Reader Reguliere Expressies

Behorende bij project Big Data

13-12-2017

NHL HOGESCHOOL

13 december 2017

Reader Reguliere Expressies

Behorende bij project Big Data

13 december 2017

Inhoud

13 december 2017

13 december 2017

1 Algemeen

Dit is de reader over Reguliere Expressies behorende bij het project Big Data in relatie tot de IMDB.

2 Introductie

In veel software wordt gebruik gemaakt van het filteren van gegevens. Als het gaat om databases zijn dat vaak SQL-statements die op gestructureerde wijze gegevens filteren. Soms bestaat zo'n filter uit een stukje (door de gebruiker ingevoerde) tekst, in combinatie met wildcards aan het begin of einde.

Wildcards zijn echter nog een grof mechanisme om gegevens te filteren. Verder is het filteren van gegevens uit bijv. tekstbestanden lastiger en moet er al gauw geprogrammeerd worden om dit te faciliteren.

Dit is waar Reguliere Expressies uitkomst bieden. Volgens Wikipedia:

> A regular expression, regex or regexp[1] (sometimes called a rational expression) is, in theoretical computer science and formal language theory, a sequence of characters that define a search pattern. Usually this pattern is then used by string searching algorithms for "find" or "find and replace" operations on strings.

Met reguliere expressies kun je dus zoeken en eventueel vervangen op basis van patronen. Dus in plaats van zoeken met een stuk tekst (artiest = "Billy Joel") zoek je naar **patronen** ("het eerste stukje tekst van de MP3-filename is de artiest").

Voordelen

- Zeer krachtig
- Kortere code
- Universeel inzetbaar
    - Linux: sed, grep, awk
    - Programmeertalen: c#, PHP, Java, javascript
    - Websites

Let op!

- Wel even je reguliere expressie toelichten in je code, want interpreteren van andermans reguliere expressies is lastig
- Er zijn nog wel subtiele verschillen in de engine die gebruikt wordt op platformen / programmeertalen

Zie de bijlagen voor voorbeelden van een eenvoudig zoek-vervang programma in C# met en zonder gebruik van reguliere expressies.

Typisch heb je dus drie zaken nodig om gebruik te kunnen maken van een reguliere expressies:

1) Een hoeveelheid tekst
2) Een stuk gereedschap dat reguliere expressie kan verwerken
3) Een reguliere expressie

13 december 2017

3 Patronen

Reguliere expressies bestaan uit patronen. Bijvoorbeeld:

[a-z]	mag alle letters van a tot en z zijn
[A-Z]	mag alle letters van A tot en Z zijn
[0-9]	mag alle cijfers van 0 tot en met 9 zijn
.	een willekeurig karakter

Op internet is uitgebreid te vinden wat de precieze set aan patronen is. Er willen nog wel eens kleine verschillen zijn in de mogelijke patronen en wat ze betekenen. Kijk dus bij twijfel goed in de juiste documentatie.

4 Greedy!

Eén van de krachten van Reguliere Expressies is de zogenaamde ‘greediness’ (***gretigheid***, ***hebberigheid***). Hiermee is het mogelijk om eenvoudig aan te geven dat bijvoorbeeld een hele reeks (willekeurige) tekens overgeslagen moet worden tot een volgend filter aangetroffen wordt.

We herkennen dit ook wel in het dagelijkse leven:

1) Loop hier rechtdoor en neem de zesde straat rechts
2) Loop hier rechtdoor tot je T-kruising tegenkomt

In het eerste voorbeeld tel je de straten die rechts van je voorbijkomen. Het is niet interessant hoe ze heten of hoe ze er uit zien: als ze maar rechts liggen.

In het tweede voorbeeld negeren we alle straten en situaties totdat we een T-kruising tegenkomen.

Als je dit vertaald naar *greediness*, dan zou je kunnen zeggen dat je met het tweede voorbeeld heel hebberig bent: je wilt alle straten ‘hebben’ totdat je de juiste situatie (T-kruising) tegenkomt.

Als we dit vertalen naar Reguliere Expressies, dan zijn hiervoor verschillende patronen. Een aantal voorbeelden met willekeurige tekens:

`.`	= willekeurig teken. Vergelijk dit met pacman die ook stippen eet….

Vervolgens kun je **achter** een teken opgeven of dat teken herhaald moet worden bij het zoeken. Bijvoorbeeld een sterretje of

`.*`	= 0 of meerdere willekeurige tekens. Het sterretje (asterisk) wordt ‘greedy’ genoemd.
`.?`	= 0 of 1 willekeurige tekens
`.+`	= 1 of meer willekeurige tekens
`.{2}`	= 2 willekeurige tekens
`.{3,}`	= 3 of meer willekeurige tekens
`.{2,4}`	= 2 tot 4 (inclusief) willekeurige tekens

Maar dit kan ook met ‘gewone’ tekens:

ABC{3} = 3 keer letterlijk ‘ABC’

[A-Z]{10} = 10 keer één letter uit de keuze A,B,C,D,E……X,Y,Z

[A-Za-z]* = willekeurige aantal herhalingen van de A,B,C,D,E……X,Y,Z,a,b,c,d,e,f…..x,y,z

Door expressies te stapelen kun je voorkomen dat de ***greediness*** uit de hand loopt. Want het kale patroon ‘.*’ zonder verdere expressie is gewoon de hele regel.

Let op: greediness kan op verschillende platformen nog wel eens andere effecten hebben, en daardoor frustrerend zijn. Het is dan een kwestie van goed lezen van de handleiding hoe greediness geregeld is en hoe er met herhalingen ({} en *) omgegaan wordt.

Hoe werkt greediness? Een voorbeeld. We gebruiken onderstaande tekst:

```
<td>hallo</td>martin
```

En de expressie

```
/<.+>/
```

Verwachtte resultaat (2 stuks)

```
<td>
</td>
```

De RegEx engine zal dit als volgt aanvliegen:

13 december 2017

Stap	RegEx part	Found	Match?	Action?	Result string
1.	<	<	Yes	Next char; next part	<
2.	.+	t	Yes	Next char	<t
3.	.+	d	Yes	Next char	<td
4.	.+	>	Yes	Next char	<td>
5.	.+	h	Yes	Next char	<td>h
6.	.+	a	Yes	Next char	<td>ha
7.	.+	l	Yes	Next char	<td>hal
8.	.+	l	Yes	Next char	<td>hall
9.	.+	o	Yes	Next char	<td>hallo
10.	.+	<	Yes	Next char	<td>hallo<
11.	.+	/	Yes	Next char	<td>hallo</
12.	.+	t	Yes	Next char	<td>hallo</t
13.	.+	d	Yes	Next char	<td>hallo</td>
14.	.+	>	Yes	Next char	<td>hallo</td>
15.	.+	m	Yes	Next char	<td>hallo</td>m
16.	.+	a	Yes	Next char	<td>hallo</td>ma
17.	.+	r	Yes	Next char	<td>hallo</td>mar
18.	.+	t	Yes	Next char	<td>hallo</td>mart
19.	.+	i	Yes	Next char	<td>hallo</td>marti
20.	>	n{EOL}	No	RegEx not found; backtrack	<td>hallo</td>martin
21.	>	i	No	backtrack	<td>hallo</td>marti
22.	>	t	No	backtrack	<td>hallo</td>mart
23.	>	r	No	backtrack	<td>hallo</td>mar
24.	>	a	No	backtrack	<td>hallo</td>ma
25.	>	m	No	backtrack	<td>hallo</td>m
26.	>	>	Yes	Done	<td>hallo</td>

In stap 20 wordt gezien dat de letter N gevolgd wordt door het einde van de regel, dus de engine gaat verder met '>' om te kijken of het laatste deel matcht. Niet dus. De RegEx-engine geeft niet op maar gaat terugzoeken of hij misschien te hebberig is geweest en op de terugweg toch nog de juiste tekens kan terugvinden. Dat noemen we '**backtracking'** (https://nl.wikipedia.org/wiki/Backtracking): als je te ver bent gegaan, keer je terug naar een vorig keuzemoment. Dit is ook weer herkenbaar met het zoeken naar de 6e straat uit ons stratenvoorbeeld: als je op een gegeven moment ergens verkeerd bent afgeslagen dan keer je terug naar de plek waar je de weg nog wel wist, en begin je deels overnieuw met je route.

Dat zien we in stappen 21 tot en met 26: de RegEx-engine gaat terug in de zoektekst totdat het laatste deel van de expressie (die nog niet gevonden was) alsnog gevonden wordt.

Gevonden resultaat:

```
<td>hallo</td>
```

De *greediness* loopt hier dus uit de hand. We krijgen namelijk niet het gewenste resultaat (<td> en <td>).

Hoe dit op te lossen? ***Laziness***!

13 december 2017

5 Laziness

In het vorige hoofdstuk hebben we gezien dat je met greediness een probleem kunnen krijgen. Er worden te grote stukken tekst aan de ‘match’ toegevoegd. Dit probleem is op te lossen door de expressive wat aan te passen, zodat de RegEx-engine zich wat luier (lazy) gaat gedragen.

Door een vraagteken achter de plus te plaatsen vertellen we de RegEx engine “herhaal zo weinig mogelijk!”.

Gebruik de onderstaande expressie

```
/<.+?>/
```

In plaats van

```
/<.+>/
```

Zie het verschil: we voegen een vraagteken toe achter het plusteken. In onderstaande tabel geeft aan hoe de verwerking plaatsvindt met lazy gedrag.

stap	RegEx part	Found	Match?	Action?	Result string
1.	<	<	Yes	Next char; next part	<
2.	.+?	t	Yes	Next char; next part	<t
3.	>	d	No	Backtrack	<t
4.	.+	d	Yes	Next char; next part	<td
5.	>	>	Yes	Next char; Done; Reset pattern	<td>
6.	<	h	No	Next char	
7.	<	a	No	Next char	
8.	<	l	No	Next char	
9.	<	l	No	Next char	
10.	<	o	No	Next char	
11.	<	<	Yes	Next char; next part	<
12.	.+?	/	Yes	Next char; next part	</
13.	>	t	No	Backtrack	</
14.	.+?	t	Yes	Next char; next part	</t
15.	>	d	No	Backtrack	</t
16.	.+?	d	Yes	Next char; next part	</td
17.	>	>	Yes	Next char; done	</td>
18.		{EOL}		Finished	

In de tweede regel wordt meteen de voldaan aan de ‘lazy’ expressie ‘.+?’: er is één karakter gevonden, dus gaat de engine in de volgende regel door naar het volgende deel van het zoekpatroon: “>”. Echter in regel 3 gaat het mis: het >-teken matcht niet met de letter d. Dus gaat we weer terug (‘backtrack’), maar dan een stap terug in de reguliere expressie! In regel 4 wordt weer voldaan aan de expressie ‘.*?’ dus schuiven we een stap op in de RegEx en vinden weer de “>”. Die voldoet wél in stap 5 en we zijn klaar. Nu gaan we opnieuw beginnen omdat een RegEx in principe vaker voor kan komen. Merk op dat backtracking vanaf stap 13 wel vaker voorkomt om tot een conclusie te komen.

6 Vervangen

Het doel van Reguliere Expressies is meestal het zoeken van gegevens. Maar ook het vervangen van gegevens is wel degelijk mogelijk. Denk bijvoorbeeld aan een editor als Notepad++ of Linux/Unix-tools als GREP en SED. Daar kun je zoeken én vervangen met reguliere expressies.

Het idee bij 'vervangen' is meestal dat de RegEx-engine je de mogelijkheid geeft om de gevonden tekst te vervangen door een ander stuk tekst. Een voorbeeld zin waarbij ik een nieuw telefoonnummer heb gekregen:

```
Mijn telefoonnummer is +316123456789 geworden.
```

Met de reguliere expressie

```
\+31[0-9]+
```

Kunnen we het telefoonnummer vinden. De RegEx-engine zal als 'match' teruggeven:

```
+316123456789
```

We zouden het telefoonnummer kunnen vervangen door het woord '`geheim`'. Dan wordt de zin:

```
Mijn telefoonnummer is geheim geworden.
```

Bij het gebruik van een 'vervang'-instructie, maak je gebruik van normale tekst én speciale RegEx tekens. Vaak zijn dit verwijzingen naar groepen (zie volgende hoofdstuk).

In de volgende paragraaf gaan we zien hoe je bijvoorbeeld de +31 kunt vervangen door een nul. Je krijgt dan:

```
Mijn telefoonnummer is 06123456789 geworden.
```

7 Groepen

Je kunt expressies groeperen zodat je er later aan kunt refereren. Een groep is een aantal RegEx expressie-tekens tussen haakjes. Bijvoorbeeld twee groepen tekens (cijfers en hoofdletters):

```
([0-9])([A-Z])
```

Je kunt op twee manieren gebruik maken van groepen:

1) In de zoek expressie
2) In de vervang expressie

Voordat we ingaan op de verschillende manieren van gebruik van groepen nog een opmerking: door gebruik te maken van groepen splits je de gevonden teksten ('matches') in onderdelen. Het is niet dat het verschillende expressies zijn die los toegepast worden.

7.1 Groepen gebruiken in de zoek-expressie

Dat kan erg handig zijn bij repeterende zaken zoals "kijk of dit telefoonnummer twee keer in één regel voorkomt" als je wilt checken of mensen wel een apart mobile nummer en thuis nummer hebben opgegeven. Hiervoor gebruik je de groep als '***back reference***'.

Stel je de volgende lijst met invoer voor: (`mobiele nummer;thuis nummer`)

```
0612345678;0581234567
0698756355;0698756355
0645454545;0581234567
+31699988877;0581234567
0651548748;0651548748
0649798646;0581234567
+31888571111;0581234567
```

We gebruiken het volgende patroon:

```
([0-9]{10});\1
```

Het geel gemarkeerde deel is een groep omdat het tussen haken () staat. Dit gaat op zoek naar 10 opeenvolgende cijfers:

`[0-9]` → alle cijfers tussen 0 en 9 (inclusief)

`{10}` → herhaal 10 keer

`;` → het vaste teken punt-komma (geen speciaal teken!)

Het groen gemarkeerde deel is een ***back-reference*** naar de eerste groep. Het betekent:

*hier moet hetzelfde gevonden worden als in de **eerste** groep.*

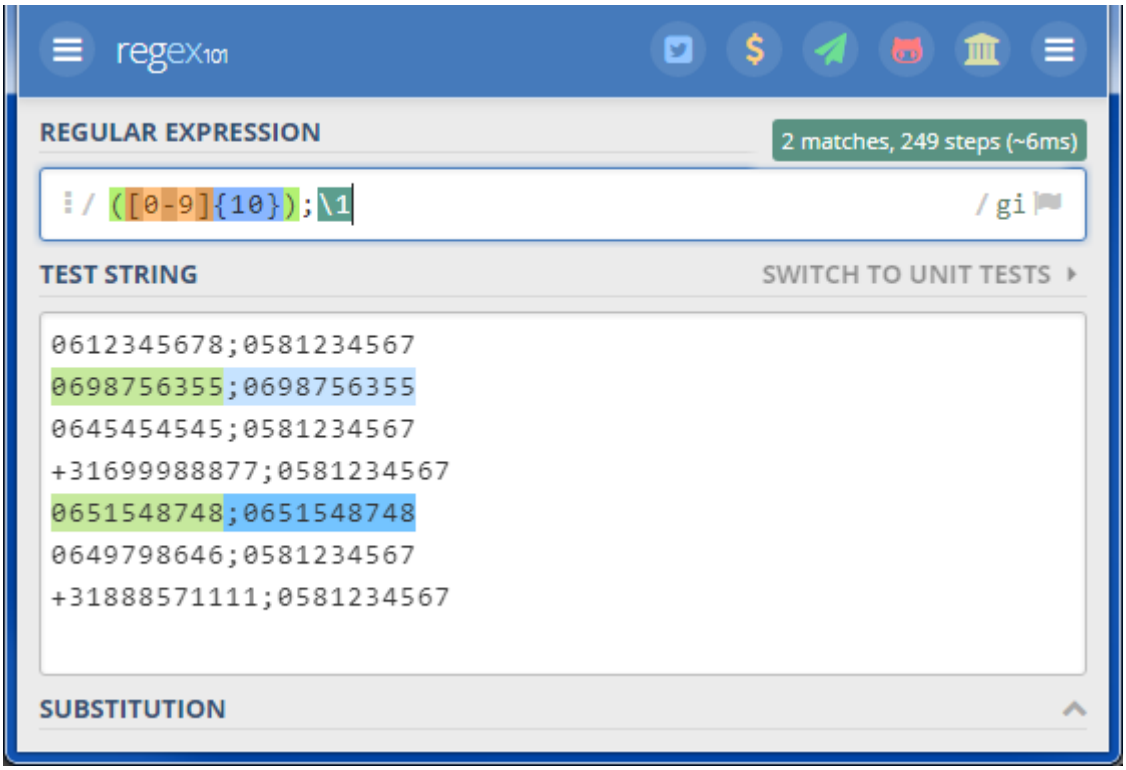
Conclusie: als er vóór en na de puntkomma hetzelfde staat, dan is de expressie geldig.

Resultaat:

```
0698756355;0698756355
0651548748;0651548748
```

We kunnen dit testen met de website http://www.regex101.com. De gemarkeerde tekst onder 'Test string' laat de gevonden resultaten zien. Let ook op de tekst rechtsboven: 2 matches in 249 (!!) stappen.

13 december 2017



7.2 Gebruik van groepen in de vervang expressive

Zoals gezegd in hoofdstuk 6 kun je niet alleen zoeken maar ook vervangen. Daar is een voorbeeld gegevens over het vervangen van een deel van een telefoonnummer.

We gebruiken weer de tekst:

```
Mijn telefoonnummer is +316123456789 geworden.
```

Maar nu met een subtiel andere reguliere expressie

```
(\+31)([0-9]+)
```

Let op de backslash voor het plusteken: dat is nodig omdat het plusteken een speciaal teken is waarvan we de werking niet willen gebruiken omdat we een letterlijk plusteken zoeken.

Merk op dat we nu twee groepen hebben (groen en geel). De gele groep is nummer 1, en de groene nummer 2. We vinden dan het telefoonnummer: de RegEx-engine zal als ‘match’ teruggeven:

```
+316123456789
```

Maar dan verdeeld in twee groepen:

```
+316123456789
```

De RegEx engine onthoudt dus hoe de match opgedeeld is in groepen.

Bij het gebruik van een ‘vervang’-instructie, maak je gebruik van normale tekst én speciale RegEx tekens. We gebruiken nu de volgende vervang-expressie:

```
0\2
```

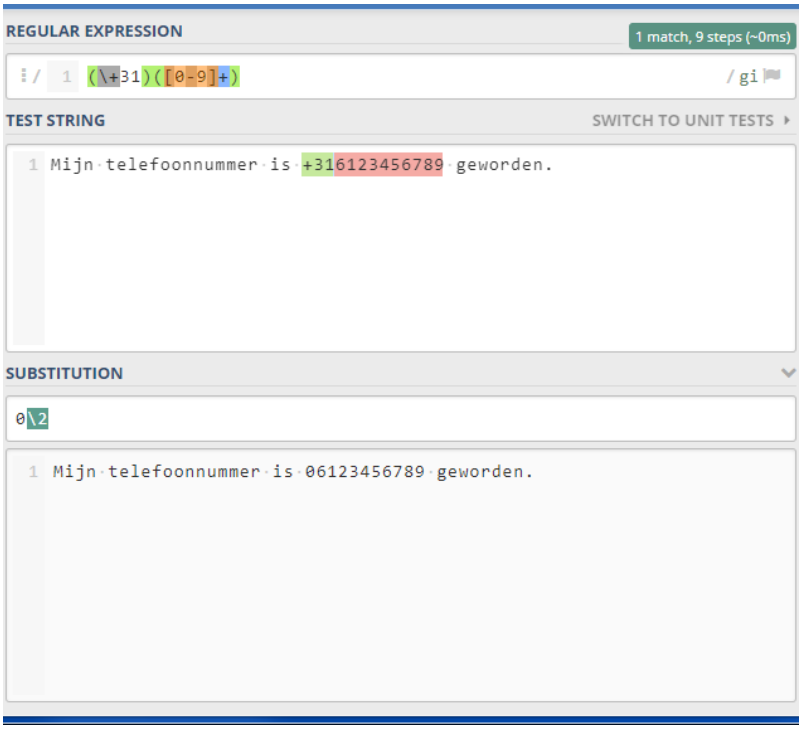
13 december 2017

De groene tekst is een referentie naar de 2e groep. De eerste nul (0) is een normaal teken (vaste tekst) die gewoon meegestuurd wordt. Wat we dus eigenlijk zeggen met deze zoek vervang actie:

> Zoek in de tekst naar twee groepen tekst: eerst +31 gevolgd door een serie getallen. Als je die combinatie gevonden hebt, dan moet je als uitvoer een nul sturen gevolgd door de tweede groep.

Dit levert dan uiteindelijk de volgende output:

```
Mijn telefoonnummer is 06123456789 geworden.
```



Je kunt het ook bijvoorbeeld het telefoonnummer **leesbaarder** maken. We gebruiken de volgende reguliere expressie om te zoeken. Let op de kleuren die consequent gebruikt zijn om de groepen aan te wijzen.

```
(\+31)(6)([0-9]{3})([0-9]{3})([0-9]{3})
```

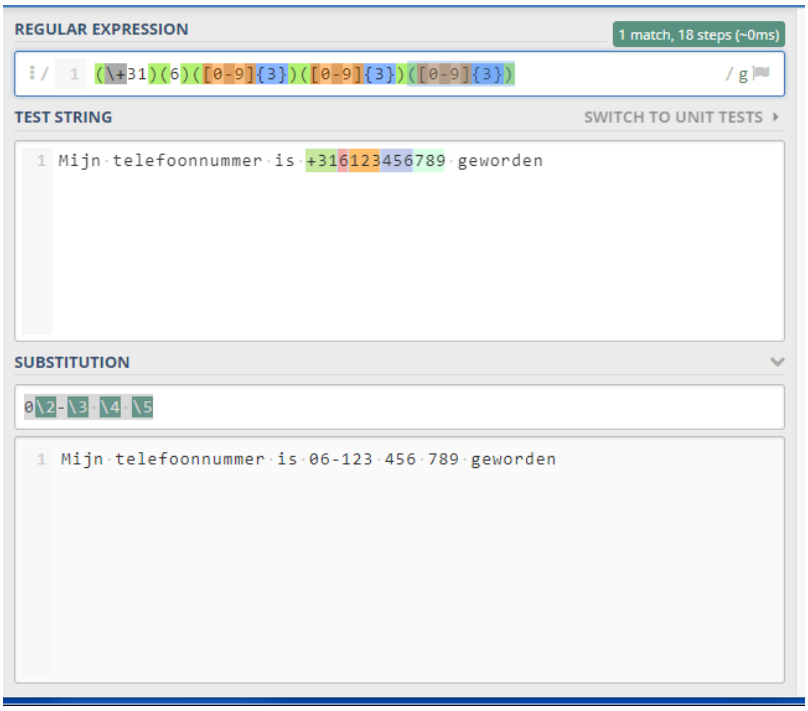
We vervangen met onderstaande expressie. Let op de spaties tussen de groep referenties.

```
0\2-\3 \4 \5
```

Levert:

```
Mijn telefoonnummer is 06-123 456 789 geworden
```

13 december 2017



8 Modifiers / RegEx flags

Je kunt het gedrag van de zoekopdracht beïnvloeden met vlaggen. Die staan achter de laatste delimiter. In de website van http://www.regex101.com staan ze achteraan en kun je ze met een muisklik eenvoudig aan- of uitzetten.

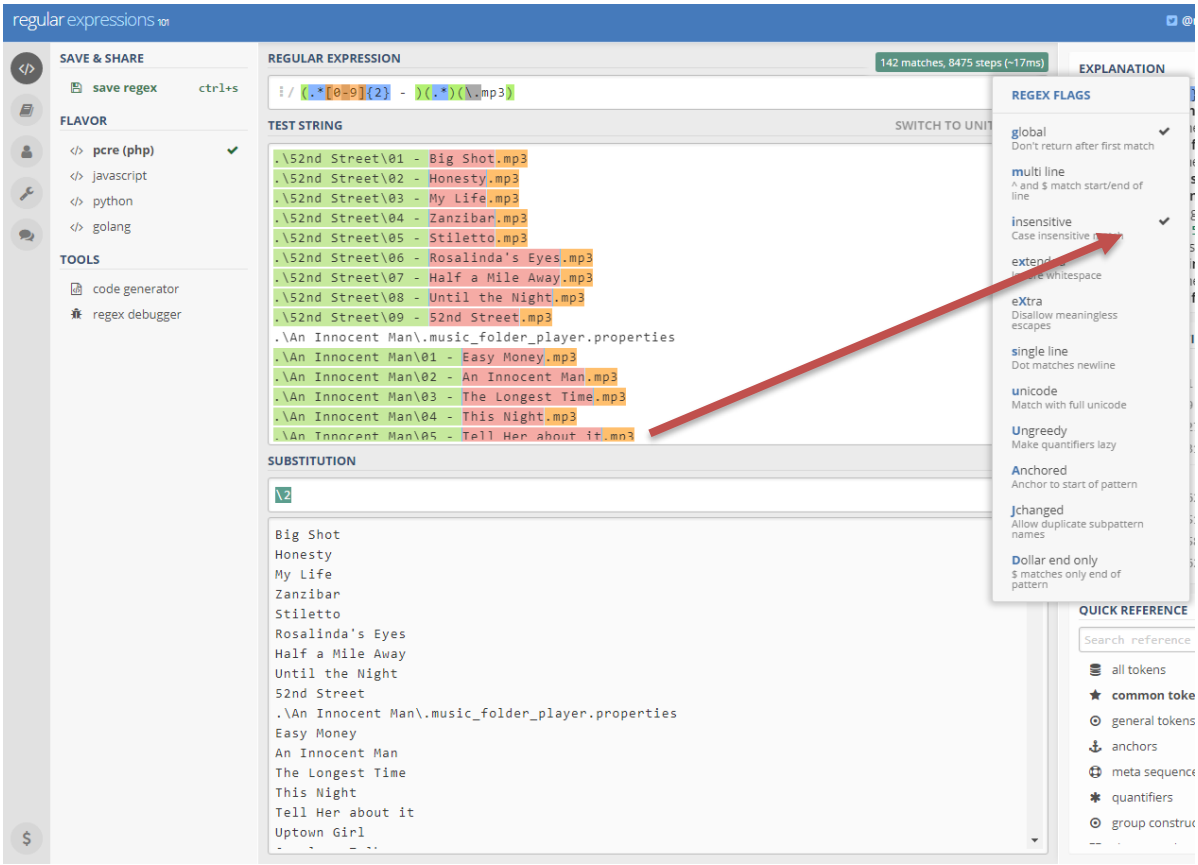
Standaard regex = `<delimiter><opdracht><delimiter>`

Typisch wordt de forward slash (‘/’) gebruikt.

Eenvoudiger = `/….../gi`

Door aan het einde de modifier ‘`i`’ te gebruiken wordt de hele matching **case-insensitive**.

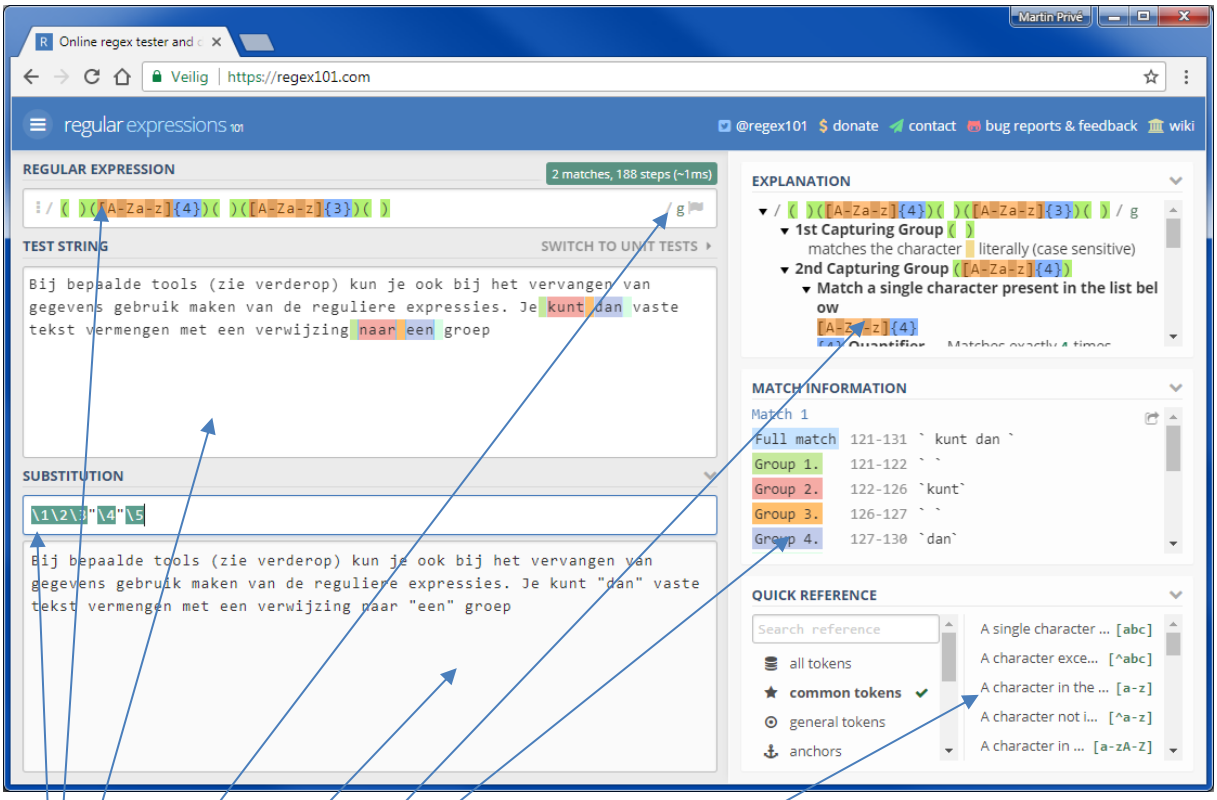
13 december 2017



9 Oefeningen

Voor deze oefeningen maken we gebruiken van de website https://regex101.com/.

13 december 2017



Deze website heeft de mogelijkheid om

- Een reguliere expressie in te vullen
- Een veld met tekst in te voeren waar je de reguliere expressie op wilt laten werken (‘***Test String***’) waar ook de *syntax highlighting* werkt!
- Vlaggen kunt besturen (‘flags’)
- Een veld waar je de vervangingsexpressie kunt invoeren (‘***Substitution***’)
- Een veld met de daadwerkelijke uitvoer
- Uitleg over hoe de engine tot zijn conclusie is gekomen
- Uitleg welke matches er gevonden zijn
- **Quick Reference** met veel gebruikte onderdelen van expressies.

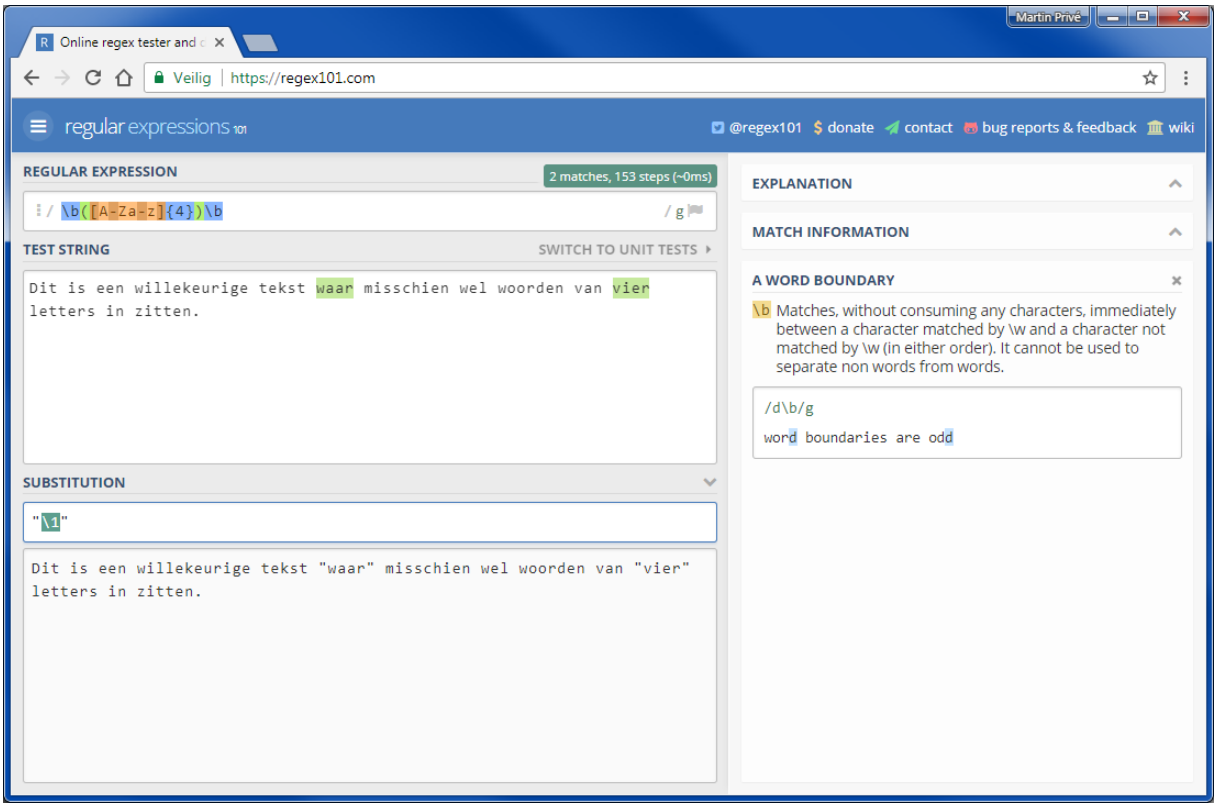
Download van Blackboard het bestand ‘**files.txt**’ en navigeer daarna naar de website Deze website gebruikt de **PHP PCRE** *engine* (http://php.net/manual/en/book.pcre.php).

Daarna kun je onderstaande oefeningen aan de slag.

9.1 Oefening 1

We gaan alle woorden zoeken van 4 letters en zetten er dubbele quotes (“) omheen:

13 december 2017



De zoek-expressie die we bij '**Regular Expression**' intypen is

```
\b([A-Za-z]{4})\b
```

Type vervolgens een willekeurige tekst in bij het vak '**Test String**'.

En voer een expressie in bij '**Substitution'** :

```
"\1"
```

En bekijk het resultaat in het vak *onder* '**Substitution'**

9.2 Oefening 2

We willen nu van een lijst met MP3-bestandsnamen alleen de naam hebben van de titel van de track hebben. Gelukkig hebben we een strakke naamgevingsconventie aangehouden waardoor we met een RegEx goed uit de voeten kunnen.

De opmaak van elke bestandsnaam is als volgt:

```
<punt><slash><albumnaam><slash><track nr><separator><filename><extensie>
```

Bijvoorbeeld:

```
.\Streetlife serenade\03 - The great suburban showdown.mp3
```

We gaan nu een zoekpatroon gebruiken om de informatie die we willen hebben in groepen op te delen, zodat we er later aan kunnen refereren als we delen willen vervangen.

Zoekpatroon ➔ `(.*[0-9]{2} - )(.*)(\.mp3)`

Toelichting op dit patroon:

13 december 2017

(	begin een groep (groep nummer 1)
.*	verzamel willekeurige karakters tot volgende opdracht
[0-9]{2}	er moeten 2 ({2}) cijfers staan
–	gevolgd door ‘spatie minteken spatie’
)	sluit de groep
(	begin een groep (groep nummer 2)
.*	verzamel willekeurige karakters tot volgende patroon
)	sluit de groep
(	begin een groep (groep nummer 3)
\.mp3	zoek naar literal ‘.mp3’
)	sluit de groep

Let op als je een gereserveerd karakter nodig hebt, je deze vooraf moet laten gaan door een backslash (‘\’). Dus in het geval van de derde groep moet de punt voor ‘mp3’ vooraf gegaan worden door een backslash.

Om te zorgen dat de website RegEx101 doet wat ie moet doen, moeten we de instellingen van de vlaggen (flags) zodanig doen dat de tekst in het invoervak als multi-line wordt beschouwd, en niet als één stuk tekst.

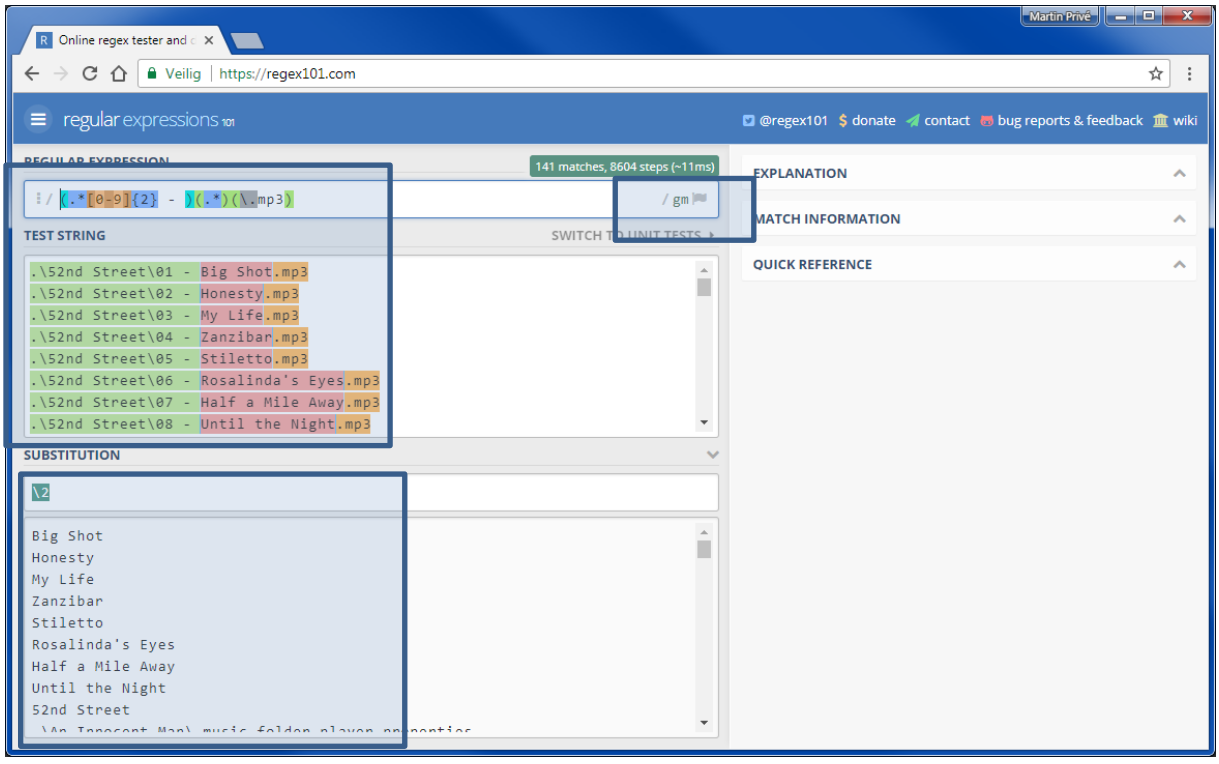
Flags ➔ `gm`

Nu willen we de uitvoer gaan bepalen: als er een match gevonden is met de reguliere expressie, dan willen we alleen de tweede groep op de uitvoer zien, en de rest moet weggelaten worden.

Daarvoor gebruiken we een ‘substitute’ opdracht.

Substitute expressie ➔ `\2`

Dat ziet er dan als volgt ui (let op de blauwe doorzichtige delen)



13 december 2017

9.3 Telefoonnummers aanvullen

We gebruiken in deze oefening een lijst met telefoonnummers, die niet helemaal consistent is. Het ene telefoonnummer is international en begint met +31, het andere begint met 06. Dat willen we uniformeren zodat ze allemaal met +31 beginnen. Het zoekpatroon kun je verwoorden als:

> Zoek alle telefoonnummers die niet beginnen met een plusteken. Als je er één gevonden hebt, moet je de eerste nul er afhakken, en vervangen door +31.

Dat vertaalt zich in het volgende zoekpatroon:

> Zoek Patroon ➔ `^([^\+])([0-9]*)`

De expressie die we gebruiken om te vervangen is relatief simple:

> Vervangen door ➔ `+31\2`

Toelichting patroon:

`^`	het patroon dat volgt moet aan het begin van de regel staan
`([^\+])`	de regel mag niet met een plus teken beginnen. Begin tevens een groep
`([0-9]*)`	maak een groep op basis van alle volgende cijfers.

Als het eerste karakter dus géén plusteken is, dan heeft de eerste groep een cijfer gevangen waarschijnlijk (afhankelijk van je testdata natuurlijk).

De tweede groep bevat dus het telefoonnummer **zonder de eerste nul** (!!). Je kunt die tweede groep dus gebruiken om er +31 vóór te zetten.

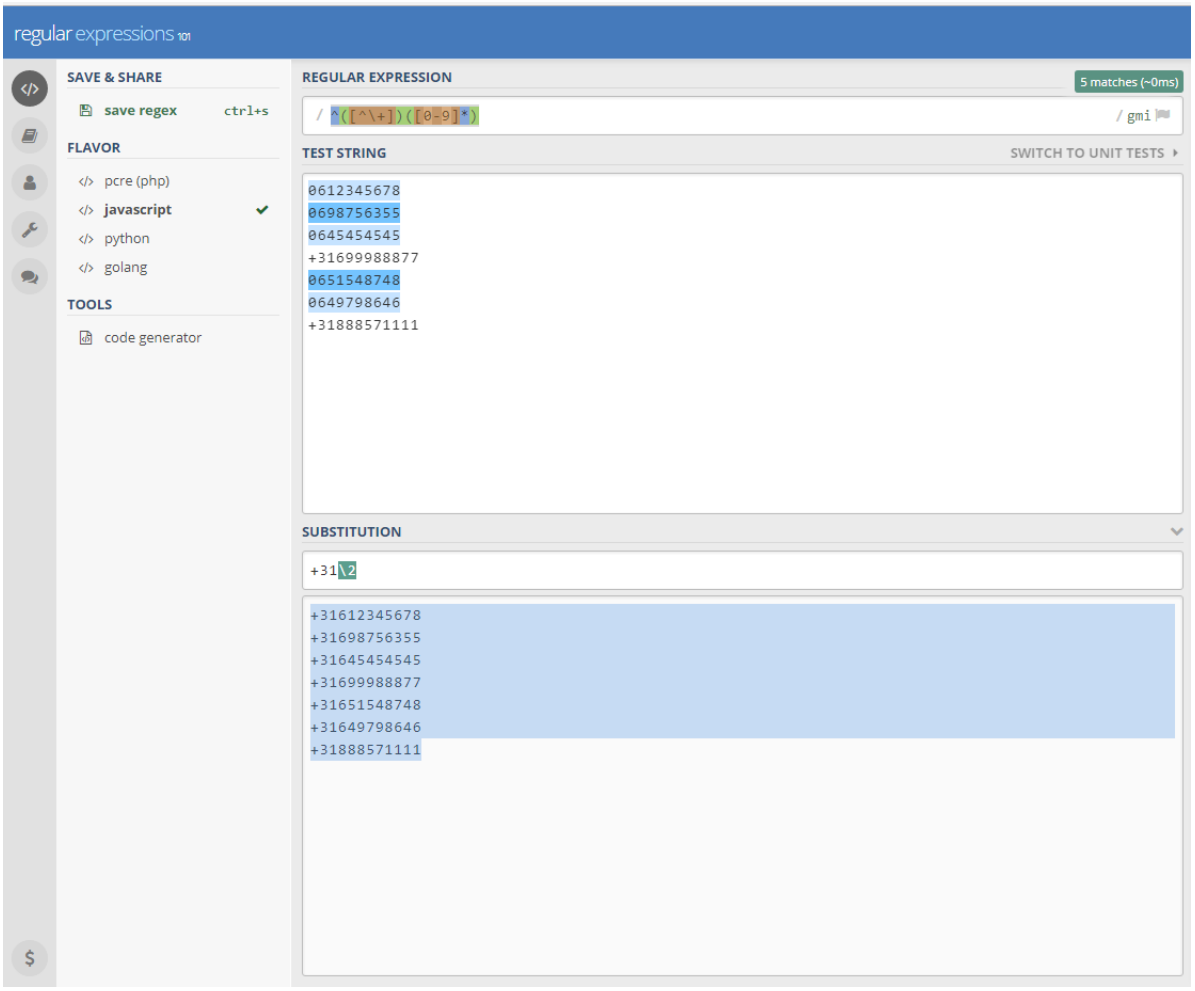
> Alternatieve zoekexpressie ➔ `^(0)([0-9]*)`

En de expressie voor vervanging is dan:

> Vervangen door ➔ `+31\2`

Let op de modifiers: `gmi` (de m is nieuw in verband gebruik van ^): op elke regel moeten we opnieuw beginnen.

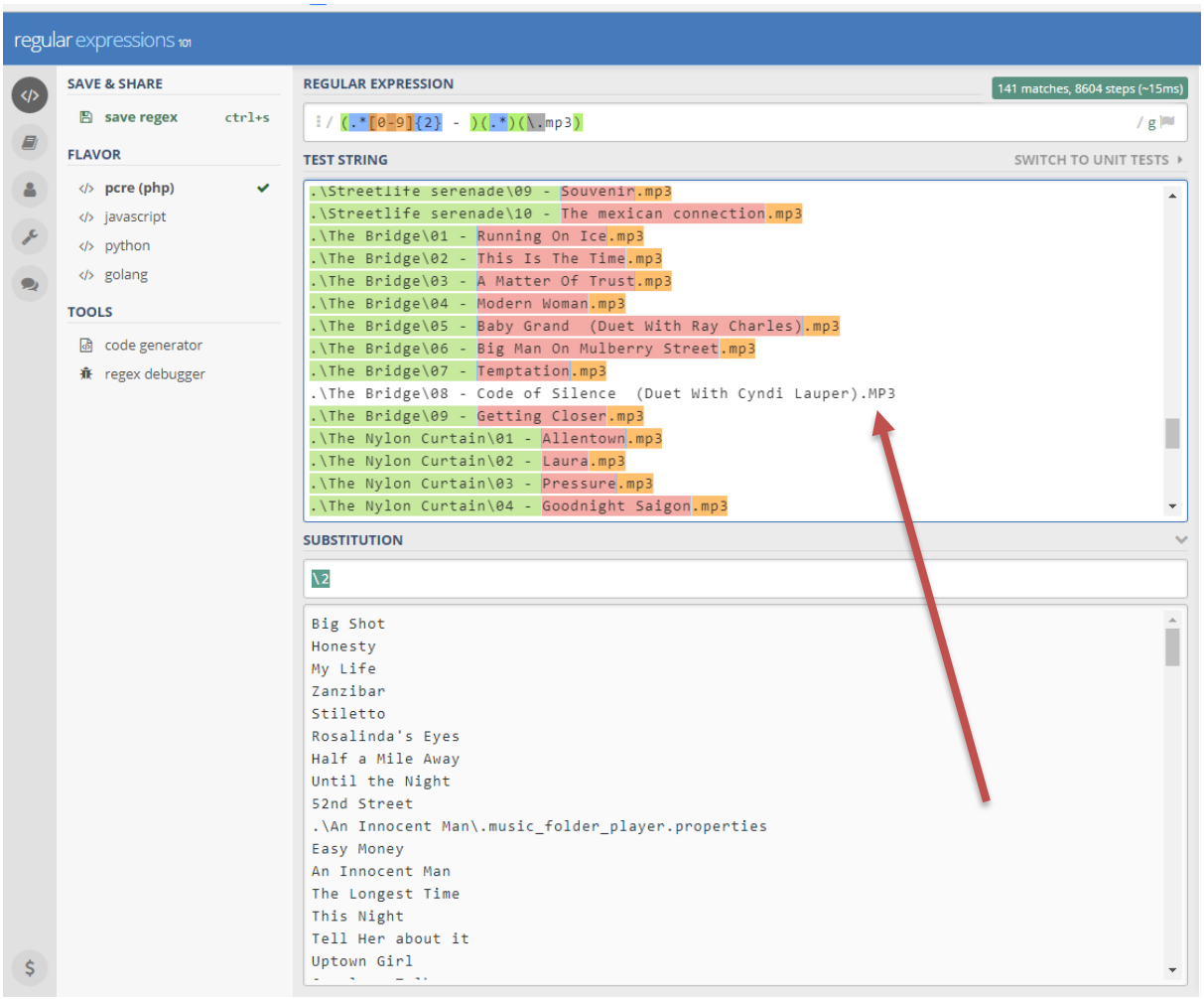
13 december 2017



10 Case sensitive

Uit onderstaande afdruk blijkt dat niet alle bestandsnamen meegenomen worden. Dit komt omdat de extensie van “Code of Silence” in hoofdletters is. Hoe kun je dat oplossen?

13 december 2017

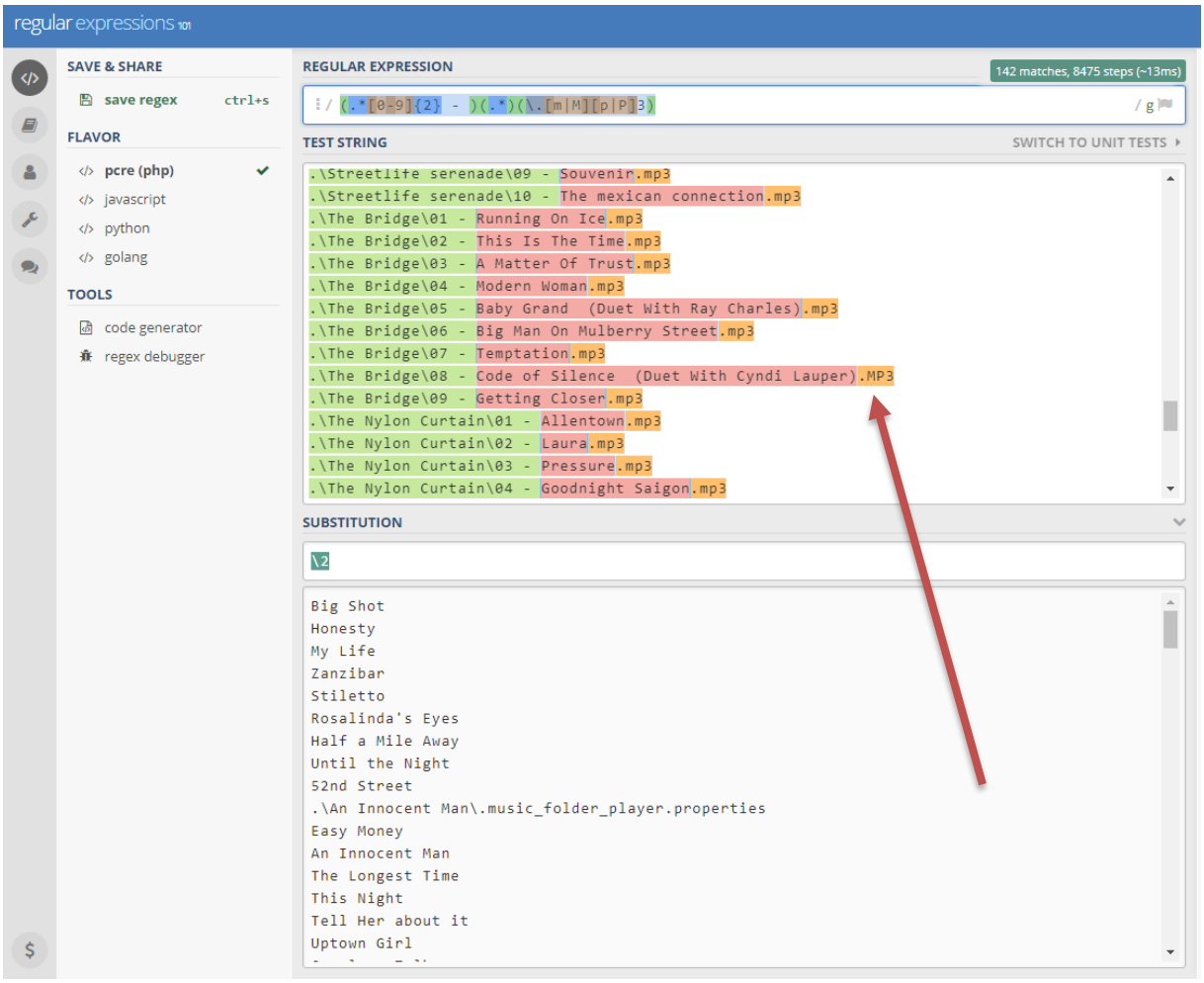


Nu case insensitive voor de extensie .MP3 of mp3 of Mp3?

```
(.*[0-9]{2} - )(.*)(\.[m|M][p|P]3)
```

Het groene stuk is nieuw. We zeggen daar eigenlijk dat elke letter een hoofd of kleine letter mag zijn.

13 december 2017



Dat kan ook eenvoudiger zoals we gezien hebben bij hoofdstuk 6: maak gebruik van de modifier letter '`i`'. Dan wordt echter wel de **hele expressie** case-insensitive behandeld!

11 Gebruik van RegEx in tooling

Binnen Linux kun je goed gebruik maken van reguliere expressies. Echter, er zijn nogal wat subtiele verschillen. Je kunt dat op verschillende manieren doen:

- Onder Windows 10 installeer je de bash-shell
- Elke windows versie: installeer cygwin (http://www.cygwin.com/). Zo kun je met één druk op de knop een bash-shell starten
- Installeer Debian of Ubuntu als 2e operating systeem of via virtualisatie (Parallels of Virtual Box).

Voordeel van de eerste twee opties is dat je gewoon met je bestaande windows omgeving aan de slag kan, en dus bij al je bestanden kan.

11.1 grep

Dezelfde expressie om deze foute telefoonnummers te zoeken via GREP (https://www.gnu.org/software/grep/manual/grep.html). Bijvoorbeeld:

```
grep '^[^\+][0-9]*' telefoonnummers.txt
```

13 december 2017

Zie onderstaande set Linux commando's

```
$ cat telefoonnummers.txt
0612345678
0698756355
0645454545
+31699988877
0651548748
0649798646
+31888571111n

$ grep '^[^\+][0-9]*' telefoonnummers.txt
0612345678
0698756355
0645454545
0651548748
0649798646

$ grep -v '^[^\+][0-9]*' telefoonnummers.txt
+31699988877
+31888571111n
```

Hier is het niet nuttig of zelfs maar **toegestaan** om met groepen via haakjes te werken. Gebruik eventueel de optie '-v' om de resultaten om te draaien.

Merk op dat je bij GREP het zoekpatroon niet tussen delimiters (/…./) hoeft te zetten zoals bij het volgende programma sed.

11.2 sed

Bijvoorbeeld in SED (Stream Editor): https://www.gnu.org/software/sed/manual/sed.html

Het programma sed is zeer krachtig om tekst te bewerken vanaf 'standard input' of bestanden. Als je eenmaal gewend bent aan sed, kun je dus zeer snel bestanden manipuleren zonder één regel code te schrijven.

Met onderstaande expressie kun je telefoonnummers verbeteren zoals in het voorbeeld van paragraaf 9.3. (zoek foute telefoonnummers en verbeter ze).

```
sed -e 's/^\([^\+]\)\([0-9]*\)/\+31\2/gi' telefoonnummers.txt
```

De Stream Editor SED geeft in het 'script' dat hierboven staat eerst de opdracht om te gaan zoeken en vervangen ('s'). Het patroon moet opgenomen worden in een soort 'opdrachtregel' tussen delimiters (/…../).

Let ook op dat er een extra backslash staat (groen). Bij het vak Operating Systems heb je geleerd dat een backslash op de commandoregel gebruikt kan worden om een speciaal teken. We hebben een backslash nodig als 'literal' en moeten er daarom twee invoeren: de eerste geeft aan dat er een speciaal teken volgt, en de tweede is het daadwerkelijke speciale teken. Vergelijk dat met expressies in C#.

11.3 awk

Met AWK kunnen kleine programmatjes maken die ideaal zijn voor het werken met .csv (comma separated values) bestanden.

Neem bijvoorbeeld een eenvoudig tekstbestand met namen en functies. De kolom volgorde is: voornaam, tussenvoegsels,achternaam,functie. De kolommen worden gescheiden door een puntkomma. Zie onderstaande Linux commando's.

```
$ cat personen.txt
Karel;de;Vries;Student
Martin;;Molema;Docent
Pietje;;Puk;Student
Jan;;Jansen;Student
Jan;;Pietersen;Student
Hans;;Janssen;Student
Peter;;Jansen;Student

$ awk -f jansen.awk personen.txt
Voornaam =  Jan
Voornaam =  Peter
```

Het awk-script voor jansen.awk ziet er als volgt uit:

```
BEGIN {FS=";"}
/Jansen/ {print "Voornaam = " $1}
```

De eerste regel verteld ons dat het veldscheidingsteken een puntkomma is. De tweede regel begint met een reguliere expressie '`/Jansen/`'. Als de naam voorkomt (willekeurig waar in de regel tekst), dan drukken we het eerste veld af (de voornaam, $1).

Merk op dat dit case-sensitive is. Dus '`/jansen/`' is wat anders dan '`/Jansen/`'. Hier zijn wel oplossingen voor (zie AWK manual).

De kracht van AWK zit hem in dat alleen regels die voldoen[1] aan het patroon behandeld worden. We passen het script iets aan:

- als je Jansen heet, willen we je voornaam weten
- heet je niet Jansen, dan willen we de tussenvoegsels + achternaam weten

De uitvoer:

```
$ awk -f jansen2.awk personen.txt
Achternaam = de Vries
Achternaam =  Molema
Achternaam =  Puk
Voornaam = Jan
Achternaam =  Pietersen
Achternaam =  Janssen
Voornaam = Peter
```

Het nieuwe script (jansen2.awk):

```
BEGIN {FS=";"}
/Jansen/ {print "Voornaam = " $1}
!/Jansen/ {print "Achternaam = " $2 " " $3}
```

Probleem is nog steeds dat we het filter (de reguliere expressie vooraan), geldt voor "iets" dat in de hele regel staat. Eigenlijk zeggen we dus

> Als er ergens in de regel "`Jansen`" staat, doe dan X, anders doe Y.

Dit is als volgt op te lossen: vertel AWK dat het veld op positie 3 ($3) gelijk moet zijn aan Jansen (of niet).

```
BEGIN {FS=";"}
$3 ~/Jansen/ {print "Voornaam = " $1}
```

[1] Of juist niet voldoen; je kunt de expressie altijd omdraaien.

```
$3 !~/Jansen/ {print "Achternaam = " $2 " " $3}
```

De tilde (~) staat voor “is gelijk aan” in AWK.

11.4 Dubbele MP3-bestanden zoeken:

```
find . -iname '*.mp3' | sed -n 's/\(.*\/[0-9]*\) - \(.*\)\.mp3/\2/gpi' | sort |
uniq -c | awk '{ if ($1>1) print $0}'
```

Toelichting

1. `find . -iname '*.mp3'`
2. `sed -n 's/\(.*\/[0-9]*\) - \(.*\)\.mp3/\2/gpi'`
3. `sort | uniq -c`
4. `awk '{ if ($1>1) print $0}'`

In stap 1 worden de bestandsnamen opgezocht vanuit de huidige directory. De extensie is case-insensitive (`-iname`).

In stap 2 wordt er gefilterd op een reguliere expressie (waarover verderop meer). De modifiers (gpi) zorgen er voor dat

- `p` ➔ alleen de regels afgedrukt worden waar een match gevonden is
- `g` ➔ stop niet na de eerst gevonden match
- `i` ➔ werk case insensitive

Door de zoek/vervang combinatie in sed (opdracht `s`➔substitute) wordt alleen het juiste resultaat afgedrukt.

In stap 3 zorgen `sort` en `uniq` dat we een lijst krijgen met unieke entries van de gevonden bestandsnamen.

In stap 4 zorgen we dat alleen als items die meer dan één keer gevonden is getoond worden.

De reguliere expressie: `sed -rn 's/.+\/[0-9]+ - (.+)\.mp3/\1/p' files.txt`

- bij sed begin je met een opdracht: substitute (vervangen) ‘s’
- daarna volgt de delimiter : forward slash (/)
- daarna de echte expressie in `blauw`.
  - let op dat je bij `sed` alle haakjes moet escapen met een backslash (\)
- daarna volgt de delimiter : forward slash (/)
- vervolgens staat er in het tweede deel (`groen`) dat de gevonden expressie vervangen moet worden door de tweede groep: `(.*\)`
- na de laatste delimiter volgen de modifiers

Input file : (zie Blackboard)

Output:

```
2 Allentown
2 An Innocent Man
2 Big Shot
2 Captain Jack
2 Everybody Loves You Now
2 Goodnight Saigon
2 Honesty
2 I've Loved These Days
2 Los Angelenos
2 Only the Good Die Young
2 Say Goodbye to Hollywood
```

```
2 Stiletto
2 Summer, Highland Falls
2 Uptown Girl
2 You're My Home
```

11.5 Conclusies Linux tooling

De conclusie is dus dat we met een drietal linux tools zeer snel in hooguit een paar regels al snel slimme operaties kunnen uitvoeren op bestanden:

- sed voor zoek- en vervang
- grep voor alleen zoeken
- awk voor .csv-achtige bestanden en werken met velden/kolommen

Door de steeds betere ondersteuning op Windows is ook een realistische optie aan het worden om deze tools in te zetten in een bedrijfsomgeving.

Let op!

Bij gebruik van reguliere expressie op de shell-commandline, vereisen de tools als grep/sed/awg dat veel meer tekens voorafgegaan worden door een backslash. In het voorbeeld van de dubbele telefoonnummers (zie paragraaf 6) moet je in grep wat meer escape tekens gebruiken:

```
$ grep '\([0-9]\{10\}\);\1' telefoonnummers-dubbelingen.txt
0698756355;0698756355
0651548748;0651548748
```

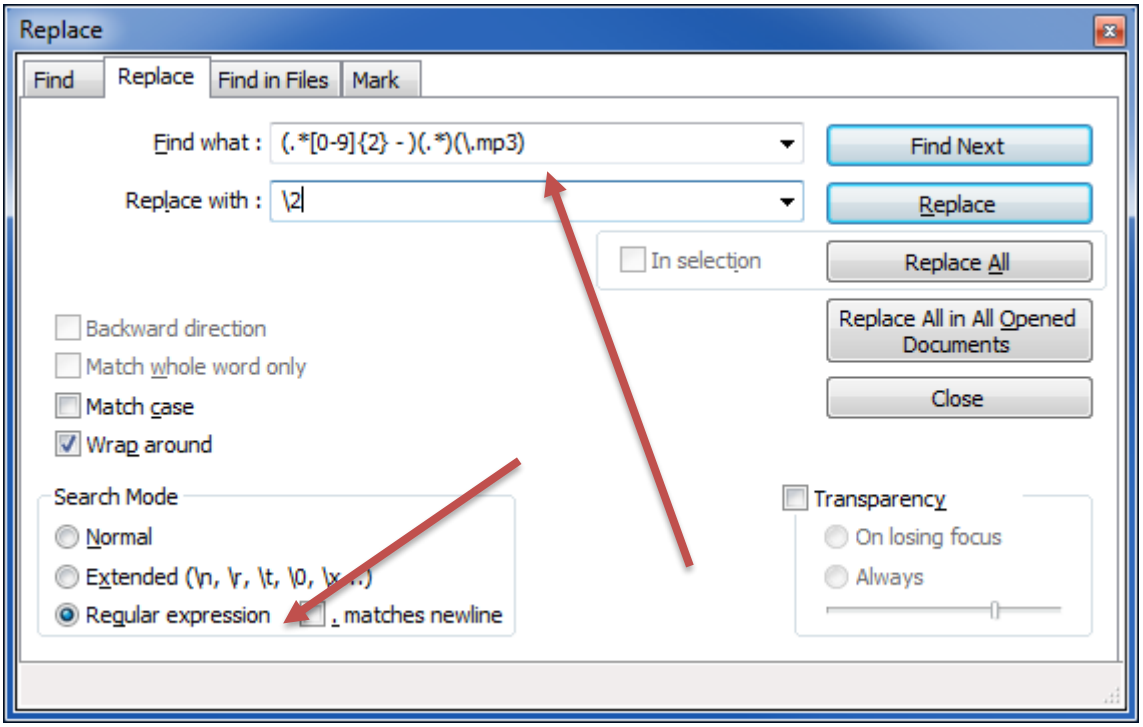
11.6 Notepad++

In Notepad++ zit een behoorlijk krachtige Reguliere Expressie engine. Je kunt Notepad++ installeren als ‘PortableApp’ (geen admin rechten nodig) of via een gewone installer. Kijk op https://notepad-plus-plus.org voor de verschillende installatie opties.

13 december 2017

Zoeken en vervangen met reguliere expressies kan via het reguliere ‘Zoek-Vervang’ dialoogvenster.

Wat opvalt is dat je delimiters (‘/’) niet hoeft op te geven bij het zoek en vervang patroon. Immers, je geeft ze op in gescheiden invoer velden.



Laad de file ‘files.txt’ in Notepad++ en voer de gegevens in zoals hierboven. Druk op ‘Replace All’ en zie wat er gebeurd.

12 Bijlagen

12.1 Broncode – ugly

In deze bijlage staat een eenvoudig programma dat hetzelfde doet als het voorbeeld in paragraaf **Fout! Verwijzingsbron niet gevonden.**: de MP3-bestandsnaam uit een lijst halen en afdrukken zonder extensie. Dit programma is traditioneel gebouwd en maakt geen gebruik van reguliere expressies. Er zijn ook andere oplossen (door de string te splitsen op basis van een separator als de backslash) maar het doel is vooral om grofweg dezelfde stappen te zetten als bij het gebruik van een Reguliere Expressie.

Een invoerregel ziet er als volgt uit:

.\An Innocent Man\10 - Keeping the Faith.mp3

Het algoritme is als volgt:

- Zoek vanaf rechts de laatste backslash
- Check of het begin van het restant met twee cijfers begint
- Haal de prefix en suffix weg
- Toon het restant (de bestandsnaam zonder cijfers, prefix en extensie mp3)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SearchReplace_Ugly
{
    class Program
    {
        static void Main(string[] args)
        {
            const string prefix = " - ";
            const string suffix = ".mp3";
```

13 december 2017

```
bool eof;
string line;

do
{
    line = Console.ReadLine();
    eof = (line == null);

    if (!eof)
    {
        int p;
        string tmp1, result;
        // Console.WriteLine(line);

        //check if there are backslashses; get index of last one.
        p = line.LastIndexOf("\\");

        // if there are backslashses p will be >=0
        if (p >= 0)
        {
            // truncate string beginning at last slash
            tmp1 = line.Substring(p+1);

            // check if the number begins with two numbers
            bool isNumeric;
            int n;
            isNumeric = int.TryParse(tmp1.Substring(0, 2), out n);

            if (isNumeric)
            {
                // remove first numbers (2 positions)
                tmp1 = tmp1.Substring(2);

                // find prefix (' - ') and suffix ('.mp3'); case insensitive search
                if (tmp1.EndsWith(suffix, StringComparison.CurrentCultureIgnoreCase) &&
                    tmp1.StartsWith(prefix, StringComparison.CurrentCultureIgnoreCase))
                {
                    //get result: remove prefix (3 chars) and suffix (4 chars); start behind prefix
                    result = tmp1.Substring(prefix.Length, tmp1.Length - suffix.Length - prefix.Length);
```

13 december 2017

```
                            Console.WriteLine(result);
                        }//if suffix and prefix are found

                    }

                }//if has backslashes
            }// if not End Of File

        } while (!eof);
    }
}
}
```

13 december 2017

12.2 Broncode – RegEx

Zie ook 12.1 voor . Deze oplossing werkt echter met een reguliere expressie om te komen tot de gezochte informatie.

Even los van de precies gebruikte statements moge duidelijk zijn dat deze broncode duidelijker en compacter is.

Let op de dubbele backslashes in de reguliere expressie.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Text.RegularExpressions;

namespace SearchReplace_Ugly
{
    class Program
    {
        static void Main(string[] args)
        {
            bool eof;
            string line;
            string pattern = "(.*\\[0-9]{2} - )(.*)(\\.mp3)";
            Regex r = new Regex(pattern, RegexOptions.IgnoreCase);

            do
            {
                line = Console.ReadLine();
                eof = (line == null);

                if (!eof)
                {
                    // Match the regular expression pattern against a text string.
                    Match m = r.Match(line);

                    // if success then process
```

```
                    if (m.Success)
                    {
                        // BEWARE: this assumes certain things about groups and captures.
                        //         this is reasonably safe because we have a good RegEx Pattern.

                        // get the "captures" in the second group
                        CaptureCollection cc = m.Groups[2].Captures;

                        // output the first capture.
                        Console.WriteLine(cc[0]);
                    }
                    // normal ==> check for possible more matches! (m.NextMatch)
                }// if not End Of File

            } while (!eof);
        }
    }
}
```

13 Bijlagen

13.1 Handige websites

http://www.grymoire.com/Unix/Regular.html#uh-12

Het gebruik van reguliere expressies in Unix sed: http://www.grymoire.com/Unix/Sed.html

https://regex101.com/