REGEXP

- Objet : Les expressions régulières
- Niveau requis :
  avisé
- Commentaires : *Les caractères spéciaux utilisés dans les commandes en ligne.*
- Suivi :
  - Création par smolski le 23/09/2013
  - Testé par smolski le 23/09/2013
- Commentaires sur le forum : Lien vers le forum concernant ce tuto[1)]

Utilisation

Les regexp servent à manipuler des chaînes de caractères : recherches, expansions, substitutions.

Manipuler les chaînes de caractères est essentiel en informatique, surtout dans les logiciels libres (dont internet !) qui favorisent dès que possible le texte comme langage commun “universel” entre les machines et l'homme aussi bien qu'entre systèmes parfois très différents.
Si tous les fichiers de configuration sont en texte dans les logiciels libres, ce n'est pas par hasard, et s'ils sont souvent en binaire dans les logiciels propriétaires non plus (-_-); .

Les regexp ont une syntaxe tortueuse qui semble provenir du fond des âges de l'informatique, mais leur puissance et la possibilité de les utiliser à l'intérieur même des chaînes de caractères les rendent incontournables et massivement employées.

Même si c'est souvent un exercice de l'esprit, ça vaut le coup de faire l'effort d'apprendre à les utiliser.

Attention, il existe plusieurs syntaxes de regexp qui diffèrent un peu selon les programmes, pas pour ennuyer l'utilisateur, mais il s'agit de choix faits pour des raisons de fonctionnalités et d'optimisation.
Il faut donc savoir si vous utilisez la norme **POSIX**, **perl**, **python**...

À noter que, concernant le shell (le shell (l'interpréteur de commandes) vous allez aussi trouver le mot **Glob**.
La nuance est que l'expression va concerner le développement des chemins et non la modification dans les chaînes de caractère.

Vous connaissez sûrement au moins deux des principales expressions Glob mises ici : `?` `*` `\` `[]`

Merci à **Haricophile** *pour cette présentation.*

Introduction

Les *regexp* des shell sont des règles de filtrage permettant de sélectionner des fichiers selon leur nom

ou leur emplacement.
Ces règles ne sont pas transmises au programme tel quel mais sont remplacées par le shell utilisé, aussi, pour la commodité de ce tuto et sauf indication contraire, nous utiliserons ici le shell bash.

Par exemple pour cette commande ls dont l'option **-d** permet à **ls** de lister uniquement les répertoires, sans leur contenu :

```
ls -d *
```

C'est bash qui va **interpréter** la regexp * (étoile) et la remplacer par **tout le contenu du répertoire** rendu ainsi lisible pour la commande **ls** et son action sur chacun d'eux comme ici par exemple :

```
fichier1 fichier2 … dossier1 …
```

ATTENTION ! De par leur statut de caractères spéciaux, les **regexp** ne peuvent pas être utilisées n'importe comment, leur rédaction demande une vrai connaissance de ce que l'on désire faire au final.

Tout d'abord, un exemple pour appréhender les notions utilisées dans cette page.
Imaginons que, dans un répertoire, nous désirons sélectionner toutes les *chaînes de caractères*[2] dont la rédaction contient **3 a** contigües (**aaa**).
Cette chaîne recherchée, formée des 3 **a** contigües (**aaa**), encadrée ou non d'autres lettres *quelconques*, se nomme dans son ensemble : `un motif`.

Pour décrire précisément ces **motifs** à un programme, on utilise des *expressions régulières* ou *regular expression* (ou encore un **regexp** en abrégé courant).

Syntaxes

Pour ces **regexp**, différents programmes *utilisent* différentes syntaxes, de ce fait leurs symboliques peuvent être différentes il faut donc bien distinguer le contexte où interviennent les **regexp**.

Par exemple, un **regexp** destiné à l'utilisation d'une commande sed, find, locate ou grep, devra être écrit différemment que dans le contexte de l'utilisation d'une commande **bash** (voir : Bash - Les metacaractères (Pattern - Glob)).

Idem pour des contextes différents...

Pour être clair dans ce wiki,

1. nous indiquerons le terme **REGEXP** pour ce qui concerne le contexte *find - locate - grep - sed*
2. et nous utiliserons le terme **GLOB** pour ce qui concerne le contexte du *bash*.

Illustration

Pour find - locate - grep - sed

Avec : `pl[oi]p` les caractères *crochets* [ ]définissent les **regexp** concernant les chaînes de caractères `plop plip`.

Pour bash :

Pour avoir la même expression avec bash, il faudra écrire : `pl{o,i}p`.
Nous voyons donc que dans le contexte **bash**, il faudra utiliser[3)] les caractères *accolades* { } pour définir les **glob** concernant les mêmes chaînes de caractères `plop plip`.

Conclusion

Il y a donc bien lieu d'*utiliser les bons termes* pour rédiger une commande au résultat identique *selon le bon contexte*.
C'est pourquoi ce tuto s'attache particulièrement à différencier ce contexte d'utilisation.

REGEXP - Utilisation pour sed find grep locate

Dans ce tuto, nous allons donc nous intéresser aux **regexp** utilisées par **sed**, **find**, **grep** et **locate** (sensiblement les mêmes).

Préparation pour executer les TP

Nous allons ici utiliser vim, l'éditeur fétiche des amoureux du terminal !

Pour réaliser les TP mis en exemple dans ce tuto, il vous faut préparer ces répertoires et fichiers.
Créer le repertoire `/home/user/tuto_regexp` :

```
mkdir ~/tuto_regexp
```

Compléter ce répertoire comme indiqué dans les TP qui suivent.
Nous seront ainsi fin prêts pour réaliser les TP mis en exemple dans ce tuto !

TP-01

Créer le fichier abraca et y inscrire le mot : `Abracadabrantesque` ainsi :

```
vim ~/tuto_regexp/abraca
```

et écrire :

```
 Abracadabrantesque
```

Enregistrer et quitter vim.

À suivre... **smolski** le 10/01/2013

Les caractères regexp

Ces caractères suivants ont un statut particulier dans les **regexp** :

- L'accent circonflexe : ^
- Le point : .
- Les crochets doubles : [ ]
- Le signe dollar : $
- L'étoile : *
- Les accolades doubles : { }
- L'anti-slash : \

Avec les **regexp**, les *shells* effectuent certaines substitutions dans les commandes entrées par les utilisateurs avant de les exécuter.

Les caractères

La première chose que l'on veut pouvoir reconnaître avec des motif, ce sont les mots que l'on peut écrire sans **REGEXP**.
Exemple :

```
Abracadabrantesque
```

Toutes ces lettres sont des caractères pouvant former une `regexp`.
`Abracadantesque` est donc déjà en soi une REGEXP reconnaissant exactement le mot `Abracadabrantesque`.

Les jokers

`A.racadabrant….e`

Comme vous l'aurez compris, les points `.` signifient *n'importe quel caractère*.
La chaîne ci-dessus est dans son ensemble une **REGEXP** qui reconnaît exactement les caractères entrant dans le mot recherché.
En effet, `.` (le point) est un caractère spécial reconnaissant *n'importe quel caractère*.

On appelle aussi ce caractère un `joker`[4)]

Les caractères échappés

Tous nos caractères spéciaux ont des significations particulières, nous l'avons vu avec le .[5], c'est vrai pour les autres.
Ainsi, pour reconnaître chacun d'eux en tant que caractère non spécial, il faut l'échapper par un \[6].

Ainsi la combinaison : `\.`[7] reconnaît simplement `.`[8]

Comme la combinaison : `\\`[9] reconnaît `\`[10]

Ou encore la combinaison : `\\\.`[11] reconnaît \.[12]

Eh oui ! 😀

Les caractères répétition

Quand on ne sait pas combien de fois un caractère va apparaître, on l'indique avec une étoile *

Exemple :
Ah* reconnaît aussi bien « Ah » que « Ahhhhhhhhhhhhhhhhhhhhhhhhhh » ou même « A »

Quand on ne sait pas si un caractère va apparaître une fois ou jamais, on l'indique avec un point d'interrogation échappé `\?`

Exemple :

La **REGEXP** : `Chevals\?` reconnaît aussi bien « Cheval » que « Chevals ».

Autre exemple : `coq\?uille`

Les ensembles de caractères possibles

`Par[lt]ez` reconnaît aussi bien « Parlez » que « Partez ».

`[a-z]` reconnaît n'importe quelle lettre minuscule (non altérée)2) de l'alphabet.

`[A-Za-z]` reconnaît n'importe quelle lettre (non altérée) de l'alphabet.

`[0-9\-]` reconnaît les chiffres et le caractère « - ».

Remarque :

Il y a aussi des classes de caractères déjà prédéfinies.

Tableau des classes prédéfinies Source (merci titia) 😀

classe	Resultat
[:alnum:]	caractères alphanumériques (équivalent à [A-Za-z0-9])

classe	Resultat
[:alpha:]	caractères alphabétiques ([A-Za-z])
[:blank:]	caractères blanc (espace, tabulation)
[:ctrl:]	caractères de contrôle (les premiers du code ASCII
[:digit:]	chiffre ([0-9])
[:graph:]	caractère d'imprimerie (qui fait une marque sur l'écran en quelque sorte)
[:print:]	caractère imprimable (qui passe à l'imprimante ... tout sauf les caractères de contrôle)
[:punct:]	caractère de ponctuation
[:space:]	caractère d'espacement
[:upper:]	caractère majuscule
[:xdigit:]	caractère hexadécimal

Les caractères interdits

`[^x]` reconnaît tous les caractères sauf « x »

`[^xy]` reconnaît tous les caractères sauf « x » et « y »

`[^a-z]` reconnaît tous les caractères sauf les lettres minuscules non altérées

Début et fin de ligne

^ désigne un début de ligne

$ désigne une fin de ligne

Exemple :

`^Pouet` reconnaîtra le motif *Pouet* s'il se trouve en début de ligne.

^$ identifie une ligne vide.

Combinaisons

`[ab]*` comme [ab] reconnaît aussi bien « a » que « b », [ab]* reconnaît aussi bien « aaaaaaaaa » que « abababbbbbbbbb » ou que « babbbaaa »

`Abra*[ca]*dabrante\?` reconnaît les chaînes commençant par « Abr », suivi par un nombre quelconque de « a », puis un nombre quelconque de « c » et de « a » suivis par « dabrant », suivi ou non par « e ».

`[^0-9]toto` identifie les lignes contenant une chaîne toto et le premier caractère ne doit pas être un chiffre, par exemple : **atoto**, **gtoto** mais pas **1toto**, 5toto.

Et :

[^a-zA-Z] n'importe quel caractère sauf une lettre minuscule ou majuscule.
Ici, attention où vous placez le point circonflexe ^, si vous tapez [1-3^], c'est désigner uniquement les

caractères `1 2, 3 et ^`.

D'une manière plus générale voici comment `[ ]` et `( - )` peuvent être utilisés :

`[A-D]` intervalle de A à D (A, B, C, D) par exemple **bof[A-D]** donne *bofA, bofB, bofC, bofD*

`[2-5]` intervalle de 2 à 5 (2, 3, 4, 5) par exemple **12[2-5]2** donne *1222, 1232, 1242, 1252*

`[2-56]` intervalle de 2 à 5 et 6 (*et non pas 56*) (2, 3, 4, 5, 6) par exemple **12[2-56]2** donne *1222, 1232, 1242, 1252, 1262*

`a-dA-D]` intervalle de a à d et A à D (a, b, c, d, A, B, C, D) par exemple **z[a-dA-D]y** donne *zay, zby, zcy, zdy, zAy, zBy, zCy, zDy*

`[1-3-]` intervalle de 1 à 3 et - (1, 2, 3, -) par exemple **[1-3-]3** donne *13, 23, 33, -3*

`[a-cI-K1-3]` intervalle de a à c, I à K et 1 à 3 (*a, b, c, I, J, K, 1, 2, 3*)

Liens

- http://cyberzoide.developpez.com/unix/sys.php3#metachar
- http://www.funix.org/fr/unix/expr-sed.htm

1)
N'hésitez pas à y faire part de vos remarques, succès, améliorations ou échecs !
2)
CHAÎNE DE CARACTÈRES :
Les *chaîne de caractères* sont des caractères accolés, un mot, une phrase ou un nom de fichier.
3)
à la place des caractères *crochets* [ ]de l'exemple précédent
4)
En bash, le caractère joker est ?
5) 8)
,
point
6) 10)
,
anti-slash
7) 12)
,
anti-slash point
9)
anti-slash antislash
11)
anti-slash anti-slash anti-slash point