SED Explained

Part—1

UNIX-like operating systems have numerous powerful utilities accessible via the command-line and shell-scripting, which are flexible enough to operate in a variety of problematic domains. Text processing is one of these. Among the many small, beautiful and efficient tools is Sed (the stream editor). You will love it, once you master the skill of using and writing Sed scripts. This article starts with its basics and goes on to solving different text-processing tasks.

Sed was written for UNIX by Lee E McMahon of Bell Labs in 1974, inspired by Perl's text-processing capabilities. It can be used for simple text operations as well as complex programs. Typical uses of Sed are simple text replacement, testing for sub-strings, complex text replacement with regular expressions, selective printing of text in a file, finding text described by regular expressions in a large text file, and solving text manipulation problems by using data structures like stack and queue.

Sed is available with all GNU/Linux distributions; there are also versions for Mac OS X and MS Windows. You can provide the input to Sed via a filename given as an argument, or via standard input (stdin). Let's look at a few simple uses of Sed that you can try out in a terminal window:

```
$ echo This is a line of text. | sed ‘s/text/replaced text/’
This is a line of replaced text.
```

In the above one-liner, we passed data into standard input. In ‘*s/text/replaced text/*’, the initial *s* stands for ‘substitute’; the ‘/’ is a delimiter. Substitution has two parts: the text to be found, and the replacement text. In this case, ‘*text*’ is the text to find, and ‘*replaced text*’ is what must be substituted; these are also separated by the ‘/’ delimiter.

```
$ seq 10 | sed ‘/[049]/d’
1
2
3
5
6
7
8
```

The seq command outputs a sequence of numbers, one to a line. For more information, run *man seq*. In the above code snippet, we have a match pattern—the set of numbers 0, 4 and 9—and ‘*d*’ specified for the action of deletion. Every matching line (containing any number from the set) is thus deleted—removed from the output. You can try different sequences—for example, *seq 99 120* | *sed ‘/[049]/d’*, to understand the operation.

A peek into regular expressions

‘Regular expressions’ (often shortened to ‘regex’) is a language used to represent patterns for matching text. Regular expressions are the primary text-matching schema in all text-processing tools, including Sed. If you are already familiar with these, you can skip this section.

The following table contains the basic elements, along with description and examples.

regex	Description	Example
^	The start-of-line marker.	*^tux* matches any line that starts with *tux.*
$	The end-of-line marker.	*tux$* matches any line that ends with *tux.*
.	Matches any one character.	*Hack.* matches *Hack1, Hacki* but not *Hack12, Hackil;* only one additional character matches.
[]	Matches any one of the character set inside [].	*coo[kl]* matches *cook* or *cool.*
[^]	Exclusion set: the carat negates the set of characters in the square brackets; text matching this set will *not* be returned as a match.	*9[^01]* matches *92, 93* but not *91* and *90*.
[-]	Matches any character within the range specified in [].	*[1-5]* matches any digits from *1* to *5*.
?	The preceding item *must* match one or zero times.	*colou?r* matches *color* or *colour* but not *colouur*.
+	The preceding item *must* match one or more times.	*Rollno-9+* matches *Rollno-99, Rollno-9* but not *Rollno-.*
*	The preceding item must match zero or more times.	*co*l* matches *cl, col, coool.*
()	Creates a sub-string in the regex match.	Explained below, in the section 'Sub-string match and back-referencing'.
{n}	The preceding item must match exactly n times.	*[0-9]{3}* matches any three-digit number. This can be expanded as: *[0-9][0-9][0-9].*
{n,}	Minimum number of times that the preceding item should match.	*[0-9]{2,}* matches any number that is two digits or more in length.
{n, m}	Specifies the minimum and maximum number of times the preceding item should match.	*[0-9]{2,5}* matches any number that is between two and five digits in length.
|	Alternation—one of the items on either side of | should match.	*Oct (1st|2nd)* matches *Oct 1st* or *Oct* 2nd.
\	The escape character for escaping any of the special characters given above.	*a\b* matches *a.b* but not *ajb*. The dot is not interpreted as the special 'match any one character' regex shown above, but instead a literal dot (period) ASCII character is sought to be matched. Another example: if you're searching for the US currency symbol '$', and not the end-of-line marker, you must precede it with a back-slash, like this: \$

There are a few character classes, called POSIX classes, in the format [:*name*:] that can be conveniently used, instead of spelling out the character set each time. Note that, as shown in the example column, you need to enclose the class itself in another pair of square brackets. For example:

```
$ echo -e "max\nOR\nMatrix" | sed '/[:alpha:]/d'
OR
$ echo -e "max\nOR\nMatrix" | sed '/[[:alpha:]]/d'
$
```

In the first case, the set is interpreted literally—the words '*max*' and '*matrix*' are deleted because they contain '*a*', one of the letters in the character set. In the second command, with another pair of square brackets around the class, all input lines are deleted, because all lines contain (at least one) alphabet.

Regex	Description	Example
[:alnum:]	Alphanumeric characters	*[[:alnum:]]+*
[:alpha:]	Alphabet character (lower-case and upper-case)	*[[:alpha:]]{4}*
[:blank:]	Space and tab	*[[:blank:]]**

[:digit:]	Digit	*[[:digit:]]?*
[:lower:]	Lower-case alphabet	*[[:lower:]]{5,}*
[:upper:]	Upper-case alphabet	*([[:upper:]]+)?*
[:punct:]	Punctuation	*[[:punct:]]*
[:space:]	All white-space characters including newline, carriage return, and so on.	*[[:space:]]+*

Meta-characters are a type of Perl-style regular expressions that are supported by a subset of text-processing utilities. Not all utilities will support the following notations.

Regex	Description	Example
\b	Word boundary	*\bcool\b* matches only *cool* and not *coolant*.
\B	Non-word boundary	*cool\B* matches *coolant* but not *cool*.
\d	Single digit character	*b\db* matches *b2b* but not *bcb*.
\D	Single non-digit	*b\Db* matches *bcb* but not *b2b*.
\w	Single word character (alnum and _)	*\w matches 1* or *a* but not &.
\W	Single non-word character	*\w* matches & but not *1* or *a*.
\n	Newline	*\n* matches a new line.
\s	Single white-space	*x\sx* matches *x x* but not *xx*.
\S	Single non-space	*x\Sx* matches *xkx* but not *x x*.
\r	Carriage return	*\r* matches carriage return.

The above tables can be used as a reference while constructing regular expression patterns.

Let us go through a few examples of regular expressions.

Treatment of special characters

Regular expressions use some characters such as $, ^, ., *, +, {, and } as special characters. But what if we want to use these characters as non-special characters (normal text character)? Let's see an example. regex: [a-z]*.[0-9]

How is this interpreted? It can be zero or more [a-z] ([a-z]*), then any one character (.), and one character in the set [0-9] such that it matches abcdeO9. It can also be interpreted as one of [a-z], then a character *, then a character . (period), and a digit such that it matches x*.8. In order to overcome this problem, we precede the character with a forward slash "\" (doing this is called "escaping the character"). The characters such as * that have multiple meanings are prefixed with "\" to make them into a special meaning or to make them non special. Whether special characters or non-special characters are to be escaped varies depending on the tool that you are using.

In short the term special meaning means that a character is considered as meaningful interpretation other than its character ascii value. For example a* means a, aa, aaa... Here * has special meaning since its not iterpreted as ascii character '*'. Certain characters to be escaped using \ to give special meaning while some others are by default taken as special meaning (Eg. *). To use it as regular ascii meaning, it should be escaped. Here is small list of characters having special meaning with escaping.

Special meaning:

\+, \{, \}, \(, \), \?

Characters that are by default special (You need to escape these inorder to use as regular ascii):

*, ., ^, $, [,]

To match any line containing ONLY the word test, and no other characters on it, use '*^test$*'. This is interpreted as 'start of line marker' followed by '*test*' followed by 'end of line marker'.

Another good example is to extract e-mail addresses from the given text. An e-mail address has the format *username@domain.root*. We can formulate the regular expression as:

[A-Za-z0-9.]+@[A-Za-z0-9.]+\.[a-zA-Z]{2,4}. The *[A-Za-z0-9.]+* before @ states that the given character class should occur one or more times, just as after the @. At the end of the e-mail address, we have the TLD (top-level domain), which can be two to four characters in length, as specified by *{2,4}*.

Points to remember

In Sed, 'one or more' (+) is always prefixed with the backslash escape character if it does not occur after a character set/class, while 'zero or more' (*) is not thus prefixed.

We can do an inverse match using /*PATTERN/!{statements}*. That is, include the bang (!) after the slash after PATTERN.

Sed essentials

Learning Sed involves understanding the basic concepts, and gaining experience with practice. Hence, I will go through different usage contexts and the basic concepts in the following section. Real scripts where Sed can be used will also be explained.

Text replacement with Sed

Sed processes the text input line by line, by default. To perform stream editing on a file, specify the operation, followed by the filename. For example:

```
$ sed 's/PATTERN/REPLACEMENT TEXT/' filename
```

We can also provide the contents of a file to Sed on its standard input, as follows:

```
$ cat filename | sed 's/pattern/replace_string/'
```

To write the output of the Sed command to a file, use shell redirection, as follows:

```
$ sed 's/MATCH/REPLACE/' filename > output.txt
```

To save the changes to the input file itself, use the -i option:

```
$ sed -i 's/MATCH/REPLACE/' filename
```

In text replacement operations such as the above example, only the first occurrence of the matched text in a line is replaced. If you have multiple occurrences of the search pattern on a single line, and need to replace all occurrences, append the '*g*' (global replacement) switch as follows:

```
$ sed 's/MATCH/REPLACE/g' filename
```

However, sometimes we may not need to replace the first '*N*' occurrences, but replace only the rest of them. There is an inbuilt option to ignore the first 'N' occurrences and replace from the 'N+1th' occurrence onwards. This is easiest to explain with examples, which make it very clear:

```
$ echo this thisthisthis | sed 's/this/THIS/2g'
this THISTHISTHIS
$ echo this thisthisthis | sed 's/this/THIS/3g'
this thisTHISTHIS
$ echo this thisthisthis | sed 's/this/THIS/4g'
this thisthisTHIS
```

Deleting lines based on how they match

To remove lines that are matched by a regular expression pattern, use the 'd' suffix. For example, *$ sed '/^$/d' filename* removes all blank lines present in the file from the output.

Printing only pattern-matched lines

We have seen different examples like the substitution of text. In the above examples, Sed makes some changes to matched lines, and prints them—or deletes them. What if we want to print only lines that match the pattern, and print only changed text? To print only matched lines, use the *-n* (no output/silent) flag, and attach a *p* suffix after the pattern, to cause printing of lines that match. Let's take a sample file named *paragraph.txt*:

```
1. introduction
some text goes here. 2nd line.
2. next heading
some text goes here. 2nd line. 3rd line.
```

On this file, we run the following command:

```
$ sed -n '/[0-9]\+\./p' paragraph.txt
1. introduction
2. next heading
```

Thus, we printed only titles (lines with numbering) from the file.

If we want the whole input text printed, with matched text printed twice, we can use */p* without the *-n* flag. Try it out: *$ seq 20 | sed '/0/p'*. Lines with 10 and 20 (containing 0) are printed twice.

Matched string reference in replacement text

In Sed, the first block of substitution is the pattern to be matched. Sometimes we need to replace matched text with some additional characters. Here, we can reference the matched text using the '&' notation in the replacement block. For example, to add a space between every character in a line, use the following code:

```
$ echo this is a line of text | sed 's/./& /g'
t h i s   i s   a   l i n e   o f   t e x t
```

Each matched character is replaced with itself plus a space. The */g* option applies this to every character matched in the line.

Sub-string match and back-referencing

We have seen how to reference and use the entire matched pattern in the replacement text. Now, how can we use sub-strings of the matched pattern in the replacement text? For example, we have (matched text): *Lakshman Sarath Mr*. This is last name, first name, and salutation. Let's suppose we want to rewrite this line to read with the salutation first, then the first name, and then the last name (think of several hundred such names in a file, which are in the 'wrong' order). If we can match each word as a sub-string, we can reference these sub-strings individually, and can specify a new order for them in the replacement text.

Sub-strings can be specified by using grouping operators—parentheses—like '\(*SUBSTR*\)'. They are referenced (in replacement text) by *N*, where *N* is the *N*th sub-string found. The grouping operators '(' and ')' must be prefixed with the \ escape character. Let us use back-referencing to solve our problem of re-ordering the parts of a name:

```
$ echo Lakshman Sarath Mr | sed 's/\(\w\+\) \(\w\+\) \(\w\+\)/\3
\2 \1/'
Mr Sarath Lakshman
```

The above (and other Sed scripts) may look complex and hard to understand initially. But once you break it up and analyse it, it is simple to understand. \w is the regex used to match character (refer to Table 3 above) and \+ to specify one or more characters. Each of the words is grouped with \(and

\). In replacement text, \3 is the third sub-string/group found; \2 the second, and \1 the first match.

Quoting

For Sed scripts, we can use single quotes or double quotes. When single quotes are used (*sed 'STATEMENTS' file*), it will not expand shell variables. But with double quotes (*Sed "STATEMENTS" file*) Bash will expand shell variables before passing the arguments to Sed. Therefore, we should use double quoting if we need to pass values into the Sed script from Bash. For example:

```
$ replace_txt="REPLACED"
$ match="MATCH"
$ echo This is a MATCH text | sed "s/$match/$replace_txt/"
This is a REPLACED text
```

Combining multiple Sed statements

Combining multiple Sed commands using a pipe (*sed 'expression'* | *sed 'expression'*) can be replaced with *sed 'expression; expression'*.

Restrict range of affected lines with patterns and line numbers

We can specify the range of line numbers for which text manipulation operations are to be performed. It is also possible to specify a range of lines by specifying a start-line pattern and an end-line pattern. To specify a range of line numbers, use something like what's shown below:

```
$ seq 30 | sed '10,20 s/[0-9]$//'
```

In the output, you can see that in lines 10-20, the last digit has been removed (replaced with an empty string). You can specify the ending line number to be the last line in the file by using a *$* character as follows (*10,$*):

```
$ seq 30 | sed '10,$ s/[0-9]$//'
```

Now, let us look at how to specify ranges of lines by using patterns. Let's suppose that I need to apply a text operation to a range of text that is between one line with the word 'start', and another with the word 'end', in a sample file (call it eg.txt) with these contents:

```
hello
next
start
line1
line2
end
next
```

We run this command, and observe in the output that only the desired lines have been modified:

```
$ sed '/start/, /end/ s/.*/& modified/' eg.txt
hello
next
a line with start  modified
line1 modified
line2 modified
a line with end modified
next
```

Here, */START_PAT/, /END PAT/* is used to identify the text range to be operated on. You can mix line numbers and text matches while specifying ranges; */START_PAT/, $* will operate on text between *START_PAT* and the end of file, while *1, /END_PAT/* will operate from the first line to a line containing *END_PAT*.

Grouping commands into subgroups

Grouping Sed statements according to contexts is a very useful operation. We can perform operations by matching some pattern of the input, then apply another text operation after the pattern is matched, after which we can again operate on a smaller subgroup, and so on. For example, here is a problem: in the file *eg2.txt* (contents listed below) I may change *only* text between the 3rd and 8th lines. In that range, I must operate on (the range of) all lines between the patterns 'single' and 'double'. In this nested range, I must replace all the lines' text with #:

```
eg2.txt:
first line
2nd line
3rd line
a word single
next line
next lines
double
test line
last line
```

After creating the text file, we run the following command on it and observe that the output satisfies the constraints and does the required operation:

```
$ sed '3,8 { /single/,/double/ { s/.*/#/ } }' eg2.txt
first line
2nd line
3rd line
#
#
#
#
test line
last line
```

While grouping, we can also negate the match condition for a group by suffixing the text match condition with the ! operator, like this: /*TEXT*/ !{ *statements* }. Such a group is executed whenever the pattern does *not* match /*TEXT*/.

Read and write from files using Sed

We can read or write to an intermediate file by specifying file-names and match/replace text in the *sed* script. Let us look at an example. We set up files as follows (file name followed by content):

file1.txt:

```
A line from file1
```

file2.txt:

```
A line from file2
```

list.txt:

```
FILE1
```

The file *list.txt* is our input file. While reading from it, if a particular match occurs, and if we wish to read in the contents of another file to replace the matched pattern, we can issue the following code:

```
$ cat list.txt | sed ‘/FILE1/ { r file1.txt
d }
/FILE2/ { r file2.txt
d }’
```

r filename is the command used. Note that when *read (r)* is used after the filename, there should be a new line. That is why, instead of continuing the command on the same line, later commands are given on new lines. Here, *d* is used to remove the matched text read from *list.txt*. Experiment with this: remove the ‘d’ commands; edit *list.txt* and insert ‘FILE2’ on the first line (before the ‘FILE1’ line)... run the command after each of your changes and see what effect it has.

We can write conditionally matched patterns or replaced text to a file, as follows:

```
$ seq 100 | sed -n ‘30,33 { w extracted.txt
}’
$ cat extracted.txt
30
31
32
33
```

Like the read command *‘r’* operates on files, we can use the write command *‘w’* to write matched text to a specified file. In the above sed script, it matches line numbers 30 to 33, which are written into a file named *extracted.txt*. As before, the file-name is immediately followed by a newline. The *-n* option is given to prevent Sed from printing input lines to standard output.

Quit operator

Sed reads input line by line and processes it. We can stop the execution of Sed at some particular matching line and quit, so that the following lines will not be read and processed, using the ‘q’ command. For example, we quit this script after line 3:

```
$ seq 10 | sed ‘3 q’
1
2
3
```

In this article, we have come across basic concepts and building blocks that can be used to solve complex problems. By making proper use of what we have learned in this article, it is possible to break up text-processing tasks into smaller, simpler tasks, and solve them. Sed has more interesting features and data structures that add more text-processing capabilities. We will learn about data structures, operators and a few text-processing problems in the second part of this article, next month. Happy hacking, till we meet again!

By: Sarath Lakshman

The author is a hacktivist of Free and Open Source Software from Kerala. He loves working on the GNU/Linux environment and contributes to the Pardus Linux distro project. He has recently authored the Linux Shell Scripting Cookbook, which gives insights into shell scripting using 119 ‘recipes’. He can be reached via his website *http://www.sarathlakshman.com.*