

Regular expressions and sed & awk

Regular expressions

- Key to powerful, efficient, and flexible text processing by allowing for variable information in the search patterns
- Defined as a string composed of letters, numbers, and special symbols, that defines one or more strings
- You have already used them in selecting files when you used asterisk (*) and question mark characters to select filenames
- Used by several Unix utilities such as `ed`, `vi`, `emacs`, `grep`, `sed`, and `awk` to search for and replace strings

- Checking the author, subject, and date of each message in a given mail folder

```
egrep "^ (From|Subject|Date): " <folder>
```

- The quotes above are not a part of the regular expression but are needed by the command shell
- The metacharacter | (or) is a convenient one to combine multiple expressions into a single expression to match any of the individual expressions contained therein

- * The subexpressions are known as *alternatives*

- A regular expression is composed of characters, delimiters, simple strings, special characters, and other metacharacters defined below

- Characters

- A character is any character on the keyboard except the newline character '\n'
- Most characters represent themselves within a regular expression
- All the characters that represent themselves are called *literals*
- A special character is one that does not represent itself (such as a metacharacter) and needs to be *quoted*
 - * The metacharacters in the example above (with `egrep`) are `"`, `^`, `(`, `|`, and `)`
- We can treat the regular expressions as a language in which the literal characters are the *words* and the metacharacters are the *grammar*

- Delimiters

- A delimiter is a character to mark the beginning and end of a regular expression
- Delimiter is always a special character for the regular expression being delimited
- The delimiter does not represent itself but marks the beginning and end of the regular expression
- Any character can be used as a delimiter as long as it (the same character) appears at both ends of the regular expression
- More often than not, people use forward slash '/' as the delimiter (guess why)
- If the second delimiter is to be immediately followed by a carriage return, it may be omitted
- Delimiters are not used with the `grep` family of utilities

- The metacharacters in the regular expressions are

```
^ $ . * [ ] \{ \} \ \ ( \)
```

- In addition, the following metacharacters have been added to the above for extended regular expressions (such as the one used by `egrep`)

```
+ ? | ( )
```

- The dash (-) is considered to be a metacharacter only within the square brackets to indicate a range; otherwise, it is treated as a literal

- * Even in this case, the dash cannot be the first character and must be enclosed between the beginning and the end of range characters

- The regular expression search is not done on a word basis but utilities like `egrep` display the entire line in which the regular expression matches

- Simple strings

- The most basic regular expression
- Matches only itself
- Examples

Reg. Exp.	Matches	Examples
<code>/ring/</code>	ring	ring spring ringing stringing
<code>/Thursday/</code>	Thursday	Thursday Thursday's
<code>/or not/</code>	or not	or not poor nothing

- Special characters

- Cause a regular expression to match more than one string
- Period
 - * Matches any character
 - * Examples

Reg. Exp.	Matches	Examples
<code>/.alk/</code>	All strings that contain a space followed by any character followed by <code>alk</code>	will talk may balk
<code>/.ing/</code>	all strings with any character preceding <code>ing</code>	singing ping before inglenook
<code>/09.17.98/</code>	Date with any separator	09/17/98 09-17-98

- Square brackets

- * Define a class of characters that matches any single character within the brackets
- * If the first character immediately following the left square bracket is a caret `^`, the square brackets define a character class that match any single character not within the brackets
- * A hyphen can be used to indicate a range of characters
- * Within a character class definition, the special characters (backslash, asterisk, and dollar signs) lose their special meaning
- * A right square bracket appearing as a member of the character class can only appear as the first character following the square bracket
- * A caret is special only if it is the first character following the square bracket
- * A dot within square brackets will not be a metacharacter
 - `/07[.-]17[.-]98/` will not match `07/17/98` but will match `07-17-98`
- * Examples

Reg. Exp.	Matches	Examples
/[bB]ill/	Member of the character class b and B followed by ill	bill Bill billed
/t[aeiou].k/	t followed by a lowercase vowel, any character, and a k	talkative stink teak tanker
/number [6-9]/	number followed by a space and a member of the character class 6 through 9	number 60 number 8: get number 9
/[^a-zA-Z]/	any character that is not a letter	1 7 @ . } Stop!

– Asterisk

- * Can follow a regular expression that represents a single character
- * Represents zero or more occurrences of a match of the regular expression
- * An asterisk following a period matches any string of characters
- * A character class definition followed by an asterisk matches any string of characters that are members of the character class
- * A regular expression that includes a special character always matches the longest possible string, starting as far toward the beginning (left) of the line as possible
- * Examples

Reg. Exp.	Matches	Examples
/ab*c/	a followed by zero or more b's followed by a c	ac abc abbc debbcaabbbc
/ab.*c/	ab followed by zero or more other characters followed by a c	abc abxc ab45c xab 756.345 x cat
/t.*ing/	t followed by zero or more characters followed by ing	thing ting I thought of going
/[a-zA-Z]*/	a string composed only of letters and spaces	1. any string without numbers or punctuation!
/(.*)/	as long a string as possible between (and)	Get (this) and (that);
/([^]*)/	the shortest string possible that starts with (and ends with)	(this) Get (this and that)

– Caret and dollar sign

- * A regular expression beginning with a caret '^' can match a string only at the beginning of a line
 - The regular expression `cat` finds the string `cat` anywhere on the line but `^cat` matches only if the string `cat` occurs at the beginning of the line
 - `^` is used to *anchor* the match to the start of the line
- * A dollar sign '\$' at the end of a regular expression matches the end of a line
 - The regular expression `cat` finds the string `cat` anywhere on the line but `cat$` matches only if the string `cat` occurs at the end of the line, it cannot be followed by any character but newline (not even space)

* Examples

Reg. Exp.	Matches	Examples
<code>/^T/</code>	a T at the beginning of a line	This line ... That time...
<code>/^[0-9]/</code>	a plus sign followed by a number at the beginning of a line	+5 + 45.72 +759 Keep this...
<code>/:\$/</code>	a colon that ends a line	...below:

– Quoting special characters

* Any special character, except a digit or a parenthesis, can be quoted by preceding it with a backslash

* Quoting a special character makes it represent itself

* Examples

Reg. Exp.	Matches	Examples
<code>/end\./</code>	all strings that contain end followed by a period	The end. send. pretend.mail
<code>/\\</code>	a single backslash	\
<code>/*/</code>	an asterisk	*.c an asterisk (*)
<code>/\[5\]/</code>	[5]	it was five [5]
<code>/and\ or/</code>	and/or	and/or

– Range metacharacters

* Used to match a number of expressions

* Described by the following rules

$r\{n\}$ Match exactly n occurrences of regular expression r

$r\{n,\}$ Match at least n occurrences of regular expression r

$r\{n,m\}$ Match between n and m occurrences of regular expression r

Both n and m above must be integers between 0 and 256

For now, r must be considered to be a single character regular expression (strings must be enclosed in bracketed regular expressions)

– Word metacharacters

* The word boundaries in the regular expressions are denoted by any whitespace character, period, end-of-line, or beginning of line

* Expressed by

`\<` beginning of word

`\>` end of word

● Rules

– Longest match possible

* A regular expression always matches the longest possible string, starting as far towards the beginning of the line as possible

– Empty regular expressions

* An empty regular expression always represents the last regular expression used

* Let us give the following command to vi

```
:s/mike/robert/
```

* If you want to make the same substitution again, the following is sufficient

```
:s//robert/
```

* You can also do the following

```
/mike/
:s//robert
```

- Bracketing expressions

- Regular expressions can be bracketed by quoted parentheses `\ (and \)`
- Quoted parentheses are also known as *tagged metacharacters*
- The string matching the bracketed regular expression can be subsequently used as quoted digits
- The regular expression does not attempt to match quoted parentheses
- A regular expression within the quoted parentheses matches exactly with what the regular expression without the quoted parentheses will match
- The expressions `/\ (regexp\) /` and `/regexp/` match the same patterns
- Quoted digits

- * Within the regular expression, a quoted digit `(\n)` takes on the value of the string that the regular expression beginning with the *n*th `\ (` matched
- * Assume a list of people in the format

```
last-name, first-name initial
```

- * It can be changed to the format

```
first-name initial last-name
```

by the following `vi` command

```
:%s/\ ([^,]*\ ), \ (.*)\ /\2 \1/
```

- Quoted parentheses can be nested
- * There is no ambiguity in identifying the nested quoted parentheses as they are identified by the opening `\ (`
- * Example

```
/\ ([a-z]\ ([A-Z]*\ )x\ )
```

matches

```
3 t dMNORx7 l u
```

- Replacement string

- `vi` and `sed` use regular expressions as search strings with the substitute command
- Ampersands (`&`) and quoted digits `(\n)` can be used to match the replacement strings within the replacement string
- An ampersand takes on the value of the string that the search string matched
- Example

```
:s/[0-9][0-9]*/Number &/
```

- Redundancy

- You can write the same regular expression in more than one way
- To search for strings `grey` and `gray` in a document, you can write the expression as `gr[ae]y`, or `grey|gray`, or `gr(a|e)y`
 - * In the last case, parentheses are required as without those, the expression will match `gra` or `ey` which is not the intention

- Regular expressions cannot be used for the newline character

sed

- Stream editor

- Derivative of `ed`
 - Takes a sequence of editor commands
 - Goes over the data line by line and performs the commands on each line

- Basic syntax

```
sed 'list of ed commands' filename[s] ...
```

- The commands are applied from the list in order to each line and the edited form is written to `stdout`
- Changing a pattern in the file

```
sed 's/pat_1/pat_2/g' in_file > out_file
```

- `sed` does not alter the contents of the input file
- Quotes around the list of commands are necessary as the `sed` metacharacters should not be translated by the shell
- Selecting range of lines
- Command to remove the mail header from a saved mail message

```
sed '1,/^$/d' in_file > out_file
```

- Removing the information from the output of the `finger` command to get only the user id and login time

```
finger | sed 's/\([a-zA-Z][a-zA-Z]*\) .* \([0-9][0-9]:[0-9][0-9]\) .*/\1 \2/'
```

- Problem: The first line should have been removed as well

```
finger | sed 's/\([a-zA-Z][a-zA-Z]*\) .* \([0-9][0-9]:[0-9][0-9]\) .*/\1 \2/' | sed '1d'
```

- Indenting a file one tab stop

```
sed 's/^/->/' file
```

- The above matches all the lines (including empty lines)
- Problem can be solved by

```
sed '/./s/^/->/' file
```

- Another way to do it

```
sed '/^$/!s/^/->/' file
```

- Multiple commands in the same invocation of `sed`

```
$ finger | sed 's/\([a-zA-Z][a-zA-Z]*\) .* \([0-9][0-9]:[0-9][0-9]\) .*/\1 \2/  
> 1d'
```

The commands must be on separate lines

- `sed` scripts

- The `sed` commands can be put into script files and can be executed by

```
sed -f cmdfile in_file
```

- Lines containing a pattern can be deleted by

```
sed '/rexp/d'
```

- Automatic printing

- By default, `sed` prints each line on the `stdout`
- This can be inhibited by using the `-n` option as follows

```
sed -n '/pattern/p'
```

- Matching conditions can be inverted by the `!`

```
sed -n '/pattern/!p'
```

- The last achieves the same effect as `grep -v`

- Inserting newlines

- Converting a document from single space to double space

```
$ sed 's/$/\
> /'
```

- Creating a list of words used in the document

```
$ sed 's/[ ->][ ->]*/\
> /g' file
```

- Counting the unique words used in the document

```
$ sed 's/[ ->, .][ ->, .]*/\
> /g' file | sort | uniq | wc -l
```

- Writing on multiple files

```
$ sed -n '/pat/w file1
> /pat/!w file2' filename
```

- Line numbering

- Line numbers can be used to select a range of lines over which the commands will operate
- Examples

```
$ sed -n '20,30p'
$ sed '1,10d'
$ sed '1,/^\$/d'
$ sed -n '/^\$/,/^\end/p'
```

- `sed` does not support relative line numbers (difference with respect to `ed`)

awk

- Acronym for the last names of its designers – Aho, Weinberger, Kernighan
- Not as good as `sed` for editing but includes arithmetic, variables, built-in functions, and a programming language like C; on the other hand, it is a more general processing model than a text editor
- Looks more like a programming language rather than a text editor

Table 1: Summary of sed commands

a\ b label c\ d i\ l p q r file s/pat1/pat2/f t label w file y/str1/str2/ = !cmd : label {	append lines to output until one not ending in \ branch to command : label change lines to following text (as in a\ delete lines insert following text before next output list line, making all non-printing characters visible (tabs appear as >; lines broken with \ print line quit (for scripts) read file, copy contents to stdout substitute pat2 for pat1 f = g, replace all occurrences f = p, print f = w file, write to file test: branch to label if substitution made to current line write line(s) to file replace each character from str1 with corresponding character from str2 (no ranges allowed) print current input line number do sed cmd if line is not selected set label for b and t commands treat commands up to the matching } as a group
--	---

- Mostly used for formatting reports, data entry, and data retrieval to generate reports
- awk is easier to use than sed but is slower
- Usage is

```
awk 'awk_script' files
```

- The awk_script looks like

```
pattern { action }
pattern { action }
...
```

- *Input-driven language*

- awk reads one line in the file at a time, compares with each pattern, and performs the corresponding action if the pattern matches
- There is no effect if the input file is empty
- Run the following commands to see the effect:

```
touch foobar
awk '{print "Hello World"}' foobar
cat "Line 1" >> foobar
awk '{print "Hello World"}' foobar
cat "Line 1" >> foobar
awk '{print "Hello World"}' foobar
```

- As it reads each line, `awk` immediately breaks those up into segments (field) based on a specified field separator (FS)
- If you want to make `awk` work on empty file, you can use the keyword `BEGIN`

```
awk 'BEGIN {print "Hello World"}'
```

- Just like `sed`, `awk` does not alter its input files
- The patterns in `awk` can be regular expressions, or C-like conditions
- `grep` can be written in `awk` as

```
awk '/regular expression/ { print }' filename
```

- Printing a message for each blank line in file

```
awk '/^$/ { print "Encountered a blank line" }' filename
```

- Either of pattern or action is optional and can be omitted

- Omitting pattern performs the action on every line

```
awk '{ print }' filename
```

- Omitting action prints matched lines

```
awk '/regular expression/' filename
```

- Just like `sed`, the `awk_script` can be presented to `awk` from a file by using

```
awk -f awk_script_file filename
```

- `awk` programming model

- Main input loop

- * Loop reads each line of input from file and makes it available for processing
- * loop iterates as many times as the lines of input
- * Loop terminates when there is no more input to be read

- Two special keywords – `BEGIN` and `END` specify the commands to be executed before the beginning of loop and at the end of loop, respectively

- * The blocks specified by these two keywords are optional

- Fields

- A field is a string of characters, separated by FS

- By default, FS is any whitespace character

- FS can be specified by a command line option

- * Changing the field separator to colon (:)

```
awk -F: '/regular expression/ { action }' file
```

- * To print the user names and real names in the `passwd` file

```
awk -F: '{print $1"\t"$5}' /etc/passwd
```

- The output of `who` has six fields as follows

```
sanjiv console Nov 18 13:26
sanjiv tty0 Nov 18 13:26 (:0.0)
sanjiv tty0 Nov 19 13:27 (:0.0)
vlad tty7 Nov 19 16:46 (arrak13.umsl.edu)
```

- The fields are called \$1, \$2, ..., \$NF
 - * NF is a variable whose value is set to the number of fields
 - * NF and \$NF are not the same
 - NF is the number of fields
 - \$NF is the contents (string) of the last field

- Printing

- The current input line (or record) is tracked by the built-in variable NR
- The entire input record is contained in the variable \$0
- To add line numbers to each line, you can use the following

```
awk '{print NR, $0}' filename
```

- Fields separated by comma are printed separated by the field separator – a blank space character by default
- Complete control of the output format can be achieved by using printf instead of print as follows

```
awk '{ printf "%4d %s\n", NR, $0 }' filename
```

- printf in awk is almost identical to the corresponding C function

- Patterns

- Checking for people who do not have a password entry in the file /etc/passwd

```
awk -F: '$2 == ""' /etc/passwd
```

- Checking for people who have a locked password entry

```
awk -F: '$2 == "*" ' /etc/passwd
```

- Other ways to check for empty string

\$2 == ""	2nd field is empty
\$2 ~ /^\$/	2nd field matches empty string
\$2 !~ /. /	2nd field does not match any character
length(\$2) == 0	length of 2nd field is zero

- The symbol ~ indicates a regular expression match while !~ indicates a regular expression non-match
- length is a built-in function to count the number of characters in the string (or field)
- Any pattern match can be preceded by ! to negate its match as follows

```
awk -F: '!( $2 == " " )' filename
```

- Data validation using the number of fields as criterion – line valid if the number of fields is odd

```
print $LINE | awk 'NF % 2 != 0'
```

- Printing excessively long lines (> 72 characters)

```
awk 'length($0) > 72' filename
```

- Above problem with more informative solution

```
$ awk '(length($0) > 72) \
{ print "Line", NR, "too long: ", substr($0,1,50)}' filename
```

- The function substr(s, m, n) produces the substring of s beginning at position m and with a length of n characters; if n is omitted, it continues to the end of string
- Extracting information with substr

```
$ date
Wed Nov 20 14:27:33 CST 1996
$ date | awk '{ print substr ( $4, 1, 5 ) }'
14:27
```

- The BEGIN and END patterns

- Special patterns used in awk scripts
- BEGIN actions are performed before the first input line has been read (used to initialize variables, print headings, and like)

- * Setting the field separator within the script

```
$ awk 'BEGIN {FS = ":"}
>      $2 == "" ' /etc/passwd
```

- END actions are done after the last line has been processed

- * Printing the number of lines in the input

```
awk 'END { printf NR }' ...
```

- Arithmetic and variables

- awk allows you to do more sophisticated arithmetic compared to the shell
- Adding the numbers in a column (first column), and printing the sum and average

```
      { s = s + $1 }
END    { print s, s/NR }
```

- Variables can be created by users and are initialized to zero by default

- awk also allows for shorthand arithmetic operators like C

```
      { s += $1 }
END    { print s, s/NR }
```

- Implementing wc in all its generality

```
$ awk '{ nc += length ( $0 ) + 1      # number of chars, 1 for \n
      nw += NF                        # number of words
      }
      END { print NR, nw, nc }' filename
```

- Variables can also store string of characters and the interpretation is based on context

- awk maintains a number of built-in variables of both types

Developing man pages with [nt]roff

“Acts oddly on nights with full moon.”

– BUGS section for catman from 4.2BSD Unix manual

- nroff and troff

- Native Unix programs to format text
- Based on requests within the documents that start with a period in the first column
- Commonly used requests are

```

.I    Italicize following line
.B    Following line in bold
.R    Following line in Roman
.br   Break the line
.ce   Center the following line
.fi   Fill lines (Align right margins)
.ft   Set font
.na   No right alignment
.nf   Do not fill lines (Preferable to .na)
.sp   One vertical line

```

- The manual page

- Stored in a subdirectory in the directory `/usr/man`
- The subdirectory is called `manx` where `x` is a digit or character to indicate the section of the manual
- The sections are numbered 1 to 8 and `n` and `l`

```

1    User commands
2    System calls
3    C Library functions
4    Devices and network interfaces
5    File formats
6    Games and demos
7    Environments, tables, and troff macros
8    Maintenance commands
1    Misc. reference manual pages (Locally developed and installed)
n    Misc. reference manual pages (New commands)

```

- Printed with the `man(1)` command
 - * A shellscript that runs `nroff -man` but may be compiled on newer machines
 - * The locally developed man pages can be tested for printing with `nroff -man` command
 - * The man pages in a given section can be printed by specifying the section number, for example, the man page for the system call `umask` can be printed by typing the command

```
man 2 umask
```

If the section number is not specified, the output will be for the user command from section 1

- The macros for `man` are discussed in section 7 of the manual and can be invoked by

```
man 7 man
```

- No manuals on the kernel

- Usual device driver man pages are user-level descriptions and not internal descriptions
- A regular joke was “Anyone needing documentation to the kernel functions probably shouldn’t be using them.”
- `/* you are not expected to understand this */` – from Unix V6 kernel source

- Layout of a Unix manual page

- The manual page is laid out as per the specifications in the `man` macro of `troff`
 - * Any text argument may be zero to six words
 - * Quotes can be used to include the space character in a “word”
 - * Some native `nroff` conventions are followed, for example, if text for a command is empty, the command is applied to the next line
 - A line starting with `.I` and with no other inputs italicizes the next line
 - * The prevailing indentation distance is remembered between successive paragraphs but not across sections
- The basic layout of a man page is described by

```
.TH COMMAND <section-number>
.SH NAME
command \- brief description of function
.B command
options
.SH DESCRIPTION
Detailed explanation of programs and options.
Paragraphs are introduced by .PP
.PP
This is a new paragraph.
.SH FILES
Files used by the command, e.g., passwd(1) mentions /etc/passwd
.SH "SEE ALSO"
References to related documents, including other manual pages
.SH DIAGNOSTICS
Description of any unusual output (e.g., see cmp(1))
.SH BUGS
Surprising features (not always bugs)
```

- If any section is empty, its header is omitted
- The `.TH` line and the `NAME`, `SYNOPSIS`, and `DESCRIPTION` sections are mandatory
- The `.TH` line
 - * Begins a reference page
 - * The full macro is described by


```
.TH command section date_last_changed left_page_footer center_header
```
 - * Sets prevailing indent and tabs to 0.5"
- The `.SH` lines
 - * Section headers
 - * Identify sections of the manual page
 - * `NAME` and `SYNOPSIS` sections are special; other sections contain ordinary prose
 - * `NAME` section
 - Names the command (in lower case)
 - Provides a one-line description of it
 - * `SYNOPSIS` section
 - Names the options, but does not describe them
 - The input is free form
 - Font changes can be described with the `.B`, `.I`, and `.R` macros
 - The name and options are bold while the rest of the information is in roman
 - * `DESCRIPTION` section
 - Describes the commands and its options
 - It tells the usage of the command
 - The man page for `cc(1)` describes how to invoke the compiler, optimizer, where the output is, but does not provide a reference page for the manual
 - The reference page can be cited in the `SEE ALSO` section
 - However, `man(7)` is the description of the language of manual macros
 - Command names and tags for options are printed in italics, using the macros `.I` (print first argument in italics) and `.IR` (print first argument in italic, second in roman)
 - * `FILES` section
 - Mentions any files implicitly used by the commands

- * **DIAGNOSTICS** section
 - Optional section and generally not present
 - Reports any unusual output produced by the command
 - May contain diagnostic messages, exit statuses, or surprising variations of the command's normal behavior
- * **BUGS** section
 - Could be called **LIMITATIONS**
 - Reports shortcomings in the program that may need to be fixed in a future release
- Other requests and macros for `man`
 - .IP x Indented paragraph with a tag x
 - .LP Left-aligned paragraph
 - .PP Same as .LP
 - .SS Section subheading