08 - Superlative Streams

CS 2043: Unix Tools and Scripting, Spring 2016 [1]

Stephen McDowell

February 12th, 2016

Cornell University

Table of contents

Some Logistics

- HW1 due today at 5pm.

Some Logistics

- HW1 due today at 5pm.
- OH Today: only 2pm - 3pm...thanks (again) Joe!

Some Logistics

- HW1 due today at 5pm.
- OH Today: only 2pm - 3pm...thanks (again) Joe!
- On my usage of **>**, which will now become **>>>** for safety.

Some Logistics

- HW1 due today at 5pm.
- OH Today: only 2pm - 3pm...thanks (again) Joe!
- On my usage of **>**, which will now become **>>>** for safety.
- Repository confusion:

Some Logistics

- HW1 due today at 5pm.
- OH Today: only 2pm - 3pm...thanks (again) Joe!
- On my usage of **>**, which will now become **>>>** for safety.
- Repository confusion:
  - *Do NOT fork the* `<usr>-assignments` *repositories*!!!!!!

Some Logistics

- HW1 due today at 5pm.
- OH Today: only 2pm - 3pm...thanks (again) Joe!
- On my usage of `>`, which will now become `>>>` for safety.
- Repository confusion:
  - *Do NOT fork the* `<usr>-assignments` *repositories*!!!!!!
  - Getting lectures easily: `clone` the `lecture-slides` repo, `pull` as needed.

Some Logistics

- HW1 due today at 5pm.
- OH Today: only 2pm - 3pm...thanks (again) Joe!
- On my usage of `>`, which will now become `>>>` for safety.
- Repository confusion:
  - *Do NOT fork the* `<usr>-assignments` *repositories*!!!!!!
  - Getting lectures easily: `clone` the `lecture-slides` repo, `pull` as needed.
  - *Only* `fork` the `lecture-demos` repo.

Some Logistics

- HW1 due today at 5pm.
- OH Today: only 2pm - 3pm...thanks (again) Joe!
- On my usage of `>`, which will now become `>>>` for safety.
- Repository confusion:
  - *Do NOT fork the* `<usr>-assignments` *repositories*!!!!!!
  - Getting lectures easily: `clone` the `lecture-slides` repo, `pull` as needed.
  - *Only* `fork` the `lecture-demos` repo.
    - This allows you to put your demo work online, get more practice with `git`.

Cutting and Pasting

Chopping up Input

Cut

`cut <options> [file]`

- *Must* specify a list of bytes, characters, or fields.
  - -.1 The `file` is optional this time, uses `STDIN` if unspecified.
- Use `-b` to extract using range of *bytes*.
- Use `-c` to extract using a range of *characters*.
- Use `-f` to extract a range of *fields* separated by a delimiter.

`N`	`N`th byte, character or field, counted from 1
`N-`	from `N`th byte, character or field, to end of line
`N-M`	from `N`th to `M`th (included) byte, character or field
`-M`	from first to `M`th (included) byte, character or field

- Use `-d` to specify a delimiter (`TAB` by default).
- Use `-s` to suppress line if `delimiter` not found.

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
```

Examples

- 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
```

- Get 11^{th} character of every line:

```
>>> cut -c 11 employees.csv
```

Splicing Input

Paste

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

- No **`options`** or **`files`** necessary...
  - ...but relatively useless program without them.
- Use **`-d`** to specify the delimiter (**TAB** by default).
- Use **`-s`** to concatenates serially instead of side-by-side.
- No options and one **`file`** specified: just like **`cat`**.
  - Use with **`-s`** to join all lines of file!

Paste Examples I

names.txt

```
Alice
Bob
Andy
```

phones.txt

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

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

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

result.csv

```
Alice,Bob,Andy
607-123-4567,607-765-4321,607-706-6007
```

Paste Examples III

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
```

```
>>> paste -d "" -s employees.csv | \
      cut -d , -f 1- --output-delimiter="" | \
      tr -d "[:space:]"
```

output (all on one line...)

```
Alicefemale607-123-456711SunnyPlaceIthacaNY14850Bobmale6
07-765-43211892RimTrailIthacaNY14850Andyn/a607-706-60071
ToRuleThemAllIthacaNY14850Bademployeedatawithoutproperde

limiter
```

Splitting and Joining

Splitting Files

Split

`split [options] [input] [prefix]`

- Use `-l` to specify how many lines in each file.
  - Default is 1000.
- Use `-b` to specify how many bytes in each file.
- The `prefix` is prepended to each file produced.
- Use `-d` to produce numeric suffixes instead of lexographic.
  - Not available on OSX.

Splitting Files

Split

`split [options] [input] [prefix]`

- Use `-l` to specify how many lines in each file.
  - Default is 1000.
- Use `-b` to specify how many bytes in each file.
- The `prefix` is prepended to each file produced.
- Use `-d` to produce numeric suffixes instead of lexographic.
  - Not available on OSX.

- Extremely useful for managing large streams of data.

Splitting Files

Split

`split [options] [input] [prefix]`

- Use `-l` to specify how many lines in each file.
  - Default is 1000.
- Use `-b` to specify how many bytes in each file.
- The `prefix` is prepended to each file produced.
- Use `-d` to produce numeric suffixes instead of lexographic.
  - Not available on OSX.

- Extremely useful for managing large streams of data.
- Remember that annoying *dungeon* folder?

Splitting Files

Split

`split [options] [input] [prefix]`

- Use `-l` to specify how many lines in each file.
  - Default is 1000.
- Use `-b` to specify how many bytes in each file.
- The `prefix` is prepended to each file produced.
- Use `-d` to produce numeric suffixes instead of lexographic.
  - Not available on OSX.

- Extremely useful for managing large streams of data.
- Remember that annoying *dungeon* folder?
  - `split -l 5` is what we did.

Joining Files

Join lines containing the same keys between two different files.

Join

`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 an alternative *single-character* delimiter.
- Use `-1 field_number` to join by the n^{th} field of `file1`.
- Use `-2 field_number` 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 ages.txt salaries.txt > results.txt
```

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 -a1 ages.txt salaries.txt > results.txt
```

results.txt

```
Alice 44
Bob 30 300,000
Candy 12 120,000
```

The Stream Editor (**sed**)

Introducing...

Stream Editor

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

- Stream editor for filtering and transforming text.
- We will focus on `sed`'s `'s/<regex>/<text>' [file]`.
  - Replace anything that matches `<regex>` with `<text>`.
- `sed` goes line by line searching for the regular expression.

Introducing...

Stream Editor

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

- Stream editor for filtering and transforming text.
- We will focus on `sed`'s `'s/<regex>/<text>' [file]`.
  - Replace anything that matches `<regex>` with `<text>`.
- `sed` goes line by line searching for the regular expression.

- We will only cover the *basics*, as `sed` is an entire programming language.

Introducing...

Stream Editor

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

- Stream editor for filtering and transforming text.
- We will focus on `sed`'s `'s/<regex>/<text>' [file]`.
  - Replace anything that matches `<regex>` with `<text>`.
- `sed` goes line by line searching for the regular expression.

- We will only cover the *basics*, as `sed` is an entire programming language.
  - As in there are entire books on it...

Introducing...

Stream Editor

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

- Stream editor for filtering and transforming text.
- We will focus on `sed`'s `'s/<regex>/<text>' [file]`.
  - Replace anything that matches `<regex>` with `<text>`.
- `sed` goes line by line searching for the regular expression.

- We will only cover the *basics*, as `sed` is an entire programming language.
  - As in there are entire books on it...
- What is the difference between `sed` and `tr`?

Introducing...

Stream Editor

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

- Stream editor for filtering and transforming text.
- We will focus on `sed`'s `'s/<regex>/<text>' [file]`.
  - Replace anything that matches `<regex>` with `<text>`.
- `sed` goes line by line searching for the regular expression.

- We will only cover the *basics*, as `sed` is an entire programming language.
  - As in there are entire books on it...
- What is the difference between `sed` and `tr`?
  - `sed` can match regular expressions!

Introducing...

Stream Editor

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

- Stream editor for filtering and transforming text.
- We will focus on `sed`'s `'s/<regex>/<text>' [file]`.
  - Replace anything that matches `<regex>` with `<text>`.
- `sed` goes line by line searching for the regular expression.

- We will only cover the *basics*, as `sed` is an entire programming language.
  - As in there are entire books on it...
- What is the difference between `sed` and `tr`?
  - `sed` can match regular expressions!
  - `sed` also does a *lot* more.

A Basic Example

```
>>> sed 's/not guilty/guilty/g' filename
```

A Basic Example

```
>>> sed 's/not guilty/guilty/g' filename
```

- Replaces *not guilty* with *guilty* everywhere in the file.

A Basic Example

```
>>> sed 's/not guilty/guilty/g' filename
```

- Replaces *not guilty* with *guilty* everywhere in the file.
- CAUTION: You should be in the habit of using *single-quotes* for strings with **`sed`**.

A Basic Example

```
>>> sed 's/not guilty/guilty/g' filename
```

- Replaces *not guilty* with *guilty* everywhere in the file.
- CAUTION: You should be in the habit of using *single-quotes* for strings with `sed`.
  - don't have to escape every double-quote (`"`).

A Basic Example

```
>>> sed 's/not guilty/guilty/g' filename
```

- Replaces *not guilty* with *guilty* everywhere in the file.
- CAUTION: You should be in the habit of using *single-quotes* for strings with `sed`.
  - don't have to escape every double-quote (`"`).
- What happens if we do not have the `g`?

A Basic Example

```
>>> sed 's/not guilty/guilty/g' filename
```

- Replaces *not guilty* with *guilty* everywhere in the file.
- CAUTION: You should be in the habit of using *single-quotes* for strings with `sed`.
  - don't have to escape every double-quote (`"`).
- What happens if we do not have the `g`?
  - Without the `g`, it will only do one substitution per line.

A Basic Example

```
>>> sed 's/not guilty/guilty/g' filename
```

- Replaces *not guilty* with *guilty* everywhere in the file.
- CAUTION: You should be in the habit of using *single-quotes* for strings with `sed`.
  - don't have to escape every double-quote (`"`).
- What happens if we do not have the `g`?
  - Without the `g`, it will only do one substitution per line.
    - There are definitely cases where you would want that!

Deletion

- Just like with `tr` we can do deletion with `sed`.

Deletion

- Just like with `tr` we can do deletion with `sed`.
- `sed '/regex/d'` - deletes all lines that contain `regex`.

Deletion

- Just like with `tr` we can do deletion with `sed`.
- `sed '/regex/d'` - deletes all lines that contain `regex`.
- Example:

Deletion

- Just like with `tr` we can do deletion with `sed`.
- `sed '/regex/d'` - deletes all lines that contain `regex`.
- Example:

```
>>> sed '/[Dd]avid/d' file1 > file2
```

Deletion

- Just like with `tr` we can do deletion with `sed`.
- `sed '/regex/d'` - deletes all lines that contain `regex`.
- Example:

```
>>> sed '/[Dd]avid/d' file1 > file2
```

  - Deletes all lines in `file1` that contain either *David* or *david*, and saves the result into `file2`.

Regular Expressions

- The power of `sed` is that it treats everything between the first pair of `/`'s as a regular expression.

Regular Expressions

- The power of `sed` is that it treats everything between the first pair of `/`'s as a regular expression.
- What does this do?

Regular Expressions

- The power of `sed` is that it treats everything between the first pair of `/`'s as a regular expression.
- What does this do?

```
>>> sed 's/[[:alpha:]]\{1,3\}[[:digit:]]*@cornell\.edu/REMOVED/g' file
```

Regular Expressions

- The power of `sed` is that it treats everything between the first pair of `/`'s as a regular expression.
- What does this do?

```
>>> sed 's/[[:alpha:]]\{1,3\}[[:digit:]]*@cornell\.edu/REMOVED/g' file
```

  - Print a file with all `netID@cornell.edu` emails removed!

Regular Expressions

- The power of `sed` is that it treats everything between the first pair of `/`'s as a regular expression.
- What does this do?

```
>>> sed 's/[[:alpha:]]\{1,3\}[[:digit:]]*@cornell\.edu/REMOVED/g' file
```

  - Print a file with all `netID@cornell.edu` emails removed!

- Use `-r` (`-E` on BSD/OSX) to use *extended* regular expressions.

Saving Strings

- What does this do?

Saving Strings

- What does this do?

```
>>> sed 's/^\([A-Z][A-Za-z]*\), \([A-Z][A-Za-z]*\)/\2 \1/' file
```

Saving Strings

- What does this do?

```
>>> sed 's/^\([A-Z][A-Za-z]*\), \([A-Z][A-Za-z]*\)/\2 \1/' file
```

- Searches for an expression at the beginning of the line of the form `e1`, `e2` where `e1` and `e2` are "words" starting with capital letters.

Saving Strings

- What does this do?

```
>>> sed 's/^\([A-Z][A-Za-z]*\), \([A-Z][A-Za-z]*\)/\2 \1/' file
```

- Searches for an expression at the beginning of the line of the form `e1`, `e2` where `e1` and `e2` are "words" starting with capital letters.
- Placing an expression inside `( )` tells the editor to *save* whatever string matches the expression.

Saving Strings

- What does this do?

```
>>> sed 's/^\([A-Z][A-Za-z]*\), \([A-Z][A-Za-z]*\)/\2 \1/' file
```

- Searches for an expression at the beginning of the line of the form `e1`, `e2` where `e1` and `e2` are "words" starting with capital letters.
- Placing an expression inside `( )` tells the editor to *save* whatever string matches the expression.
- Since `( )` are special characters, we escape them e.g. with `\(\)`.

Saving Strings

- What does this do?

```
>>> sed 's/^\([A-Z][A-Za-z]*\), \([A-Z][A-Za-z]*\)/\2 \1/' file
```

- Searches for an expression at the beginning of the line of the form `e1`, `e2` where `e1` and `e2` are "words" starting with capital letters.
- Placing an expression inside `()` tells the editor to *save* whatever string matches the expression.
- Since `()` are special characters, we escape them e.g. with `\(\)`.
- We access the saved strings as `\1` and `\2`.

Saving Strings

- What does this do?

```
>>> sed 's/^\([A-Z][A-Za-z]*\), \([A-Z][A-Za-z]*\)/\2 \1/' file
```

- Searches for an expression at the beginning of the line of the form `e1`, `e2` where `e1` and `e2` are "words" starting with capital letters.
- Placing an expression inside `()` tells the editor to *save* whatever string matches the expression.
- Since `()` are special characters, we escape them e.g. with `\(\)`.
- We access the saved strings as `\1` and `\2`.
- This script for example could convert a database file from `Lastname, Firstname` - to - `Firstname Lastname`

More **sed**

- You can specify which lines to check by numbers or with regular expressions:

More **sed**

- You can specify which lines to check by numbers or with regular expressions:

```
# checks lines 1 to 20
>>> sed '1,20s/john/John/g' file
# checks lines beginning with "The"
>>> sed '/^The/s/john/John/g' file
```

More **sed**

- You can specify which lines to check by numbers or with regular expressions:

```
# 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:

More **sed**

- You can specify which lines to check by numbers or with regular expressions:

```
# 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 quotes
>>> sed 's/[a-z]\+/"&"/g' file
```

More **sed**

- You can specify which lines to check by numbers or with regular expressions:

```
# 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 quotes
>>> sed 's/[a-z]\+/"&"/g' file
```

- Many more resources here: http://www.grymoire.com/Unix/Sed.html

Sed Practice

Extra Practice

Can be found here: https://github.com/cs2043-sp16/lecture-demos/tree/master/lec08

References I

[1] B. Abrahao, H. Abu-Libdeh, N. Savva, D. Slater, and others over the years.
Previous cornell cs 2043 course slides.