Scripting & Computer Environments

Advanced Filters

IIIT-H

Aug 29, 2015

...Previously & Today...

Previously:

- Basics of Regex
- Regex metacharacters
- `grep`: a regex-aware filter

Today:

- More regex-aware Filters:
  - `sed`✓
  - `awk` ✓

Brainstorm

1. Shell Wildcards vs Regex Metacharacters?
2. Using the filters discussed thus far, how would you:
   - Search and replace a specific pattern?
   - Do number crunching on an input text?

Brainstorm

1. Shell Wildcards vs Regex Metacharacters?

2. Using the filters discussed thus far, how would you:
   - Search and replace a specific pattern?
   - Do number crunching on an input text?

Regex-Aware Filters: 1. sed

- Sed stands for **s**tream **ed**itor.
- Derived from ed, the original unix editor.
- A powerful *noninteractive* text manipulation tool.
- Operates on a stream of text it receives (e.g. from `STDIN`, pipeline) on the fly and writes the output to `STDOUT`.
- Line-based processing cycle.
  - `read` → `buffer (aka pattern space)` → `edit` → `print`
- A complete programming language. (see this game written in sed).

sed usage

```
sed [options] 'instruction' file
```

- `instruction` - is user-supplied edit command with the form: 'address action'.
- address - specifies where in the text to take the action at.
- action - specifies action commands (substitute, delete, print, etc).
- Common [options] include:
  - -n: suppress default printing when using the print (p) command.
  - -e: for multiple instructions per line, each preceded by it.
  - -f <file>: read instruction from <file>.
  - -r: use the ERE metacharacter set (sed defaults to BRE).

Address Specifiers

```
sed [option] 'address action' <filename>
```

address can be specified as:

- 'n action' → take <action> at line number n.
- 'm,n action' → take <action> between lines m and n.
- 'm~n action' → starting from line m, take <action> on every n^{th} line from m.
- '$ action' → take <action> on the last line.
- 'n! action'→ take <action> on all but line n.

Action Specifiers

`sed [options] 'address action' <filename>`

`action` can be:

- `p` print line(s).
- `d` delete line(s).
- `s/old/new` substitute *first occurrence* of 'old' by 'new'.
- `w <filename>` write edited output to `<filename>`.
- `q` quit after reading specified lines.

sed Operations: Print (without regex)

```
sed -n 'address p' filename
```

Example

```
sed -n '3p' file.txt                          (try without -n)

sed -n '1,5p' file.txt

sed -n '2~2p' file.txt

sed -n '$p' file.txt

sed -n '4,$!p' file.txt

sed -n -e '1p' -e '3,5p' file.txt

sed -n '1p;3,5p' file.txt                     (; is delimiter)

sed '10q' file.txt                            (head??)
```

Print Format (with regex)

```
sed -n '/regex/p' filename          (1)
sed -n '/regex/, Np' filename       (2)
sed -n 'N, /regex/p' filename       (3)
sed -n '/regex1/,/regex2/p' filename (4)
```

1. emulates `grep`
2. matches `regex` upto the N^{th} line
3. matches `regex` from N^{th} line onwards
4. matches lines between the two regexes.

Example

```
ls -l | sed -nr '/^.{5}w/p'

ls -l | sed -n '/^....w/p'

sed -n '/foo/,/bar/p' MyFile.txt
```

Print Format (with regex)

```
sed -n '/regex/p' filename          (1)
sed -n '/regex/, Np' filename       (2)
sed -n 'N, /regex/p' filename       (3)
sed -n '/regex1/,/regex2/p' filename (4)
```

1. emulates `grep`
2. matches `regex` upto the Nth line
3. matches `regex` from Nth line onwards
4. matches lines between the two regexes.

Example

```
ls -l | sed -nr '/^.{5}w/p'

ls -l | sed -n '/^.....w/p'

sed -n '/foo/,/bar/p' MyFile.txt
```

sed Operations: Deletion

Delete Format

(1) `sed 'address d' filename`

(2) `sed '/regex/d' filename`

1. without regex
2. with regex

Example

```
sed '1,5d' Myfile.txt

sed '/b[oa]*/d' Myfile.txt

sed '/^$/d' Myfile.txt

cat Myfile.txt | sed '/^....$/d'
```

sed Operations: Deletion

Delete Format

(1) `sed 'address d' filename`

(2) `sed '/regex/d' filename`

1. without regex
2. with regex

Example

```
sed '1,5d' Myfile.txt

sed '/b[oa]*/d' Myfile.txt

sed '/^$/d' Myfile.txt

cat Myfile.txt | sed '/^....$/d'
```

Substitution (without regex)

Find-and-replace is what sed is best at.

Substitution Operator (s//)

```
sed '[address] s/old/new/flags' filename
```

- Searches for occurrence of <old> and substitutes it with <new> at the specified address (optional).
- Common flags include :
  - `a number` specifies which occurrence must be replaced.
  - `g` replaces every (global) occurrence of `old` with `new`.
  - `i` case-insensitive operation.
  - `w filename` writes to the given file.

Example

```
sed ‘s/one/ek/’ hinglish.txt                    (try sed -i)

sed -n ‘s/four/char/gp’ hinglish.txt

sed -n ‘s/three/teen/gpw output.txt’ hinglish.txt

sed -n ‘1,3s/four/char/pw output.txt’ hinglish.txt
```

Substitution Format (with regex)

```
sed '/regex/s/old/new/flags' filename
```

- Searches for pattern <old> and replaces with <new> string wherever <regex> matches.
- The expression /regex/ is optional.

Example

```
sed '/#/s/include/define/g' input.txt        (@ lines with #)

sed 's/saviou\?r/SAVIOR/g' input.txt

sed 's/singer/lead &/' input.txt             (& is an operator)

sed 's/\(Days\)\(of\)\(Ancient\)/\3 \2 \1 /g' input.txt

sed -r 's/(Days).*(of).*(Ancient)/\3 \2 \1 /g' input.txt
```

sed: Pros & Cons

Some Pros:

- Regex handling ✓
- Search and replace feature ✓
- Fast ✓

Some cons:

- No feature for numeric computation
- Going backward in the file not possible

Regex-Aware Filters: 2. `Awk`

- Named after its authors: **A**lfred Aho, Peter **W**einberger, and Brian **K**ernighan.
- A powerful *programming language* for text manipulation + report writing (precursor to `perl`).
- C-like syntax (functions, arrays, `if`, `for` & `while` constructs, etc).
- Combines features from many filters (e.g. `grep`, `sed`).
- Flavors: `new awk (nawk)`, `GNU awk (gawk)`, `...`

Awk: Salient Features

- Processes a line at a time (like sed)
- Numeric processing
- Can manipulate *fields* of a line (N.B. sed processes lines)
- Regex-aware (ERE)
- Report formatting capabilities
- C-like. The implication?

Awk Usage

```
awk [options] 'pattern {action}' file(s)
```

- Searches for `pattern` and applies `action` on it.
- Default action is to print current record on `STDOUT`.
- Default pattern is to match all lines.
- If file(s) not specified, input taken from??
- Common options:
  - `-f` read program/pattern from a file
  - `-F` sets field separator (FS) value (default is " ")

In Awk,

- Each line in the file ≡ record ($0)
- Each column ≡ field. ($1, $2, $3, ...)

Example

```
ls -l | awk '{print}'

ls -l | awk '{print $0}'          (How about $1, $2 ...?)

ls -l | awk '/^d/ {print $1,$8}'          (The comma??)

ls -l | awk '$5>100 {print $8}'

awk '/Sa[mt]r*/' file.txt

awk -F: '{print $7}' /etc/passwd
```

print vs printf

- Both write to STDOUT
- (un)?formatted output
- The C-like printf takes format specifiers (%d,%f,%s)

```
awk '{ print $1, $2, $3 }' sales.txt
```

```
awk '{ printf "%6s %4d %-8f \n", $1, $2, $3 }' sales.txt
```

(The '-' symbol left-justifies, printf requires '\n' to start new line)

Operators

Arithmetic

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
$\wedge$	Exponentiation

Relational

<, <=	less, less or equal
>, >=	greater, greater or equal
==, !=	equal to, not equal to
$\sim$, !$\sim$	for regex comparison

Logical

&&	AND
||	OR
!	NOT

Example

```
echo 100 8 | awk ‘{print $1 ^ $2}’

ls -l | awk ‘$2==2 {print}’          (vs $2=2 ??)

awk ‘$2 * $3 > 50 {print}’ sales.txt

awk ‘$3 > 10 && $4 > 20 {print}’ sales.txt

awk -F: ‘$1 ~ /^root/ {print}’ /etc/passwd
```

Awk: Variables

- No primitive data types (`char, int, float ...`).
- Either `string` or `number` (implicitly set to " " and 0 resp).
- Built-in + user-defined variables (no need to declare them).

`FS`	Field separator (default is space/tab)
`RS`	Record separator (default is newline)
`NF`	# of fields in the current line
`NR`	# of lines read so far
`FILENAME`	Name of the current input file

```
ls -l | awk '{print $1, NF}'
```

```
awk -F, 'NR==2, NR==10 {print NR, $1}' file.csv
```

The BEGIN...END Sections

```
BEGIN {action}
END {action}
```

- Optional sections for pre- and post-processing works.
- Way of telling Awk to do something before and after scanning through the file.
- Example usage:
  - BEGIN: generate report header, initialize variables, etc
  - END: print final result of computation, print output status, etc

Example

```
awk ‘BEGIN {n=0} {print n++, $0} END {print "Bye"}’ file.txt



ls -l | awk ‘BEGIN {printf "Permissions \t File Name \n"}

            { printf "%s \t %s \n", $1, $9 } ’



ls | awk ‘BEGIN { print "List of C files:" } /\.c$/ {print}

          END { print "Done!" } ’
```

Awk: The if Statement

- Awk provides control flow statements:

  Branching (if...else) + loop (for, while & do...while).

if...else

```
{ if (condition) {statement 1} else {statement 2} }
```

Example

```
ls -l | awk '$5 > 1000 { print }'
```

```
ls -l | awk '{ if ($5 > 1000) { print "Big file"} else
    { print "Small file" } }'
```

Awk: The if Statement

- Awk provides control flow statements:

  Branching (if...else) + loop (for, while & do...while).

if...else

```
{ if (condition) {statement 1} else {statement 2} }
```

Example

```
ls -l | awk '$5 > 1000 { print }'
```

```
ls -l | awk '{ if ($5 > 1000) { print "Big file"} else
              { print "Small file" } } '
```

Awk in Action

1. Swap the order of any two columns of a file.

```
awk '{print $2, $1}' < input.txt > output1.txt
```

2. Delete the 3rd column of `ls -l`.
3. Find the maximum/minimum value of a column.
4. Find the average of a column of data.
5. Calculate the sum of all columns of data.
6. Read about loop statements!

Awk in Action

1. Swap the order of any two columns of a file.

```
awk '{print $2, $1} < input.txt > output1.txt
```

2. Delete the 3rd column of `ls -l`.

3. Find the maximum/minimum value of a column.

4. Find the average of a column of data.

5. Calculate the sum of all columns of data.

6. Read about loop statements!

Awk in Action

1. Swap the order of any two columns of a file.

```
awk '{print $2, $1} < input.txt > output1.txt
```

2. Delete the 3rd column of `ls -l`.

```
ls -l | awk '{$3 = ""; print}' > output2.txt
```

3. Find the maximum/minimum value of a column.

4. Find the average of a column of data.

5. Calculate the sum of all columns of data.

6. Read about loop statements!

Awk in Action

1. Swap the order of any two columns of a file.

```
awk '{print $2, $1} < input.txt > output1.txt
```

2. Delete the 3rd column of `ls -l`.

```
ls -l | awk '{$3 = ""; print}' > output2.txt
```

3. Find the maximum/minimum value of a column.

4. Find the average of a column of data.

5. Calculate the sum of all columns of data.

6. Read about loop statements!

Awk in Action

1. Swap the order of any two columns of a file.

```
awk '{print $2, $1} < input.txt > output1.txt
```

2. Delete the 3rd column of `ls -l`.

```
ls -l | awk '{$3 = ""; print}' > output2.txt
```

3. Find the maximum/minimum value of a column.

```
awk 'BEGIN {max = 0} { if ($1>max) max=$1 } END {print max}'
```

4. Find the average of a column of data.

5. Calculate the sum of all columns of data.

6. Read about loop statements!

Awk in Action

1. Swap the order of any two columns of a file.

```
awk '{print $2, $1} < input.txt > output1.txt
```

2. Delete the 3rd column of `ls -l`.

```
ls -l | awk '{$3 = ""; print}' > output2.txt
```

3. Find the maximum/minimum value of a column.

```
awk 'BEGIN {max = 0} { if ($1>max) max=$1 } END {print max}'
```

4. Find the average of a column of data.

5. Calculate the sum of all columns of data.

6. Read about loop statements!

Awk in Action

1. Swap the order of any two columns of a file.

```
awk '{print $2, $1} < input.txt > output1.txt
```

2. Delete the 3rd column of `ls -l`.

```
ls -l | awk '{$3 = ""; print}' > output2.txt
```

3. Find the maximum/minimum value of a column.

```
awk 'BEGIN {max = 0} { if ($1>max) max=$1 } END {print max}'
```

4. Find the average of a column of data.

```
cat input.txt | awk 'BEGIN {ave=0} {ave+=$1} END {print ave/NR}'
```

5. Calculate the sum of all columns of data.

Read about loop statements!

Awk in Action

1. Swap the order of any two columns of a file.

```
awk '{print $2, $1} < input.txt > output1.txt
```

2. Delete the 3rd column of `ls -l`.

```
ls -l | awk '{$3 = ""; print}' > output2.txt
```

3. Find the maximum/minimum value of a column.

```
awk 'BEGIN {max = 0} { if ($1>max) max=$1 } END {print max}'
```

4. Find the average of a column of data.

```
cat input.txt | awk 'BEGIN {ave=0} {ave+=$1} END {print ave/NR}'
```

5. Calculate the sum of all columns of data.

Read about loop statements!

Awk in Action

1. Swap the order of any two columns of a file.

```
awk '{print $2, $1} < input.txt > output1.txt
```

2. Delete the 3rd column of `ls -l`.

```
ls -l | awk '{$3 = ""; print}' > output2.txt
```

3. Find the maximum/minimum value of a column.

```
awk 'BEGIN {max = 0} { if ($1>max) max=$1 } END {print max}'
```

4. Find the average of a column of data.

```
cat input.txt | awk 'BEGIN {ave=0} {ave+=$1} END {print ave/NR}'
```

5. Calculate the sum of all columns of data.

Read about loop statements!

Awk: Final Remarks

- What has been discussed so far is just tip of the iceberg.

- Awk's programming features not discussed today:
  - Loop statements (`for`, `while`, `do...while`)
  - Arrays
  - Functions

- There are many Awk one-liners. Check out Commandlinefu and this too.