

10 – Sed, cut, and paste

CS 2043: Unix Tools and Scripting, Spring 2019 [1]

Matthew Milano

February 13, 2019

Cornell University

Table of Contents

1. Cutting
2. The Stream Editor (**sed**)
3. Interlude: xargs and shift
4. Pasting
5. Splitting and Joining

Cutting

Chopping up Input

cut out sections of input (filtering)

cut <options> [**file**]

- Must specify list of *bytes* (-b), *characters* (-c), or *fields* (-f).
- The **file** is optional, uses **stdin** if unspecified.

N	Only N^{th} byte, character, or field, counted from 1 .
N-	N^{th} byte, character, or field, to end of line.
M-N	M^{th} to N^{th} (inclusive) byte, character, or field.
-N	First to N^{th} (inclusive) byte, character, or field.
M,N,...,X	Extract individual items (1,4,6 : first, fourth, and sixth bytes, characters, or fields).

- E.g., **-b 2** is “2nd byte”, **-f 3-** is “3rd field to end of line”.
- Use **-d** to specify a delimiter (**TAB** by default).
 - E.g., **echo 'a:b:c:d' | cut -d : -f 2** \Rightarrow **b**

cut Examples

employees.csv

```
Alice,female,607-123-4567,11 Sunny Place,Ithaca,NY,14850  
Bob,male,607-765-4321,1892 Rim Trail,Ithaca,NY,14850  
Andy,n/a,607-706-6007,1 To Rule Them All,Ithaca,NY,14850  
Bad employee data without proper delimiter
```

- `/course/cs2043/demos/10-demos/employees.csv`

- Get names, ignore improper lines:

```
$ cut -d , -f 1 -s employees.csv
```

- Get names and phone numbers, ignore improper lines:

```
$ cut -d , -f 1,3 -s employees.csv
```

- Get address (4th col and after), ignore improper lines:

```
$ cut -d , -f 4- -s employees.csv
```

The Stream Editor (**sed**)

Introducing...

The Stream Editor

`sed [options] [script] [file]`

- Stream editor for filtering and transforming text.
- If no **file** provided, **stdin** is used.
- We will focus on **sed**'s '`s/<regex>/<replacement>/`':
 - Replace anything matching `<regex>` with `<replacement>`.
 - E.g., `echo 'hello' | sed 's/lo/p!/'` \Rightarrow `help!`
- **sed** goes line by line searching for the regular expression.
- Only covering *basics*, **sed** is a full programming language.
- Main difference between **sed** and **tr** for scripting?
 - **sed** can match regular expressions, and perform *captures*!
- Extended regular expressions: use the **-E** flag (not **-r**).
 - GNU **sed** supports both **-r** and **-E**, BSD **sed** only **-E**.
- See examples for more.

A Basic Example

- Luke, there is *no spoon* (demo file `no_spoon.txt`).

```
$ head -1 no_spoon.txt
There is no spoon. There is no spoon. There is no spoon. There is no spoon.

$ sed 's/no spoon/a fork/g' no_spoon.txt
There is a fork. There is a fork. There is a fork. There is a fork.
...
There is a fork. There is a fork. There is a fork. There is a fork.
```

- Replaces **no spoon** with **a fork** for every line.
- No ending `/g`? Only one substitution per line:

```
$ sed 's/no spoon/a fork/' no_spoon.txt
There is a fork. There is no spoon. There is no spoon. There is no spoon.
...
There is a fork. There is no spoon. There is no spoon. There is no spoon.
```

- **Caution:** get in habit of using *single-quotes* for with **sed**.
 - Otherwise special shell characters (like `*`) may expand in *double-quotes* causing you sadness and pain.

Deletion

- Delete all lines that contain regex: `sed '/regex/d'`

david.txt

```
Hi, my name is david.  
Hi, my name is DAVID.  
Hi, my name is David.  
Hi, my name is dAVID.
```

- Delete all lines in demo file `david.txt` matching `[Dd]avid`:

```
$ sed '/[Dd]avid/d' david.txt  
Hi, my name is DAVID.  
Hi, my name is dAVID.
```

- To delete pattern per-line, just do an empty replacement:

```
$ sed 's/[ ]\?[Dd][Aa][Vv][Ii][Dd].//g' david.txt  
Hi, my name is  
Hi, my name is  
Hi, my name is  
Hi, my name is
```

Regular Expressions

- What does this **REMOVED** from **demo file data.txt**?

```
$ sed 's/[a-zA-Z]\{1,3\}[0-9]*@cornell\.edu/REMOVED/g' data.txt
```

- Only removes **netID@cornell.edu** emails, not the others!
- The `\{1,3\}`.{bash} specifies a number of occurrences
- “Regular” regex: escape specials (**(parens)**, **{braces}**, etc.).

```
$ sed 's/[[:alnum:]]\{1,11\}@/REMOVED@/g' data.txt
```

- We have to escape the curly braces: `\{1,11\}`
- “Extended” regex (using `-E` flag): escaping rules **reversed**!

```
$ sed -E 's/[[:alnum:]]\{1,11\}@/REMOVED@/g' data.txt
```

- No replacements, `\{1,11\}` now means literal string `{1,11}`.

```
$ sed -E 's/[[:alnum:]]{1,11}@/REMOVED@/g' data.txt
```

- Works! `\{1,11\} ⇒ {1,11}`

Capture Groups

- Like most regular expressions, **(parens)** form capture groups.
- You can use the capture groups in the replacement text.
 - If you have one capture group: `\1` in replacement text.
 - Two groups? `\1` and `\2` are available in replacement text.

- A contrived example:

```
$ echo 'hello world' | \
    sed 's/\(hello\) \(world\)\/\2 say \1 back/'
world say hello back
```

- And using regular expressions?

```
$ echo 'I have a spoon.' | \
    sed -E 's/([a-z+)\.\/super shiny silver \1!/'
I have a super shiny silver spoon!
```

- Notice that those **(parens)** are not escaped because of `-E`!

More sed

- Can specify lines to check by numbers or with regex:

```
# checks lines 1 to 20
$ sed '1,20s/john/John/g' file
```

```
# checks lines beginning with 'The'
$ sed '/^The/s/john/John/g' file
```

- The & corresponds to the pattern found:

```
# replace words with words in double quotes
$ sed 's/[a-zA-Z]\+/"&"/g' no_spoon.txt
"There" "is" "no" "spoon". ....
```

- Many more resources [available here](#).

See **sed Practice** demo folder.

Interlude: xargs and shift

Xargs

- Use the output of a command as arguments to another command
- Option 1: `cmd2 $(command1)`
 - usually works fine, order looks weird
- Option 2: `command1 | xargs cmd`
 - no subshell
 - commands written in the “right” order

Use standard input as arguments

```
xargs <command> [args for command...]
```

- pipe input to xargs or redirect file to xargs
- becomes arguments for xargs' command
- like `find`'s `-exec`, except no `{}` \;

shift

Ignore some arguments

shift <number>

- used in shell scripts only!
- drop the first arguments
- renumber remaining arguments
 - after **shift**; \$2 is \$1, \$3 is \$2, etc.

- Also effects \$* and \$@.
- Want to use \$* but ignore the first argument? **shift** is your answer.
 - can keep **shifting** to keep ignoring arguments.

Pasting

Merge Lines of Files

```
paste [options] [file1] [file2] ... [fileN]
```

- Neither **options** nor **files** are *required*.
- Use **-d** to specify the delimiter (**TAB** by default).
- Use **-s** to concatenate serially instead of side-by-side.
- No **options** and one **file** specified: same as **cat**.
 - Use with **-s** to join all lines of a file.

paste Examples I

```
names.txt
```

```
Alice
```

```
Bob
```

```
Andy
```

```
phones.txt
```

```
607-123-4567
```

```
607-765-4321
```

```
607-706-6007
```

- `paste cut_paste/names.txt` and
`cut_paste/phones.txt` line by line:

```
$ paste -d , names.txt phones.txt > result.csv
```

```
$ cat result.csv
```

```
Alice,607-123-4567
```

```
Bob,607-765-4321
```

```
Andy,607-706-6007
```

paste Examples II

names.txt

Alice

Bob

Andy

phones.txt

607-123-4567

607-765-4321

607-706-6007

- `paste names.txt and phones.txt` *serially* (-s):

```
$ paste -d , -s names.txt phones.txt > result.csv
```

```
$ cat result.csv
```

Alice,Bob,Andy

607-123-4567,607-765-4321,607-706-6007

Splitting and Joining

Splitting Files

split a file into pieces

```
split [options] [file [prefix]]
```

- Use **-l** to specify how many lines in each file
 - Default: **1000**
- Use **-b** to specify how many *bytes* in each file.
- The **prefix** is prepended to *each file* produced.
- If no **file** provided (or if **file** is **-**), **stdin** is used.
- Use **-d** to produce numeric suffixes instead of lexographic.
 - Not available on BSD / macOS.

split Examples I

```
ages.txt
```

```
Alice 44
```

```
Bob 30
```

```
Candy 12
```

- `split split_join/ages.txt` into files of one line each:

```
$ split -l 1 ages.txt
```

```
$ ls
```

```
ages.txt  salaries.txt  xaa  xab  xac
```

```
$ cat xaa
```

```
Alice 44
```

```
$ cat xab
```

```
Bob 30
```

```
$ cat xac
```

```
Candy 12
```


split Examples II

```
ages.txt
```

```
Alice 44
```

```
Bob 30
```

```
Candy 12
```

- `split split_join/ages.txt` into files of one line each,
 - with numeric suffixes (`-d`) (GNU / Linux), and with `ages_` prefix

```
$ split -l 1 -d ages.txt ages_
```

```
$ ls
```

```
ages_00  ages_01  ages_02  ages.txt  salaries.txt
```

```
$ cat ages_00
```

```
Alice 44
```

```
$ cat ages_01
```

```
Bob 30
```

```
$ cat ages_02
```

```
Candy 12
```

Joining Files

join lines of two files on a common field

```
join [options] file1 file2
```

- Join two files at a time, no more, no less.
- Default: files are assumed to be delimited by *whitespace*.
- Use **-t <char>** to specify alternative *single-character* delimiter.
- Use **-1 n** to join by the n^{th} field of **file1**.
- Use **-2 n** to join by the n^{th} field of **file2**.
 - Field numbers start at **1**, like **cut** and **paste**.
- Use **-a f_num** to display unpaired lines of file **f_num**.

join Examples I

ages.txt

Alice 44

Bob 30

Candy 12

salaries.txt

Bob 300,000

Candy 120,000

- join split_join/ages.txt and split_join/salaries.txt files into results.txt:

```
$ join ages.txt salaries.txt > results.txt
```

```
$ cat results.txt
```

```
Bob 30 300,000
```

```
Candy 12 120,000
```

join Examples II

ages.txt

Alice 44
Bob 30
Candy 12

salaries.txt

Bob 300,000
Candy 120,000

- join split_join/ages.txt and split_join/salaries.txt files into results.txt:

```
$ join -a1 ages.txt salaries.txt > results.txt  
$ cat results.txt  
Alice 44  
Bob 30 300,000  
Candy 12 120,000
```

References

- [1] Stephen McDowell, Bruno Abrahao, Hussam Abu-Libdeh, Nicolas Savva, David Slater, and others over the years. “Previous Cornell CS 2043 Course Slides”.