EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Dr. Eberle Zentrum für digitale Kompetenzen

DataBASHing - Reguläre Ausdrücke

Martin Raden

Einführung von

- Reguläre Ausdrücke
- `grep` in Zeilen filtern
- `tr, sed` in Text ersetzen
- Probleme mit Zeilenumbrüchen

Zum Warmwerden gibts die folgenden Videos.

Video: Bashino - #02: was soll das Ganze & mehr von egrep [5 min]

Video: Bashino - #01 cat und egrep [11 min]

Video: Bashino - #10 mit sed suchen und ersetzen [7 min]

Im (optionalen) Folgevideo #11 Gezieltes Ersetzen mit sed [8 min] wird noch einmal mehr auf sed und reguläre Ausdrücke eingegangen.

Wer immer noch nicht genug hat, kann sich im sed Video von Pedagogy (en) [18 min] noch haufenweise andere “sed magics” anschauen.

Reguläre Ausdrücke

Reguläre Ausdrücke sind einer der Grundpfeiler für die konsolengestützte Datenprozessierung. Damit kann man automatisiert und/oder mit wenig Aufwand

- **nach** Vorkommen von **Textmustern suchen** (z.B. via `grep`), um
  - nur gesuchte Muster anzeigen
  - alle Zeilen(nummern) anzuzeigen, die das Muster enthalten
  - alle Dateien zu finden, die ein Textmuster beinhalten
- Textblöcke **verändern und ersetzen** (z.B. via `sed`)

Dabei ist unter “Textmuster” nicht ein exaktes Wort oder dergleichen zu verstehen, sondern eine Suchkodierung, welche Variabilität zulässt.

Als Einsteig ein kleines Beispiel: Wir haben die folgenden Worte

soll gefunden werden	soll nicht gefunden werden
Frida	Fritz
Erna	Hugo
Lisa	Lino

Die einfachste (und am wenigsten flexible) Lösung wäre es, die gesuchten Worte alle als Alternativen hintereinander zu hängen:

- “ `Frida|Erna|Lisa` ” = wobei “ `|` ” für ein “ODER” steht

Wenn man reguläre Ausdrücke baut, muss man genau schauen, was die gesuchten Wörter (oder Textpassagen) gemeinsam haben. In unserem Fall enden alle gesuchten Namen mit “a”, was wir verwenden können. Was davor für Buchstaben kommen (und wieviele), ist erst einmal unerheblich, um die Worte von den nicht gesuchten zu unterscheiden. Daher könnten wir folgendes verwenden:

- “ `.*a` ” = wobei “ `.` ” für ein beliebiges Zeichen (Buchstaben, Zahlen, Leer-, Satzzeichen, . . .) steht und “ `*` ” für eine beliebige Anzahl (auch 0) des vorherigen Teils (also hier “ `.` ”)

Das funktioniert, liefert aber ggf. zu viel Variabilität, was ein häufiger Fallstrick bei regulären Ausdrücken ist. Sprich sie sind zu unkonkret. In unserem Fall passt der Ausdruck auch auf: “..a”, “lala”, “Hans-Anna”, . . . Sie sehen das Problem? Um dieses zu Umgehen, könnte man folgende Sachen machen:

- “ `.+a` ” = wir fordern mit “ `+` ”, dass vor dem “a” mindestens ein Buchstabe ist
- “ `[a-zA-Z]+a` ” = wir geben in “ `[]` ” ganz genau an, welche Buchstaben wir erlauben (hier z.B. keine Umlaute)
- “ `\w+a` ” = dazu könnten wir auch eine vordefinierte character class (hier `\w` für Wortbuchstabe) nehmen
- “ `[A-Z][a-z]*a` ” = mit “ `[A-Z` ” sichern wir, dass der erste Buchstabe grossgeschrieben ist und anschliessend können (da “ `*` ”) nur Kleinbuchstaben in der Wortmitte sein
- . . . und so weiter

Das Beispiel zeigt, dass reguläre Ausdrücke ganz allgemein (“ `.*a` ”) oder sehr präzise (“ `Frida|Erna|Lisa` ”) sein können. Die Kunst ist es nun, den für die anstehende Aufgabe passenden regulären Ausdruck zu erdenken. Dazu hilft einem das beiliegende CheatSheet, welches man sicher auch in Zukunft immer wieder heraus holen muss.

Um regulären Ausdrücken zu entwickeln oder um damit herumzuspielen um Effekte von Änderungen zu beobachten, eigenen sich online regex tester. Diese könnten ihnen u.a. auch beim Bearbeiten der Übungen oder Tests behilflich sein. Ein solcher ist zum Beispiel

https://www.regextester.com/

CheatSheet

regularexpressions

Anchors

^	Start of line +
\A	Start of string +
$	End of line +
\Z	End of string +
\b	Word boundary +
\B	Not word boundary +
\<	Start of word
\>	End of word

Character Classes

\c	Control character
\s	White space
\S	Not white space
\d	Digit
\D	Not digit
\w	Word
\W	Not word
\xhh	Hexadecimal character hh
\Oxxx	Octal character xxx

POSIX Character Classes

[:upper:]	Upper case letters
[:lower:]	Lower case letters
[:alpha:]	All letters
[:alnum:]	Digits and letters
[:digit:]	Digits
[:xdigit:]	Hexadecimal digits
[:punct:]	Punctuation
[:blank:]	Space and tab
[:space:]	Blank characters
[:cntrl:]	Control characters
[:graph:]	Printed characters
[:print:]	Printed characters and spaces
[:word:]	Digits, letters and underscore

Assertions

?=	Lookahead assertion +
?!	Negative lookahead +
?<=	Lookbehind assertion +
?!= or ?<!	Negative lookbehind +
?>	Once-only Subexpression
?()	Condition [if then]
?()|	Condition [if then else]
?#	Comment

Note Items marked + should work in most regular expression implementations.

Sample Patterns

([A-Za-z0-9-]+)	Letters, numbers and hyphens
(\d{1,2}\/\d{1,2}\/\d{4})	Date (e.g. 21/3/2006)
([^\s]+(?=\.(jpg|gif|png))\.\2)	jpg, gif or png image
(^[1-9]{1}$\|^[1-4]{1}[0-9]{1}$|^50$)	Any number from 1 to 50 inclusive
(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)	Valid hexadecimal colour code
((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})	8 to 15 character string with at least one upper case letter, one lower case letter, and one digit (useful for passwords).
(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Email addresses
(\<(/?[^\>]+)\>)	HTML Tags

Note *These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.*

Quantifiers

*	0 or more +
*?	0 or more, ungreedy +
+	1 or more +
+?	1 or more, ungreedy +
?	0 or 1 +
??	0 or 1, ungreedy +
{3}	Exactly 3 +
{3,}	3 or more +
{3,5}	3, 4 or 5 +
{3,5}?	3, 4 or 5, ungreedy +

Special Characters

\	Escape Character +
\n	New line +
\r	Carriage return +
\t	Tab +
\v	Vertical tab +
\f	Form feed +
\a	Alarm
[\b]	Backspace
\e	Escape
\N{name}	Named Character

String Replacement (Backreferences)

$n	nth non-passive group
$2	"xyz" in /^(abc(xyz))$/
$1	"xyz" in /^(?:abc)(xyz)$/
$`	Before matched string
$'	After matched string
$+	Last matched string
$&	Entire matched string
$_	Entire input string
$$	Literal "$"

Ranges

.	Any character except new line (\n) +
(a|b)	a or b +
(...)	Group +
(?:...)	Passive Group +
[abc]	Range (a or b or c) +
[^abc]	Not a or b or c +
[a-q]	Letter between a and q +
[A-Q]	Upper case letter + between A and Q +
[0-7]	Digit between 0 and 7 +
\n	nth group/subpattern +

Note *Ranges are inclusive.*

Pattern Modifiers

g	Global match
i	Case-insensitive
m	Multiple lines
s	Treat string as single line
x	Allow comments and white space in pattern
e	Evaluate replacement
U	Ungreedy pattern

Metacharacters (must be escaped)

^	[	.
$	{	*
(	\	+
)	|	?
<	>	

Available free from AddedBytes.com

> Tutorials <

- step-by-step Tutorial zu regulären Ausdrücken von regexone.com
- Spielerisch können sie mit RegEx-Kreuzworträtseln das lesen (und verstehen) von regulären Ausdrücken üben. Bis zu welchem Level kommen sie?
- Zum "RegEx schreiben üben" sind die "Exercises" von regextutorials.com hilfreich!

Zeilen filtern

- `grep` = Suche (und Ausgabe) von Zeilen (oder Teilen) die ein gegebenes Textmuster oder -stück enthalten
  - wichtige Suchmodi
    - " `-P` " = Perl RegEx Modus = seeehr umfangreicher RegEx Support
    - " `-F` " = (fixiertes) Stringmatching (z.B. zur Suche von RegEx-Syntax in Texten)
    - " `-E` " = egrep = extended POSIX RegEx matching = Suche mit standardisierten Ausdrücken
  - " `-i` " = Gross-Klein-Schreibung egal
  - " `-c` " = Anzahl der Zeilen, die einen Match enthalten
  - " `-v` " = Gegenteilige Ausgabe (inverted), d.h. alle Zeilen die NICHT matchen
  - " `-n` " = Zeilennummern
  - " `-o` " = nur matchig parts (z.B. für reguläre Ausdrücke wichtig)
  - " `-A XX` " = matching line und XX nachfolgende Zeilen ausgeben (append)
  - " `-B XX` " = wie -A nur für vorangehende Zeilen (before)
  - Beispiele

RegEx mit grep

- am besten **immer** im " `-P` " **(Perl) Modus nutzen**, da nur so wirklich alle regex-Feature wie " `\` "-based character classes (z.B. " `\d` " etc) verfügbar sind, z.B. `grep --help | grep -P "^\s+-\w\W"`
- Ausdrücke müssen i.d.R. ge**quote**t werden, s.o.
  - ggf. sogar mit single quotes, wenn " `$` " etc. im Ausdruck verwendet wird

---

> Tutorials <

Dieses Online-grep-regex-Tutorial gibt einen guten Einstieg in die Nutzung von regulären Ausdrücken und `grep`!

Text ersetzen

- `tr` = ersetzt/löscht **EINZELbuchstaben**/-character (translate)
  - Bsp. `echo "1,2,3" | tr "," "\n"` ersetzt Kommas mit Zeilenumbrüchen
  - **mehrere Zeichen auf einmal möglich**, aber gleiche Anzahl an Ersetzungen nötig, z.B. `echo "1,2 3" | tr ", " "\n\n"` ersetzt sowohl Komma als auch Leerzeichen JEWEILS mit Zeilenumbruch (unabhängig voneinander!)

  - " `-d` " löscht entsprechende Zeichen, z.B. `echo "Hans-Peter" | tr -d "-"` löscht alle Trennstriche
  - versteht auch einige **regex-Gruppen** (siehe tr Manpage)
- `sed` = stream editor = u.a. für **beliebige Textersetzung**
  - d.h. man definiert vorher, was passieren soll, und lässt das "Programm" dann auf einen Text etc. anwenden (keine Benutzeroberfläche!)
  - Eingabe wird nur einmal verarbeitet (input -> Verarbeitung -> output), sodass sed gut in pipelines verwendet werden kann (analog zu awk)
  - " `s/text/replacement/` " = (ersten) **suchen und ersetzen von Texten** (substitute), z.B. `echo "1,2,3" | sed "s/,/ - /"`
    - " `s/../../g` " = **ALLE** ersetzen, z.B. `echo "1,2,3" | sed "s/,/ - /g"`
    - " `s/../../4` " = vierten Treffer (pro Zeile) ersetzen
    - " `s/../../I` " = Gross-/Kleinschreibung ignorieren (ignore case), z.B. `echo "a,A" | sed "s/a/b/Ig"`
    - Kombinationen (siehe " `/Ig` " in "ignore case" Beispiel
  - " `-r` " ermöglicht die Verwendung von **erweiterten RegEx Ausdrücken als Suchtext**
    - z.B. `echo "1. a." | sed "s/[^a]?\./x-ter/g"`
  - **mehrere Ersetzungen mit Semikolon trennen**, z.B. `echo "a,A" | sed "s/a/b/g; s/A/B/g"` (werden nacheinander ausgeführt, pro Zeile)
  - **Filter**n der verarbeiteten Zeilen mittels **vorangestellt**er Bedingungen
    - **Zeilennummer**, z.B. `echo -e "Zeile 1\nZeile 2" | sed "2 s/i/l/"`
    - **Zeilenbereich**, z.B. `echo "1,2,33,4" | tr "," "\n" | sed "2,3 s/./X/"`
    - **ab Zeile X** via " `$` " (" `$` " = Platzhalter für "letzte Zeile")
      z.B. `echo "1,2,3,4" | tr "," "\n" | sed "2,$ s/./X/"`
    - **RegEx**-Match (in Zeile), z.B. `echo -e "#1\n2" | sed -r "/^#/ s/[:digit:]/X/"`
      (dieser Aufruf geht zwar auch ohne " `-r` ", aber sicher ist sicher! ;))
    - alle gematchten Zeilen werden verarbeitet und am Ende ausgeben
  - " `-n` " verhindert den automatischen output
    - sinnvoll in Kombination mit " `p` " Befehl (print)
      z.B. `echo "1,2,3,4" | tr "," "\n" | sed -n "2 p"`
      liefert die zweite Zeile (ohne " `-n` " würde jeder Zeile standardmässig ausgegeben und die Zweite noch ein zweites mal aufgrund des print Befehls!)

Zeilenumbrüche

- je nach Betriebssystem unterschiedliche "Kodierung"
  - " `\n` " = LF : Linux (line feed = ASCII 10)
  - " `\r` " = CR : (altes) MacOs (carriage return = ASCII 13)
  - " `\r\n` " = CR LF : Windows (DOS)
- entweder via Texteditor ändern, oder via Kommandozeile
  - `dos2unix` = einfaches Tool um Eingabe von Windows-Kodierung in Linux-Kodierung umzuwandeln
    - installieren via `sudo apt install dos2unix` (ggf. zuvor noch `sudo apt-get update` notwendig, falls Paket noch nicht verfügbar)

Kodierung von Zeilenumbrüchen sorgt immer wieder für Probleme bei Textersetzungen/-formatierung! Daher immer (wieder) dran denken!
