manager':

```
$ sed '/Manager/q' employee.txt
101,John Doe,CEO
```

```
102,Jason Smith,IT Manager
```

Note: q command doesn't take range of address. It works only on a single address. (or a single pattern)

37. Read from File (r command)

The sed r command will read the content of another file and print it at a specified location while processing the input-file. The following example will read the content of log.txt file and print it after printing the last line of employee.txt. Basically this combines both employee.txt and log.txt and prints the result.

```
$ sed '$ r log.txt' employee.txt
```

You can also specify a pattern with the r command. The following example will read the content of log.txt and print it after the line that matches 'Raj' in the employee.txt.

Insert the log.txt file after the 'Raj' keyword in the employee.txt file:

```
$ sed '/Raj/ r log.txt' employee.txt
```

38. Simulating Unix commands in sed (cat, grep, head)

We have already seen examples that worked very much like other standard UNIX commands. Using sed you can simulate many commands. Do this just to learn how sed works.

Cat in sed

```
cat employee.txt
```

Each of the following sed commands produces the same output as the cat command above.

```
sed 's/JUNK/&/p' employee.txt
sed -n 'p' employee.txt
sed 'n' employee.txt
sed 'N' employee.txt
```

Grep in sed

Simple grep:

```
grep Jane employee.txt
```

Each of the following sed commands produces the same output as the grep command above.

```
sed -n 's/Jane/&/p' employee.txt
sed -n '/Jane/ p' employee.txt
```

grep -v (print non-matching lines):

```
grep -v Jane employee.txt
```

The following sed command is equivalent to the above "grep -v" command.

```
sed -n '/Jane/ !p' employee.txt
```

Head in sed

```
head -10 /etc/passwd
```

Each of the following sed commands produces the same output as the head command above.

```
sed '11,$ d' /etc/passwd
sed -n '1,10 p' /etc/passwd
sed '10 q' /etc/passwd
```

39. Sed Command Line Options

-n option

We already discussed this option and we have used it in many examples. The sed option -n suppresses the default printing that happens as part of the standard sed flow.

You can also use --quiet, or –-silent instead of -n. They are identical in function.

All of the following commands are the same:

```
sed -n 'p' employee.txt
sed --quiet 'p' employee.txt
sed --silent 'p' employee.txt
```

-f option

You can also combine multiple sed-commands in a file and call the sed script file using the -f option. We demonstrated this earlier. You can also use -–file.

All of the following commands are the same:

```
sed -n -f test-script.sed /etc/passwd
sed -n --file=test-script.sed /etc/passwd
```

-e option

Use -e to execute a sed command script from the command line. You can use multiple -e options from the command line. You can also use –-expression.

All of the following commands are the same:

```
sed -n -e '/root/ p' /etc/passwd
sed -n --expression '/root/ p' /etc/passwd
```

-i option

As we already discussed sed doesn't touch the input file. It always prints to standard output, Or you can use the w command to write the output to a different file. We also showed how sed can use the -i option to modify the input file directly.

Replace John with Johnny in the original employee.txt file:

```
sed -i 's/John/Johnny/' employee.txt
```

Perform the same command but take a backup by passing an extension to -i.

```
sed -ibak 's/John/Johnny/' employee.txt
```

Instead of -i, you can also use –-in-place.

Both of the following commands are the same:

```
sed -ibak 's/John/Johnny/' employee.txt
sed --in-place=bak 's/John/Johnny/' employee.txt
```

-c option

This should be used in conjunction with sed option -i. Sed option -i typically uses a temporary file to create the changes and renames it to the original input-file when the operation is completed. This might cause file ownership to change. When you use -c along with -i, the input file ownership will not change. You can also use –-copy.

Both of the following commands are the same:

```
sed -ibak -c 's/John/Johnny/' employee.txt
sed --in-place=bak --copy 's/John/Johnny/' employee.txt
```

-l option

Specify the line length. This needs to be used in conjunction with the sed l command. The value you specify in the -l option will be used as the line size. You can also use –-line-length.

All the following commands are the same.

```
sed -n -l 20 'l' employee.txt
sed -n --line-length=20 employee.txt
```

Please note that you can also achieve the same output without specifying -n option as shown below.

```
sed -n 'l 20' employee.txt --posix option
```

40. Print Pattern Space (n command)

The sed n command prints the current pattern space and fetches the next line from the input-file. This happens in the middle of command execution, and so it can change the normal flow if it occurs between other commands.

Print the pattern space for each line:

```
$ sed n employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

If you specify -n flag when you are using the n command, sed will not print anything.

```
$ sed -n n employee.txt
```

As we discussed earlier, normal sed execution flow is Read, Execute (all available sed commands), Print, Repeat.

The sed n command lets you change that flow. The sed n command will print the current pattern space, clear the current pattern space, read the next line from the input-file, and continue the command flow.

Let us assume that you have 2 sed commands before and 2 after the n command as shown below.

```
sed-command-1
sed-command-2
n
sed-command-3
sed-command-4
```

In this case, sed-command-1 and sed-command-2 will be applied to the current line in the pattern space; when sed encounters the n command, it will clear the current line from the pattern space, read the next line from the input-file, and apply sed-command-3 and sed-command-4 to this newly read line in the sed pattern space.

Note: The sed n command by itself is relatively useless as you see in the above examples. However, it is extremely powerful when combined with the sed hold pattern commands that are discussed in the following hacks.

Chapter 6. Sed Hold and Pattern Space Commands

Sed has two types of internal storage space:

- **Pattern space:** You already know about pattern space, which is used as part of the typical sed execution flow. Pattern space is the internal sed buffer where sed places, and modifies, the line it reads from the input file.
- **Hold space:** This is an additional buffer available where sed can hold temporary data. Sed allows you to move data back and forth between pattern space and hold space, but you cannot execute the typical sed commands on the hold space. As we already discussed, pattern space gets deleted at the end of every cycle in a typical sed execution flow. However, the content of the hold space will is retained from one cycle to the next; it is not deleted between cycles.

Please create a new text file to be used for the sed hold space examples:

```
$ vi empnametitle.txt
John Doe
CEO
Jason Smith
IT Manager
Raj Reddy
Sysadmin
Anand Ram
Developer
Jane Miller
Sales Manager
```

As you can see, for each employee this file contains name and title on two consecutive lines.

41. Swap Pattern Space with Hold Space (x command)

The sed Exchange (x) command swaps pattern space with hold space. This command in itself is not that helpful, unless it is combined with other sed commands; however, in conjunction with other commands, it is quite powerful.

Suppose that pattern space contains "line 1" and hold space contains "line 2". After the x command is executed, pattern space will have "line 2", and hold space will have "line 1".

The following example prints the names of the managers. It looks for the keyword 'Manager' and prints the previous line.

Print manager names from empnametitle.txt:

```
$ sed -n -e 'x;n' -e '/Manager/{x;p}' empnametitle.txt
Jason Smith
Jane Miller
```

In the above example:

- **{x;n}** - x swaps pattern space to the hold space; n reads the next line into the pattern space. So, this command saves the current line in hold space and reads the next line into pattern space. For the example file, it is saving employee name to hold space and fetching employee title into pattern space.
- **/Manager/{x;p}** - If the content of the pattern space contains the keyword 'Manager', this command swaps pattern space with hold space and then prints pattern space. This means that if the employee title contains 'Manager' the employee name will be printed.

You can also save this in a sed script file and execute it as shown below.

```
$ vi x.sed
#!/bin/sed -nf
```

```
x;n
/Manager/{x;p}

$ chmod u+x empnametitle.txt

$ ./x.sed empnametitle.txt
Jason Smith
Jane Miller
```

42. Copy Pattern Space to Hold Space (h command)

The hold command (h) copies pattern space to hold space. Unlike the x command, the h command does not change the content of pattern space. The previous content of the hold space is overwritten with the content from the pattern space.

Suppose pattern space contains "line 1" and hold space contains "line 2"; after the h command is executed, pattern space is not changed and will still have "line 1", but hold space will also have "line 1".

Print the names of the managers:

```
$ sed -n -e '/Manager/!h' -e '/Manager/{x;p}'
empnametitle.txt
Jason Smith
Jane Miller
```

The above command should be executed in a single line as shown below.

```
sed -n -e '/Manager/!h' -e '/Manager/{x;p}' empnametitle.txt
```

In the above example:

- **/Manager/!h** - If the content of the pattern space doesn't contain Manager (the ! after the pattern means "not equal to" the pattern), copy the content of the pattern space to the hold space. (In this case, this might be employee name (or) a title that is not "Manager".) Note that, unlike the previous example,

this one does not use the 'n' command to get the next line; instead, the next line is fetched via normal execution flow.

- **/Manager/{x;p}** - If the content of the pattern space contains the keyword 'Manager', this command swaps pattern space with hold space and prints. This is identical to the command we used for printing in the example for the x command.

You can also save this in a sed script file and execute it as shown below.

```
$ vi h.sed
#!/bin/sed -nf
/Manager/!h
/Manager/{x;p}

$ chmod u+x empnametitle.txt

$ ./h.sed empnametitle.txt
Jason Smith
Jane Miller
```

43. Append Pattern Space to Hold Space (H command)

Capital H is the command to append pattern space to hold space with a new line. The previous content of hold space is not overwritten; instead the content of pattern space is appended to the existing content of hold space by adding a new line at the end.

Suppose pattern space contains "line 1" and hold space contains "line 2"; after the H command is executed, pattern space is not changed and will still have "line 1", but hold space will have "line 2\nline 1".

Print the name and title (in separate lines) of the managers:

```
$ sed -n -e '/Manager/!h' -e '/Manager/{H;x;p}'
empnametitle.txt
Jason Smith
```

```
IT Manager
Jane Miller
Sales Manager
```

The above command should be executed in a single line as shown below.

```
sed -n -e '/Manager/!h' -e '/Manager/{H;x;p}' empnametitle.txt
```

In the above example:

- **/Manager/!h** - If the content of the pattern space doesn't contain Manager (the ! after the pattern means "not equal to" the pattern), copy the content of the pattern space to the hold space. (In this case, this might employee name (or) a title that is not "Manager".) This is the same command we used in the h command example.
- **/Manager/{H;x;p}** - If the content of the pattern space contains the keyword 'Manager', the H command appends pattern space (which is Manager) to hold space with a new line. So, the hold space at this stage will have "Employee Name\nTitle" (which contains the keyword manager). The x command swaps hold space back into pattern space, and p prints the pattern space.

You can also save this in a sed script file and execute it as shown below.

```
$ vi H-upper.sed
#!/bin/sed -nf
/Manager/!h
/Manager/{H;x;p}

$ chmod u+x H-upper.sed

$ ./H-upper.sed empnametitle.txt
Jason Smith
IT Manager
Jane Miller
```

```
Sales Manager
```

The above example can be slightly modified, if you want the employee name and title to be printed on the same line with colon : as a delimiter:

```
$ sed -n -e '/Manager/!h' -e '/Manager/{H;x;s/\n/:/;p}'
empnametitle.txt
Jason Smith:IT Manager
Jane Miller:Sales Manager
```

The above command should be executed in a single line as shown below.

```
sed -n -e '/Manager/!h' -e '/Manager/{H;x;s/\n/:/;p}' empnametitle.txt
```

In the second example everything is same as the previous example except for the substitute command added to the 2nd -e option. The H, x, and p commands do the same thing as before; the s command replaces \n with : after swapping but before printing. Therefore the name and title are printed on one line, separated by a colon.

You can also save this in a sed script file and execute it as shown below.

```
$ vi H1-upper.sed
#!/bin/sed -nf
/Manager/!h
/Manager/{H;x;s/\n/:/;p}

$ chmod u+x H1-upper.sed

$ ./H1-upper.sed empnametitle.txt
Jason Smith:IT Manager
Jane Miller:Sales Manager
```

44. Copy Hold Space to Pattern Space (g command)

The sed get (g) command copies the content of hold space to pattern space.

Think of it this way: h command "holds" it in the hold space, g command "gets" it from the hold space.

Suppose pattern space contains "line 1" and hold space contains "line 2"; after the g command is executed, pattern space is changed and now contains "line 2", while hold space is not changed and still contains "line 2".

Print the names of the managers:

```
$ sed -n -e '/Manager/!h' -e '/Manager/{g;p}'
empnametitle.txt
Jason Smith
Jane Miller
```

The above command should be executed in a single line as shown below.

```
sed -n -e '/Manager/!h' -e '/Manager/{g;p}' empnametitle.txt
```

In the above example:

- **/Manager/!h** – we've been using this one for the last few examples. If the content of the pattern space doesn't contain Manager, copy the content of pattern space to hold space.
- **/Manager/{g;p}** – g gets the line from hold space and puts it in pattern space, then prints it.

You can also save this in a sed script file and execute it as shown below.

```
$ vi g.sed
#!/bin/sed -nf
/Manager/!h
/Manager/{g;p}
```

```
$ chmod u+x g.sed

$ ./g.sed empnametitle.txt
Jason Smith
Jane Miller
```

45. Append Hold Space to Pattern Space (G command)

Upper case G appends the content of hold space to pattern space with a new line. The previous content in the pattern space is not overwritten; instead the content from hold space is appended to the existing content in pattern space by adding a new line at the end.

G and g are related in the same way as H and h; the lower case version replaces the content while the upper case one appends to it.

Suppose pattern space contains "line 1" and hold space contains "line 2"; after the G command is executed, pattern space is changed to contain "line 1\nline 2" while hold space is not changed and still contains "line 2".

Prints the employee name and title of the managers separated by colon.

```
$ sed -n -e '/Manager/!h' -e '/Manager/{x;G;s/\n/:/;p}'
empnametitle.txt
Jason Smith:IT Manager
Jane Miller:Sales Manager
```

The above command should be executed in a single line as shown below.

```
sed -n -e '/Manager/!h' -e '/Manager/{x;G;s/\n/:/;p}' empnametitle.txt
```

In the above example:

- **/Manager/!h** – As in previous examples, if the content of pattern space doesn't contain Manager, copy pattern space to hold space.
- **/Manager/{x;G;s/\n/:/;p}** - If the content of the pattern space contains Manager, do the following:
  - **x** - Swap the content of pattern space with hold space. So, the employee name stored in hold space will now be in pattern space, while the title will be in hold space.
  - **G** - Appends the content of hold space (title) to pattern space (employee name). So, the pattern space at this stage will have "Employee Name\nTitle"
  - **s/\n/:/** This replaces the \n that separates the "Employee Name\nTitle" with a colon :
  - **p** prints the result (i.e. the content of pattern space).
  - Note that if we left out the x command, i.e. if we used /Manager/{G;s/\n/:/;p}, we would print the title:name instead of name:title for each manager.

You can also save this in a sed script file and execute it as shown below.

```
$ vi G-upper.sed
#!/bin/sed -nf
/Manager/!h
/Manager/{x;G;s/\n/:/;p}

$ chmod u+x G-upper.sed

$ ./G-upper.sed empnametitle.txt
Jason Smith:IT Manager
Jane Miller:Sales Manager
```

Chapter 7. Sed Multi-Line Commands and loops

Sed by default always handles one line at a time, unless we use the H, G, or N command to create multiple lines separated by new line.

This chapter will describe sed commands applicable to such multi-line buffers.

Note: When we have multiple lines, please keep in mind that ^ matches only the 1st character of the buffer, i.e. of all the multiple lines combined together, and $ matches only the last character in the buffer, i.e. the newline of the last line.

46. Append Next Line to Pattern Space (N command)

Just as upper case H and G append rather than replacing, the N command appends the next line from input-file to the pattern buffer, rather than replacing the current line.

As we discussed earlier the lower case n command prints the current pattern space, clears the pattern space, reads the next line from the input-file into pattern space and resumes command execution where it left off.

The upper case N command does not print the current pattern space and does not clear the pattern space. Instead, it adds a newline (\n) at the end of the current pattern space, appends the next line from the input-file to the current pattern space, and continues with the sed standard flow by executing the rest of the sed commands.

Print employee names and titles separated by colon:

```
$ sed -e '{N;s/\n/:/}' empnametitle.txt
John Doe:CEO
Jason Smith:IT Manager
```

```
Raj Reddy:Sysadmin
Anand Ram:Developer
Jane Miller:Sales Manager
```

In the above example:

- **N** appends new line to current pattern space (which has employee name) and appends the next line from input-file to the current pattern space. So, the pattern space will contain (employee name\ntitle).
- **s/\n/:/** This replaces the \n that separates the "Employee Name\nTitle" with a colon :

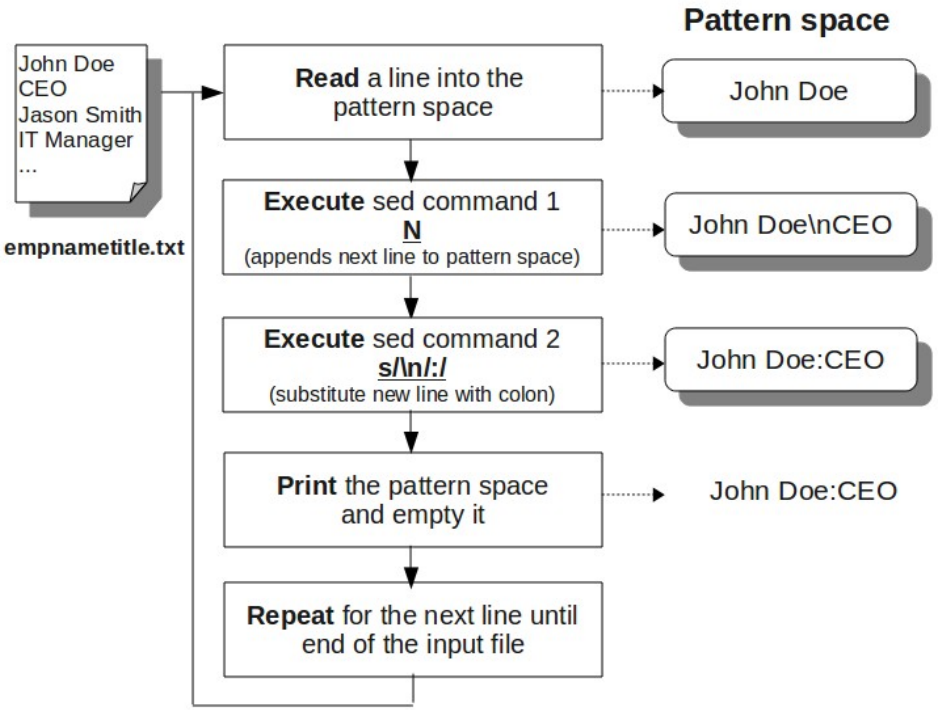

Fig: Illustration of the above example

The following example demonstrates the use of the N command to print the line number on the same line as the text, while printing each line from employee.txt.

Print line numbers:

```
$ sed -e '=' employee.txt | sed -e '{N;s/\n/ /}'
1 101,John Doe,CEO
2 102,Jason Smith,IT Manager
3 103,Raj Reddy,Sysadmin
4 104,Anand Ram,Developer
5 105,Jane Miller,Sales Manager
```

As we saw in our previous examples, the sed = command prints the line number first, and the original line next.

In this example, the N command adds \n to the current pattern space (which contains the line number), then reads the next line and appends it. So, the pattern space will contain "line-number\nOriginal-line-content". Then we execute s/\n/ / to change the newline (\n) to a space.

47. Print 1st Line in MultiLine (P command)

We have seen three upper case commands so far, each of which appended to rather than replacing the content of a buffer. We will now see that upper case P and D operate in a fashion similar to their lower case equivalents, but that they also do something special related to MultiLine buffers.

As we discussed earlier the lower case p command prints the pattern space. Upper case P command also prints the pattern space, but only until it encounters a new line (\n). The following example prints all the managers names from the empnametitle.txt file

```
$ sed -n -e 'N' -e '/Manager/P' empnametitle.txt
Jason Smith
Jane Miller
```

48. Delete 1st Line in MultiLine (D command)

As we discussed earlier the lower case d command deletes the current pattern space, reads the next line from the input-file to the pattern space, aborts the rest of the sed commands and starts the loop again.

The upper case D command does not read the next line to the pattern space after deleting it, nor does it completely clear the pattern buffer (unless it only has one line). Instead, it does the following:

- Deletes part of the pattern space until it encounters new line (\n).
- Aborts the rest of the sed commands and starts command execution from the beginning on the remaining content in the pattern buffer.

Consider the following file, which has comments enclosed between @ and @ for every title. Note that this comment also spans across the lines in some cases. For example @Information Technology officer@ spans across two rows. Create the following sample file.

```
$ vi empnametitle-with-comment.txt
John Doe
CEO @Chief Executive Officer@
Jason Smith
IT Manager @Information Technology
Officer@
Raj Reddy
Sysadmin @System Administrator@
Anand Ram
Developer @Senior
Programmer@
Jane Miller
Sales Manager @Sales
Manager@
```

Our goal is to remove these comments from this file. This can be done as shown below.

```
$ sed -e '/@/{N;/@.*@/{s/@.*@//;P;D}}' empnametitle-
with-comment.txt
John Doe
CEO
Jason Smith
IT Manager
Raj Reddy
Sysadmin
Anand Ram
Developer
Jane Miller
Sales Manager
```

The above command should be executed in a single line as shown below.

sed -e '/@/{N;/@.*@/{s/@.*@//;P;D}}' empnametitle-with-comment.txt

You can also save this in a sed script file and execute it as shown below.

```
$ vi D-upper.sed
#!/bin/sed -f
/@/ {
N
/@.*@/ {s/@.*@//;P;D }
}

$ chmod u+x D-upper.sed

$ ./D-upper.sed empnametitle-with-comment.txt
```

In the above example:

- **/@/ {** - This is the outer loop. Sed looks for any line that contains @ symbol. If it finds one, it executes the rest of the logic. If not, it reads the next line. For example, let us take line 4, which is "@Information Technology" (the comment spans to multiple column and goes to line 5 also). There is an @ symbol on line 4, so the rest of the commands are executed.
- **N** - Get the next line from the input file and append it to the pattern space. For example, this will read line 5 "Officer@", and append it to pattern space. So, pattern space will contain "@Information Technology\nOfficer@".
- **/@.*@/** - Searches whether pattern space has the pattern "@.*@", which means anything enclosed between @ and @. The expression is true for the current pattern space, so, it goes to the next step.
- **s/@.*@//;P;D** - This substitutes the whole text "@Information Technology\nOfficer@" with nothing (basically it deletes the text). P prints the 1st portion of the line. D deletes the rest of the content of pattern space. And the logic continues from the top again.

49. Loop and Branch (b command and :label)

You can change the execution flow of the sed commands by using label and branch (b command).

- **:label** defines the label.
- **b label** branches the execution flow to the label. Sed jumps to the line marked by the label and continues executing the rest of the commands from there.
- Note: You can also execute just the b command (without any label name). In this case, sed jumps to the end of the sed script file.

The following example combines the employee name and title (from the empnametitle.txt file) to a single line separated by : between the fields, and also adds a "*" in front of the employee name, when that employee's title contains the keyword "Manager".

```
$ vi label.sed
#!/bin/sed -nf
h;n;H;x
s/\n/:/
/Manager/!b end
s/^/*/
:end
p
```

In the above example, you already know what "h;n;H;x" and "s/\n/:/" does, as we discussed those in our previous examples. Following are the branching related lines in this file.

- /Manager/!b end - If the lines doesn't contain the keyword "Manager", it goes to the label called "end". Please note that the name of the label can be anything you want. So, this executes "s/^/*/" (add a * in the front), only for the Managers.
- :end - This is the label.

Execute the above label.sed script:

```
$ chmod u+x label.sed

$ ./label.sed empnametitle.txt
John Doe:CEO
*Jason Smith:IT Manager
Raj Reddy:Sysadmin
Anand Ram:Developer
*Jane Miller:Sales Manager
```

50. Loop Using t command

The sed command t label branches the execution flow to the label only if the previous substitute command was successful. That is, when the previous substitution was successful, sed jumps to the line marked by the label and continues executing the rest of the commands from there, otherwise it continues normal execution flow.

The following example combines the employee name and title (from the empnametitle.txt file) to a single line separated by : between the fields, and also adds three "*" in front of the employee name, when that employee's title contains the keyword "Manager".

Note: We could've just changed the substitute command in the previous example to "s/^/***/" (instead of s/^/*/) to achieve the same result. This example is given only to explain how the sed t command works.

```
$ vi label-t.sed
#!/bin/sed -nf
h;n;H;x
s/\n/:/
:repeat
/Manager/s/^/*/
/\*\*\*/!t repeat
p

$ chmod u+x label-t.sed

$ ./label-t.sed empnametitle.txt
John Doe:CEO
***Jason Smith:IT Manager
Raj Reddy:Sysadmin
Anand Ram:Developer
***Jane Miller:Sales Manager
```

In the above example:

- The following code snippet does the looping.

```
:repeat
/Manager/s/^/*/
/\*\*\*/!t repeat
```

- **/Manager/s/^/*/** - If it is Manager, it adds a single * in front of the line.
- **/***/!t repeat** - If the line doesn't contain three *s (represented by /***/!), and if the previous substitute command is successful by adding a single star in front of the line, sed jumps to the label called repeat (this is represented by t repeat)
- **:repeat** - This is just the label

Chapter 8. Awk Syntax and Basic Commands

Awk is a powerful language to manipulate and process text files. It is especially helpful when the lines in a text files are in a record format, i.e, when each line (record) contains multiple fields separated by a delimiter. Even when the input file is not in a record format, you can still use awk to do some basic file and data processing. You can also write programming logic using awk even when there are no input files that needs to be processed. In short, AWK is a powerful language that can come in handy to do daily routine jobs.

The learning curve on AWK is much smaller than the learning curve on any other language. If you know C programming already, you'll appreciate how simple and easy it is to learn AWK.

AWK was originally written by three developers -- A. Aho, B. W. Kernighan and P. Weinberger. So, the name AWK came from the initials of those three developers.

The following are the three variations of AWK:

- AWK is original AWK.
- NAWK is new AWK.
- GAWK is GNU AWK. All Linux distributions comes with GAWK. This is fully compatible with AWK and NAWK.

This book covers all the fundamentals of original AWK, and some advanced features available only in GAWK. On the systems that have either NAWK, or GAWK installed, you can still type awk, which will invoke nawk or gawk correspondingly.

For example, on Linux, you'll see that awk is a symbolic link to gawk. So, executing awk (or) gawk on Linux system will invoke gawk.

```
$ ls -l /bin/awk /bin/gawk
```

```
lrwxrwxrwx 1 root root   4 Sep 1 07:38 /bin/awk -> gawk
-rwxr-xr-x 1 root root 320416 Mar 14 2007 /bin/gawk
```

For most of the awk examples in this book, the following three sample files are used. Please create these sample files in your home directory, and use them to try out all the awk examples shown in this book.

employee.txt sample file

employee.txt is a comma delimited file that contains 5 employee records in the following format:

```
employee-number,employee-name,employee-title
```

Create the file:

```
$ vi employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

items.txt sample file

items.txt is a comma delimited text file that contains 5 item records in the following format:

```
item-number,item-description,item-category,cost,quantity-available
```

Create the file:

```
$ vi items.txt
101,HD Camcorder,Video,210,10
102,Refrigerator,Appliance,850,2
103,MP3 Player,Audio,270,15
```

```
104,Tennis Racket,Sports,190,20
105,Laser Printer,Office,475,5
```

items-sold.txt sample file

items-sold.txt is a space delimited text file that contains 5 item records. Each record is for one particular item that contains the item number followed by number of items sold for that month (during the last 6 months). So, you'll see 7 fields in every record. Field 1 is the item-number. Field 2 through Field 7 are the total number of items sold in every month during the last 6 months.

Following is the format of the items-sold.txt file.

```
item-number qty-sold-month1 qty-sold-month2 qty-sold-month3
qty-sold-month4 qty-sold-month5 qty-sold-month6
```

Create the file:

```
$ vi items-sold.txt
101 2 10 5 8 10 12
102 0 1 4 3 0 2
103 10 6 11 20 5 13
104 2 3 4 0 6 5
105 10 2 5 7 12 6
```

51. Awk Command Syntax

Basic Awk Syntax:

```
awk -Fs '/pattern/ {action}' input-file
(or)
awk -Fs '{action}' intput-file
```

In the above syntax:

- -F is the field separator. If you don't specify, it will use an empty space as field delimiter.
- The /pattern/ and the {action} should be enclosed inside single quotes.

- /pattern/ is optional. If you don't provide it, awk will process all the records from the input-file. If you specify a pattern, it will process only those records from the input-file that match the given pattern.
- {action} - These are the awk programming commands, which can be one or multiple awk commands. The whole action block (including all the awk commands together) should be closed between { and }
- input-file - The input file that needs to be processed.

Following is a very simple example demonstrating the awk syntax:

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

In the above simple example:

- -F: This indicates that the field separator in the input-file is colon :, i.e. the fields are separated by a colon. Please note that you can also enclose the field separator within double quotes. -F ":" is also valid.
- /mail/ - This is the pattern. awk will process only the records that contains the keyword mail.
- {print $1} - This is the action. This action block contains only one awk command, that prints the 1st field of the record that matches the pattern "mail"
- /etc/passwd - This is the input file.

Awk Commands in a Separate File

When you have to process a lot of awk commands, you can specify the '/pattern/ {action}' inside an awk script file and invoke it as shown below.

```
awk -Fs -f myscript.awk input-file
```

The myscript.awk can have any file extension (or no extension). But, it is easier to keep the extension as .awk for easy maintenance. You

can also specify the field separator in script file itself (more on this later), and just invoke it as shown below.

```
awk -f myscript.awk input-file
```

52. Awk Program Structure (BEGIN, body, END block)

A typical awk program has following three blocks.

1. BEGIN Block

Syntax of begin block:

```
BEGIN { awk-commands }
```

The begin block gets executed only once at the beginning, before awk starts executing the body block for all the lines in the input file.

- The begin block is a good place to print report headers, and initialize variables.
- You can have one or more awk commands in the begin block.
- The keyword BEGIN should be specified in upper case.
- Begin block is optional.

2. Body Block

Syntax of body block:

```
/pattern/ {action}
```

The body block gets executed once for every line in the input file.

- If the input file has 10 records, the commands in the body block will be executed 10 times (once for each record in the input file).
- There is no keyword for the body block. We discussed pattern and action previously.

3. END Block

Syntax of end block:

```
END { awk-commands }
```

The end block gets executed only once at the end, after awk completes executing the body block for all the lines in the input-file.

- The end block is a good place to print a report footer and do any clean-up activities.
- You can have one or more awk commands in the end block.
- The keyword END should be specified in upper case.
- End block is optional.

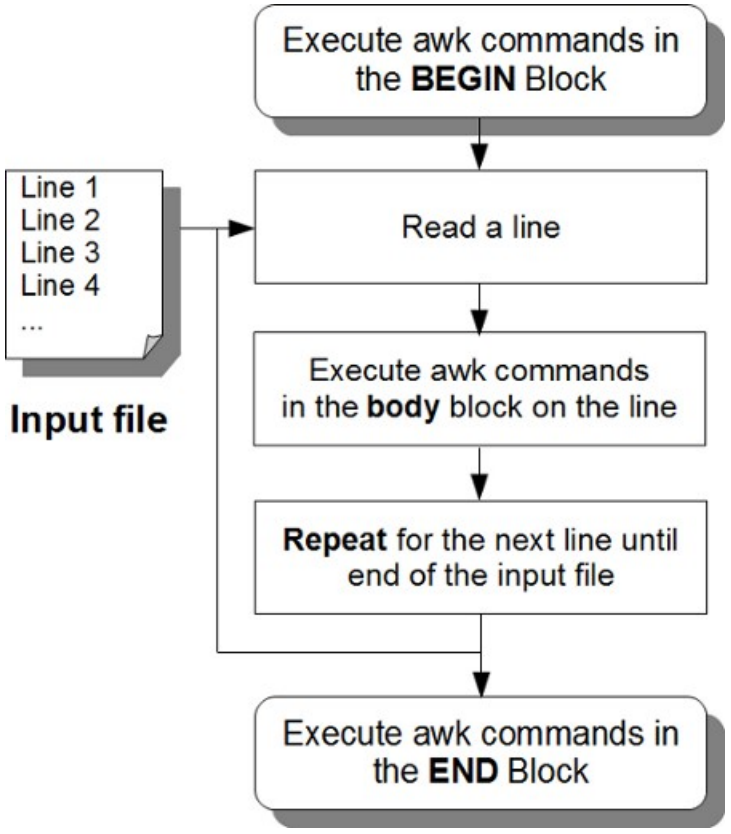

Fig: Awk Workflow

The following simple example shows the three awk blocks in action.

```
$ awk 'BEGIN { FS=":";print "---header---" } \
/mail/ {print $1} \
END { print "---footer---"}' /etc/passwd
---header---
mail
mailnull
---footer---
```

Note: When you have a very long command, you can either type is on a single line, or split it to multiple lines by specifying a \ at the end of each line. The above example is typed in 3 lines with a \ at the end of line 1 and line 2.

In the above example:

- BEGIN { FS=":";print "---header---" } is the begin block, that sets the field separator variable FS (more on this later), and prints the header. This gets executed only once before the body loop.
- /mail/ {print $1} is the body loop, that contains a pattern and an action. i.e. This searches for the keyword "mail" in the input file and prints the 1st field.
- END { print "---footer---"}' is the end block, that prints the footer.
- /etc/passwd is the input file. The body loop gets executed for every records in this file.

Instead of executing the above simple example from the command line, you can also execute it from a file.

First, create the following myscript.awk file that contains the begin, body, and end loop:

```
$ vi myscript.awk
BEGIN {
 FS=":"
```

```
 print "---header---"
}
/mail/ {
 print $1
}
END {
 print "---footer---"
}
```

Next, execute the myscript.awk as shown below for the input file /etc/passwd:

```
$ awk -f myscript.awk /etc/passwd
---header---
mail
mailnull
---footer---
```

Please note that a comment inside a awk script starts with #. If you are writing a complex awk script, follow the best practice: write enough comments inside the *.awk file so that it will be easier for you to understand when you look at the file later.

Following are some random simple examples that show you various combinations of awk blocks.

Only the body block:

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

Begin, body, and end block:

```
awk -F: 'BEGIN { printf "username\n------\n"} \
{ print $1 } \
END { print "------" }' /etc/passwd
```

Begin, and body block:

```
awk -F: 'BEGIN { print "UID"} { print $3 }' /etc/passwd
```

A Note on using only a BEGIN Block:

Specifying only the begin block is valid awk syntax. When you don't specify a body loop, there is no point in specifying a input file, since only the body loop gets executed for the lines in the input file. So, use only the BEGIN block when you want to use an awk program to do things not related to file processing. In many of our examples below, we'll have only the BEGIN block, to explain how some of the awk programming components work. You can use this idea for anything that you see fit.

A simple begin only example:

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

Multiple Input Files

Please note that you can specify multiple input files. If you specify two input files, first the body block will be executed for all the lines in input-file1, next the body block will be executed for all the lines in input-file2.

Multiple input file example:

```
$ awk 'BEGIN { FS=":";print "---header---" } \
/mail/ {print $1} \
END { print "---footer---"}' /etc/passwd /etc/group
---header---
mail
mailnull
mail
mailnull
---footer---
```

Please note that the BEGIN block and the END block will be executed only once, even when you specify multiple input-files.

53. Print Command

By default, the awk print command (without any argument) prints the full record as shown. The following example is equivalent to "cat employee.txt" command.

```
$ awk '{print}' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

You can also print specific fields in a record by passing $field-number as a print command argument. The following example is supposed to print only the employee name (field number 2) of every record.

```
$ awk '{print $2}' employee.txt
Doe,CEO
Smith,IT
Reddy,Sysadmin
Ram,Developer
Miller,Sales
```

Wait. It didn't work as expected. It printed from the last name until the end of the record. This is because the default field delimiter in Awk is space. Awk did exactly what we asked; it did print the 2nd field considering space as a delimiter. When the default space is used as delimiter, "101,John" became field-1 and "Doe,CEO" became field-2 of the 1st record. So, the above awk example printed "Doe,CEO" as field-2.

To solve this issue, we should instruct Awk to use comma (,) as field delimiter. Use option -F to indicate the field separator.

```
$ awk -F ',' '{print $2}' employee.txt
John Doe
Jason Smith
Raj Reddy
Anand Ram
Jane Miller
```

When there is only one character used for delimiter, any of the following forms works, i.e. you can specify the field delimiter within single quotes, or double quotes, or without any quotes as shown below.

```
awk -F ',' '{print $2}' employee.txt
awk -F "," '{print $2}' employee.txt
awk -F, '{print $2}' employee.txt
```

Note: You can also use the FS variable for this purpose. We'll review that in the awk built-in variables section.

A simple report that prints employee name and title with a header and footer:

```
$ awk -F ',' 'BEGIN \
{ print "-------------\nName\tTitle\n-------------"} \
{ print $2,"\t",$3;} \
END { print "-------------"; }' employee.txt
-------------
Name  Title
-------------
John Doe     CEO
Jason Smith   IT Manager
Raj Reddy    Sysadmin
Anand Ram    Developer
Jane Miller   Sales Manager
-------------
```

In the above report the fields are not aligned properly. We'll look at how to do that in later sections. The above example does show how you can use BEGIN to print a header, and END to print a footer.

Please note that field $0 represents the whole record. Both of the following examples are the same; each prints the whole lines from employee.txt.

```
awk '{print}' employee.txt
awk '{print $0}' employee.txt
```

54. Pattern Matching

You can execute awk commands only for lines that match a particular pattern.

For example, the following prints the names and titles of the Managers:

```
$ awk -F ',' '/Manager/ {print $2, $3}' employee.txt
Jason Smith IT Manager
Jane Miller Sales Manager
```

The following example prints the employee name whose Emp id is 102:

```
$ awk -F ',' '/^102/ {print "Emp id 102 is", $2}' \
employee.txt
Emp id 102 is Jason Smith
```

Chapter 9. Awk Built-in Variables

55. FS - Input Field Separator

The default field separator recognized by awk is space. If the records in your input file are delimited by anything other than space, you already know that you can specify the input field separator in the awk command line using option -F as shown below.

```
awk -F ',' '{print $2, $3}' employee.txt
```

You can also do the same using the FS (field separator) Awk built-in variable. You have to specify the FS in the BEGIN block as shown below.

```
awk 'BEGIN {FS=","} {print $2, $3}' employee.txt
```

You can have multiple awk statements in the BEGIN block. In the following example, we have both FS and a print command to print the headers inside the BEGIN block. Multiple commands inside the BEGIN or END block are separated by semi-colon.

```
awk 'BEGIN { FS=","; \
print "-------------\nName\tTitle\n-------------" } \
{ print $2,"\t",$3; } \
END {print "-------------"}' employee.txt
```

Please note that the default field separator is not just a single space. It actually matches one or more whitespace characters.

The following employee-multiple-fs.txt file contains three different field separators in each record:

- , Comma is the field separator after emp id
- : Colon is the field separator after name
- % Percentage is the field separator after title

Create the file:

```
$ vi employee-multiple-fs.txt
101,John Doe:CEO%10000
102,Jason Smith:IT Manager%5000
103,Raj Reddy:Sysadmin%4500
104,Anand Ram:Developer%4500
105,Jane Miller:Sales Manager%3000
```

When you encounter a file that contains different field separators, don't worry, FS can come to your rescue. You can specify MULTIPLE field separators using a regular expression. For example FS = "[,:%]" indicates that the field separator can be , or : or %

So, the following example will print the name and the title from the employee-multiple-fs.txt file that contains different field separators.

```
$ awk 'BEGIN {FS="[,:%]"} {print $2, $3}' \
employee-multiple-fs.txt
John Doe CEO
Jason Smith IT Manager
Raj Reddy Sysadmin
Anand Ram Developer
Jane Miller Sales Manager
```

56. OFS - Output Field Separator

FS is for input field separator. OFS is for output field separator. OFS is printed between consecutive fields in the output. By default, awk prints the output fields with space between the fields.

Please note that we don't specify IFS for input field separator, we simply refer to it as FS.

The following example prints the name and the salary with space between them. When you use a single print statement to print two

variables by separating them with comma (as shown below), it will print the values of those two variables separated by space.

```
$ awk -F ',' '{print $2, $3}' employee.txt
John Doe CEO
Jason Smith IT Manager
Raj Reddy Sysadmin
Anand Ram Developer
Jane Miller Sales Manager
```

If you try to include a colon manually in the print statement between the fields, following will the output. Please note how there is an additional space before and after the colon. That is because, awk is still using space as the output field separator.

The following print statement really printing three values (that are separated by comma) -- $2, :, and $4. As you already know when you use one print statement to print multiple values, the output will contain space in between them.

```
$ awk -F ',' '{print $2, ":", $3}' employee.txt
John Doe : CEO
Jason Smith : IT Manager
Raj Reddy : Sysadmin
Anand Ram : Developer
Jane Miller : Sales Manager
```

The right way to do is use the awk built-in variable OFS (output field separator), as shown below. Please note that there is no space before and after the colon in this example, as OFS replaces the default awk OFS (which is space) with the colon.

The following print statement is printing two variables ($2 and $4) separated by comma, however the output will have colon separating them (instead of space), as our OFS is set to colon.

```
$ awk -F ',' 'BEGIN { OFS=":" } \
{ print $2, $3 }' employee.txt
```

```
John Doe:CEO
Jason Smith:IT Manager
Raj Reddy:Sysadmin
Anand Ram:Developer
Jane Miller:Sales Manager
```

Please also note the subtle difference between including a comma vs not including a comma in the print statement (when printing multiple variables). When you specify a comma in the print statement between different print values, awk will use the OFS. In the following example, the default OFS is used, so you'll see a space between the values in the output.

```
$ awk 'BEGIN { print "test1","test2" }'
test1 test2
```

When you don't separate values with a comma in the print statement, awk will not use the OFS; instead it will print the values with nothing in between.

```
$ awk 'BEGIN { print "test1" "test2" }'
test1test2
```

57. RS - Record Separator

Let us assume that you have the following text file which contains the employee ids and names in a single line.

```
$ vi employee-one-line.txt
101,John Doe:102,Jason Smith:103,Raj Reddy:104,Anand
Ram:105,Jane Miller
```

In the above example, every record contains two fields (empid and name), and every record is separated by : (instead of a new line). The individual fields (empid and name) in the records are separated by comma.

The default record separator used by awk is new line. If you are trying to print only the employee name, the following will not work for this example.

```
$ awk -F, '{print $2}' employee-one-line.txt
John Doe:102
```

In the above example, it is treating employee-one-line.txt as one single record, and comma as field delimiter. So, it prints "John Doe:102", as the 2nd field.

If you want awk to treat this as 5 different lines (instead of a single line), and print employee name from each record, then you must specify the record separator as colon : as shown below.

```
$ awk -F, 'BEGIN { RS=":" } \
{ print $2 }' employee-one-line.txt
John Doe
Jason Smith
Raj Reddy
Anand Ram
Jane Miller
```

Let us assume that you have the following input file, where the records are separated by a "-" on it's own line. All the fields are on a separate line.

```
$ vi employee-change-fs-ofs.txt
101
John Doe
CEO
-
102
Jason Smith
IT Manager
-
```

```
103
Raj Reddy
Sysadmin
-
104
Anand Ram
Developer
-
105
Jane Miller
Sales Manager
```

In the above example, the field separator FS is new line, the record separator RS is "-" followed by a new line. So, if you want to print employee name and salary, you should do the following.

```
$ awk 'BEGIN { FS="\n"; RS="-\n"; OFS=":" } \
{print $2, $3}' employee-change-fs-ofs.txt
John Doe:CEO
Jason Smith:IT Manager
Raj Reddy:Sysadmin
Anand Ram:Developer
Jane Miller:Sales Manager
```

58. ORS - Output Record Separator

RS is for input record separator. ORS is for output record separator.

Please note that we don't specify IRS for input record separator, we simply refer to it as RS.

The following example adds a new line with "---" after each and every line output that is printed. By default, awk uses "\n" as ORS. In this example, we are using "\n---\n" as ORS to get the output as shown below.

```
$ awk 'BEGIN { FS=","; ORS="\n---\n" } \
{print $2, $3}' employee.txt
John Doe CEO
---
Jason Smith IT Manager
---
Raj Reddy Sysadmin
---
Anand Ram Developer
---
Jane Miller Sales Manager
---
```

The following example takes the records in employee.txt, and prints every field in its own line, separating each record with a separate line with "---".

```
$ awk 'BEGIN { FS=","; OFS="\n";ORS="\n---\n" } \
{print $1,$2,$3}' employee.txt
101
John Doe
CEO
---
102
Jason Smith
IT Manager
---
103
Raj Reddy
Sysadmin
---
104
Anand Ram
Developer
```

```
---
105
Jane Miller
Sales Manager
---
```

59. NR - Number of Records

NR is very helpful. When used inside the loop, this gives the line number. When used in the END block, this gives the total number of records in the file.

Even thought NR stands for "Number of Records", it might be appropriate to call this as "Number of the Record", as it really gives you the line number of the current record.

The following example shows how NR works in the body block, and in the END block:

```
$ awk 'BEGIN {FS=","} \
{print "Emp Id of record number",NR,"is",$1;} \
END {print "Total number of records:",NR}' employee.txt
Emp Id of record number 1 is 101
Emp Id of record number 2 is 102
Emp Id of record number 3 is 103
Emp Id of record number 4 is 104
Emp Id of record number 5 is 105
Total number of records: 5
```

60. FILENAME – Current File Name

FILENAME is helpful when you are specifying multiple input-files to the awk program. This will give you the name of the file Awk is currently processing.

```
$ awk '{ print FILENAME }' \
employee.txt employee-multiple-fs.txt
employee.txt
employee.txt
employee.txt
employee.txt
employee.txt
employee-multiple-fs.txt
employee-multiple-fs.txt
employee-multiple-fs.txt
employee-multiple-fs.txt
employee-multiple-fs.txt
```

When you read the values from the standard input, FILENAME variable will be set to the value of "-" as shown below. In the following example, since we didn't give any input-file, you should type the record in the standard input.

In this example, I typed the 1st line "John Doe", and awk printed the last two lines. You have to press “Ctrl-C” to stop reading from stdin.

```
$ awk '{print "Last name:", $2; \
print "Filename:", FILENAME}'
John Doe
Last name: Doe
Filename: -
```

The above is also true when you pipe the input to awk from another program, as shown below. The following also will print FILENAME as "-".

```
$ echo "John Doe" | awk '{print "Last name:", $2; \
print "Filename:", FILENAME}'
Last name: Doe
Filename: -
```

Note: FILENAME inside the BEGIN block will return empty value "", as the BEGIN block is for the whole awk program, and not for any specific file.

61. FNR - File "Number of Record"

We already know that "NR" is "Number of Records" (or "Number of the Record"), which prints the current line number of the file that is getting processed.

How will NR behave when we give have two input files? NR keeps growing between multiple files. When the body block starts processing the 2nd file, NR will not be reset to 1, instead it will continue from the last NR number value of the previous file.

In the following example 1st file has 5 records, 2nd file has 5 records. As you see below, when the body loop is processing the 2nd file, NR starts from 6 (instead of 1). Finally, in the END block, NR gives the total number of records of both the files combined.

```
$ awk 'BEGIN {FS=","} \
{print FILENAME ": record number",NR,"is",$1;} \
END {print "Total number of records:",NR}' \
employee.txt employee-multiple-fs.txt
employee.txt: record number 1 is 101
employee.txt: record number 2 is 102
employee.txt: record number 3 is 103
employee.txt: record number 4 is 104
employee.txt: record number 5 is 105
employee-multiple-fs.txt: record number 6 is 101
employee-multiple-fs.txt: record number 7 is 102
employee-multiple-fs.txt: record number 8 is 103
employee-multiple-fs.txt: record number 9 is 104
employee-multiple-fs.txt: record number 10 is 105
Total number of records: 10
```

In the above example, we have two input files (employee.txt and employee-multiple-fs.txt). Each file has 5 records each. So, NR continued incrementing after the 1st file is processed.

FNR will give you record number within the current file. So, when awk finishes executing the body block for the 1st file and starts the body block the next file, FNR will start from 1 again.

```
$ awk 'BEGIN {FS=","} \
{print FILENAME ": record number",FNR,"is",$1;} \
END {print "Total number of records:",NR}' \
employee.txt employee-multiple-fs.txt
employee.txt: record number 1 is 101
employee.txt: record number 2 is 102
employee.txt: record number 3 is 103
employee.txt: record number 4 is 104
employee.txt: record number 5 is 105
employee-multiple-fs.txt: record number 1 is 101
employee-multiple-fs.txt: record number 2 is 102
employee-multiple-fs.txt: record number 3 is 103
employee-multiple-fs.txt: record number 4 is 104
employee-multiple-fs.txt: record number 5 is 105
Total number of records: 10
```

The following example shows both NR and FNR:

```
$ vi fnr.awk
BEGIN {
 FS=","
}
{
 printf "FILENAME=%s NR=%s FNR=%s\n", FILENAME, NR,
FNR;
}
END {
```

```
 printf "END Block: NR=%s FNR=%s\n", NR, FNR
}

$ awk -f fnr.awk employee.txt employee-multiple-fs.txt
FILENAME=employee.txt NR=1 FNR=1
FILENAME=employee.txt NR=2 FNR=2
FILENAME=employee.txt NR=3 FNR=3
FILENAME=employee.txt NR=4 FNR=4
FILENAME=employee.txt NR=5 FNR=5
FILENAME=employee-multiple-fs.txt NR=6 FNR=1
FILENAME=employee-multiple-fs.txt NR=7 FNR=2
FILENAME=employee-multiple-fs.txt NR=8 FNR=3
FILENAME=employee-multiple-fs.txt NR=9 FNR=4
FILENAME=employee-multiple-fs.txt NR=10 FNR=5
END Block: NR=10 FNR=5
```

Chapter 10. Awk Variables and Operators

62. Variables

Awk variables should begin with an alphabetic character; the rest of the characters can be numbers, or letters, or underscore. Keywords cannot be used as an awk variable name.

Unlike other programming languages, you don't need to declare an variable to use it. If you wish to initialize an awk variable, it is better to do it in the BEGIN section, which will be executed only once.

There are no data types in Awk. Whether an awk variable is a number or a string depends on the context in which the variable is used in.

employee-sal.txt sample file

employee-sal.txt is a comma delimited file that contains 5 employee records in the following format:

```
employee-number,employee-name,employee-title,salary
```

Create the file:

```
$ vi employee-sal.txt
101,John Doe,CEO,10000
102,Jason Smith,IT Manager,5000
103,Raj Reddy,Sysadmin,4500
104,Anand Ram,Developer,4500
105,Jane Miller,Sales Manager,3000
```

The following example shows how to create and use your own variable inside an awk script. In this example, "total" is the user defined Awk variable that is used to calculate the total salary of all the employees in the company.

```
$ cat total-company-salary.awk
BEGIN {
 FS=",";
 total=0;
}
{
 print $2 "'s salary is: " $4;
 total=total+$4
}
END {
 print "---\nTotal company salary = $"total;
}

$ awk -f total-company-salary.awk employee-sal.txt
John Doe's salary is: 10000
Jason Smith's salary is: 5000
Raj Reddy's salary is: 4500
Anand Ram's salary is: 4500
Jane Miller's salary is: 3000
---
Total company salary = $27000
```

63. Unary Operators

An operator which accepts a single operand is called a unary operator.

Operator	Description
+	The number (returns the number itself)
-	Negate the number
++	Auto Increment
--	Auto Decrement

The following example negates the number using unary operator minus:

```
$ awk -F, '{print -$4}' employee-sal.txt
-10000
-5000
-4500
-4500
-3000
```

The following example demonstrates how plus and minus unary operators affect negative numbers stored in a text file:

```
$ vi negative.txt
-1
-2
-3

$ awk '{print +$1}' negative.txt
-1
-2
-3

$ awk '{print -$1}' negative.txt
1
2
3
```

Auto Increment and Auto Decrement

Auto increment and auto decrement operators change the associated variable's value; when used inside an expression their interpreted value can be either 'pre' or 'post' the change of value.

Pre means you'll add ++ (or --) *before* the variable name. This will first increase (or decrease) the value of the variable by one, and then execute the rest of the statement in which it is used.

Post means you'll add ++ (or --) *after* the variable name. This will first execute the containing statement and then increase (or decrease) the value of the variable by one.

Example of pre-auto-increment:

```
$ awk -F, '{print ++$4}' employee-sal.txt
10001
5001
4501
4501
3001
```

Example of pre-auto-decrement:

```
$ awk -F, '{print --$4}' employee-sal.txt
9999
4999
4499
4499
2999
```

Example of post-auto-increment:
(since ++ is in the print statement the original value is printed):

```
$ awk -F ',' '{print $4++}' employee-sal.txt
10000
5000
4500
4500
3000
```

Example of post-auto-increment:
(since ++ is in a separate statement the resulting value is printed):

```
$ awk -F ',' '{$4++; print $4}' employee-sal.txt
10001
5001
4501
4501
3001
```

Example of post-auto-decrement:
(since -- is in the print statement the original value is printed):

```
$ awk -F ',' '{print $4--}' employee-sal.txt
10000
5000
4500
4500
3000
```

Example of post-auto-decrement:
(since -- is in a separate statement the resulting value is printed):

```
$ awk -F ',' '{$4--; print $4}' employee-sal.txt
9999
4999
4499
4499
2999
```

The following useful example displays the total number of users who have a login shell, i.e. who can log in to the system and reach a command prompt.

- This uses the post-increment unary operator (although since the variable is not printed till the END block pre-increment would produce the same result).

- The body block of this script includes a pattern match so that the contained code executes only if the last field of the line contains the pattern /bin/bash.
- Note: Regular expressions should be enclosed between // but that means that the frontslash (/) character must be escaped in the regular expression so that it is not interpreted as the end-of-expression.
- When a line matches, variable 'n' gets incremented by one. The final value is printed from the END block.

Example: Print number of shell users.

```
$ awk -F ':' '$NF ~ /\/bin\/bash/ { n++ }; END { print
n }' /etc/passwd
2
```

64. Arithmetic Operators

An operator that accepts two operands is called a binary operator. There are different kinds of binary operators that are classified based on usage. (arithmetic, string, assignment, etc.)

The following operators are used for performing arithmetic calculations.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Division

The following example shows the usage of the binary operators +, -, * and /

This examples does two things:

1. Reduces the price of every single item by 20%
2. Reduces the quantity of every single item by 1.

Create and run awk arithmetic example:

```
$ vi arithmetic.awk
BEGIN {
 FS=",";
 OFS=",";
 item_discount=0;
}
{
 item_discount=$4*20/100;
 print $1,$2,$3,$4-item_discount,$5-1
}

$ awk -f arithmetic.awk items.txt
101,HD Camcorder,Video,168,9
102,Refrigerator,Appliance,680,1
103,MP3 Player,Audio,216,14
104,Tennis Racket,Sports,152,19
105,Laser Printer,Office,380,4
```

The following example prints all the even numbered lines from the input file. The row number of each line is checked to see if it is a multiple of 2, and if so the default operation (print the whole line) is executed.

Demonstrate modulo division:

```
$ awk 'NR % 2 == 0' items.txt
102,Refrigerator,Appliance,850,2
104,Tennis Racket,Sports,190,20
```

65. String Operator

(space) is a string operator that does string concatenation.

In the following example, string concatenation happens at three locations. In the statement "string3=string1 string2", string3 contains the concatenated value of string1 and string2. Each print statement does a string concatenation with a static string and an awk variable.

Note: This operator is why you must separate the values in a print statement with a comma if you want to print the OFS in between. If you do not include a comma to separate the values, the values are concatenated instead.

```
$ cat string.awk
BEGIN {
 FS=",";
 OFS=",";
 string1="Audio";
 string2="Video";
 numberstring="100";
 string3=string1 string2;
 print "Concatenate string is:" string3;
 numberstring=numberstring+1;
 print "String to number:" numberstring;
}

$ awk -f string.awk items.txt
Concatenate string is:AudioVideo
String to number:101
```

66. Assignment Operators

Just like most other programming languages, awk uses = as the assignment operator. Like C, awk also supports shortcut assignment operators that modify a variable rather than replacing its value.

Operator	Description
=	Assignment
+=	Shortcut addition assignment

-=	Shortcut subtraction assignment
*=	Shortcut multiplication assignment
/=	Shortcut division assignment
%=	Shortcut modulo division assignment

The following example shows how to use the assignment operators:

```
$ cat assignment.awk
BEGIN {
 FS=",";
 OFS=",";
 total1 = total2 = total3 = total4 = total5 = 10;
 total1 += 5; print total1;
 total2 -= 5; print total2;
 total3 *= 5; print total3;
 total4 /= 5; print total4;
 total5 %= 5; print total5;
}

$ awk -f assignment.awk
15
5
50
2
0
```

The following example uses the += shortcut assignment operator.

Display the total amount of inventory available across all items:

```
$ awk -F ',' 'BEGIN { total=0 } { total+=$5 } END
{print "Total Quantity: " total}' items.txt
Total Quantity: 52
```

The next example counts the total number of fields in a file. The awk script matches all lines and keeps adding the number of fields in each line using the shortcut addition assignment operator. The number of fields seen so far is kept in a variable named 'total'. Once the input file is processed, the END block is executed, which prints the total number of fields.

Count total number of fields in items.txt:

```
$ awk -F ',' 'BEGIN { total=0 } { total += NF }; END
{ print total }' items.txt
25
```

67. Comparison Operators

Awk supports the standard comparison operators that are listed below.

Operator	Description
>	Is greater than
>=	Is greater than or equal to
<	Is less than
<=	Is less than or equal to
==	Is equal to
!=	Is not equal to
&&	Both the conditional expressions are true
||	Either one of the conditional expressions is true

A note on the following examples: If you don't specify any action, awk will print the whole record if it matches the conditional comparison.

The following example uses <= condition. This displays all the items that are under the critical inventory level of 5:

```
$ awk -F "," '$5 <= 5' items.txt
102,Refrigerator,Appliance,850,2
105,Laser Printer,Office,475,5
```

The following example uses == condition. This displays the record with the item number 103:

```
$ awk -F "," '$1 == 103' items.txt
103,MP3 Player,Audio,270,15
```

Note: don't confuse the == (exact match) operator with = (Assignment).

Print only the description of the item with number 103:

```
$ awk -F "," '$1 == 103 {print $2}' items.txt
MP3 Player
```

The following example uses != condition. This prints all items except those in the category Video:

```
$ awk -F "," '$3 != "Video"' items.txt
102,Refrigerator,Appliance,850,2
103,MP3 Player,Audio,270,15
104,Tennis Racket,Sports,190,20
105,Laser Printer,Office,475,5
```

Same as above, but prints only the item description:

```
$ awk -F "," '$3 != "Video" {print $2}' items.txt
Refrigerator
MP3 Player
Tennis Racket
Laser Printer
```

The following example uses && (AND operator) to check two conditions. This prints the record where the cost is under 900 AND the quantity is less than or equal to the critical inventory level of 5.

```
$ awk -F "," '$4 < 900 && $5 <= 5' items.txt
102,Refrigerator,Appliance,850,2
105,Laser Printer,Office,475,5
```

Same as above, but prints only the item description:

```
$ awk -F "," '$4 < 900 && $5 <= 5 {print $2}' items.txt
Refrigerator
Laser Printer
```

The following example uses || (OR operator) to check two conditions. This prints records where the cost is less than 900 OR the quantity is at or under the critical inventory level of 5.

```
$ awk -F "," '$4 < 900 || $5 <= 5' items.txt
101,HD Camcorder,Video,210,10
102,Refrigerator,Appliance,850,2
103,MP3 Player,Audio,270,15
104,Tennis Racket,Sports,190,20
105,Laser Printer,Office,475,5
```

Same as above. But prints only the item description:

```
$ awk -F "," '$4 < 900 || $5 <= 5 {print $2}' items.txt
HD Camcorder
Refrigerator
MP3 Player
Tennis Racket
Laser Printer
```

The following example uses > (Greater than) condition. This example displays the uid (and the full line) from the /etc/passwd that has the highest USER ID value. This awk script keeps track of the largest number (of field3) in the variable 'maxuid' and keeps a copy of the corresponding line in the variable 'maxline'. Once it has looped over all the lines, it prints the uid and the line.

```
$ awk -F ':' '$3 > maxuid { maxuid=$3; maxline=$0 }; \
 END { print maxuid, maxline }' /etc/passwd
112 gdm:x:112:119:Gnome Display
Manager:/var/lib/gdm:/bin/false
```

The following example uses == condition. This example prints every line from the /etc/passwd file that has the same USER ID and GROUP ID. This awk script prints the line only if $3 (USER ID) and $4 (GROUP ID) are equal.

```
$ awk -F ':' '$3==$4' /etc/passwd
gnats:x:41:41:Gnats Bug-Reporting System
(admin):/var/lib/gnats:/bin/sh
```

The following example uses >= and && conditions. This example prints any line from /etc/passwd where the USER ID >= 100 AND the user's shell is /bin/sh.

```
$ awk -F ':' '$3>=100 && $NF ~ /\/bin\/sh/' /etc/passwd
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
```

The following example uses == condition. This example prints all the lines from /etc/passwd that doesn't have a comment (field 5).

```
$ awk -F ':' '$5 == "" ' /etc/passwd
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
syslog:x:101:102::/home/syslog:/bin/false
saned:x:110:116::/home/saned:/bin/false
```

68. Regular Expression Operators

Operator	Description
~	Match operator
!~	No Match operator

When you use the == condition, awk looks for a full match. The following example doesn't print anything, as none of the 2nd fields in the items.txt file exactly matches the keyword "Tennis". "Tennis Racket" is not a full match.

Print lines where field two *is* “Tennis”:

```
awk -F "," '$2 == "Tennis"' items.txt
```

When you use the match operator ~, awk looks for a partial match, i.e. it looks for a field that “contains” the match string.

Print lines where field two *contains* “Tennis”:

```
$ awk -F "," '$2 ~ "Tennis"' items.txt
104,Tennis Racket,Sports,190,20
```

The !~ operator is the opposite of ~, i.e. “does not contain”.

Print lines where field two does not contain “Tennis”:

```
$ awk -F "," '$2 !~ "Tennis"' items.txt
101,HD Camcorder,Video,210,10
102,Refrigerator,Appliance,850,2
103,MP3 Player,Audio,270,15
105,Laser Printer,Office,475,5
```

The next example prints the total number of users who use /bin/bash as their shell. In this awk script, when the last field of a line contains the pattern "/bin/bash", the awk variable ‘n’ gets incremented by one.

```
$ awk -F ':' '$NF ~ /\/bin\/sh/ { n++ }; END { print
n }' /etc/passwd
2
```

Chapter 11. Awk Conditional Statements and Loops

Awk supports conditional statements to control the flow of the program. Most of the Awk conditional statement syntax is similar to the 'C' programming language conditional statements.

Awk supports the following three kinds of if statements.

- Awk Simple If statement
- Awk If-Else statement
- Awk If-ElseIf-Ladder

69. Simple If Statement

The simple if statement tests a condition, and if the condition returns true, performs the corresponding action(s).

Single Action

Syntax:

```
if (conditional-expression)
    action
```

- if is a keyword
- conditional-expression represents the condition to be tested
- action is an awk statement to perform

Multiple Actions

If more than one action needs to be performed when the condition is true, those actions should be enclosed in curly braces. The individual actions (awk statements) should be separated by new line or semicolon as shown below.

Syntax:

```
if (conditional-expression)
{
    action1;
    action2;
}
```

If the condition is true, all the actions enclosed in braces will be performed in the given order. After all the actions are performed, awk continues to execute the next statement.

Print all the items with quantity <=5:

```
$ awk -F "," '{ if ($5 <= 5) \
print "Only",$5,"qty of",$2, "is available"; }' \
items.txt
Only 2 qty of Refrigerator is available
Only 5 qty of Laser Printer is available
```

You can also have multiple conditional operators in an if statement as shown below. This example prints all the items with price between 500 and 1000, and the total quantity <= 5

```
$ awk -F "," \
'{ if ( ($4 >= 500 && $4 <= 1000) && ($5 <= 5)) \
print "Only",$5,"qty of",$2,"is available";}' items.txt
Only 2 qty of Refrigerator is available
```

70. If Else Statement

In the awk "If Else" statement you can also provide list of actions to perform if the condition is false. In the following syntax, if the condition is true action1 will be performed, if the condition is false action 2 will be performed.

Syntax:

```
if (conditional-expression)
```

```
    action1
else
    action2
```

Awk also has a conditional operator, the 'ternary operator' (?:) which works like the equivalent one in C.

Just like in the if-else statement, if the conditional-expression is true action1 will be performed, and if the conditional-expression is false action2 will be performed.

Ternary Operator Syntax:

```
conditional-expression ? action1 : action2 ;
```

The following example displays the message "Buy More" when the total quantity is <= 5, and prints "Sell More" when the total quantity is not <=5.

```
$ cat if-else.awk
BEGIN {
 FS=",";
}
{
 if ( $5 <= 5 )
  print "Buy More: Order", $2, "immediately!"
 else
  print "Sell More: Give discount on", $2,
"immediately!"
}

$ awk -f if-else.awk items.txt
Sell More: Give discount on HD Camcorder immediately!
Buy More: Order Refrigerator immediately!
Sell More: Give discount on MP3 Player immediately!
Sell More: Give discount on Tennis Racket immediately!
```

```
Buy More: Order Laser Printer immediately!
```

The following example uses the ternary operator to concatenate every 2 lines from the items.txt file, with a comma in between.

We discussed the awk ORS (output record separator) built-in variable earlier. In this example, the value of ORS is changed back and forth between comma and newline. When the line number modulo 2 (NR %2) produces a remainder (i.e. for odd lines) ORS is set to comma; otherwise it's a newline. So, lines 1 and 2 combine and print as a single line, lines 3 and 4 combine and print as a single line, and line 5 prints by itself, with a comma and no newline character.

Print concatenated pairs of records:

```
$ awk 'ORS=NR%2?",":"\n"' items.txt
101,HD Camcorder,Video,210,10,102,Refrigerator,Appliance,850,2
103,MP3 Player,Audio,270,15,104,Tennis Racket,Sports,190,20
105,Laser Printer,Office,475,5,
```

71. While Loop

Awk looping statements are used to perform a set of actions again and again in succession. Awk keeps executing a statement as long as the loop condition is true. Just like a C program, awk supports various looping statements.

First, let us look at the While loop statement.

Syntax:

```
while(condition)
    actions
```

- while is awk keyword.
- condition is conditional expression.
- actions are the body of the while loop. If there are more than one action, the actions must be enclosed within curly braces.

The awk while loop checks the condition first; if the condition is true, it executes the actions. After executing all the actions, the condition is checked again, and if it is true, the actions are performed again. This process is repeated until the condition becomes false.

Please note that if the condition returns false in the first iteration, the actions are never executed.

The example below uses the BEGIN block that gets executed before anything else in an Awk program. The awk while loop appends the character 'x' to the variable 'string' 50 times. The variable count is post-incremented each time it is checked, and the actions are performed if it was less than 50 before being incremented. So the loop executes exactly 50 times. After the loop, the value of the 'string' variable is printed.

```
$ awk 'BEGIN \
{ while (count++<50) string=string "x"; print string }'
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The following awk program prints the total number of items sold from the items-sold.txt file for each item.

For each line, the program has to add the values of field 2 through field 7. (Field 1 is the item number so its value is not added to the total). So, the while condition starts from 2nd field (as i=2 before while), and checks whether it has reached the last field in the record (i <= NF). N represents the total number of fields in the record.

```
$ cat while.awk
{
 i=2; total=0;
 while (i <= NF) {
 total = total + $i;
 i++;
 }
```

```
 print "Item", $1, ":", total, "quantities sold";
}

$ awk -f while.awk items-sold.txt
Item 101 : 47 quantities sold
Item 102 : 10 quantities sold
Item 103 : 65 quantities sold
Item 104 : 20 quantities sold
Item 105 : 42 quantities sold
```

72. Do-While Loop

The awk while loop is an entry-controlled loop, as the condition is checked at the entry. The do-while loop is an exit-controlled loop; the condition is checked at exit. The do-while loop always executes at least once; it repeats as long as the condition is true.

Syntax:

```
do
action
while(condition)
```

In the example below, the print statement is executed exactly once because we ensure that the condition will be false. If this were a while statement, with the same initialization and condition, the actions would not be executed at all.

```
$ awk 'BEGIN{
count=1;
do
print "This gets printed at least once";
while(count!=1)
}'
This gets printed at least once
```

The following awk program prints the total number of quantities sold from the items-sold.txt file for each item. The output of this program is exactly the same as the while.awk program, but this uses do-while.

```
$ cat dowhile.awk
{
 i=2; total=0;
 do
 {
 total = total + $i;
 i++;
 } while (i <= NF)
 print "Item", $1, ":", total, "quantities sold";
}

$ awk -f dowhile.awk items-sold.txt
Item 101 : 47 quantities sold
Item 102 : 10 quantities sold
Item 103 : 65 quantities sold
Item 104 : 20 quantities sold
Item 105 : 42 quantities sold
```

73. For Loop Statement

The awk for statement is functionally the same as the awk while loop, but the for statement syntax is much easier to use.

Syntax:

```
for(initialization;condition;increment/decrement)
actions
```

The awk for statement starts by executing initialization, then checks the condition; if the condition is true, it executes the actions, then does the increment or decrement. As long as the condition is true, awk repeatedly executes the action and then the increment/decrement.

The following example prints the sum of fields in a line. Initially the variable i is initialized to 1; if i is less than or equal to the total number of fields, the current field is added to the total; I is incremented and the test is repeated.

```
$ echo "1 2 3 4" | awk \
'{ for (i = 1; i <= NF; i++) total = total+$i }; \
END { print total }'
10
```

The following example prints all the fields in the file in the reverse order using a for loop. Please note that this uses decrement rather than increment in the for loop.

Note: After reading in each line, Awk sets the NF variable to the number of fields found on that line.

This example loops in reverse order starting from NF to 1 and outputs the fields one by one. It starts with field $NF, then $(NF-1),..., $1. After that it prints a newline character.

Reverse For Example:

```
$ cat forreverse.awk
BEGIN {
 ORS="";
}
{
 for (i=NF; i >0; i--)
 print $i," "
 print "\n";
}

$ awk -f forreverse.awk items-sold.txt
12 10 8 5 10 2 101
2 0 3 4 1 0 102
13 5 20 11 6 10 103
5 6 0 4 3 2 104
```

```
6 12 7 5 2 10 105
```

Now we will present the for-loop version of the program we used to print the total quantity sold for each item in the items-sold.txt file. We previously showed a while-loop and do-while-loop version.

```
$ cat for.awk
{
 total=0;
 for (i=2; i <= NF; i++)
 total = total + $i;
 print "Item", $1, ":", total, "quantities sold";
}

$ awk -f for.awk items-sold.txt
Item 101 : 47 quantities sold
Item 102 : 10 quantities sold
Item 103 : 65 quantities sold
Item 104 : 20 quantities sold
Item 105 : 42 quantities sold
```

74. Break Statement

The break statement is used for jumping out of the innermost loop (while, do-while, or for loop) that encloses it. Please note that the break statement has meaning only if you use it with in the loop.

The following example prints any item number that has a month with no sold items, i.e. that has 0 for any one of the values field2 through field7.

```
$ cat break.awk
{
 i=2; total=0;
 while (i++ <= NF)
```

```
 if ($i == 0) {
  print "Item", $1, "had a month with no item sold"
  break;
 }
}

$ awk -f break.awk items-sold.txt
Item 102 had a month with no item sold
Item 104 had a month with no item sold
```

If you execute the following command, press Ctrl-C to stop the script and break out of it.

```
$ awk 'BEGIN{while(1) print "forever"}'
```

The above awk while loop prints the string “forever” forever, because the condition never fails. Usually this is not a good thing—although forever loops are used in process control or operating system applications!

Let us modify the loop so that it executes exactly ten times and is terminated by a break statement.

```
$ awk 'BEGIN{
x=1;
while(1)
{
print "Iteration";
if ( x==10 )
break;
x++;
}}'
```

The above command produces the following output:

```
Iteration
```

```
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
```

75. Continue Statement

The continue statement skips over the rest of the loop body causing the next cycle around the loop to begin immediately. Please note that the continue statement has meaning only if you use it with in the loop.

The following awk program prints the total number of quantities sold from the items-sold.txt file for each item. The output of this program is exactly same as the while.awk, dowhile.awk, and for.awk program, but this uses the while loop with continue instead of starting the loop at 2.

```
$ cat continue.awk
{
 i=1;
 total=0;
 while (i++ <= NF) {
 if (i == 1) continue;
 total = total + $i;
 }
 print "Item", $1, ":", total, "quantities sold";
}

$ awk -f continue.awk items-sold.txt
```

```
Item 101 : 47 quantities sold
Item 102 : 10 quantities sold
Item 103 : 65 quantities sold
Item 104 : 20 quantities sold
Item 105 : 42 quantities sold
```

The following awk script prints the value of x at each iteration except the 5th, where a continue statement skips the printing.

```
$ awk 'BEGIN{
x=1;
while(x<=10)
{
if(x==5){
x++;
continue;
}
print "Value of x",x;x++;
}
}'
```

The above command produces the following output.

```
Value of x 1
Value of x 2
Value of x 3
Value of x 4
Value of x 6
Value of x 7
Value of x 8
Value of x 9
Value of x 10
```

76. Exit Statement

The exit statement causes the script to immediately stop executing the current commands, and also ignores the remaining lines from the input file.

Exit accepts any integer as an argument which will be the exit status code for the awk process. If no argument is supplied, exit returns status zero.

The following awk script exits during the 5th iteration. Since the print statement is after the exit statement, the value of x is printed only till 4, and once it reaches 5 awk exits.

```
$ awk 'BEGIN{
x=1;
while(x<=10)
{
if(x==5){
exit;}
print "Value of x",x;x++;
}
}'
```

The above command produces the following output.

```
Value of x 1
Value of x 2
Value of x 3
Value of x 4
```

The following example prints the first item number that has a month when no items were sold. This is similar to the break.awk example, except that it exits when it finds a month with no sales for an item, rather than going on to look at the other items.

```
$ cat exit.awk
{
 i=2; total=0;
 while (i++ <= NF)
 if ($i == 0) {
  print "Item", $1, "had a month with no item sold"
  exit;
 }
}

$ awk -f exit.awk items-sold.txt
Item 102 had a month with no item sold
```

Note: Item 104 also had a month with no item sold. But, it was not displayed above, as we used exit in the while loop.

Chapter 12. Awk Associative Arrays

77. Assigning Array Elements

Arrays in awk are extremely powerful when compared to the traditional arrays that you might have used in other programming languages.

In Awk, arrays are associative, i.e. an array contains multiple index/value pairs. The index doesn't need to be a continuous set of numbers; in fact it can be a string or a number, and you don't need to specify the size of the array.

Syntax:

```
arrayname[string]=value
```

- arrayname is the name of the array.
- string is the index of an array.
- value is any value assigning to the element of the array.

Accessing elements of the AWK array

If you want to access a particular element in an array, you use the format arrayname[index], which gives you the value assigned to that index.

The following is a simple array assignment example:

```
$ cat array-assign.awk
BEGIN {
 item[101]="HD Camcorder";
 item[102]="Refrigerator";
 item[103]="MP3 Player";
 item[104]="Tennis Racket";
 item[105]="Laser Printer";
```

```
 item[1001]="Tennis Ball";
 item[55]="Laptop";
 item["na"]="Not Available";
 print item["101"];
 print item[102];
 print item["103"];
 print item[104];
 print item["105"];
 print item[1001];
 print item[55];
 print item["na"];
}

$ awk -f array-assign.awk
HD Camcorder
Refrigerator
MP3 Player
Tennis Racket
Laser Printer
Tennis Ball
Laptop
Not Available
```

Please note the following in the above example:

- Array indexes are not in sequence. It didn't even have to start from 0 or 1. It really started from 101 .. 105, then jumped to 1001, then came down to 55, then it had a string index "na".
- Array indexes can be string. The last item in this array has an index string. i.e. "na" is the index.
- You don't need to initialize or even define the array in awk; you don't need to specify the total array size before you have to use it.
- The naming convention of an awk array is same as the naming convention of an awk variable.

From awk's point of view, the index of the array is always a string. Even when you pass a number for the index, awk will treat it as string index. Both of the following are the same.

```
item[101]="HD Camcorder"
item["101"]="HD Camcorder"
```

78. Referring to Array Elements

You can directly print an array element using print command as shown below, or you can assign the array item to another variable for additional manipulation inside awk program.

```
print item[101]
x=item[105]
```

If you refer to an array element that doesn't exist, awk will automatically create that array element with the given index, and assign null value to it. If you want to avoid this, check if the index is valid before accessing the array element.

You can check whether a particular array index exists by using the following if condition syntax. This will return true, if the index exists in the array.

```
if ( index in array-name )
```

The following is a simple array reference example:

```
$ cat array-refer.awk
BEGIN {
 x = item[55];
 if ( 55 in item )
  print "Array index 55 contains",item[55];
 item[101]="HD Camcorder";
 if ( 101 in item )
```

```
  print "Array index 101 contains",item[101];
 if ( 1010 in item )
  print "Array index 1010 contains",item[1010];
}

$ awk -f array-refer.awk
Array index 55 contains
Array index 101 contains HD Camcorder
```

In the above example:

- item[55] is not assigned with any value earlier. But it is referred in "x = item[55]", so awk will automatically create this array element with null value.
- item[101] is assigned a value. So, when you check for index 101, it is present.
- item[1010] does not exist. So, when you check for index 1010, it is not present.

79. Browse the Array using For Loop

If you want to access all the array elements, you can use a special instance of the for loop to go through all the indexes of an array:

Syntax:

```
for (var in arrayname)
actions
```

- var is any variable name
- in is a keyword
- arrayname is the name of the array.
- actions are list of awk statements to be executed. If you want to execute more than one action, it has to be enclosed within braces. The loop executes list of actions for each element in the array, by setting the variable var to the index of the corresponding element.

In the following example:

In "for (x in item)", x can be any variable, which holds the index.

Please note that we don't have any conditions to verify how many times the condition should loop through. We really don't care how many items are there in the array, as the awk for loop will automatically take care of it, and loop through all the items before exiting the for loop.

The following is a simple for loop example that loops through all the elements in the item array and prints it.

```
$ cat array-for-loop.awk
BEGIN {
 item[101]="HD Camcorder";
 item[102]="Refrigerator";
 item[103]="MP3 Player";
 item[104]="Tennis Racket";
 item[105]="Laser Printer";
 item[1001]="Tennis Ball";
 item[55]="Laptop";
 item["na"]="Not Available";

 for (x in item)
 print item[x];
}

$ awk -f array-for-loop.awk
Laptop
HD Camcorder
Refrigerator
MP3 Player
Tennis Racket
Laser Printer
Not Available
Tennis Ball
```

80. Delete Array Element

If you want to remove an element from a particular index of an array, use awk delete statement. Once you delete an element from an awk array, you can no longer obtain its value.

Syntax:

```
delete arrayname[index];
```

The loop command below removes all elements from an array.

```
for (var in array)
   delete array[var]
```

In GAWK, you can specify the following single command to delete all the elements from an array.

```
delete array
```

Also, as shown in the example below, item[103]="" does not delete the array element. It just stores null values in it.

```
$ cat array-delete.awk
BEGIN {
 item[101]="HD Camcorder";
 item[102]="Refrigerator";
 item[103]="MP3 Player";
 item[104]="Tennis Racket";
 item[105]="Laser Printer";
 item[1001]="Tennis Ball";
 item[55]="Laptop";
 item["na"]="Not Available";

 delete item[102];
 item[103]="";
```

```
 delete item[104];
 delete item[1001];
 delete item["na"];

 for (x in item)
 print "Index",x,"contains",item[x];
}

$ awk -f array-delete.awk
Index 55 contains Laptop
Index 101 contains HD Camcorder
Index 103 contains
Index 105 contains Laser Printer
```

81. Multi Dimensional Array

Awk has only one dimensional array. But, the beauty of awk is that you can simulate a multi dimensional array using the single dimensional array itself.

Suppose you want to create the following 2 x 2 multi dimensional array.

```
10 20
30 40
```

In the above example, item at location "1,1" is 10, item at location "1,2" is 20, etc. Do the following to assign 10 to location "1,1".

```
item["1,1"]=10
```

Even though you've given "1,1" as index, it is not two indexes. It is just one index with the string "1,1". So, in the above example, you are really storing the value 10 at a single dimensional array with index "1,1".

```
$ cat array-multi.awk
BEGIN {
```

```
 item["1,1"]=10;
 item["1,2"]=20;
 item["2,1"]=30;
 item["2,2"]=40;

 for (x in item)
 print item[x];
}

$ awk -f array-multi.awk
10
20
30
40
```

Now, what happens when you don't enclose the indexes within quotes? i.e. item[1,1] (instead of item["1,1"]), as shown in the example below.

```
$ cat array-multi2.awk
BEGIN {
 item[1,1]=10;
 item[1,2]=20;
 item[2,1]=30;
 item[2,2]=40;

 for (x in item)
 print item[x];
}

$ awk -f array-multi2.awk
30
40
10
20
```

The above sample program will still work. But, there is a difference. In a multi-dimensional awk array, when you don't enclose the indexes within quotes, awk uses a subscript separator with default value of "\034".

When you specify item[1,2], it will be translated to item["1\0342"]. Awk will combine both the subscripts using \034 in between and convert them to string.

When you specify item["1,2"], it will not be translated, as it will be treated just as a one dimensional array with no subscripts.

This is demonstrated in the example below.

```
$ cat array-multi3.awk
BEGIN {
 item["1,1"]=10;
 item["1,2"]=20;

 item[2,1]=30;
 item[2,2]=40;

 for (x in item)
 print "Index",x,"contains",item[x];
}

$ awk -f array-multi3.awk
Index 1,1 contains 10
Index 1,2 contains 20
Index 2#1 contains 30
Index 2#2 contains 40
```

In the above example:

- Indexes "1,1" and "1,2" are enclosed in quotes. So, this is treated as a one dimensional array index, no subscript separator is used by awk. So, the index gets printed as is.
- Indexes 2,1 and 2,2 are not enclosed in quotes. So, this is treated as a multi-dimensional array index, and awk uses a subscript separator. So, the index is "2\0341" and "2\0342", which is printed with the non-printable character "\034" between the subscripts.

82. SUBSEP - Subscript Separator

You can change the default subscript separator to anything you like using the SUBSEP variable. In the following example, SUBSEP is set to colon.

```
$ cat array-multi4.awk
BEGIN {
 SUBSEP=":";

 item["1,1"]=10;
 item["1,2"]=20;

 item[2,1]=30;
 item[2,2]=40;

 for (x in item)
 print "Index",x,"contains",item[x];
}

$ awk -f array-multi4.awk
Index 1,1 contains 10
Index 1,2 contains 20
Index 2:1 contains 30
Index 2:2 contains 40
```

In the above example, indexes "1,1" and "1,2" didn't use the SUBSEP because they were enclosed in quotes.

So, for a multi-dimensional awk array, the best practice is not to enclose any of the indexes within quotes, as shown below.

```
$ cat array-multi5.awk
BEGIN {
 SUBSEP=":";
 item[1,1]=10;
 item[1,2]=20;

 item[2,1]=30;
 item[2,2]=40;

 for (x in item)
 print "Index",x,"contains",item[x];
}

$ awk -f array-multi5.awk
Index 1:1 contains 10
Index 1:2 contains 20
Index 2:1 contains 30
Index 2:2 contains 40
```

83. Sort Array Values using asort

The asort function sorts the array values and stores them in indexes from 1 through n. Where n is the total number of elements in the array.

Suppose you have two elements in the array: item["something"]="B - I'm big b" and item["notsure"]="A - I'm big a". After an asort function call, the array will be sorted based on the values to: item[1]="A - I'm big a" and item[2]="B - I'm big b".

In the following example, we have array indexes with various non-consecutive numbers and strings. After the asort, the array values

will be sorted and stored in the indexes 1,2,3,4,... Please note that asort returns the total number of items in the array.

```
$ cat asort.awk
BEGIN {
 item[101]="HD Camcorder";
 item[102]="Refrigerator";
 item[103]="MP3 Player";
 item[104]="Tennis Racket";
 item[105]="Laser Printer";
 item[1001]="Tennis Ball";
 item[55]="Laptop";
 item["na"]="Not Available";

 print "------Before asort------"
 for (x in item)
 print "Index",x,"contains",item[x];
 total = asort(item);

 print "------After asort------"
 for (x in item)
 print "Index",x,"contains",item[x];
 print "Return value from asort:", total;
}

$ awk -f asort.awk
------Before asort------
Index 55 contains Laptop
Index 101 contains HD Camcorder
Index 102 contains Refrigerator
Index 103 contains MP3 Player
Index 104 contains Tennis Racket
Index 105 contains Laser Printer
Index na contains Not Available
```

```
Index 1001 contains Tennis Ball
------After asort------
Index 4 contains MP3 Player
Index 5 contains Not Available
Index 6 contains Refrigerator
Index 7 contains Tennis Ball
Index 8 contains Tennis Racket
Index 1 contains HD Camcorder
Index 2 contains Laptop
Index 3 contains Laser Printer
Return value from asort: 8
```

In the above example, after the asort, the array elements are not printed from indexes 1 through 8. Instead, it is random. You can print them from 1 through 8 as shown in the example below.

```
$ cat asort1.awk
BEGIN {
 item[101]="HD Camcorder";
 item[102]="Refrigerator";
 item[103]="MP3 Player";
 item[104]="Tennis Racket";
 item[105]="Laser Printer";
 item[1001]="Tennis Ball";
 item[55]="Laptop";
 item["na"]="Not Available";

 total = asort(item);
 for (i=1; i<= total; i++)
 print "Index",i,"contains",item[i];
}

$ awk -f asort1.awk
Index 1 contains HD Camcorder
```

```
Index 2 contains Laptop
Index 3 contains Laser Printer
Index 4 contains MP3 Player
Index 5 contains Not Available
Index 6 contains Refrigerator
Index 7 contains Tennis Ball
Index 8 contains Tennis Racket
```

As you may have noticed in the above examples, once asort is executed, you'll lose the original indexes forever. So, instead of overwriting the original array with the new indexes, you might want to create a new array with the new indexes.

In the following example, the original array "item" is not modified. Instead, the "itemnew" array will contain the new indexes. i.e. itemnew[1], itemnew[2], itemnew[3], etc.

```
total = asort(item, itemnew);
```

Again, remember that asort sorts the array values. But, instead of using the original indexes, it uses new indexes from 1 through n. Original indexes are lost.

84. Sort Array Indexes using asorti

Just like sorting array values, you can take all the array indexes, sort them, and store them in a new array using asorti.

The following example shows how asorti differs from asort. Keep the following in mind:

- asorti sorts the indexes (not the values) and stores them as values.
- If you specify asorti(state), you'll lose the original values. i.e. the indexes will now become the values. So, to be on safe side, always specify two parameters to the asorti function. i.e. asorti(state,stateabbr). This way, the original array (state), it not overwritten.

```
$ cat asorti.awk
BEGIN {
 state["TX"]="Texas";
 state["PA"]="Pennsylvania";
 state["NV"]="Nevada";
 state["CA"]="California";
 state["AL"]="Alabama";

 print "----- Function: asort -----"
 total = asort(state,statedesc);
 for (i=1; i<= total; i++)
 print "Index",i,"contains",statedesc[i];

 print "----- Function: asorti -----"
 total = asorti(state,stateabbr);
 for (i=1; i<= total; i++)
 print "Index",i,"contains",stateabbr[i];
}

$ awk -f asorti.awk
----- Function: asort -----
Index 1 contains Alabama
Index 2 contains California
Index 3 contains Nevada
Index 4 contains Pennsylvania
Index 5 contains Texas
----- Function: asorti -----
Index 1 contains AL
Index 2 contains CA
Index 3 contains NV
Index 4 contains PA
Index 5 contains TX
```

Chapter 13. Additional Awk Commands

85. Pretty Printing Using printf

Printf is very flexible and makes report printing job relatively easier by allowing you to print the output in the way you want it.

Syntax:

```
printf "print format", variable1, variable2, etc.
```

Special Characters in the printf Format

Following are some of the special characters that can be used inside a printf.

Special Character	Description
\n	New Line
\t	Tab
\v	Vertical Tab
\b	Backspace
\r	Carriage Return
\f	Form Feed

The following prints "Line 1" and "Line 2" in separate lines using newline:

```
$ awk 'BEGIN { printf "Line 1\nLine 2\n" }'
Line 1
Line 2
```

The following prints different fields separated by tabs, with 2 tabs after "Field 1":

```
$ awk 'BEGIN \
{ printf "Field 1\t\tField 2\tField 3\tField 4\n" }'
Field 1         Field 2     Field 3     Field 4
```

The following prints vertical tabs after every field:

```
$ awk 'BEGIN \
{ printf "Field 1\vField 2\vField 3\vField 4\n" }'
Field 1
    Field 2
       Field 3
           Field 4
```

The following prints a backspace after every field except Field4. This erases the last number in each of the first three fields. For example "Field 1" is displayed as "Field ", because the last character is erased with backspace. However the last field "Field 4" is displayed as it is, as we didn't have a \b after "Field 4".

```
$ awk 'BEGIN \
{ printf "Field 1\bField 2\bField 3\bField 4\n" }'
Field Field Field Field 4
```

In the following example, after printing every field, we do a "Carriage Return" and print the next value on top of the current printed value. This means, in the final output you see is only "Field 4", as it was the last thing to be printed on top of all the previous fields.

```
$ awk 'BEGIN \
{ printf "Field 1\rField 2\rField 3\rField 4\n" }'
Field 4
```

Print Uses OFS, ORS Values

When you print multiple values separated by comma using print command (not printf), it uses the OFS and RS built-in variable values to decide how to print the fields.

The following example show how the simple print statement "print $2,$3" gets affected by using OFS and ORS values.

```
$ cat print.awk
BEGIN {
 FS=",";
 OFS=":";
 ORS="\n--\n";
}
{
 print $2,$3
}

$ awk -f print.awk items.txt
HD Camcorder:Video
--
Refrigerator:Appliance
--
MP3 Player:Audio
--
Tennis Racket:Sports
--
Laser Printer:Office
--
```

Printf doesn't Use OFS, ORS Values

Printf doesn't use the OFS and ORS values. It uses only what is specified in the "format" field of the printf command as shown in the example below.

```
$ cat printf1.awk
BEGIN {
 FS=",";
 OFS=":";
 ORS="\n--\n";
}
{
 printf "%s^^%s\n\n", $2, $3
}

$ awk -f printf1.awk items.txt
HD Camcorder^^Video
Refrigerator^^Appliance
MP3 Player^^Audio
Tennis Racket^^Sports
Laser Printer^^Office
```

Printf Format Specifiers

Format Specifier	Description
s	String
c	Single Character
d	Decimal
e	Exponential Floating point
f	Fixed Floating point
g	Uses either e or f depending on which is smaller for the given input
o	Octal
x	Hexadecimal
%	Prints the percentage symbol

The following example shows the basic usage of the format specifiers:

```
$ cat printf-format.awk
BEGIN {
 printf "s--> %s\n", "String"
 printf "c--> %c\n", "String"
 printf "s--> %s\n", 101.23
 printf "d--> %d\n", 101.23
 printf "e--> %e\n", 101.23
 printf "f--> %f\n", 101.23
 printf "g--> %g\n", 101.23
 printf "o--> %o\n", 0x8
 printf "x--> %x\n", 16
 printf "percentage--> %%\n", 17
}

$ awk -f printf-format.awk
s--> String
c--> S
s--> 101.23
d--> 101
e--> 1.012300e+02
f--> 101.230000
g--> 101.23
o--> 10
x--> 10
percentage--> %
```

Print with Fixed Column Width (Basic)

To create a fixed column width report, you have to specify a number immediately after the % in the format specifier. This number indicates the minimum number of character to be printed. When the input-string is smaller than the specified number, spaces are added to the left to make it fixed width.

The following example displays the basic use of the printf statement with number specified immediately after %

```
$ cat printf-width.awk
BEGIN {
 FS=","
 printf "%3s\t%10s\t%10s\t%5s\t%3s\n",
"Num","Description","Type","Price","Qty"
 printf
"-----------------------------------------------------\
n"
}
{
 printf "%3d\t%10s\t%10s\t%g\t%d\n", $1,$2,$3,$4,$5
}

$ awk -f printf-width.awk items.txt
Num  Description     Type   Price  Qty
-------------------------------------------------
101  HD Camcorder    Video   210   10
102  Refrigerator  Appliance   850   2
103  MP3 Player     Audio   270   15
104  Tennis Racket   Sports   190   20
105  Laser Printer   Office   475   5
```

Notice that the output is a bit ragged, even though we specified the exact width. That's because the width we specify is actually the *minimum* width, not the absolute size; if the input string has more characters than that, the whole string will be printed. So, you should really pay attention to how many characters you want to print.

If you want to print a fixed column width even when the input string is longer than the number specified, you should use the substr function (or) add a decimal before the number in the format identifier (as explained later).

In the previous example, the second field was wider than the 10 character width specified, so the result was not what was intended.

Spaces are added to the left to print “Good” as a 6 character string:

```
$ awk 'BEGIN { printf "%6s\n", "Good" }'
 Good
```

The whole string is printed here even though you specified 6 character width:

```
$ awk 'BEGIN { printf "%6s\n", "Good Boy!" }'
Good Boy!
```

Print with Fixed Width (Left Justified)

When the input-string is less than the number of characters specified, and you would like it to be left justified (by adding spaces to the right), use a minus symbol (-) immediately after the % and before the number.

"%6s" is right justified as shown below:

```
$ awk 'BEGIN { printf "|%6s|\n", "Good" }'
| Good|
```

"%-6s" is left justified as shown below:

```
$ awk 'BEGIN { printf "|%-6s|\n", "Good" }'
|Good |
```

Print with Dollar Amount

To add a dollar symbol before the price value, just add the dollar symbol before the identifier in the printf as shown below.

```
$ cat printf-width2.awk
BEGIN {
 FS=","
 printf "%-3s\t%-10s\t%-10s\t%-5s\t%-3s\n",
"Num","Description","Type","Price","Qty"
```

```
 printf
"-------------------------------------------------\
n"
}
{
 printf "%-3d\t%-10s\t%-10s\t$%-.2f\t%-d\n",
$1,$2,$3,$4,$5
}

$ awk -f printf-width2.awk items.txt
Num  Description   Type     Price  Qty
-------------------------------------------------
101  HD Camcorder   Video     $210.00 10
102  Refrigerator   Appliance   $850.00 2
103  MP3 Player    Audio     $270.00 15
104  Tennis Racket  Sports    $190.00 20
105  Laser Printer  Office    $475.00 5
```

Print with Leading Zeros

By default values are right justified with space added to the left

```
$ awk 'BEGIN { printf "|%5s|\n", "100" }'
| 100|
```

For right justified with 0's in front of the number (instead of the space), add a zero (0) before the number. i.e. Instead of "%5s", use "%05s" as the format identifier.

```
$ awk 'BEGIN { printf "|%05s|\n", "100" }'
|00100|
```

The following example uses the leading zero format identifier for the Qty field.

```
$ cat printf-width3.awk
BEGIN {
 FS=","
 printf "%-3s\t%-10s\t%-10s\t%-5s\t%-3s\n",
"Num","Description","Type","Price","Qty"
 printf
"-----------------------------------------------------\
n"
}
{
 printf "%-3d\t%-10s\t%-10s\t$%-.2f\t%03d\n",
$1,$2,$3,$4,$5
}

$ awk -f printf-width3.awk items.txt
Num  Description   Type     Price  Qty
-------------------------------------------------
101  HD Camcorder   Video     $210.00 010
102  Refrigerator   Appliance   $850.00 002
103  MP3 Player    Audio     $270.00 015
104  Tennis Racket  Sports    $190.00 020
105  Laser Printer  Office    $475.00 005
```

Print Absolute Fixed Width String Value

As we already shown you, when the input string contains more characters than what is specified in the format specifier it prints the whole thing as shown below.

```
$ awk 'BEGIN { printf "%6s\n", "Good Boy!" }'
Good Boy!
```

To print maximum of ONLY 6 characters, add a decimal before the number. i.e. Instead of "%6s", give "%.6s", which will print only 6 characters from the input string, even when the input string is longer than that as shown below.

```
$ awk 'BEGIN { printf "%.6s\n", "Good Boy!" }'
Good B
```

The above doesn't work on all versions of awk. On GAWK 3.1.5 it worked. But on GAWK 3.1.7 it didn't work.

So, the reliable way to print a fixed character might be to use the substr function as shown below.

```
$ awk 'BEGIN \
{ printf "%6s\n", substr("Good Boy!",1,6) }'
Good B
```

Dot . Precision

A dot before the number in format identifier indicates the precision.

The following example shows how a dot before a number for the numeric format identifier works. This example shows how the number "101.23" is printed differently when using using .1 and .4 (using d, e, f, and g format specifier).

```
$ cat dot.awk
BEGIN {
 print "----Using .1----"
 printf ".1d--> %.1d\n", 101.23
 printf ".1e--> %.1e\n", 101.23
 printf ".1f--> %.1f\n", 101.23
 printf ".1g--> %.1g\n", 101.23
 print "----Using .4----"
 printf ".4d--> %.4d\n", 101.23
 printf ".4e--> %.4e\n", 101.23
 printf ".4f--> %.4f\n", 101.23
 printf ".4g--> %.4g\n", 101.23
}

$ awk -f dot.awk
```

```
----Using .1----
.1d--> 101
.1e--> 1.0e+02
.1f--> 101.2
.1g--> 1e+02
----Using .4----
.4d--> 0101
.4e--> 1.0123e+02
.4f--> 101.2300
.4g--> 101.2
```

Print Report to File

You can redirect the output of a print statement to a specific output file inside the awk script. In the following example the 1st print statement has "> report.txt", which creates the report.txt file and sends the output of the prints statement to it. All the subsequent print statements have ">> report.txt", which appends the output to the existing report.txt file.

```
$ cat printf-width4.awk
BEGIN {
 FS=","
 printf "%-3s\t%-10s\t%-10s\t%-5s\t%-3s\n",
"Num","Description","Type","Price","Qty" > "report.txt"
 printf
"-----------------------------------------------------\
n" >> "report.txt"
}
{
 if ($5 > 10)
 printf "%-3d\t%-10s\t%-10s\t$%-.2f\t%03d\n",
$1,$2,$3,$4,$5 >> "report.txt"
}

$ awk -f printf-width4.awk items.txt
```

```
$ cat report.txt
Num  Description   Type     Price  Qty
-------------------------------------------------
103  MP3 Player    Audio     $270.00 015
104  Tennis Racket Sports    $190.00 020
```

The other method is not to specify the "> report.txt" or ">> report.txt" in the print statement. Instead, while executing the awk script, redirect the output to the report.xt as shown below.

```
$ cat printf-width5.awk
BEGIN {
 FS=","
 printf "%-3s\t%-10s\t%-10s\t%-5s\t%-3s\n",
"Num","Description","Type","Price","Qty"
 printf
"-----------------------------------------------------\
n"
}
{
 if ($5 > 10)
 printf "%-3d\t%-10s\t%-10s\t$%-.2f\t%03d\n",
$1,$2,$3,$4,$5
}

$ awk -f printf-width5.awk items.txt > report.txt

$ cat report.txt
Num  Description   Type     Price  Qty
-------------------------------------------------
103  MP3 Player    Audio     $270.00 015
104  Tennis Racket  Sports    $190.00 020
```

86. Built-in Numeric Functions

Awk has built-in functions for several numeric, string, input, and output operations. We discuss some of them here.

Awk int(n) Function

int() function gives you the integer part of the given argument. This produces the lowest integer part of given n. n is any number with or with out floating point. If you give a whole number as an argument, this function returns the same number; for a floating point number, it truncates.

Init Function Example:

```
$ awk 'BEGIN{
print int(3.534);
print int(4);
print int(-5.223);
print int(-5);
}'
```

The above command produces the following output.

```
3
4
-5
-5
```

Awk log(n) Function

The log(n) function provides the natural logarithm of given argument n. The number n must be positive, or an error will be thrown.

Log Function Example:

```
$ awk 'BEGIN{
print log(12);
print log(0);
print log(1);
```

```
print log(-1);
}'
2.48491
-inf
0
nan
```

In the above output you can identify that log(0) is infinity which was shown as -inf, and log(-1) gives you the error nan (Not a Number).

Note: You might also get the following warning message for the log(-1): awk: cmd. line:4: warning: log: received negative argument -1

Awk sqrt(n) Function

sqrt function gives the positive square root for the given integer n. This function also requires a positive number, and it returns nan error if you give the negative number as an argument.

Sqrt Function Example:

```
$ awk 'BEGIN{
print sqrt(16);
print sqrt(0);
print sqrt(-12);
}'
4
0
nan
```

Awk exp(n) Function

The exp(n) function provides e to the power of n.

Exp Function Example:

```
$ awk 'BEGIN{
print exp(123434346);
print exp(0);
```

```
print exp(-12);
}'
inf
1
6.14421e-06
```

In the above output, for exp(1234346), it gives you the output infinity, because this is out of range.

Awk sin(n) Function

The sin(n) function gives the sine of n, with n in radians.

Sine Function Example:

```
$ awk 'BEGIN {
print sin(90);
print sin(45);
}'
0.893997
0.850904
```

Awk cos(n) Function

The cos(n) returns the cosine of n, with n in radians.

Cosine Function Example:

```
$ awk 'BEGIN {
print cos(90);
print cos(45);
}'
-0.448074
0.525322
```

Awk atan2(m,n) Function

This function gives you the arc-tangent of m/n in radians.

Atan2 Function Example:

```
$ awk 'BEGIN { print atan2(30,45) }'
0.588003
```

87. Random Number Generator

Awk rand() Function

rand() is used to generate a random number between 0 and 1. It never returns 0 or 1, always a value between 0 and 1. Numbers are random within one awk run, but predictable from run to run.

Awk uses an algorithm to generate the random numbers, and since this algorithm is fixed, the numbers are repeatable.

The following example generates 1000 random numbers between 0 and 100, and shows how often each number was generated.

Generate 1000 random numbers (between 0 and 100):

```
$ cat rand.awk
BEGIN {
while(i<1000)
{
    n = int(rand()*100);
    rnd[n]++;
    i++;
}
for(i=0;i<=100;i++) {
    print i,"Occured", rnd[i], "times";
}
}
```

```
$ awk -f rand.awk
0 Occured 6 times
1 Occured 16 times
2 Occured 12 times
3 Occured 6 times
4 Occured 13 times
5 Occured 13 times
6 Occured 8 times
7 Occured 7 times
8 Occured 16 times
9 Occured 9 times
10 Occured 6 times
11 Occured 9 times
12 Occured 17 times
13 Occured 12 times
```

From the above output, we can see that the rand() function can generate repeatable numbers very often.

Awk srand(n) Function

srand(n) is used to initialize the random number generation with a given argument n. Whenever program execution starts, awk starts generating its random numbers from n. If no argument were given, awk would use the time of the day to generate the seed.

Generate 5 random numbers starting from 5 to 50:

```
$ cat srand.awk
BEGIN {
 # Initialize the seed with 5.
 srand(5);

 # Totally I want to generate 5 numbers.
 total=5;

 #maximum number is 50.
```

```
 max=50;
 count=0;
 while(count < total) {
    rnd = int(rand() * max);
    if ( array[rnd] == 0 ) {
          count++;
          array[rnd]++;
    }
 }

 for ( i=5; i<=max; i++) {
    if ( array[i] )
          print i;
 }
}

$ awk -f srand.awk
9
15
26
37
39
```

The above srand.awk does the following:

- Uses rand() function to generate a random number that is multiplied with the maximum desired value to produce a number < 50.
- Checks if the generated random number already exists in the array. If it does not exist, it increments the index and loop count. It generates 5 numbers using this logic.
- Finally in the for loop, it loops from minimum to maximum, and prints each index that contains any value.

88. Generic String Functions

Following are the common awk string functions that are available on all flavors of awk.

Index Function

The index function can be used to get the index (location) of the given string (or character) in an input string.

In the following example, string "Cali" is located in the string "CA is California" at location number 7.

You can also use index to check whether a given string (or character) is present in an input string. If the given string is not present, it will return the location as 0, which means the given string doesn't exist, as shown below.

```
$ cat index.awk
BEGIN {
 state="CA is California"
 print "String CA starts at
location",index(state,"CA");
 print "String Cali starts at
location",index(state,"Cali");

 if (index(state,"NY")==0)
 print "String NY is not found in:", state
}

$ awk -f index.awk
String CA starts at location 1
String Cali starts at location 7
String NY is not found in: CA is California
```

Length Function

The length function returns the length of a string. In the following example, we print the total number of characters in each record of the items.txt file.

```
$ awk '{print length($0)}' items.txt
29
32
27
31
30
```

Split Function

Syntax:

```
split(input-string,output-array,separator)
```

This split function splits a string into individual array elements. It takes following three arguments.

- input-string: This is the input string that needs to be split into multiple strings.
- output-array: This array will contain the split strings as individual elements.
- separator: The separator that should be used to split the input-string.

For this example, the original items-sold.txt file is slightly changed to have different field delimiters, i.e. a colon to separate the item number and the quantity sold. Within quantity sold, the individual quantities are separated by comma.

So, in order for us to calculate the total number of items sold for a particular item, we should take the 2nd field (which is all the quantities sold delimited by comma), split them using comma separator and store the substrings in an array, then loop through the array to add the quantities.

```
$ cat items-sold1.txt
101:2,10,5,8,10,12
102:0,1,4,3,0,2
103:10,6,11,20,5,13
104:2,3,4,0,6,5
```

```
105:10,2,5,7,12,6

$ cat split.awk
BEGIN {
 FS=":"
}
{
 split($2,quantity,",");
 total=0;
 for (x in quantity)
 total=total+quantity[x];
 print "Item", $1, ":", total, "quantities sold";
}

$ awk -f split.awk items-sold1.txt
Item 101 : 47 quantities sold
Item 102 : 10 quantities sold
Item 103 : 65 quantities sold
Item 104 : 20 quantities sold
Item 105 : 42 quantities sold
```

Substr Function

Syntax:

```
substr(input-string, location, length)
```

The substr function extracts a portion of a given string. In the above syntax:

- input-string: The input string containing the substring.
- location: The starting location of the substring.
- length: The total number of characters to extract from the starting location. This parameter is optional. When you don't specify it extracts the rest of the characters from the starting location.

The following example starts extracting the string from 5th the character and prints the rest of the line. The 1st 3 characters are the item number, 4th character is the comma delimiter. So, this skips the item number and prints the rest.

```
$ awk '{print substr($0,5)}' items.txt
HD Camcorder,Video,210,10
Refrigerator,Appliance,850,2
MP3 Player,Audio,270,15
Tennis Racket,Sports,190,20
Laser Printer,Office,475,5
```

Start from the 1st character (of the 2nd field) and prints 5 characters:

```
$ awk -F"," '{print substr($2,1,5)}' items.txt
HD Ca
Refri
MP3 P
Tenni
Laser
```

89. GAWK/NAWK String Functions

These string functions are available only in GAWK and NAWK flavors.

Sub Function

syntax:

```
sub(original-string,replacement-string,string-variable)
```

- sub stands for substitution.
- original-string: This is the original string that needs to be replaced. This can also be a regular expression.
- replacement-string: This is the replacement string.

- string-variable: This acts as both input and output string variable. You have to be careful with this, as after the successful substitution, you lose the original value in this string-variable.

In the following example:

- original-string: This is the regular expression C[Aa], which matches either "CA" or "Ca"
- replacement-string: When the original-string is found, replace it with "KA"
- string-variable: Before executing the sub, the variable contains the input string. Once the replacement is done, the variable contains the output string.

Please note that sub replaces only the 1st occurrence of the match.

```
$ cat sub.awk
BEGIN {
 state="CA is California"
 sub("C[Aa]","KA",state);
 print state;
}

$ awk -f sub.awk
KA is California
```

The 3rd parameter string-variable is optional. When it is not specified, awk will use $0 (the current line), as shown below. This example changes the first 2 characters of the record from "10" to "20". So, the item number 101 becomes 201, 102 becomes 202, etc.

```
$ awk '{ sub("10","20"); print $0; }' items.txt
201,HD Camcorder,Video,210,10
202,Refrigerator,Appliance,850,2
203,MP3 Player,Audio,270,15
204,Tennis Racket,Sports,190,20
205,Laser Printer,Office,475,5
```

When a successful substitution happens, the sub function returns 1, otherwise it returns 0.

Print the record only when a successful substitution occurs:

```
$ awk '{ if (sub("HD","High-Def")) print $0; }' \
items.txt
101,High-Def Camcorder,Video,210,10
```

Gsub Function

gsub stands for global substitution. gsub is exactly same as sub, except that all occurrences of original-string are changed to replacement-string.

In the following example, both "CA" and "Ca" are changed to "KA":

```
$ cat gsub.awk
BEGIN {
 state="CA is California"
 gsub("C[Aa]","KA",state);
 print state;
}

$ awk -f gsub.awk
KA is KAlifornia
```

As with sub, the 3rd parameter is optional. When it is not specified, awk will use $0 as shown below.

The following example replaces all the occurrences of "10" in the line with "20". So, other than changing the item-number, it also changes other numeric fields in the record, if it contains "10".

```
$ awk '{ gsub("10","20"); print $0; }' items.txt
201,HD Camcorder,Video,220,20
```

```
202,Refrigerator,Appliance,850,2
203,MP3 Player,Audio,270,15
204,Tennis Racket,Sports,190,20
205,Laser Printer,Office,475,5
```

Match Function () and RSTART, RLENGTH variables

Match function searches for a given string (or regular expression) in the input-string, and returns a positive value when a successful match occurs.

Syntax:

```
match(input-string,search-string)
```

- input-string: This is the input-string that needs to be searched.
- search-string: This is the search-string, that needs to be search in the input-string. This can also be a regular expression.

The following example searches for the string "Cali" in the state string variable. If present, it prints a successful message.

```
$ cat match.awk
BEGIN {
 state="CA is California"
 if (match(state,"Cali")) {
 print substr(state,RSTART,RLENGTH),"is present in:",
state;
 }
}

$ awk -f match.awk
Cali is present in: CA is California
```

Match sets the following two special variables. The above example uses these in the substring function call, to print the pattern in the success message.

- RSTART - The starting location of the search-string
- RLENGTH - The length of the search-string.

90. GAWK String Functions

tolower and toupper are available only in Gawk. As the name suggests the function converts the given string to lower case or upper case as shown below.

```
$ awk '{print tolower($0)}' items.txt
101,hd camcorder,video,210,10
102,refrigerator,appliance,850,2
103,mp3 player,audio,270,15
104,tennis racket,sports,190,20
105,laser printer,office,475,5

$ awk '{print toupper($0)}' items.txt
101,HD CAMCORDER,VIDEO,210,10
102,REFRIGERATOR,APPLIANCE,850,2
103,MP3 PLAYER,AUDIO,270,15
104,TENNIS RACKET,SPORTS,190,20
105,LASER PRINTER,OFFICE,475,5
```

91. Argument Processing (ARGC, ARGV, ARGIND)

The built-in variables we discussed earlier, FS, NFS, RS, NR, FILENAME, OFS, and ORS, are all available on all versions of awk (including nawk, and gawk).

- The environment variables discussed in this hack are available only on nawk and gawk.
- Use ARGC and ARGV to pass some parameters to the awk script from the command line.

- ARGC contains the total number of arguments passed to the awk script.
- ARGV is an array contains all the arguments passed to the awk script in the index from 0 through ARGC
- When you pass 5 arguments, ARGC will contain the value of 6.
- ARGV[0] will always contain awk.

The following simple arguments.awk shows how ARGC and ARGV behave:

```
$ cat arguments.awk
BEGIN {
 print "ARGC=",ARGC
 for (i = 0; i < ARGC; i++)
  print ARGV[i]
}

$ awk -f arguments.awk arg1 arg2 arg3 arg4 arg5
ARGC= 6
awk
arg1
arg2
arg3
arg4
arg5
```

In the following example:

- We are passing parameters to the script in the format "--paramname paramvalue".
- The awk script can take item number and the quantity as arguments.
- if you use "--item 104 --qty 25" as argument to the awk script, it will set quantity as 25 for the item number 104.

- if you use "--item 105 --qty 3" as argument to the awk script, it will set quantity as 3 for the item number 105.

```
$ cat argc-argv.awk
BEGIN {
 FS=",";
 OFS=",";
 for (i=0; i<ARGC; i++) {
 if (ARGV[i]=="--item") {
  itemnumber=ARGV[i+1];
  delete ARGV[i]
  i++;
  delete ARGV[i]
 } else if (ARGV[i]=="--qty") {
  quantity=ARGV[i+1];
  delete ARGV[i]
  i++;
  delete ARGV[i]
 }
 }
}
{
  if ($1==itemnumber)
  print $1,$2,$3,$4,quantity
  else
  print $0;
}

$ awk -f argc-argv.awk --item 104 --qty 25 items.txt
101,HD Camcorder,Video,210,10
102,Refrigerator,Appliance,850,2
103,MP3 Player,Audio,270,15
104,Tennis Racket,Sports,190,25
105,Laser Printer,Office,475,5
```

In gawk the file that is currently getting processed is stored in the ARGV array that is accessed from the body loop. The ARGIND is the index to this ARGV array to retrieve the current file.

When you are processing only one file in an awk script, the ARGIND will be 1, and ARGV[ARGIND] will give the file name that is currently getting processed.

The following example contains only the body block, that prints the value of the ARGIND, and the current file name from the ARGV[ARGIND]

```
$ cat argind.awk
{
 print "ARGIND:", ARGIND
 print "Current file:", ARGV[ARGIND]
}
```

When you call the above example with two files, while processing each and every line of the input-file, it will print the two lines. This just gives you the idea of what is getting stored in the ARGIND and ARGV[ARGIND].

```
$ awk -f argind.awk items.txt items-sold.txt
ARGIND: 1
Current file: items.txt
ARGIND: 1
Current file: items.txt
ARGIND: 1
Current file: items.txt
ARGIND: 1
Current file: items.txt
ARGIND: 1
Current file: items.txt
ARGIND: 1
Current file: items.txt
ARGIND: 2
Current file: items-sold.txt
```

```
ARGIND: 2
Current file: items-sold.txt
ARGIND: 2
Current file: items-sold.txt
ARGIND: 2
Current file: items-sold.txt
ARGIND: 2
Current file: items-sold.txt
```

92. OFMT

The OFMT built-in variable is available only in NAWK and GAWK.

When a number is converted to a string for printing, awk uses the OFMT format to decide how to print the values. The default value is "%.6g", which will print a total of 6 characters including both sides of the dot in a number.

When using g, you have to count all the characters on both sides of the dot. For example, "%.4g" means total of 4 characters will be printed including characters on both sides of the dot.

When using f, you are counting ONLY the characters on the right side of the dot. For example, "%.4f" means 4 characters will be printed on the right side of the dot. The total number of characters on the left side of the dot doesn't matter here.

The following ofmt.awk example shows how the output will be printed when using various OFMT values (for both g and f).

```
$ cat ofmt.awk
BEGIN {
 total=143.123456789;
 print "---using g----"
 print "Default OFMT:", total;
 OFMT="%.3g";
```

```
 print "%.3g OFMT:", total;
 OFMT="%.4g";
 print "%.4g OFMT:", total;
 OFMT="%.5g";
 print "%.5g OFMT:", total;
 OFMT="%.6g";
 print "%.6g OFMT:", total;
 print "---using f----"
 OFMT="%.0f";
 print "%.0f OFMT:", total;
 OFMT="%.1f";
 print "%.1f OFMT:", total;
 OFMT="%.2f";
 print "%.2f OFMT:", total;
 OFMT="%.3f";
 print "%.3f OFMT:", total;
}

$ awk -f ofmt.awk
---using g----
Default OFMT: 143.123
%.3g OFMT: 143
%.4g OFMT: 143.1
%.5g OFMT: 143.12
%.6g OFMT: 143.123
---using f----
%.0f OFMT: 143
%.1f OFMT: 143.1
%.2f OFMT: 143.12
%.3f OFMT: 143.123
```

93. GAWK Built-in Environment Variables

The built-in variables discussed in this section are available only in GAWK.

ENVIRON

This is very helpful when you want to access the shell environment variable in your awk script. ENVIRON is an array that contains all the environment values. The index to the ENVIRON array is the environment variable name.

For example, the array element ENVIRON["PATH"] will contain the value of the PATH environment variable.

The following example prints all the available environment variables and their values.

```
$ cat environ.awk
BEGIN {
 OFS="="
 for(x in ENVIRON)
 print x,ENVIRON[x];
}
```

Partial output is shown below.

```
$ awk -f environ.awk
SHELL=/bin/bash
PATH=/home/ramesh/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
HOME=/home/ramesh
TERM=xterm
USERNAME=ramesh
DISPLAY=:0.0
AWKPATH=.:/usr/share/awk
```

IGNORECASE

By default IGNORECASE is set to 0. So, the awk program is case sensitive.

When you set IGNORECASE to 1, the awk program becomes case insensitive. This will affect regular expression and string comparisons.

The following will not print anything, as it is looking for "video" with lower case "v". But, the items.txt file contains only "Video" with upper case "V".

```
awk '/video/ {print}' items.txt
```

However when you set IGNORECASE to 1, and search for "video", it will print the line containing "Video", as it will not do a case sensitive pattern match.

```
$ awk 'BEGIN{IGNORECASE=1} /video/ {print}' items.txt
101,HD Camcorder,Video,210,10
```

As you see in the example below, this works for both string and regular expression comparisons.

```
$ cat ignorecase.awk
BEGIN {
 FS=",";
 IGNORECASE=1;
}
{
 if ($3 == "video") print $0;
 if ($2 ~ "TENNIS") print $0;
}

$ awk -f ignorecase.awk items.txt
101,HD Camcorder,Video,210,10
104,Tennis Racket,Sports,190,20
```

ERRNO

When there is an error while using I/O operations (for example: getline), the ERRNO variable will contain the corresponding error message.

The following example is trying to read a file that doesn't exist using getline. In this case the ERRNO variable will contain "No such file or directory" message.

```
$ vi errno.awk
{
 print $0;
 x = getline < "dummy-file.txt"
 if ( x == -1 )
  print ERRNO
 else
  print $0;
}

$ awk -f errno.awk items.txt
101,HD Camcorder,Video,210,10
No such file or directory
102,Refrigerator,Appliance,850,2
No such file or directory
103,MP3 Player,Audio,270,15
No such file or directory
104,Tennis Racket,Sports,190,20
No such file or directory
105,Laser Printer,Office,475,5
No such file or directory
```

94. Awk Profiler - pgawk

The pgawk program is used to create an execution profile of your awk program. Using pgawk you can view how many time each awk statement (and custom user defined functions) were executed.

First, create a sample awk program that we'll run through the pgawk to see how the profiler output looks like.

```
$ cat profiler.awk
BEGIN {
 FS=",";
 print "Report Generated On:" strftime("%a %b %d %H:%M:
%S %Z %Y",systime());
}
{
 if ( $5 <= 5 )
  print "Buy More: Order", $2, "immediately!"
 else
  print "Sell More: Give discount on", $2,
"immediately!"
}
END {
 print "----"
}
```

Next, execute the sample awk program using pgawk (instead of just calling awk).

```
$ pgawk -f profiler.awk items.txt
Report Generated On:Mon Jan 31 08:35:59 PST 2011
Sell More: Give discount on HD Camcorder immediately!
Buy More: Order Refrigerator immediately!
Sell More: Give discount on MP3 Player immediately!
Sell More: Give discount on Tennis Racket immediately!
Buy More: Order Laser Printer immediately!
----
```

By default pgawk creates a file called profiler.out (or awkprof.out). You can specify your own profiler output file name using --profiler option as shown below.

```
$ pgawk --profile=myprofiler.out -f profiler.awk
items.txt
```

View the default awkprof.out to understand the execution counts of the individual awk statements.

```
$ cat awkprof.out
    # gawk profile, created Mon Jan 31 08:35:59 2011
    # BEGIN block(s)
    BEGIN {
   1     FS = ","
   1     print ("Report Generated On:" strftime("%a %b
%d %H:%M:%S %Z %Y", systime()))
    }
    # Rule(s)
   5 {
   5     if ($5 <= 5) { # 2
   2         print "Buy More: Order", $2,
"immediately!"
   3     } else {
   3         print "Sell More: Give discount on", $2,
"immediately!"
        }
    }
    # END block(s)
    END {
   1     print "----"
    }
```

While reading the awkprof.out, please keep the following in mind:

- The column on the left contains a number. This indicates how many times that particular awk command has executed. For example, the print statement in begin executed only once (duh!). The while lop executed 6 times.
- For any condition checking, one on the left side, another on the right side after the parenthesis. The left side indicates how many times the pattern was checked. The right side indicate how many times it was successful. In the above example, if was executed 5 times, but it was successful 2 times as indicated by (# 2) next to the if statement.

95. Bit Manipulation

Just like C, awk can manipulate bits. You might not need this on your day to day awk programming. But, this goes to show how much you can do with the awk program.

Following table shows the single digit decimal number and its binary equivalent.

Decimal	Binary
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

AND

For an AND output to be 1, both the bits should be 1.

- 0 and 0 = 0
- 0 and 1 = 0

- 1 and 0 = 0
- 1 and 1 = 1

For example, let us do AND between the decimal 15 and 25. The and output of 15 and 25 is binary 01001, which is decimal 9.

- 15 = 01111
- 25 = 11001
- 15 and 25 = 01001

OR

For an OR output to be 1, either one of the bits should be 1.

- 0 or 0 = 0
- 0 or 1 = 1
- 1 or 0 = 1
- 1 or 1 = 1

For example, let us do OR between the decimal 15 and 25. The or output of 15 and 25 is binary 11111, which is decimal 31.

- 15 = 01111
- 25 = 11001
- 15 or 25 = 11111

XOR

For XOR output to be 1, only one of the bits should be 1. When both the bits are 1, xor will return 0.

- 0 xor 0 = 0
- 0 xor 1 = 1
- 1 xor 0 = 1
- 1 xor 1 = 0

For example, let us do XOR between the decimal 15 and 25. The xor output of 15 and 25 is binary 10110, which is decimal 22.

- 15 = 01111
- 25 = 11001
- 15 xor 25 = 10110

Complement

Complement Makes 0 as 1, and 1 as 0.

For example, let us complement decimal 15.

- 15 = 01111
- 15 compl = 10000

Left Shift

This function shifts the bits to the left side; you can specify how many times it should do the shift. 0s are shifted in from the right side.

For example, let us left shift (two times) decimal 15. The lshift twice output of 15 is binary 111100, which is decimal 60.

- 15 = 1111
- lshift twice = 111100

Right Shift

This function shifts the bits to the right side; you can specify how many times it should do the shift. 0s are shifted in from the left side.

For example, let us right shift (two times) decimal 15. The lshift twice output of 15 is binary 0011, which is decimal 3.

- 15 = 1111
- lshift twice = 0011

Awk Example using Bit Functions

```
$ cat bits.awk
BEGIN {
 number1=15
 number2=25
 print "AND: " and(number1,number2);
 print "OR: " or(number1,number2)
 print "XOR: " xor(number1,number2)
 print "LSHIFT: " lshift(number1,2)
 print "RSHIFT: " rshift(number1,2)
}

$ awk -f bits.awk
AND: 9
OR: 31
XOR: 22
LSHIFT: 60
RSHIFT: 3
```

96. User Defined Functions

Awk allows you to define user defined functions. This is extremely helpful when you are writing a lot of awk code and end-up repeating certain pieces of code every time. Those pieces could be fit into a user defined function.

Syntax:

```
function fn-name(parameters)
{
 function-body
}
```

In the above syntax:

- fn-name is the function name: Just like an awk variable, an awk user defined function name should begin with a letter. The rest of the characters can be numbers, or alphabetic characters, or underscore. Keywords cannot be used as function name.
- parameters: Multiple parameters are separated by comma. You can also create a user defined function without any parameter.
- function-body: One or more awk statements.

If you've already used a name for a variable inside the awk program, you cannot use the same name for your user defined function.

The following example creates a simple user defined function called discount that gives a discount in the prices for the specified percentage. For example, discount(10) gives 10% discount on the price.

For any items where the quantity is <= 10, it gives 10% discount, otherwise it gives 50% discount.

```
$ cat function.awk
BEGIN {
 FS=","
 OFS=","
}
{
 if ($5 <= 10)
 print $1,$2,$3,discount(10),$5
 else
 print $1,$2,$3,discount(50),$5
}
function discount(percentage)
{
 return $4 - ($4*percentage/100)
}
```

```
$ awk -f function.awk items.txt
101,HD Camcorder,Video,189,10
102,Refrigerator,Appliance,765,2
103,MP3 Player,Audio,135,15
104,Tennis Racket,Sports,95,20
105,Laser Printer,Office,427.5,5
```

Another good use of creating a custom function is to print debug messages.

Following is a simple mydebug function:

```
$ cat function-debug.awk
{
 i=2; total=0;
 while (i <= NF) {
 mydebug("quantity is " $i);
 total = total + $i;
 i++;
 }
 print "Item", $1, ":", total, "quantities sold";
}
function mydebug ( message ) {
  printf("DEBUG[%d]>%s\n", NR, message )
}
```

Partial output is shown below.

```
$ awk -f function-debug.awk items-sold.txt
DEBUG[1]>quantity is 2
DEBUG[1]>quantity is 10
DEBUG[1]>quantity is 5
DEBUG[1]>quantity is 8
```

```
DEBUG[1]>quantity is 10
DEBUG[1]>quantity is 12
Item 101 : 47 quantities sold
DEBUG[2]>quantity is 0
DEBUG[2]>quantity is 1
DEBUG[2]>quantity is 4
DEBUG[2]>quantity is 3
DEBUG[2]>quantity is 0
DEBUG[2]>quantity is 2
Item 102 : 10 quantities sold
```

97. Language Independent Output (Internationalization)

When you write an awk script to print a report, you might specify the report header and footer information using the print command. You might define the header and footer static values in English. What if you want to execute the report output for some other language? You might end-up copying this awk script to another awk script and modify all the print statements to have the static values displayed in appropriate values.

Probably an easier way is to use internationalization where you can use the same awk script, but change the static values of the output during run time.

This technique is also helpful when you have a huge program, but you end-up changing the printed static output frequently for some reason. Or you might want the users to customize the awk output by changing the static displayed text to something of their own.

This simple example shows the 4 high level steps to implement internalization in awk.

Step 1 - Create text domain

Create a text domain and bind it to the directory where the awk program should look for the text domain. In this example it is set to the current directory.

```
$ cat iteminfo.awk
BEGIN {
 FS=","
 TEXTDOMAIN = "item"
 bindtextdomain(".")
 print _"START_TIME:" strftime("%a %b %d %H:%M:%S %Z
%Y",systime());
 printf "%-3s\t", _"Num";
 printf "%-10s\t", _"Description"
 printf "%-10s\t", _"Type"
 printf "%-5s\t", _"Price"
 printf "%-3s\n", _"Qty"
 printf
_"-----------------------------------------------------
\n"
}
{
 printf "%-3d\t%-10s\t%-10s\t$%-.2f\t%03d\n",
$1,$2,$3,$4,$5
}
```

Note: The above example has _ in front of all the strings that are allowed to be customized. Having _ (underscore) in front of a string doesn't change the way how it is printed, i.e. it will print without any issues as shown below.

```
$ awk -f iteminfo.awk items.txt
START_TIME:Sat Mar 05 09:15:13 PST 2011
Num   Description   Type      Price  Qty
```

```
-------------------------------------------------
101   HD Camcorder  Video      $210.00 010
102   Refrigerator  Appliance    $850.00 002
103   MP3 Player   Audio      $270.00 015
104   Tennis Racket  Sports     $190.00 020
105   Laser Printer  Office     $475.00 005
```

Step 2: Generate .po

Generate portable object file (extension .po) as shown below. Please note that instead of --gen-po, you can also use "-W gen-po"

```
$ gawk --gen-po -f iteminfo.awk > iteminfo.po

$ cat iteminfo.po
#: iteminfo.awk:5
msgid "START_TIME:"
msgstr ""

#: iteminfo.awk:6
msgid "Num"
msgstr ""
#: iteminfo.awk:7
msgid "Description"
msgstr ""
#: iteminfo.awk:8
msgid "Type"
msgstr ""
#: iteminfo.awk:9
msgid "Price"
msgstr ""
#: iteminfo.awk:10
msgid "Qty"
msgstr ""
```

```
#: iteminfo.awk:11
msgid
"-------------------------------------------------\
n"
""
msgstr ""
```

Now, modify this portable object file and change the message string accordingly. For example, if you want to call "Report Generated on:" (Instead of the "START_TIME:"), edit the iteminfo.po file and change the msgstr right below the msgid for "START_TIME:"

```
$ cat iteminfo.po
#: iteminfo.awk:5
msgid "START_TIME:"
msgstr "Report Generated On:"
```

Note: In this example, the rest of the msgstr strings are left empty.

Step 3: Create message object

Create message Object file (from the portable object file) using msgfmt command.

If the iteminfo.po has all the msgstr empty, it will not produce any message object file, as shown below.

```
$ msgfmt -v iteminfo.po
0 translated messages, 7 untranslated messages.
```

Since we created one message translation, it will create the messages.mo file.

```
$ msgfmt -v iteminfo.po
1 translated message, 6 untranslated messages.
$ ls -1 messages.mo
messages.mo
```

Copy this message object file to the message directory that you should create under current directory.

```
$ mkdir -p en_US/LC_MESSAGES

$ mv messages.mo en_US/LC_MESSAGES/item.mo
```

Note: The destination file name should match the name we gave in the TEXTDOMAIN variable of the original awk file. TEXTDOMAIN = "item"

Step 4: Verify the message

Now you see that it doesn't display "START TIME:" anymore. It should the translated string "Report Generated On:" in the output.

```
$ gawk -f iteminfo.awk items.txt
Report Generated On:Sat Mar 05 09:19:19 PST 2011
Num   Description   Type      Price  Qty
-------------------------------------------------------
101   HD Camcorder  Video      $210.00 010
102   Refrigerator  Appliance    $850.00 002
103   MP3 Player   Audio      $270.00 015
104   Tennis Racket  Sports     $190.00 020
105   Laser Printer  Office     $475.00 005
```

98. Two Way Communication

Awk can communication to an external process using "|&", which is two way communication.

The following simple sed example substitutes the word "Awk" with "Sed and Awk".

```
$ echo "Awk is great" | sed 's/Awk/Sed and Awk/'
Sed and Awk is great
```

To understand how the two way communication from Awk works, the following awk script simulates the above simple example using "|&"

```
$ cat two-way.awk
BEGIN {
 command = "sed 's/Awk/Sed and Awk/'"
 print "Awk is Great!" |& command
 close(command,"to");
 command |& getline tmp
 print tmp;
 close(command);
}

$ awk -f two-way.awk
Sed and Awk is Great!
```

In the above example:

- command = "sed 's/Awk/Sed and Awk/'" -- This is the command to which we are going to establish the two way communication from awk. This is a simple sed substitute command, that will replace "Awk" with "Sed and Awk".
- print "Awk is Great!" |& command -- The input to the command. i.e. The input to the sed substitute command is "Awk is Great!". The "|&" indicates that it is a two way communication. The input to the command on the right side to the "|&" comes from the left side.
- close(command,"to") - Once the process is executed, you should close the "to" process.
- command |& getline tmp - Now that the process is completed, it is time to get the output of the process using the getline. The output of the previously executed command will now be stored in the variable "tmp".
- print tmp - This prints the output.
- close(command) - Finally, close the command.

Two way communication can come-in handy when you rely heavily on output from external programs.

99. System Function

You can use the system built-in function to execute system commands. Please note that there is a difference between two way communication and system command.

In "|&", you can pass the output of any awk command as input to an external command, and you can receive the output from the external command in your awk program (basically it is two way communication).

Using the system command, you can pass any string as a parameter, which will get executed exactly as given in the OS command line, and the output will be returned (which is not same as the two way communication).

The following are some simple examples of calling pwd and date command from awk:

```
$ awk 'BEGIN { system("pwd") }'
/home/ramesh

$ awk 'BEGIN { system("date") }'
Sat Mar 5 09:19:47 PST 2011
```

When you are executing a long awk program, you might want it to send an email when the program starts and when it ends. The following example shows how you can use system command in the BEGIN and END block to send you an email when it starts and completes.

```
$ cat system.awk
BEGIN {
 system("echo 'Started' | mail -s 'Program system.awk
started..' ramesh@thegeekstuff.com");
}
{
 split($2,quantity,",");
```

```
 total=0;
 for (x in quantity)
 total=total+quantity[x];
 print "Item", $1, ":", total, "quantities sold";
}
END {
 system("echo 'Completed' | mail -s 'Program system.awk
completed..' ramesh@thegeekstuff.com");
}

$ awk -f system.awk items-sold.txt
Item 101 : 2 quantities sold
Item 102 : 0 quantities sold
Item 103 : 10 quantities sold
Item 104 : 2 quantities sold
Item 105 : 10 quantities sold
```

100. Timestamp Functions

These are available only in GAWK.

As you see from the example below, systime() returns the time in POSIX epoch time, i.e. the number of seconds elapsed since January 1, 1970.

```
$ awk 'BEGIN { print systime() }'
1299345651
```

The systime function becomes more useful when you use the strftime function to convert the epoch time to a readable format.

The following example displays the current timestamp in a readable format using systime and strftime function.

```
$ awk 'BEGIN { print strftime("%c",systime()) }'
```

```
Sat 05 Mar 2011 09:21:10 AM PST
```

The following awk script shows various possible date formats.

```
$ cat strftime.awk
BEGIN {
 print "--- basic formats --"
 print strftime("Format 1: %m/%d/%Y %H:%M:
%S",systime())
 print strftime("Format 2: %m/%d/%y %I:%M:%S
%p",systime())
 print strftime("Format 3: %m-%b-%Y %H:%M:
%S",systime())
 print strftime("Format 4: %m-%b-%Y %H:%M:%S
%Z",systime())
 print strftime("Format 5: %a %b %d %H:%M:%S %Z
%Y",systime())
 print strftime("Format 6: %A %B %d %H:%M:%S %Z
%Y",systime())
 print "--- quick formats --"
 print strftime("Format 7: %c",systime())
 print strftime("Format 8: %D",systime())
 print strftime("Format 8: %F",systime())
 print strftime("Format 9: %T",systime())
 print strftime("Format 10: %x",systime())
 print strftime("Format 11: %X",systime())
 print "--- single line format with %t--"
 print strftime("%Y %t%B %t%d",systime())
 print "--- multi line format with %n --"
 print strftime("%Y%n%B%n%d",systime())
}

$ awk -f strftime.awk
--- basic formats --
Format 1: 03/05/2011 09:26:03
```

```
Format 2: 03/05/11 09:26:03 AM
Format 3: 03-Mar-2011 09:26:03
Format 4: 03-Mar-2011 09:26:03 PST
Format 5: Sat Mar 05 09:26:03 PST 2011
Format 6: Saturday March 05 09:26:03 PST 2011
--- quick formats --
Format 7: Sat 05 Mar 2011 09:26:03 AM PST
Format 8: 03/05/11
Format 8: 2011-03-05
Format 9: 09:26:03
Format 10: 03/05/2011
Format 11: 09:26:03 AM
--- single line format with %t--
2011      March       05
--- multi line format with %n --
2011
March
05
```

Following are the various time format identifiers you can use in the strftime function. Please note that all the abbreviations shown below depend on your locale setting. These examples are shown for English (en).

Basic Time Formats:

Format Identifier	Description
%m	Month in two number format. January is shown as 01
%b	Month abbreviated. January is shown as Jan
%B	Month displayed fully. January is shown as January.
%d	Day in two number format. 4th of the month is shown as 04.

%Y	Year in four number format. For example: 2011
%y	Year in two number format. 2011 is shown as 11.
%H	Hour in 24 hour format. 1 p.m is shown as 13
%I	Hour in 12 hour format. 1 p.m is shown as 01.
%p	Displays AM or PM. Use this along with %I 12 hour format.
%M	Minute in two character format. 9 minute is shown as 09.
%S	Seconds in two character format. 5 seconds is shown as 05
%a	Day of the week shown in three character format. Monday is shown as Mon.
%A	Day of the week shown fully. Monday is shown as Monday.
%Z	Time zone. Pacific standard time is shown as PST.
%n	Displays a new line character
%t	Displays a tab character

Quick Time Formats:

Format Identifier	Description
%c	Displays the date in current locale full format. For example: Fri 11 Feb 2011 02:45:03 AM PST
%D	Quick date format. Same as %m/%d/%y
%F	Quick date format. Same as %Y-%m-%d
%T	Quick time format. Same as %H:%M:%S
%x	Date format based on your locale.
%X	Time format based on your locale.

101. getline Command

As you already know, the body block of an awk script gets executed once for every line in the input file. You don't have any control over it, as awk does it automatically.

However using the getline command, you can control the reading of lines from the input-file (or from some other file). Note that after getline is executed, the awk script sets the value of NF, NR, FNR, and $0 built-in variables appropriately.

Simple getline

```
$ awk -F"," '{getline; print $0;}' items.txt
102,Refrigerator,Appliance,850,2
104,Tennis Racket,Sports,190,20
105,Laser Printer,Office,475,5
```

When you just specify getline in the body block, awk reads the next line from the input-file. In this example, the 1st statement in the body block is getline. So, even though awk already read the 1st line from the input-file, getline reads the next line, as we are explicitly requesting the next line from the input-file. So, executing 'print $0' after getline makes awk print the 2nd line.

Here is how it works:

- At the beginning of the body block, before executing any statement, awk reads the 1st line of the items.txt and stores it in $0
- getline - we are forcing awk to read the next line from the input file and store it in the built-in $0 variable.
- print $0 - since the 2nd line is read into $0, print $0 will print the 2nd line (And not the 1st line).
- The body block continues in the same way for rest of the lines in the items.txt and prints only the even numbered lines.

getline to a variable

You can also get the next line from the input file into a variable (instead of reading it to $0).

The following example prints only the odd numbered lines.

```
$ awk -F"," '{getline tmp; print $0;}' items.txt
101,HD Camcorder,Video,210,10
103,MP3 Player,Audio,270,15
105,Laser Printer,Office,475,5
```

Here is how it works:

- At the beginning of the body block, before executing any statement, awk reads the 1st line of the items.txt and stores it in $0
- getline tmp - We are forcing awk to read the next line from the input file and store it in the tmp variable.
- print $0 - $0 still contains the 1st line, as "getline tmp" didn't overwrite the value of $0. So, print $0 will print the 1st line (and not the 2nd line).
- The body block continues in the same way for rest of the lines in the items.txt and prints only the odd numbered lines.

The following example prints both $0 and tmp. As you see below, $0 contains the odd numbered lines and tmp contains the even numbered lines.

```
$ awk -F"," '{getline tmp; print "$0->", $0; print
"tmp->", tmp;}' items.txt
$0-> 101,HD Camcorder,Video,210,10
tmp-> 102,Refrigerator,Appliance,850,2
$0-> 103,MP3 Player,Audio,270,15
tmp-> 104,Tennis Racket,Sports,190,20
$0-> 105,Laser Printer,Office,475,5
tmp-> 104,Tennis Racket,Sports,190,20
```

getline from a different file

The previous two examples read the line from the given input-file itself. Using getline you can also read lines from a different file (than the current input-file) as shown below.

Switch back and forth between two files, printing lines from each.

```
$ awk -F"," '{print $0; getline < "items-sold.txt";
print $0;}' items.txt
101,HD Camcorder,Video,210,10
101 2 10 5 8 10 12
102,Refrigerator,Appliance,850,2
102 0 1 4 3 0 2
103,MP3 Player,Audio,270,15
103 10 6 11 20 5 13
104,Tennis Racket,Sports,190,20
104 2 3 4 0 6 5
105,Laser Printer,Office,475,5
105 10 2 5 7 12 6
```

Here is how it works:

- At the beginning of the body block, before executing any statement, awk reads the 1st line of items.txt and stores it in $0
- print $0 - Prints the 1st line from items.txt
- getline < "items-sold.txt" - Reads the 1st line from items-sold.txt and stores it in $0.
- print $0 - Prints the 1st line from items-sold.txt (not from items.txt)
- The body block continues in the same way for the rest of the lines in items.txt and items-sold.txt

getline from a different file to a variable

Rather than reading both files into $0, you can also use the "getline var" format to read lines from a different file into a variable.

Switch back and forth between two files, printing lines from each (using tmp var).

```
$ awk -F"," '{print $0; getline tmp < "items-sold.txt";
print tmp;}' items.txt
101,HD Camcorder,Video,210,10
101 2 10 5 8 10 12
102,Refrigerator,Appliance,850,2
102 0 1 4 3 0 2
103,MP3 Player,Audio,270,15
103 10 6 11 20 5 13
104,Tennis Racket,Sports,190,20
104 2 3 4 0 6 5
105,Laser Printer,Office,475,5
105 10 2 5 7 12 6
```

This is identical to the previous example except that it stores the lines from the second file in the variable tmp.

getline to execute external command

You can also use getline to execute a UNIX command and get its output.

The following example gets the output of the date command and prints it. Please note that you should also close the command that you just executed as shown below. The output of the date command is stored in the $0 variable.

Use this method to print timestamp on your report's header or footer.

```
$ cat getline1.awk
BEGIN {
 FS=",";
 "date" | getline
 close("date")
 print "Timestamp:" $0
}
{
 if ( $5 <= 5 )
  print "Buy More: Order", $2, "immediately!"
 else
  print "Sell More: Give discount on", $2,
"immediately!"
}

$ awk -f getline1.awk items.txt
Timestamp:Sat Mar 5 09:29:22 PST 2011
Sell More: Give discount on HD Camcorder immediately!
Buy More: Order Refrigerator immediately!
Sell More: Give discount on MP3 Player immediately!
Sell More: Give discount on Tennis Racket immediately!
Buy More: Order Laser Printer immediately!
```

Instead of storing the output in the $0 variable, you can also store it in any awk variable (for example: timestamp) as shown below.

```
$ cat getline2.awk
BEGIN {
 FS=",";
 "date" | getline timestamp
 close("date")
 print "Timestamp:" timestamp
}
{
```

```
 if ( $5 <= 5 )
  print "Buy More: Order", $2, "immediately!"
 else
  print "Sell More: Give discount on", $2,
"immediately!"
}

$ awk -f getline2.awk items.txt
Timestamp:Sat Mar 5 09:38:22 PST 2011
Sell More: Give discount on HD Camcorder immediately!
Buy More: Order Refrigerator immediately!
Sell More: Give discount on MP3 Player immediately!
Sell More: Give discount on Tennis Racket immediately!
Buy More: Order Laser Printer immediately!
```

Thank You

I hope you found the ***Sed and Awk 101 Hacks*** eBook helpful.

I sincerely appreciate all the support given by you and other regular readers of my thegeekstuff.com blog.

You have encouraged me in more ways than you know.

Please use the contact form thegeekstuff.com/contact/ to send me your suggestions, or feedback, or questions on this eBook.

Ramesh Natarajan

ramesh@thegeekstuff.com