Sed One-Liners Explained

by

@pkrumins
Peteris Krumins
peter@catonmat.net
http://www.catonmat.net
good coders code, great reuse

Contents

Preface

Thanks!

Thank you for purchasing my "Sed One-Liners Explained" e-book! This is my second e-book and I based it on the "Famous Sed One-Liners Explained" article series that I wrote on my www.catonmat.net blog. I went through all the one-liners in the articles, improved them, added 8 new ones (bringing the total count to 100), fixed a lot of mistakes, added an Introduction to sed one-liners and three new chapters. The three new chapters are Sed commands that summarizes all the sed commands, Sed addresses and ranges that summarizes the sed ranges, and Debugging sed scripts with sed-sed that shows how useful the sed-sed utility is.

You might wonder why I called the article series "famous"? Well, because I based the articles on the famous sed1line.txt file by Eric Pement. This file has been circulating around Unix newsgroups and forums for years and it's very popular among Unix programmers. That's how I actually really learned sed myself. I went through all the one-liners in this file, tried them out and endeavored to understand exactly how they work. Then I decided that it would be a good idea to explain them on my blog, which I did, and after that I thought, why not turn it into a book? That's how I ended up writing this book.

As I mentioned, this is my second e-book. My first e-book is "Awk One-Liners Explained" which I based on Eric Pement's awk1line.txt file. I really learned Awk from this file, too.

My third e-book is now in development. It's "Perl One-Liners Explained". The Perl book will be based on my "Famous Perl One-Liners Explained" article series. I was so inspired by sed1line.txt and awk1line.txt files that I decided to create my own perl1line.txt file. I am still working on it but I am one chapter away from finishing it. If you're interested, subscribe to my blog, follow me on Twitter or follow me on Google+. That

way you'll be the first to know when I publish perl1line.txt and my next book!

Credits

I'd like to thank Eric Pement who made the famous sed1line.txt file that I learned Sed from and that I based this book on. I'd also like to thank waldner and DJ Mills from #sed channel on FreeNode IRC network for always helping me with sed. Also thanks goes to Madars Virza for the book review, Przemysław Pawełczyk for teaching me LaTeX, Abraham Alhashmy for advice on how to improve the design of the book, Blake Sitney for proof-reading the book, everyone who commented on my blog while I was writing the sed one-liners article series, and everyone else who helped me with sed and this book.

One

Introduction

1.1 Sed One-Liners

Mastering sed can be reduced to understanding and manipulating *the four spaces* of sed. These four spaces are:

- Input Stream
- Pattern Space
- Hold Buffer
- Output Stream

Think about the spaces this way – sed reads the input stream and produces the output stream. Internally it has the pattern space and the hold buffer. Sed reads data from the input stream until it finds the newline character `\n`. Then it places the data read so far, without the newline, into the pattern space. Most of the sed commands operate on the data in the pattern space. The hold buffer is there for your convenience. Think about it as temporary buffer. You can copy or exchange data between the pattern space and the hold buffer. Once sed has executed all the commands, it outputs the pattern space and adds a `\n` at the end.

It's possible to modify the behavior of sed with the `-n` command line switch. When `-n` is specified, sed doesn't output the pattern space and you have to explicitly print it with either `p` or `P` commands.

Let's look at several examples to understand the four spaces and sed. You'll learn much more about the sed commands and spaces as you work through the chapters of the book. These are just examples to illustrate what sed looks like and what it's all about.

Here is the simplest possible sed program:

```
sed 's/foo/bar/'
```

This program replaces text "foo" with "bar" on every line. Here is how it works. Suppose you have a file with these lines:

```
abc
foo
123-foo-456
```

Sed opens the file as the input stream and starts reading the data. After reading "abc" it finds a newline \n. It places the text "abc" in the pattern space and now it applies the s/foo/bar/ command. Since we have "abc" in the pattern space and there is no "foo" anywhere, sed does nothing to the pattern space. At this moment sed has executed all the commands (in this case just one). The default action when all the commands have been executed is to print the pattern space, followed by newline. So the output from the first line is "abc\n".

Now sed reads in the second line "foo" and executes s/foo/bar/. This replaces "foo" with "bar". The pattern space now contains just "bar". The end of the script has been reached and sed prints out the pattern space, followed by newline. The output from the second line is "bar\n".

Now the 3rd line is read in. The pattern space is now "123-foo-456". Since there is "foo" in the text, the s/foo/bar/ is successful and the pattern space is now "123-bar-456". The end is reached and sed prints the pattern space. The output is "123-bar-456\n".

All the lines of the input have been read at this moment and sed exits. The output from running the script on our example file is:

```
abc
bar
123-bar-456
```

In this example we never used the hold buffer because there was no need for temporary storage.

Before we look at an example with temporary storage, let's take a look at three command line switches – -n, -e and -i. First -n.

If you specify -n to sed, like this:

```
sed -n 's/foo/bar/'
```

Then sed will no longer print the pattern space when it reaches the end of the script. So if you run this program on our sample file above, there will be no output. You must use sed's `p` command to force sed to print the line:

```
sed -n 's/foo/bar/; p'
```

As you can see, sed commands are separated by the `;` character. You can also use `-e` switch to separate the commands:

```
sed -n -e 's/foo/bar/' -e 'p'
```

It's the same as if you used `;`. Next, let's take a look at the `-i` command line argument. This one forces sed to do in-place editing of the file, meaning it reads the contents of the file, executes the commands, and places the new contents back in the file.

Here is an example. Suppose you have a file called `"users"`, with the following content:

```
pkrumins:hacker
esr:guru
rms:geek
```

And you wish to replace the `":"` symbol with `";"` in the whole file. Then you can do it as easily as:

```
sed -i 's/:/;/' users
```

It will silently execute the `s/:/;/` command on all lines in the file and do all substitutions. **Be very careful when using `-i` as it's destructive and it's not reversible!** It's always safer to run sed without `-i`, and then replace the file yourself.

Actually, before we look at the hold buffer, let's take a look at addresses and ranges. Addresses allow you to restrict sed commands to certain lines, or ranges of lines.

The simplest address is a single number that limits sed commands to the given line number:

```
sed '5s/foo/bar/'
```

This limits the `s/foo/bar/` only to the 5th line of file or input stream. So if there is a `"foo"` on the 5th line, it will be replaced with `"bar"`. No other lines will be touched.

The addresses can be also inverted with the `!` after the address. To match all lines that are not the 5th line (lines 1-4, plus lines 6-...), do this:

```
sed '5!s/foo/bar/'
```

The inversion can be applied to any address.

Next, you can also limit sed commands to a range of lines by specifying two numbers, separated by a comma:

```
sed '5,10s/foo/bar/'
```

In this one-liner the `s/foo/bar/` is executed only on lines 5 – 10, inclusive. Here is a quick, useful one-liner. Suppose you want to print lines 5 – 10 in the file. You can first disable implicit line printing with the `-n` command line switch, and then use the `p` command on lines 5 – 10:

```
sed -n '5,10p'
```

This will execute the `p` command only on lines 5 – 10. No other lines will be output. Pretty neat, isn't it?

There is a special address `$` that matches the last line of the file. Here is an example that prints the last line of the file:

```
sed -n '$p'
```

As you can see, the `p` command has been limited to `$`, which is the last line of input.

Next, there is also a single regular expression address match like this `/regex/`. If you specify a regex before a command, then the command will only get executed on lines that match the regex. Check this out:

```
sed -n '/a+b+/p'
```

Here the `p` command will get called only on lines that match `a+b+` regular expression, which means one or more letters "`a`" followed by one or more letters "`b`". For example, it prints lines like "`ab`", "`aab`", "`aaabbbbb`", "`foo-123-ab`", etc.

There is also an expression to match a range between two regexes. Here is an example,

```
sed '/foo/,/bar/d'
```

This one-liner matches all lines between the first line that matches "`/foo/`" regex and the first line that matches "`/bar/`" regex, inclusive. It applies the `d` command that stands for delete. In other words, it deletes a range of lines between the first line that matches "`/foo/`" and the first line after "`/foo/`" that matches "`/bar/`", inclusive.

Now let's take a look at the hold buffer. Suppose you have a problem where you want to print the line before the line that matches a regular expression. How do you do this? If sed didn't have a hold buffer, things would be tough, but with hold buffer we can always save the current line to the hold buffer, and then let sed read in the next line. Now if the next line matches the regex, we would just print the hold buffer, which holds the previous line. Easy, right?

The command for copying the current pattern space to the hold buffer is `h`. The command for copying the hold buffer back to the pattern space is `g`. The command for exchanging the hold buffer and the pattern space is `x`. We just have to choose the right commands to solve this problem. Here is the solution:

```
sed -n '/regex/{x;p;x}; h'
```

It works this way – every line gets copied to the hold buffer with the `h` command at the end of the script. However, for every line that matches the `/regex/`, we exchange the hold buffer with the pattern space by using the `x` command, print it with the `p` command, and then exchange the buffers back, so that if the next line matches the `/regex/` again, we could print the current line.

Also notice the command grouping. Several commands can be grouped and executed only for a specific address or range. In this one-liner the command group is `{x;p;x}` and it gets executed only if the current line matches `/regex/`.

Note that this one-liner doesn't work if it's the first line of the input matches `/regex/`. To fix this, we can limit the `p` command to all lines that are not the first line with the `1!` inverted address match:

```
sed -n '/regex/{x;1!p;x}; h'
```

Notice the `1!p`. This says – call the `p` command on all the lines that are not the 1st line. This prevents anything to be printed in case the first line matches `/regex/`.

I think that this is a great introduction to sed one-liners that covers the most important concepts in sed. Overall this book contains exactly 100 well-explained one-liners. Once you work through them, you'll have rewired your brain to "think in sed". In other words, you'll have learned how to manipulate the pattern space, the hold buffer and you'll know when to print the data to get the results that you need. Enjoy this book!

Two

Line Spacing

2.1 Double-space a file

```
sed G
```

This sed one-liner uses the `G` command. The `G` command appends a newline `\n` followed by the contents of the hold buffer to the pattern space. In this one-liner the hold buffer is empty all the time (only three commands - `h`, `H` and `x` modify the hold buffer), so we end up simply appending a newline to the pattern space for every line. Once all the commands have been processed (in this case just the `G` command), sed sends the contents of pattern space to the output stream followed by a newline. So there we have it, each line now ends with two newlines – one added by the `G` command and the other implicitly by sed's printing mechanism. File has been double spaced.

2.2 Double-space a file which already has blank lines in it

```
sed '/^$/d;G'
```

Sed allows the commands to be restricted only to certain lines. This one-liner operates only on lines that match the regular expression `/^$/`. But which are those? Those are the empty lines, because only the empty lines have a property that the beginning of the line `^` is the same as the end of the line `$`. Remember that before doing the regular expression match sed pushes the input line to pattern space. When doing it, sed strips the trailing