FREE eBook

LEARNING

sed

Free unaffiliated eBook created from
Stack Overflow contributors.

#sed

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sed

It is an unofficial and free sed ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sed.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with sed

Remarks

References

- FreeBSD sed man-page
- NetBSD sed man-page
- OpenBSD sed man-page
- Illumos sed man-page
- macOS (OS X) man-page
- Plan9 sed man-page
- GNU sed online manual

Versions

Name	Initial Release	Version	Release Date
POSIX sed	1992	IEEE Std 1003.1, 2013 Edition	2013-04-19
BSD sed	1992	FreeBSD 10.3 / NetBSD 7.0 / OpenBSD 5.9	2016-04-04
GNU sed	1989	4.2.2	2012-12-22

Examples

Hello World

One of the most common use of Sed is text substitution that can be achieved with the `s` command.

In a terminal, type `echo "Hello sed" | sed 's/sed/World/'` and press `Enter`:

```
$ echo "Hello sed" | sed 's/sed/World/'
Hello World
```

"Hello World" should be output to the terminal.

The string "Hello, sed" is sent via pipe as input to the sed command that replace the word `sed` with `World`.

The syntax of a basic substitution command is `s` followed by the string or pattern to be searched and the substitution text. `s` command and strings are separated with a default `/` delimiter.

Read Getting started with sed online: https://riptutorial.com/sed/topic/934/getting-started-with-sed

Chapter 2: Additional Options

Syntax

- -a - (BSD sed) Create / Truncate all files written to before processing
- -E | -r - Use Extended Regular Expressions
- -i | -I - Refer to the topic on In-Place Editing
- -l - (BSD sed) Use line-buffered output
- -l length - (GNU sed) Specify the length for `l` command line-wrapping
- -s - (GNU sed) Treat files as separate streams
- -u - Do not buffer the output
- -z - (GNU sed) Use the NUL character to separate records
- --quiet | --silent - (GNU sed) Synonyms for `-n`
- --expression=command - (GNU sed) Synonym for `-e`
- --file=command_file - (GNU sed) Synonym for `-f`
- --follow-symlinks - (GNU sed) Follow symlinks
- --in-place[=extension] - (GNU sed) Synonym for `-i`
- --line-length=length - (GNU sed) Synonym for `-l`
- --separate - (GNU sed) Synonym for `-s`
- --unbuffered - (GNU sed) Synonym for `-u`
- --null-data - (GNU sed) Synonym for `-z`
- --help - (GNU sed) Print usage
- --version - (GNU sed) Print version

Remarks

The `-E` option is to be standardized in the next major version, see the relevant issue.

Examples

Delay Creation/Truncation of Files

Files written to with the `w` command are created/truncated before any commands are run.

```
$ sed 'w created-file' < /dev/null && ls created-file && rm created-file
created-file
```

From the standard:

> Each wfile shall be created before processing begins. Implementations shall support at least ten wfile arguments in the script; the actual number (greater than or equal to 10) that is supported by the implementation is unspecified. The use of the wfile parameter shall cause that file to be initially created, if it does not exist, or shall replace the contents of an existing file.

BSD `sed` provides the `-a` option to delay creating/truncating files until they are written to with the `w` command.

```
$ if sed -a 'w created-file' < /dev/null && [ ! -e created-file ]; then
>       echo The file was not created
> fi
The file was not created
```

'l' Line-Wrapping

The length of line-wrapping when using the `l` command is implementation defined.

From the standard:

> Long lines shall be folded, with the point of folding indicated by writing a followed by a ; the length at which folding occurs is unspecified, but should be appropriate for the output device.

GNU `sed` provides the `-l` option to specify the length at which to split long lines when printing with the `l` command, defaulting to seventy characters.

```
$ yes | head -c100 | tr '\n' ' ' | sed -n l | head -n1 | wc -c
      71
$ yes | head -c100 | tr '\n' ' ' | sed -nl50 l | head -n1 | wc -c
      51
```

BSD `sed` splits long lines at the number provided by the environment variable `COLUMNS`, if `COLUMNS` is not provided then it splits at the terminal width, and if `COULMNS` is not provided and the output is not a terminal then it defaults to sixty characters.

```
$ yes | head -c100 | tr '\n' ' ' | sed -n l | head -n1 | wc -c
      61
$ yes | head -c100 | tr '\n' ' ' | COLUMNS=50 sed -n l | head -n1 | wc -c
      51
$ yes | head -c100 | tr '\n' ' ' | sed -n l | head -n1
y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y\
 y y y y y y y y y y $
```

Read Additional Options online: https://riptutorial.com/sed/topic/7922/additional-options

Chapter 3: Address and address range

Introduction

Sed commands can be specified to act only on certain lines by using *addresses* or *address ranges* .

Examples

Specific line

```
$ cat ip.txt
address
range
substitution
pattern
sample
```

- *N*th line

  ```
  $ sed -n '2p' ip.txt
  range

  $ sed '3d' ip.txt
  address
  range
  pattern
  sample
  ```

- Last line

  ```
  $ sed -n '$p' ip.txt
  sample
  ```

Specific range of lines

```
$ cat ip.txt
address
range
substitution
pattern
sample
```

- Range specified is inclusive of those line numbers

  ```
  $ sed -n '2,4p' ip.txt
  range
  substitution
  ```

```
pattern
```

- `$` can be used to specify last line. Space can be used between address and command for clarity

```
$ sed -n '3,$ s/[aeiou]//gp' ip.txt
sbstttn
pttrn
smpl
```

GNU sed

- *i*th line to *i+j*th line

```
$ sed '2,+2d' ip.txt
address
sample
```

- *i*th line and *i+j*, *i*+2*j*, *i*+3*j*, etc.

```
$ sed -n '1~2p' ip.txt
address
substitution
sample
```

Lines matching regular expression pattern

```
$ cat ip.txt
address
range
substitution
pattern
sample
Add Sub Mul Div
```

- Lines matching a pattern

```
$ sed '/add/d' ip.txt
range
substitution
pattern
sample
Add Sub Mul Div

$ sed -n '/t/p' ip.txt
substitution
pattern

$ sed -n '/[A-Z]/ s| |/|gp' ip.txt
Add/Sub/Mul/Div
```

- Range of patterns

```
$ sed -n '/add/,/sub/p' ip.txt
address
range
substitution

$ sed -n '/a/,/e/p' ip.txt
address
range
pattern
sample
```

Note

- In the second example, it matched two ranges - lines `1,2` and lines `4,5`
- See Using different delimiters on how to use other characters instead of `/` for specifying the pattern

GNU sed

- Case-insensitive match

```
$ sed -n '/add/Ip' ip.txt
address
Add Sub Mul Div

$ sed -n '/add/I,/sub/p' ip.txt
address
range
substitution
Add Sub Mul Div
```

Specifying range using both number and pattern

```
$ cat ip.txt
address
range
substitution
pattern
sample
Add Sub Mul Div
```

- Line number to line matching pattern

```
$ sed -n '2,/pat/p' ip.txt
range
substitution
pattern
```

- Line matching pattern to line number

```
$ sed '/pat/,$d' ip.txt
address
range
substitution
```

GNU sed

- Line matching pattern plus number of lines following it

```
$ sed -n '/add/I,+1p' ip.txt
address
range
Add Sub Mul Div
```

- `0` can be used as starting line number to signal end of range when pattern matches first line of input

```
$ sed -n '0,/r/p' ip.txt
address

$ sed -n '1,/r/p' ip.txt
address
range

$ sed -n '0,/u/p' ip.txt
address
range
substitution
```

Negating address range

```
$ cat ip.txt
address
range
substitution
1234
search pattern
sample
Add Sub Mul Div
```

- Deleting lines other than address specified

```
$ sed '/[0-9]/!d' ip.txt
1234

$ sed -n '/[0-9]/p' ip.txt
1234

$ sed '$!d' ip.txt
Add Sub Mul Div
```

```
$ sed -n '$p' ip.txt
Add Sub Mul Div
```

- Search and replace on lines not matching a pattern

```
$ sed '/ /! s/^/#/' ip.txt
#address
#range
#substitution
#1234
search pattern
#sample
Add Sub Mul Div

$ sed '/add/,/sub/! s/[aeiou]//gi' ip.txt
address
range
substitution
1234
srch pttrn
smpl
dd Sb Ml Dv
```

Read Address and address range online: https://riptutorial.com/sed/topic/3120/address-and-address-range

Chapter 4: Advanced sed commands

Examples

Insert a new line before matching pattern - using eXchange

Given a file file.txt with the following content:

```
line 1
line 2
line 3
```

You can add a new line using below command

```
sed '/line 2/{x;p;x;}' file.txt
```

The above command will output

```
line 1

line 2
line 3
```

Explanation:

`x` command is eXchange. sed has a buffer that you can use to store some lines. This command exchanges this buffer with current line (so current line goes to this buffer and buffer content becomes current line).

`p` command prints current line.

Read Advanced sed commands online: https://riptutorial.com/sed/topic/3946/advanced-sed-commands

Chapter 5: Append command

Examples

Insert line after first match

Given a file file.txt with the following content:

```
line 1
line 2
line 3
```

You can add a new line after first matching line with the `a` command.

For portable use the `a` command must be followed immediately by an escaped newline, with the text-to-append on its own line or lines.

```
sed '
/line 2/a\
new line 2.2
' file.txt
```

GNU sed

Some versions of `sed` allow the text-to-append to be inline with the `a` command:

```
sed '/line 2/a new line 2.2' file.txt
```

The above commands will output:

```
line 1
line 2
new line 2.2
line 3
```

Read Append command online: https://riptutorial.com/sed/topic/2835/append-command

Chapter 6: Branching Operation

Introduction

The branching operation of `sed` can help control the flow of the program.

Examples

Do multiple line regexp replacing with unconditional branch

Assume that I have a file named `in.txt`:

```
$ cat in.txt
a
b
a
c
a
d
```

I only want to replace the `a\nc` with `deleted`, but not `a\nb` or `a\nd`.

```
$ sed -e ':loop          # create a branch/label named `loop`
 $!{
 N                       # append the next line of input into the pattern space
 /\n$/!b loop            # If it is not the last line go to the `loop` branch again
 }
 s/a\nc/"deleted"/' in.txt    # do replacing in the pattern space

a
b
"deleted"    # see! succeed
a
d
```

Read Branching Operation online: https://riptutorial.com/sed/topic/8821/branching-operation

Chapter 7: BSD/macOS Sed vs. GNU Sed vs. the POSIX Sed specification

Introduction

To quote from @SnoringFrog's topic-creation request:

"One of the biggest gotchas using sed is scripts that fail (or succeed in an unexpected way) because they were written for one and not the other. Simple run-down of the more major differences would be good."

Remarks

***macOS* uses the *BSD* version of `sed`[1], which differs in many respects from the *GNU* `sed` version that comes with *Linux* distros.**

Their **common denominator** is the functionality decreed by **POSIX**: see the POSIX `sed` spec.

The **most portable approach** is to **use POSIX features only**, which, however, **limits functionality**:

- Notably, POSIX specifies **support only for *basic* regular expressions**, which have many limitations (e.g., no support for `|` (alternation) at all, no direct support for `+` and `?`) and different escaping requirements.
    - Caveat: **GNU `sed` (without `-r`), *does* support `\|`, `\+` and `\?`, which is NOT POSIX-compliant; use `--posix` to disable** (see below).
- **To use POSIX features only**:
    - (both versions): use *only* the `-n` and `-e` options (notably, do not use `-E` or `-r` to turn on support for *extended* regular expressions)
    - GNU `sed`: add option `--posix` to ensure POSIX-only functionality (you don't strictly need this, but without it you could end up inadvertently using non-POSIX features without noticing; *caveat*: `--posix` *itself* is *not* POSIX-compliant)
    - Using POSIX-only features means stricter formatting requirements (forgoing many conveniences available in GNU `sed`):
        - Control-character sequences such as `\n` and `\t` are generally NOT supported.
        - Labels and branching commands (e.g., `b`) *must* be followed by an *actual* newline or continuation via a separate `-e` option.
        - See below for details.

However, *both* versions implement *extensions* to the POSIX standard:

- ***what* extensions they implement differs** (GNU `sed` implements more).
- even those **extensions they *both* implement *partially differ in syntax***.

If you need to support BOTH platforms (discussion of differences):

- **Incompatible** features:
  - Use of the **`-i` option *without* an argument** (in-place updating without backup) is incompatible:
    - BSD `sed`: MUST use `-i ''`
    - GNU `sed`: MUST use just `-i` (equivalent: `-i''`) - using `-i ''` does NOT work.
  - **`-i` sensibly turns on *per-input-file* line numbering** in *GNU* `sed` and *recent* versions of *BSD* `sed` (e.g., on FreeBSD 10), but does **NOT on macOS as of 10.12**.
    Note that in the absence of `-i` *all* versions number lines *cumulatively* across input files.
  - If the ***last* input line does *not* have a trailing newline** (and is printed):
    - BSD `sed`: *always appends a newline* on output, even if the input line doesn't end in one.
    - GNU `sed`: *preserves the trailing-newline status*, i.e., it appends a newline only if the input line ended in one.
- **Common** features:
  - If you restrict your `sed` scripts to what BSD `sed` supports, they will generally work in GNU `sed` too - with the notable exception of using platform-specific *extended* regex features with `-E`. Obviously, you'll also forgo extensions that are specific to the GNU version. See next section.

Guidelines for *cross-platform support* (OS X/BSD, Linux), driven by the stricter requirements of the BSD version:

Note that that the shorthands *macOS* and *Linux* are occasionally used below to refer to the BSD and GNU versions of `sed`, respectively, because they are the *stock* versions on each platform. However, it is possible to install GNU `sed` on macOS, for instance, using Homebrew with `brew install gnu-sed`.

Note: ***Except when the `-r` and `-E` flags are used*** (*extended* regexes), the instructions below amount to writing **POSIX-compliant** `sed` scripts.

- For POSIX compliance, you must restrict yourself to POSIX BREs (*basic* regular expressions), which are, unfortunately, as the name suggests, quite basic.
  Caveat: do not assume that `\|`, `\+` and `\?` are supported: While GNU `sed` supports them (unless `--posix` is used), BSD `sed` does not - these features are *not* POSIX-compliant.
  While **`\+` and `\?` *can* be emulated** in POSIX-compliant fashion :
  `\{1,\}` for `\+`,
  `\{0,1\}` for `\?`,
  `\|` (alternation) *cannot*, unfortunately.

- For more powerful regular expressions, **use `-E`** (rather than `-r`) to support **EREs (*extended* regular expressions)** (GNU `sed` doesn't document `-E`, but it does work there as an alias of `-r`; *newer* version of BSD `sed`, such as on FreeBSD 10, now also support `-r`, but the macOS version as of 10.12 does *not*).
  Caveat: Even though use of `-r` / `-E` means that your command is by definition *not* POSIX-compliant, you must still **restrict yourself to POSIX EREs (extended regular expressions)**. Sadly, this means that you won't be able to use several useful constructs, notably:
  - word-boundary assertions, because they're *platform-specific* (e.g., `\<` on Linux, `[[:<]]` on OS X).
  - back-references *inside regular expressions* (as opposed to the "back-references" to capture-group matches in the replacement string of `s` function calls), because BSD `sed` doesn't support them in *extended* regexes (but, curiously, does so in *basic* ones, where they are POSIX-mandated).

- **Control-character escape sequences such as `\n` and `\t`:**
  - In **regexes** (both in patterns for line selection and the first argument to the `s` function), assume that only `\n` is recognized as an escape sequence (rarely used, since the pattern space is usually a *single* line (without terminating `\n`), but not inside a *character class*, so that, e.g., `[^\n]` doesn't work; (if your input contains no control chars. other than `\t`, you can emulate `[^\n]` with `[[:print:][:blank:]]`; otherwise, splice control chars. in as *literals*[2]) - **generally, include control characters as *literals*, either via spliced-in *ANSI C-quoted strings* (e.g., `$'\t'`) in shells that support it (`bash`, `ksh`, `zsh`), or via *command substitutions using `printf`* (e.g., `"$(printf '\t')"`)**.
    - Linux only:
      ```
      sed 's/\t/-/' <<<$'a\tb' # -> 'a-b'
      ```
    - OSX *and* Linux:
      ```
      sed 's/'$'\t''/-/' <<<$'a\tb' # ANSI C-quoted string
      sed 's/'"$(printf '\t')"'/-/' <<<$'a\tb' # command subst. with printf
      ```
  - In **replacement strings** used with the `s` command, **assume that NO control-character escape sequences are supported**, so, again, include control chars. as *literals*, as above.
    - Linux only:
      ```
      sed 's/-/\t/' <<<$'a-b' # -> 'a<tab>b'
      ```
    - macOS *and* Linux:
      ```
      sed 's/-/'$'\t''/' <<<'a-b'
      sed 's/-/'"$(printf '\t')"'/' <<<'a-b'
      ```
  - Ditto for the **text arguments** to the **`i` and `a` functions**: **do not use control-character sequences** - see below.

- **Labels and branching**: labels as well as the label-name *argument* to the `b` and `t` functions **must be followed by either by a *literal* newline or a spliced-in `$'\n'`**. Alternatively, use multiple `-e` options and terminate each right after the label name.
  - Linux only:
    ```
    sed -n '/a/ bLBL; d; :LBL p' <<<$'a\nb' # -> 'a'
    ```

    - macOS *and* Linux:
        - EITHER (actual newlines):
          `sed -n '/a/ bLBL d; :LBL p' <<<$'a\nb'`
        - OR (spliced-in `$\n` instances):
          `sed -n '/a/ bLBL'$'\n''d; :LBL'$'\n''p' <<<$'a\nb'`
        - OR (multiple `-e` options):
          `sed -n -e '/a/ bLBL' -e 'd; :LBL' -e 'p' <<<$'a\nb'`

- Functions **`i` and `a` for inserting/appending text**: **follow the function name by `\`, followed either by a *literal* newline or a spliced-in `$'\n'`** before specifying the text argument.

    - Linux only:
      `sed '1 i new first line' <<<$'a\nb' # -> 'new first line<nl>a<nl>b'`
    - OSX *and* Linux:
      `sed -e '1 i\'$'\n''new first line' <<<$'a\nb'`
    - Note:
        - Without `-e`, the text argument is inexplicably not newline-terminated on output on macOS (bug?).
        - **Do not use control-character escapes** such as `\n` and `\t` in the text argument, as they're only supported on Linux.
        - If the text argument therefore has actual interior newlines, `\`-escape them.
        - If you want to place additional commands after the text argument, you must terminate it with an (unescaped) newline (whether literal or spliced in), or continue with a separate `-e` option (this is a general requirement that applies to all versions).

- Inside **function *lists*** (multiple function calls enclosed in `{...}`), **be sure to also terminate the *last* function, before the closing `}`, with `;`.**

    - Linux only:
        - `sed -n '1 {p;q}' <<<$'a\nb' # -> 'a'`
    - macOS *and* Linux:
        - `sed -n '1 {p;q;}' <<<$'a\nb'`

GNU `sed`-specific features missing from BSD `sed` altogether:

GNU features you'll miss out on if you need to support both platforms:

- Various **regex-matching and substitution options** (both in patterns for line selection and the first argument to the `s` function):

    - **The `I` option for case-INsensitive regex matching (incredibly, BSD `sed` doesn't support this at all).**
    - The `M` option for multi-line matching (where `^` / `$` match the start / end of *each line*)
    - For additional options that are specific to the `s` function, see https://www.gnu.org/software/sed/manual/sed.html#The-_0022s_0022-Command

- **Escape sequences**

    - Substitution-related escape sequences such as `\u` in the replacement argument of the

`s///` function that allow *substring manipulation*, within limits; e.g., `sed 's/^./\u&/' <<<'dog' # -> 'Dog'` - see http://www.gnu.org/software/sed/manual/sed.html#The-_0022s_0022-Command

- Control-character escape sequences: in addition to `\n`, `\t`, ..., codepoint-based escapes; for instance, all of the following escapes (hex., octal, decimal) represent a single quote (`'`): `\x27`, `\o047`, `\d039` - see https://www.gnu.org/software/sed/manual/sed.html#Escapes

- **Address extensions**, such as `first~step` to match every step-th line, `addr, +N` to match N lines following `addr`, ... - see http://www.gnu.org/software/sed/manual/sed.html#Addresses

---

[1] The macOS `sed` version is *older* than the version on other BSD-like systems such as FreeBSD and PC-BSD. Unfortunately, this means that you cannot assume that features that work in FreeBSD, for instance, will work [the same] on macOS.

[2] The ANSI C-quoted string
`$'\001\002\003\004\005\006\007\010\011\013\014\015\016\017\020\021\022\023\024\025\026\027\030\031\032\0`
contains all ASCII control characters except `\n` (and NUL), so you can use it in combination with `[:print:]` for a pretty robust emulation of `[^\n]`:

```
'[[:print:]'$'\001\002\003\004\005\006\007\010\011\013\014\015\016\017\020\021\022\023\024\025\026\027\0
```

Examples

Replace all newlines with tabs

Note: For brevity, the commands use here-strings (`<<<`) and ANSI C-quoted strings (`$'...'`). Both these shell features work in `bash`, `ksh`, and `zsh`.

```
# GNU Sed
$ sed ':a;$!{N;ba}; s/\n/\t/g' <<<$'line_1\nline_2\nline_3'
line_1 line_2 line_3

# BSD Sed equivalent (multi-line form)
sed <<<$'line_1\nline_2\nline_3' '
:a
$!{N;ba
}; s/\n/'$'\t''/g'

# BSD Sed equivalent (single-line form, via separate -e options)
sed -e ':a' -e '$!{N;ba' -e '}; s/\n/'$'\t''/g' <<<$'line 1\nline 2\nline 3'
```

BSD Sed notes:

- Note the need to terminate labels (`:a`) and branching commands (`ba`) either with actual newlines or with separate `-e` options.

- Since control-character escape sequences such as `\t` aren't supported in the replacement string, an ANSI C-quoted tab *literal* is spliced into the replacement string.
(In the *regex* part, BSD Sed *only* recognizes `\n` as an escape sequence).

Append literal text to a line with function 'a'

Note: For brevity, the commands use here-strings (`<<<`) and ANSI C-quoted strings (`$'...'`). Both these shell features work in `bash`, `ksh`, and `zsh`.

```
# GNU Sed
$ sed '1 a appended text' <<<'line 1'
line 1
appended text

# BSD Sed (multi-line form)
sed '1 a\
appended text' <<<'line 1'

# BSD Sed (single-line form via a Bash/Ksh/Zsh ANSI C-quoted string)
sed $'1 a\\\nappended text' <<<'line 1'
```

Note how BSD Seed requires a `\` followed by an *actual newline* to pass the text to append. The same applies to the related `i` (insert) and `c` (delete and insert) functions.

Read BSD/macOS Sed vs. GNU Sed vs. the POSIX Sed specification online: https://riptutorial.com/sed/topic/9436/bsd-macos-sed-vs--gnu-sed-vs--the-posix-sed-specification

Chapter 8: Delete command

Examples

Delete one line containing a pattern

Given a file **file.txt** with the following content:

```
line 1
line 2
line 3
```

You can delete a line from file content with the `d` command.

The pattern to match is surrounded with default `/` delimiter and the `d` command follows the pattern:

```
sed '/line 2/d' file.txt
```

The above command will output:

```
line 1
line 3
```

To edit the file *in place*, use the `-i` option:

```
sed -i '/line 2/d' file.txt
```

Read Delete command online: https://riptutorial.com/sed/topic/2177/delete-command

Chapter 9: In-Place Editing

Syntax

- sed -I *extension* - FreeBSD sed (continuous line-counter)
- sed -I[*extension*] - NetBSD and Illumos sed (continuous line-counter)
- sed -i *extension* - FreeBSD sed
- sed -i[*extension*] - NetBSD, OpenBSD, Illumos, BusyBox and GNU sed
- sed --in-place[=*extension*] - Illumos, BusyBox, and GNU sed

Parameters

Parameter	Details
`extension`	Save a backup file with the specified extension, or no backup file when `extension` is a zero-length string.

Remarks

In-place editing is a common but non-standard extension present in the majority of recent systems.

From a BSD `sed` manual

(a section like this appears in all current BSD `sed` manuals, and those of their derivatives)

> It is not recommended to give a zero length extension when in place editing files, as it risks corruption or partial content in situations where disk space is exhausted, etc.

Don't forget the mighty `ed`

There is definitely a use for `sed` and for in-place editing features of `sed`, but when the UNIX standard is extended, we should always ask why the old UNIX standard did not include that feature. Though UNIX is not perfect, the orthogonality and completeness of the tools has been developed to be quite near to perfection, at least for purposes that where visible around 1970: *Text editing and automated text editing was surely visible around that time.*

Actually, the idea of `sed` is not to edit a *file* in place, but to edit a *stream*. That's why the name `sed` is a short form of *stream editor*. Take away the `s`, and you get the tool that was actually designed for *file* editing: `ed`:

```
printf 'g/what to replace/s//with what to replace/g\nw\nq\n' | ed file
```

or `cat file_edit_commands | ed file`.

Examples

Replacing strings in a file in-place

```
sed -i s/"what to replace"/"with what to replace"/g $file
```

We use `-i` to select in-place editing on the `$file` file. In some systems it is required to add suffix after `-i` flag which will be used to create backup of original file. You can add empty string like `-i ''` to omit the backup creation. Look at *Remarks* in this topic about `-i` option.

The `g` terminator means do a global find/replace in each line.

```
$ cat example
one
two
three
total
$ sed -i s/"t"/"g"/g example
$ cat example
one
gwo
ghree
gogal
```

Portable Use

In-place editing, while common, is a non-standard feature. A viable alternative would be to use an intermediate file to either store the original, or the output.

```
sed 'sed commands' > file.out && mv file.out file
# or
mv file file.orig && sed 'sed commands' file.orig > file
```

To use the `-i` option with both the GNU and FreeBSD syntax an extension must be specified and appended to the `-i` option. The following will be accepted by both, and produce two files, the original version at `file.orig` and the edited version at `file`:

```
sed -i.orig 'sed commands' file
```

See a basic example given a file `file`:

```
$ cat file
one
two
three
$ sed -i.orig 's/one/XX/' file
$ cat file                        # the original file has changed its content
XX
```

```
two
three
$ cat file.orig                  # the original content is now in file.orig
one
two
three
```

A more complex example, replacing each line with line number:

```
$ printf 'one\ntwo\n' | tee file1 | tr a-z A-Z > file2
$ sed -ni.orig = file1 file2
$ cat file1.orig file2.orig
one
two
ONE
TWO
$ cat file1 file2
1
2
1
2
```

Why a backup file is required

In order to use in-place editing without a backup file, `-i` must be given a zero-length argument and FreeBSD `sed` requires an argument to `-i`, either appended or separate, while the GNU optional argument extension requires the argument be appended to `-i`. Both support appending the argument to `-i`, but without it being required `-i'' command` is indistinguishable from `-i extension`, and so a zero-length argument can not be appended to `-i`.

In-place editing without specifying a backup file overrides read-only permissions

`sed -i -e cmd file` will modify `file` even if its permissions are set to read-only.

This command behaves similarly to

```
sed -e cmd file > tmp; mv -f tmp file
```

rather than

```
sed -e cmd file > tmp; cat tmp > file; rm tmp
```

The following example uses gnu `sed`:

```
$ echo 'Extremely important data' > input
$ chmod 400 input  # Protect that data by removing write access
$ echo 'data destroyed' > input
-bash: input: Permission denied
$ cat input
Extremely important data (#phew! Data is intact)
$ sed -i s/important/destroyed/ input
$ cat input
```

```
Extremely destroyed data (#see, data changed)
```

This can be mitigated by creating a backup by specifying a `SUFFIX` with the `i` option:

```
$ sed -i.bak s/important/destroyed/ input
$ cat input
Extremely destroyed data
$ cat input.bak
Extremely important data
```

Read In-Place Editing online: https://riptutorial.com/sed/topic/3640/in-place-editing

Chapter 10: Regular expressions

Examples

Using different delimiters

Given a file like this:

```
$ cat file
hello/how/are/you
i am fine
```

You can use `/pattern/` to match specific lines:

```
$ sed -n '/hello/p' file
hello/how/are/you
```

If the pattern contains slashes itself, you can use another delimiter using `\cBREc`:

```
$ sed -n '\#hello/how#p' file
hello/how/are/you
$ sed -n '\_hello/how_p' file
hello/how/are/you
```

As defined by POSIX in:

> **Regular Expressions in sed**
> In a context address, the construction `\cBREc`, where c is any character other than backslash or , shall be identical to `/BRE/`. If the character designated by c appears following a backslash, then it shall be considered to be that literal character, which shall not terminate the BRE. For example, in the context address "\xabc\xdefx", the second x stands for itself, so that the BRE is "abcxdef".

Read Regular expressions online: https://riptutorial.com/sed/topic/7720/regular-expressions

Chapter 11: Substitution

Examples

Substitution Using Shell Variables

Variables inside single quotes `'` don't get expanded by POSIX compatible shells, so using a shell variable in a `sed` substitution requires the use of double quotes `"` instead of single quotes `'`:

```
$ var="he"
$ echo "hello" | sed "s/$var/XX/"
XXllo

$ var="he"
$ echo "hello" | sed 's/$var/XX/'
hello
```

Be careful of command injection when evaluating variables:

```
$ var='./&/;x;w/etc/passwd
> x;s/he'
$ echo "hello" | sed "s/$var/XX/"
sed: /etc/passwd: Permission denied
```

If the above was run as root the output would have been indistinguishable from the first example, and the contents of `/etc/passwd` would be destroyed.

Backreference

Using escaped brackets, you can define a capturing group in a pattern that can be backreferenced in the substitution string with `\1`:

```
$ echo Hello world! | sed 's/\(Hello\) world!/\1 sed/'
Hello sed
```

With multiple groups:

```
$ echo one two three | sed 's/\(one\) \(two\) \(three\)/\3 \2 \1/'
three two one
```

BSD sedGNU sed

When using extended regular expressions (see Additional Options) parenthesis perform grouping by default, and do not have to be escaped:

```
$ echo one two three | sed -E 's/(one) (two) (three)/\3 \2 \1/'
three two one
```

Words consisting of letter, digits and underscores can be matched using the expression `[[:alnum:]_]\{1,\}`:

```
$ echo Hello 123 reg_exp | sed 's/\([[:alnum:]_]\{1,\}\) \([[:alnum:]_]\{1,\}\)
\([[:alnum:]_]\{1,\}\)/\3 \2 \1/'
reg_exp 123 Hello
```

GNU sed

The sequence `\w` is equivalent to `[[:alnum:]_]`

```
$ echo Hello 123 reg_exp | sed 's/\(\w\w*\) \(\w\w*\) \(\w\w*\)/\3 \2 \1/'
reg_exp 123 Hello
```

Using different delimiters

POSIX/IEEE Open Group Base Specification says:

> **[2addr] s/BRE/replacement/flags**
>
> Substitute the replacement string for instances of the BRE in the pattern space. **Any character other than backslash or newline** can be used instead of a slash to delimit the BRE and the replacement. Within the BRE and the replacement, the BRE delimiter itself can be used as a literal character if it is preceded by a backslash.

There are cases when the delimiter `/` for `sed` replacement is in the BRE or replacement, triggering errors like:

```
$ echo "2/3/4" | sed "s/2/3/X/"
sed: -e expression #1, char 7: unknown option to `s'
```

For this, we can use different delimiters such as `#` or _ or even a space:

```
$ echo "2/3/4" | sed "s#2/3#X#"
X/4
$ echo "2/3/4" | sed "s_2/3_X_"
X/4
$ echo "2/3/4" | sed "s 2/3 X "
X/4
```

Pattern flags - occurrence replacement

If we want to replace only the first occurrence in a line, we use `sed` as usual:

```
$ cat example
aaaaabbbbb
aaaaaccccc
aaaaaddddd
$ sed 's/a/x/' example
xaaaabbbbb
xaaaaccccc
```

```
xaaaaddddd
```

But what if we want to replace all occurrences?

We just add the `g` pattern flag at the end:

```
$ sed 's/a/x/g' example
xxxxxbbbbb
xxxxxccccc
xxxxxddddd
```

And if we want to replace one specific occurrence, we can actually specify which one:

```
$ sed 's/a/x/3' example
aaxaabbbbb
aaxaaccccc
aaxaaddddd
```

`/3` being the 3rd occurrence.

GNU sed

From `info sed`, see GNU sed manual for online version

> the POSIX standard does not specify what should happen when you mix the `g` and NUMBER modifiers, and currently there is no widely agreed upon meaning across `sed` implementations. For GNU `sed`, the interaction is defined to be: ignore matches before the NUMBERth, and then match and replace all matches from the NUMBERth on.

```
$ sed 's/b/y/2g' example
aaaaabyyyy
aaaaaccccc
aaaaaddddd

$ sed 's/c/z/g3' example
aaaaabbbbb
aaaaacczzz
aaaaaddddd
```

Read Substitution online: https://riptutorial.com/sed/topic/1096/substitution

Credits

S. No	Chapters	Contributors
1	Getting started with sed	Community, fedorqui, kdhp, SLePort
2	Additional Options	kdhp, mklement0
3	Address and address range	Benjamin W., Sundeep
4	Advanced sed commands	Raju
5	Append command	kdhp, Slawomir Jaranowski
6	Branching Operation	Ekeyme Mo
7	BSD/macOS Sed vs. GNU Sed vs. the POSIX Sed specification	mklement0
8	Delete command	SLePort
9	In-Place Editing	AstraSerg, Ekeyme Mo, Emil Burzo, fedorqui, ghostarbeiter, ikrabbe, kdhp, mklement0, Oleg Arkhipov, William Pursell
10	Regular expressions	fedorqui
11	Substitution	Emil Burzo, fedorqui, kdhp, SLePort, Sundeep, thanasisp