eBook Gratuit

APPRENEZ

Regular Expressions

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#regex

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: regular-expressions

It is an unofficial and free Regular Expressions ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Regular Expressions.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec les expressions régulières

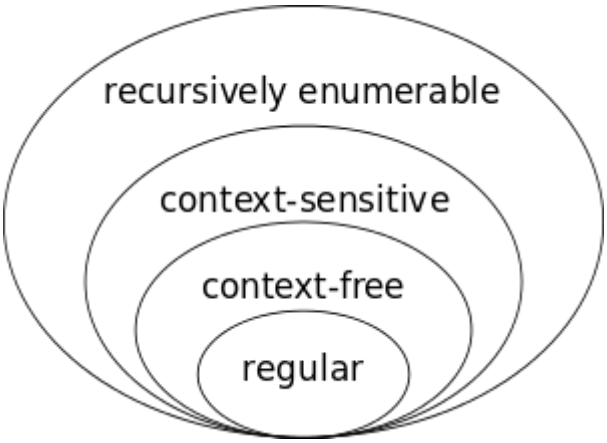
Remarques

Pour beaucoup de programmeurs, le *regex* est une sorte d'épée magique qu'ils lancent pour résoudre tout type d'analyse de texte. Mais cet outil n'a rien de magique, et même si c'est très bien, ce n'est pas un langage de programmation complet (*c'est*- à- *dire* qu'il n'est **pas** complet).

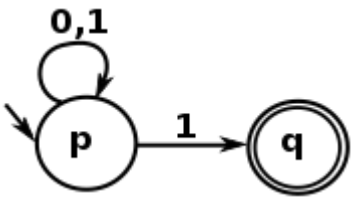
Que signifie l'expression régulière?

Les expressions régulières expriment un langage défini par une *grammaire régulière* pouvant être résolue par un *automate fini non déterministe* (NFA), où la correspondance est représentée par les états.

Une *grammaire régulière* est la *grammaire* la plus simple exprimée par la *hiérarchie de Chomsky* .



En termes simples, un langage régulier est exprimé visuellement par ce qu'une NFA peut exprimer, et voici un exemple très simple de NFA:



Et le langage *Regular Expression* est une représentation textuelle d'un tel automate. Ce dernier exemple est exprimé par l'expression rationnelle suivante:

```
^[01]*1$
```

Qui correspond à une chaîne commençant par `0` ou `1` , en répétant 0 fois ou plus, qui se termine par `1` . En d'autres termes, c'est une expression rationnelle qui permet de faire correspondre les

nombres impairs de leur représentation binaire.

Toutes les regex sont-elles réellement une grammaire *régulière* ?

En fait, ils ne le sont pas. De nombreux moteurs regex ont été améliorés et utilisent *des automates déroulants* , qui peuvent s'accumuler et afficher des informations au fur et à mesure de leur exécution. Ces automates définissent ce qu'on appelle *des* grammaires *sans* contexte dans la hiérarchie de Chomsky. L'utilisation la plus courante de celles dans les *regex* non régulières est l'utilisation d'un modèle récursif pour la correspondance entre parenthèses.

Une expression rationnelle récursive comme celle qui suit (qui correspond à une parenthèse) est un exemple d'une telle implémentation:

```
{((?>[^\(\)]+|(?R))*)}
```

(cet exemple ne fonctionne pas avec le moteur `re` de python, mais avec le moteur `regex` ou avec le moteur PCRE).

Ressources

Pour plus d'informations sur la théorie derrière les expressions régulières, vous pouvez vous référer aux cours suivants mis à disposition par le MIT:

- Automates, calculabilité et complexité
- Expressions régulières et Grammars
- Spécification de langues avec des expressions régulières et des grammaires sans contexte

Lorsque vous écrivez ou déboguez une expression rationnelle complexe, il existe des outils en ligne qui peuvent aider à visualiser les expressions rationnelles en tant qu'automates, comme le site debuggex .

Versions

PCRE

Version	Libéré
2	2015-01-05
1	1997-06-01

Utilisé par: PHP 4.2.0 (et supérieur), Delphi XE (et supérieur), Julia , Notepad ++

Perl

Version	Libéré
1	1987-12-18
2	1988-06-05
3	1989-10-18
4	1991-03-21
5	1994-10-17
6	2009-07-28

.NET

Version	Libéré
1	2002-02-13
4	2010-04-12

Langues: C #

Java

Version	Libéré
4	2002-02-06
5	2004-10-04
7	2011-07-07
SE8	2014-03-18

JavaScript

Version	Libéré
1.2	1997-06-11
1.8.5	2010-07-27

Python

Version	Libéré
1.4	1996-10-25
2.0	2000-10-16
3.0	2008-12-03
3.5.2	2016-06-07

Oniguruma

Version	Libéré
Initiale	2002-02-25
5.9.6	2014-12-12
Onigmo	2015-01-20

Renforcer

Version	Libéré
0	1999-12-14
1,61.0	2016-05-13

POSIX

Version	Libéré
BRE	1997-01-01
AVANT	2008-01-01

Langues: Bash

Examples

Guide du personnage

Notez que certains éléments de syntaxe ont un comportement différent selon l'expression.

Syntaxe	La description
`?`	Faites correspondre le caractère ou la sous-expression précédent 0 ou 1 fois. Également utilisé pour les groupes non capturés et les groupes de capture nommés.
`*`	Faites correspondre le caractère précédent ou la sous-expression 0 ou plusieurs fois.
`+`	Faites correspondre le caractère ou la sous-expression précédent une ou plusieurs fois.
`{n}`	Faites correspondre le caractère ou la sous-expression précédent exactement *n* fois.
`{min,}`	Faites correspondre le caractère précédent ou la sous-expression au *moins* plusieurs fois.
`{,max}`	Faites correspondre le caractère précédent ou la sous-expression *max* ou moins.
`{min,max}`	Faites correspondre le caractère ou la sous-expression précédent avec les temps *minimum* , mais pas plus que les durées `max` .
`-`	Lorsque inclus entre crochets indique `to` ; Par exemple, [3-6] correspond aux caractères 3, 4, 5 ou 6.
`^`	Début de chaîne (ou début de ligne si l'option multiline `/m` est spécifiée) ou annule une liste d'options (par exemple, si elle est entre crochets `[]`)
`$`	Fin de chaîne (ou fin d'une ligne si l'option multiligne `/m` est spécifiée).
`( ... )`	Groupes sous-expressions, capture le contenu correspondant dans des variables spéciales (`\1` , `\2` , etc.) pouvant être utilisées ultérieurement dans la même expression régulière, par exemple `(\w+)\s\1\s` correspond à la répétition des mots
`(?<name> ... )`	Regroupe les sous-expressions et les capture dans un groupe nommé
`(?: ... )`	Regrouper les sous-expressions sans capturer
`.`	Correspond à n'importe quel caractère sauf les sauts de ligne (`\n` et généralement `\r`).
`[ ... ]`	Tout caractère entre ces parenthèses devrait être associé une fois. NB: `^` suivre le crochet ouvert annule cet effet. `-` dans les parenthèses, une plage de valeurs peut être spécifiée (à moins que ce soit le premier ou le dernier caractère, auquel

Syntaxe	La description
	cas il ne représente qu'un tiret régulier).
`\`	Échappe au caractère suivant. Également utilisé dans les méta-séquences - les jetons regex ayant une signification particulière.
`\$`	dollar (c.-à-d. un caractère spécial échappé)
`\(`	parenthèse ouverte (c'est-à-dire un caractère spécial échappé)
`\)`	parenthèse proche (c.-à-d. un caractère spécial échappé)
`\*`	astérisque (c'est-à-dire un caractère spécial échappé)
`\.`	dot (c'est-à-dire un caractère spécial échappé)
`\?`	point d'interrogation (c'est-à-dire un caractère spécial échappé)
`\[`	crochet gauche (ouvert) (c'est-à-dire un caractère spécial échappé)
`\\`	barre oblique inverse (c'est-à-dire un caractère spécial échappé)
`\]`	crochet droit (proche) (c'est-à-dire un caractère spécial échappé)
`\^`	caret (c'est-à-dire un caractère spécial échappé)
`\{`	accolade / accolade gauche (ouverte) (c'est-à-dire un caractère spécial échappé)
`|`	pipe (c'est-à-dire un caractère spécial échappé)
`\}`	accolades / accolades (c'est-à-dire un caractère spécial échappé)
`\+`	plus (c'est-à-dire un caractère spécial échappé)
`\A`	début d'une chaîne
`\Z`	fin d'une chaîne
`\z`	absolu d'une chaîne
`\b`	limite de mot (séquence alphanumérique)
`\1` , `\2` , etc.	back-references à des sous-expressions précédentes, regroupées par `()` , `\1` signifie la première correspondance, `\2` signifie deuxième correspondance, etc.
`[\b]`	backspace - quand `\b` trouve dans une classe de caractères (`[]`) correspond à un retour arrière
`\B`	négation `\b` - correspond à n'importe quelle position entre caractères à deux mots et à n'importe quelle position entre deux caractères non-mots

Syntaxe	La description
`\D`	non-chiffre
`\d`	chiffre
`\e`	échapper
`\f`	aliment de forme
`\n`	saut de ligne
`\r`	retour de chariot
`\S`	espace non blanc
`\s`	espace blanc
`\t`	languette
`\v`	onglet vertical
`\W`	non-mot
`\w`	mot (caractère alphanumérique)
`{ ... }`	jeu de caractères nommé
`|`	ou; c'est-à-dire délimite les options précédentes et précédentes.

Lire Démarrer avec les expressions régulières en ligne:
https://riptutorial.com/fr/regex/topic/259/demarrer-avec-les-expressions-regulieres

Chapitre 2: Caractères d'ancre: Dollar ($)

Remarques

Un grand nombre de moteurs regex utilisent un mode "multi-lignes" afin de rechercher plusieurs lignes dans un fichier de manière indépendante.

Par conséquent, lorsque vous utilisez `$` , ces moteurs correspondent à la fin de toutes les lignes. Cependant, les moteurs qui n'utilisent pas ce type de mode multiligne ne correspondront qu'à la dernière position de la chaîne fournie pour la recherche.

Examples

Faire correspondre une lettre à la fin d'une ligne ou d'une chaîne

```
g$
```

Ce qui précède correspond à une lettre (la lettre `g`) à la fin d'une *chaîne* dans la plupart des moteurs d'expression régulière (pas dans Oniguruma , où `$` anchor correspond à la fin d'une ligne par défaut et le modificateur `m` (*MULTILINE*) est utilisé pour créer une `.` correspondre tous les caractères, y compris les caractères de saut de ligne, en tant que modificateur DOTALL dans la plupart des autres expressions de regex NFA). `$` Anchor correspondra à la première occurrence d'une lettre `g` avant la fin des chaînes suivantes:

Dans les phrases suivantes, seules les lettres en **gras** correspondent:

> Les ancres sont des caractères qui, en fait, ne correspondent à aucun caractère d'une strin **g**
>
> Leur objectif est de correspondre à une position spécifique dans cette chaîne.
>
> Bob était helpin **g**
>
> Mais son édition a introduit des exemples qui ne correspondaient pas!

Dans la plupart des expressions rationnelles, `$` anchor peut également correspondre à un caractère de retour à la ligne ou à un saut de ligne (séquence), dans un mode *MULTILINE* , où `$` correspond à la fin de chaque ligne au lieu d'une chaîne. Par exemple, en utilisant à nouveau `g$` comme regex, en mode multiligne, les caractères en italique dans la chaîne suivante correspondent:

```
tvxlt obofh necpu riist g\n aelxk zlhdx lyogu vcbke pzyay wtsea wbrju jztg\n drosf ywhed bykie
lqmzg wgyhc lg\n qewrx ozrvm jwenx
```

Lire Caractères d'ancre: Dollar ($) en ligne: https://riptutorial.com/fr/regex/topic/1603/caracteres-d-ancre--dollar----

Chapitre 3: Classes de caractères

Remarques

Classes simples

Regex	Allumettes
`[abc]`	N'importe lequel des caractères suivants: `a` , `b` ou `c`
`[az]`	Tout caractère **compris** entre `a` et `z` , **inclus** (on appelle cela une *plage*)
`[0-9]`	N'importe quel chiffre de `0` à `9` **inclus**

Cours communs

Certains groupes / plages de caractères sont si souvent utilisés, ils ont des abréviations spéciales:

Regex	Allumettes
`\w`	Caractères alphanumériques plus le trait de soulignement (également appelé "caractères de mots")
`\W`	Caractères non verbaux (identiques à `[^\w]`)
`\d`	Chiffres (*plus larges* que `[0-9]` puisque les chiffres persans, indiens, etc.)
`\D`	Non-chiffres (*plus courts* que `[^0-9]` depuis le rejet des chiffres persans, indiens, etc.)
`\s`	Caractères d'espacement (espaces, tabulations, etc.) **Remarque** : peut varier en fonction de votre moteur / contexte
`\S`	Caractères non blancs

Classes négatives

Un caret (^) après le carré ouvrant fonctionne comme une négation des caractères qui le suivent. Cela correspond à tous les caractères qui ne sont pas dans la classe de caractères.

Les classes de caractères négatives correspondent également aux caractères de saut de ligne. Par conséquent, si elles ne doivent pas être comparées, les caractères de saut de ligne

spécifiques doivent être ajoutés à la classe (\ r et / ou \ n).

Regex	Allumettes
`[^AB]`	Tout caractère **autre** que `A` et `B`
`[^\d]`	N'importe quel caractère, **sauf les** chiffres

Examples

Les bases

Supposons que nous ayons une liste d'équipes nommées comme ceci: `Team A` , `Team B` ,..., `Team Z` Alors:

- `Team [AB]` : Cela correspondra à l' `Team A` ou à l' `Team B`
- `Team [^AB]` : Cela correspond à n'importe quelle équipe à l' **exception de l'** `Team A` ou de l' `Team B`

Nous avons souvent besoin de faire correspondre des caractères qui "appartiennent" ensemble dans un contexte ou un autre (comme des lettres de `A` à `Z`), et c'est à cela que servent les classes de caractères.

Match différent, mots similaires

Considérons la classe de caractère `[aeiou]` . Cette classe de caractères peut être utilisée dans une expression régulière pour correspondre à un ensemble de mots épelés de la même manière.

`b[aeiou]t` correspond à:

- chauve souris
- pari
- bit
- bot
- mais

Il ne correspond pas:

- combat
- btt
- bt

Les classes de personnage correspondent à un et un seul personnage à la fois.

Correspondance non alphanumérique (classe de caractères nuls)

```
[^0-9a-zA-Z]
```

Cela correspondra à tous les caractères qui ne sont ni des chiffres ni des lettres (caractères alphanumériques). Si le caractère de soulignement _ est également à nier, l'expression peut être raccourcie à:

```
[^\w]
```

Ou:

```
\W
```

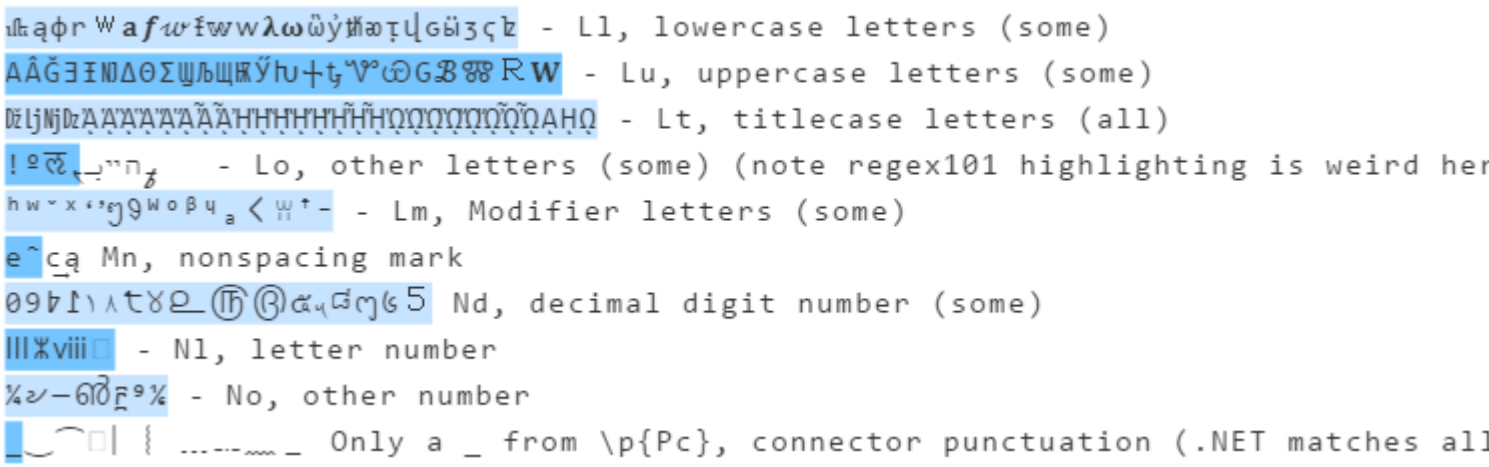
Dans les phrases suivantes:

1. Salut, ça va?
2. J'ai hâte pour 2017 !!!

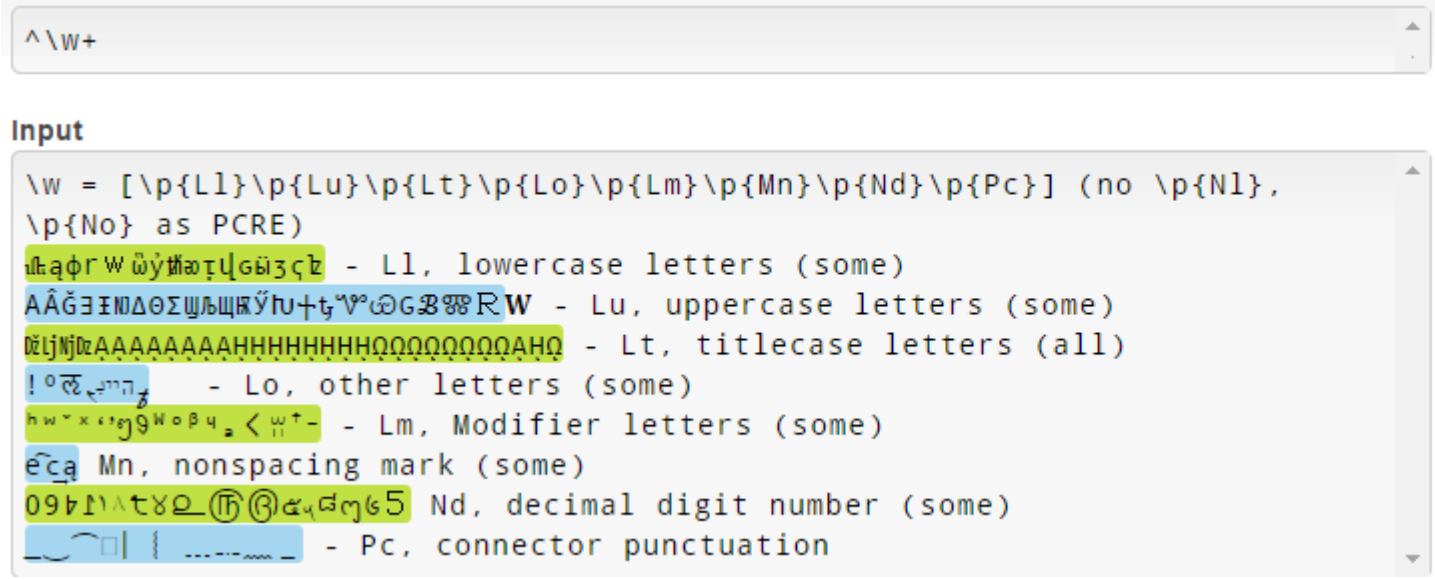
Les caractères suivants correspondent à:

1. `,` , `,` , `'` , `?` et le caractère de fin de ligne.
2. `'` , `,` , `!` et le caractère de fin de ligne.

UNICODE NOTE
Notez que certaines variantes prenant en charge les propriétés de caractères Unicode peuvent interpréter `\w` et `\W` comme `[\p{L}\p{N}_]` et `[^\p{L}\p{N}_]` ce qui signifie d'autres lettres Unicode et les caractères numériques seront également inclus (voir documents PCRE). Voici un test PCRE `\w` :

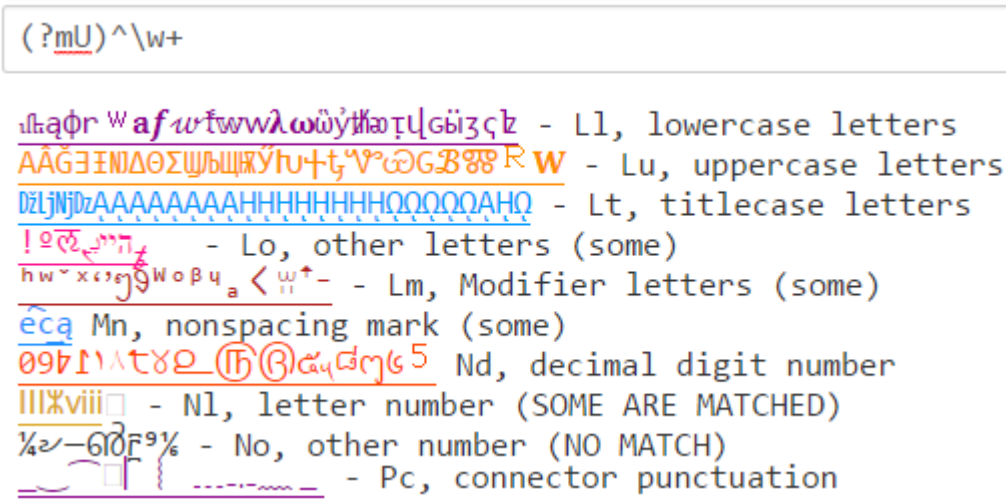


Dans .NET, `\w` = `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Lm}\p{Mn}\p{Nd}\p{Pc}]` , et notez qu'il ne correspond pas à `\p{Nl}` et à `\p{No}` contrairement à PCRE (voir la documentation de `\w` .NET):



Notez que pour une raison quelconque, les lettres minuscules Unicode 3.1 (comme ▯▯▯▯▯▯▯▯▯) ne correspondent pas.

Java `(?U)\w` correspondra à un mélange de ce que `\w` correspond dans PCRE et .NET:



Correspondance sans chiffres (classe de caractères négative)

```
[^0-9]
```

Cela correspondra à tous les caractères qui ne sont pas des chiffres ASCII.

Si les chiffres Unicode doivent également être annulés, l'expression suivante peut être utilisée, en fonction de vos paramètres de saveur / langue:

```
[^\d]
```

Cela peut être raccourci à:

```
\D
```

Vous devrez peut-être activer explicitement la prise en charge des propriétés de caractère Unicode en utilisant le modificateur `u` ou par programmation dans certaines langues, mais cela peut ne pas être évident. Pour transmettre explicitement l'intention, la construction suivante peut être utilisée (lorsque le support est disponible):

```
\P{N}
```

Ce qui signifie *par définition* : tout caractère qui n'est pas un caractère numérique dans un script. Dans une plage de caractères négative, vous pouvez utiliser:

```
[^\p{N}]
```

Dans les phrases suivantes:

1. Salut, ça va?
2. J'ai hâte pour 2017 !!!

Les caractères suivants seront appariés:

1. `,` , , `'` , `?` , le caractère de fin de ligne et toutes les lettres (minuscules et majuscules).
2. `'` , , `!` , le caractère de fin de ligne et toutes les lettres (minuscules et majuscules).

Classe de personnage et problèmes communs rencontrés par les débutants

1. Classe de caractère

La classe de caractères est désignée par `[]` . Le contenu d'une classe de caractères est traité `single character separately` . par exemple, supposons que nous utilisions

```
[12345]
```

Dans l'exemple ci-dessus, cela signifie correspondre à `1 or 2 or 3 or 4 or 5` . En termes simples, il peut être compris comme `or condition for single characters` (**accent sur un seul caractère**)

1.1 Mot de prudence

- Dans la classe de caractères, il n'y a aucun concept de correspondance d'une chaîne. Donc, si vous utilisez regex `[cat]` , cela ne signifie pas qu'il devrait correspondre littéralement au mot `cat` mais cela devrait correspondre à `c` ou `a` ou `t` . Il s'agit d'un malentendu très courant chez les personnes les plus récentes.
- Parfois, les gens utilisent `|` (alternance) à l'intérieur de la classe de caractères en pensant qu'il agira comme une `OR condition` qui est incorrecte. par exemple en utilisant `[a|b]` signifie en fait correspondre à `a` ou `|` (littéralement) ou `b` .

2. Gamme dans la classe de caractères

La plage de la classe de caractères est indiquée par un signe `-` . Supposons que nous voulons trouver un caractère dans les alphabets anglais de `A` à `Z` Cela peut être fait en utilisant la classe de caractères suivante

```
[A-Z]
```

Cela pourrait être fait pour toute plage ASCII ou unicode valide. Les gammes les plus couramment utilisées comprennent `[AZ]` , `[az]` ou `[0-9]` . En outre, ces plages peuvent être combinées en classe de caractères

```
[A-Za-z0-9]
```

Cela signifie que tous les caractères compris entre `A to Z` ou `a to z` ou `0 to 9` . La commande peut être n'importe quoi. Donc, ce qui précède est équivalent à `[a-zA-Z0-9]` tant que la plage que vous définissez est correcte.

2.1 Mot de prudence

- Parfois, lorsque vous écrivez des plages pour `A` à `Z` gens l'écrivent comme `[Az]` . C'est faux dans la plupart des cas car nous utilisons `z` au lieu de `Z` Donc, cela correspond à n'importe quel caractère de la plage ASCII `65` (de A) à `122` (de z), qui inclut de nombreux caractères non intentionnels après la plage ASCII `90` (de Z). **Cependant** , `[Az]` peut être utilisé pour faire correspondre toutes les lettres `[a-zA-Z]` dans une expression régulière de type POSIX lorsque le classement est défini pour une langue particulière. `[[ "ABCEDEF[]_abcdef" =~ ([Az]+) ]] && echo "${BASH_REMATCH[1]}"` sur Cygwin avec `LC_COLLATE="en_US.UTF-8"` produit `ABCEDF` . Si vous définissez `LC_COLLATE` sur `C` (sur Cygwin, fait avec `export`), cela donnera le `ABCEDEF[]_abcdef` attendu.

- Signification de `-` classe de caractères à l'intérieur est spécial. Il désigne la plage comme expliqué ci-dessus. *Que faire si nous voulons correspondre `-` caractère littéralement?* Nous ne pouvons pas le mettre ailleurs, sinon il indiquera des plages s'il est placé entre deux caractères. Dans ce cas , nous devons mettre `-` en début de classe de caractères comme `[-AZ]` ou en fin de classe de caractères comme `[AZ-]` ou `escape it` si vous voulez l' utiliser au milieu comme `[AZ\-az]` .

3. Classe de caractère nié

La classe de caractère négatif est désignée par `[^..]` . Le signe caret `^` correspond à n'importe quel caractère à l'exception de celui présent dans la classe de caractères. par exemple

```
[^cat]
```

signifie que n'importe quel caractère, sauf `c` ou `a` ou `t` .

3.1 Mot de prudence

- Le sens du signe caret `^` correspond à la négation que s'il se trouve au début de la classe de caractères. S'il est ailleurs dans la classe de caractères, il est traité comme un caractère

littéral sans signification particulière.
- Certaines personnes écrivent des regex comme `[^]` . Dans la plupart des moteurs de regex, cela génère une erreur. La raison en est lorsque vous utilisez `^` dans la position de départ, il attend au moins un caractère qui devrait être annulé. En *JavaScript* cependant, il s'agit d'une construction valide correspondant à *tout sauf à rien* , c.-à-d. Qu'elle correspond à n'importe quel symbole possible (sauf les signes diacritiques, au moins dans ES5).

Classes de caractères POSIX

Les classes de caractère POSIX sont des séquences prédéfinies pour un certain ensemble de caractères.

Classe de personnage	La description
`[:alpha:]`	Caractères alphabétiques
`[:alnum:]`	Caractères alphabétiques et chiffres
`[:digit:]`	Chiffres
`[:xdigit:]`	Chiffres hexadécimaux
`[:blank:]`	Espace et onglet
`[:cntrl:]`	Caractères de contrôle
`[:graph:]`	Caractères visibles (tout sauf les espaces et les caractères de contrôle)
`[:print:]`	Caractères et espaces visibles
`[:lower:]`	Minuscules
`[:upper:]`	Lettres capitales
`[:punct:]`	Ponctuation et symboles
`[:space:]`	Tous les caractères d'espacement, y compris les sauts de ligne

Des classes de caractères supplémentaires peuvent être disponibles en fonction de l'implémentation et / ou des paramètres régionaux.

Classe de personnage	La description
`[:<:]`	Début de mot
`[:>:]`	Fin de mot

Classe de personnage	La description
`[:ascii:]`	Caractères ASCII
`[:word:]`	Lettres, chiffres et soulignement. Équivalent à `\w`

Pour utiliser l'intérieur d'une séquence de parenthèses (alias classe de caractères), vous devriez également inclure les crochets. Exemple:

```
[[:alpha:]]
```

Cela correspondra à un caractère alphabétique.

```
[[:digit:]-]{2}
```

Cela correspondra à 2 caractères, soit des chiffres, soit `-` . Ce qui suit correspondra à:

- `--`
- `11`
- `-2`
- `3-`

Plus d'informations sont disponibles sur: Regular-expressions.info

Lire Classes de caractères en ligne: https://riptutorial.com/fr/regex/topic/1757/classes-de-caracteres

Chapitre 4: Échapper

Examples

Littéraux bruts

Il est préférable pour la lisibilité (et votre santé mentale) d'éviter de fuir les fuites. C'est là que les littéraux de chaînes brutes entrent en jeu. (Notez que certaines langues autorisent les délimiteurs, qui sont préférables aux chaînes de caractères. Mais c'est une autre section.)

Ils fonctionnent généralement de la même manière que cette réponse décrit :

> [A] backslash, `\` , est considéré comme signifiant "juste une barre oblique inverse" (sauf quand il vient juste avant une citation qui terminerait le littéral) - pas de "séquences d'échappement" pour représenter les nouvelles lignes, les tabulations, les backspaces, les flux , etc.

Toutes les langues ne les ont pas et celles qui utilisent une syntaxe variable. C # les appelle réellement *littéraux littéraux* , mais c'est la même chose.

Python

```
pattern = r"regex"
```

```
pattern = r'regex'
```

C ++ (11+)

La syntaxe ici est extrêmement polyvalente. La seule règle consiste à utiliser un délimiteur qui n'apparaît nulle part dans le regex. Si vous faites cela, aucune fuite supplémentaire n'est nécessaire pour rien dans la chaîne. Notez que les parenthèses `()` ne font pas partie de l'expression rationnelle:

```
pattern = R"delimiter(regex)delimiter";
```

VB.NET

Utilisez simplement une chaîne normale. Les barres obliques inverses sont TOUJOURS littérales .

C

```
pattern = @"regex";
```

Notez que cette syntaxe autorise également `""` (deux guillemets) comme une forme échappée de `"` .

Cordes

Dans la plupart des langages de programmation, pour obtenir une barre oblique inverse dans une chaîne générée à partir d'un littéral de chaîne, chaque barre oblique inverse doit être doublée dans le littéral de chaîne. Sinon, il sera interprété comme une évasion pour le prochain caractère.

Malheureusement, toute barre oblique inverse requise par l'expression rationnelle doit être une barre oblique inverse littérale. C'est pourquoi il devient nécessaire d'avoir des "échappements" (`\\`) lorsque des expressions rationnelles sont générées à partir de littéraux de chaîne.

De plus, les guillemets (`"` ou `'`) dans le littéral de chaîne peuvent devoir être échappés, en fonction de ce qui entoure le littéral de chaîne. Dans certaines langues, il est possible d'utiliser l'un ou l'autre style de guillemets pour une chaîne échapper à toute la chaîne littérale).

Dans certaines langues (par exemple: Java <= 7), les expressions rationnelles ne peuvent pas être exprimées directement en tant que littéraux tels que `/\w/` ; ils doivent être générés à partir de chaînes, et normalement les littéraux de chaîne sont utilisés - dans ce cas, `"\\w"` . Dans ces cas, les caractères littéraux tels que les guillemets, les barres obliques inverses, etc. doivent être échappés. La manière la plus simple d'y parvenir est d'utiliser un outil (comme *RegexPlanet*). Cet outil spécifique est conçu pour Java, mais il fonctionnera pour tout langage avec une syntaxe de chaîne similaire.

Quels personnages doivent être échappés?

L'échappement de caractères est ce qui permet de rechercher littéralement certains caractères (réservés par le moteur de regex pour la manipulation des recherches) dans la chaîne d'entrée. L'échappée dépend du contexte. Par conséquent, cet exemple ne couvre pas les chaînes ou les délimiteurs qui s'échappent.

Des backslashes

Dire que la barre oblique inverse est le caractère "évasion" est un peu trompeur. Le backslash s'échappe et le backslash apporte; il active ou désactive le métacaractère et le statut littéral du personnage qui le précède.

Pour utiliser un backslash littéral n'importe où dans une regex, il doit être échappé par une autre barre oblique inverse.

S'échapper (en dehors des classes de caractères)

Plusieurs caractères doivent être échappés pour être pris à la lettre (au moins en dehors des classes de caractères):

- Supports: `[]`
- Parenthèses: `()`
- Accolades: `{}`
- Opérateurs: `*` , `+` `?` , `|`
- Ancres: `^` , `$`
- Autres: `.` , `\`
- Pour utiliser un littéral `^` au début ou un littéral `$` à la fin d'une expression régulière, le caractère doit être échappé.
- Certaines saveurs n'utilisent que `^` et `$` comme métacaractères respectivement au début ou à la fin de l'expression rationnelle. Dans ces saveurs, aucune fuite supplémentaire n'est nécessaire. En général, il vaut mieux y échapper.

S'échapper dans les classes de personnages

- Il est préférable d'échapper aux crochets (`[` et `]`) lorsqu'ils apparaissent comme des littéraux dans une classe de caractères. Sous certaines conditions, cela n'est pas nécessaire, en fonction de la saveur , mais cela nuit à la lisibilité.
- Le caret, `^` , est un méta lorsqu'il est placé en tant que premier caractère dans une classe de caractères: `[^aeiou]` . Partout ailleurs dans la classe char, c'est juste un caractère littéral.
- Le tiret, `-` , est un caractère méta, sauf s'il est au début ou à la fin d'une classe de caractères. Si le premier caractère de la classe char est un caret `^` , alors ce sera un littéral si c'est le deuxième caractère de la classe char.

Échapper au remplacement

Il y a aussi des règles pour échapper au remplacement, mais aucune des règles ci-dessus ne s'applique. Les seuls métacaractères sont `$` et `\` , du moins lorsque `$` peut être utilisé pour référencer des groupes de capture (comme `$1` pour le groupe 1). Pour utiliser un littéral `$` , échappez-y: `\$5.00` . De même `\` : `C:\\Program Files\\` .

Exceptions BRE

Alors que ERE (expressions régulières étendues) reflète la syntaxe typique de Perl, BRE (expressions régulières de base) présente des différences significatives en matière d'échappée:

- Il existe une syntaxe abrégée différente. Tous les `\d` , `\s` , `\w` et ainsi de suite ont disparu. Au lieu de cela, il a sa propre syntaxe (que POSIX appelle confusément "classes de caractères"), comme `[:digit:]` . Ces constructions doivent être dans une classe de caractères.

- Il y a peu de métacaractères (`.` , `*` , `^` , `$`) Qui peuvent être utilisés normalement. TOUS les autres métacaractères doivent être échappés différemment:

Bretelles `{}`

- `a{1,2}` correspond `a{1,2}` . Pour faire correspondre `a` ou `aa` , utilisez `a\{1,2\}`

Parenthèses `()`

- `(ab)\1` n'est pas valide, car il n'y a pas de groupe de capture 1. Pour le réparer et faire correspondre `abab` utilisez `\(ab\)\1`

Barre oblique inverse

- À l'intérieur des classes de caractères (appelées expressions de parenthèses dans POSIX), la barre oblique inverse n'est pas un métacaractère (et n'a pas besoin d'être échappé). `[\d]` correspond à `\` ou à `d` .
- Partout ailleurs, échappez-vous comme d'habitude.

Autre

- `+` et `?` sont des littéraux. Si le moteur BRE les prend en charge en tant que métacaractères, ils doivent être échappés en tant que `\?` et `\+` .

/ Délimiteurs /

De nombreuses langues permettent à l'expression régulière d'être délimitée ou délimitée entre deux caractères spécifiques, généralement la barre oblique `/` .

Les délimiteurs ont un impact sur les échappements: si le délimiteur est `/` et que l'expression rationnelle doit chercher `/` littéraux, la barre oblique doit être échappée avant de pouvoir être un littéral (`\/`).

La fuite excessive nuit à la lisibilité, il est donc important de considérer les options disponibles:

Le javascript est unique car il permet d'utiliser une barre oblique comme délimiteur, mais rien d'autre (bien qu'il autorise les expressions rationnelles).

Perl 1

Perl, par exemple, permet presque n'importe quoi d'être un délimiteur. Même les caractères arabes:

```
$str =~ m ش ش
```

Des règles spécifiques sont mentionnées dans la documentation de Perl .

PCRE autorise deux types de délimiteurs: les délimiteurs appariés et les délimiteurs de style crochet. Les délimiteurs appariés utilisent la paire d'un seul personnage, tandis que les délimiteurs de style crochet utilisent un couple de caractères qui représente une paire d'ouverture et de

fermeture.

- Délimiteurs correspondants:! `!"#$%&'*+,./:;=?@^_`|~-`
- Délimiteurs de style crochets: `()` , `{}` , `[]` , `<>`

Lire Échapper en ligne: https://riptutorial.com/fr/regex/topic/4524/echapper

Chapitre 5: Frontière de mot

Syntaxe

- Style POSIX, fin du mot: `[[:>:]]`
- Style POSIX, début du mot: `[[:<:]]`
- Style POSIX, limite de mot: `[[:<:][:>:]]`
- SVR4 / GNU, fin du mot: `\>`
- SVR4 / GNU, début du mot: `\<`
- Perl / GNU, limite de mot: `\b`
- Tcl, fin de mot: `\M`
- Tcl, début du mot: `\m`
- Tcl, limite de mot: `\y`
- Portable ERE, début du mot: `(^|[^[:alnum:]_])`
- Portable ERE, fin du mot: `([^[:alnum:]_]|$)`

Remarques

Ressources additionnelles

- Chapitre POSIX sur les expressions régulières
- Documentation sur les expressions régulières Perl
- Tcl page de manuel re_syntax
- Expressions de barre oblique inversée GNU grep
- BSD re_format
- Plus de lecture

Examples

Match mot complet

```
\bfoo\b
```

correspondra au mot complet sans alphanumérique et _ précédant ou suivant par lui.

Prenant de regularexpression.info

> Il y a trois positions différentes qui qualifient de limites de mots:
>
> 1. Avant le premier caractère de la chaîne, si le premier caractère est un caractère de mot.
> 2. Après le dernier caractère de la chaîne, si le dernier caractère est un caractère de mot.
> 3. Entre deux caractères dans la chaîne, l'un est un caractère de mot et l'autre n'est

pas un caractère de mot.

Le terme *caractère de mot* signifie ici l'un des suivants

1. Alphabet (`[a-zA-Z]`)
2. Nombre (`[0-9]`)
3. Souligner _

En bref, *mot caractère* = `\w` = `[a-zA-Z0-9_]`

Rechercher des motifs au début ou à la fin d'un mot

Examinez les chaînes suivantes:

```
foobarfoo
bar
foobar
barfoo
```

- la `bar` expression régulière correspondra aux quatre chaînes,
- `\bbar\b` ne fera que correspondre à la 2ème,
- `bar\b` sera capable de faire correspondre les 2ème et 3ème chaînes, et
- `\bbar` correspondra aux 2ème et 4ème chaînes.

Limites de mots

Le métacaractère `\b`

Pour faciliter la recherche de mots entiers, nous pouvons utiliser le métacaractère `\b` . Il marque le **début** *et* la **fin** d'une séquence alphanumérique *. En outre, comme il ne sert qu'à marquer ces emplacements, il ne correspond en réalité à aucun caractère.

**: Il est courant d'appeler une séquence alphanumérique par un mot, puisque nous pouvons capturer ses caractères avec un `\w` (la classe des caractères du mot). Cela peut être trompeur, car `\w` inclut également des nombres et, dans la plupart des cas, le trait de soulignement.*

Exemples:

Regex	Contribution	Allumettes?
`\bstack\b`	`stackoverflow`	**Non** , car il n'y a pas d'occurrence de la `stack` mots **entière**
`\bstack\b`	`foo stack bar`	**Oui** , car il n'y a rien avant ou après la `stack`
`\bstack\b`	`stack!overflow`	**Oui,** il n'y a rien avant la `stack` et `!` n'est pas un caractère de mot

Regex	Contribution	Allumettes?
`\bstack`	`stackoverflow`	**Oui** , car il n'y a rien avant la `stack`
`overflow\b`	`stackoverflow`	**Oui** , car il n'y a rien après le `overflow`

Le métacaractère `\B`

Ceci est l'opposé de `\b` , correspondant à l'emplacement de chaque caractère non-frontière. Comme `\b` , puisqu'il correspond aux emplacements, il ne correspond à aucun caractère. C'est utile pour trouver *des* mots *non* entiers.

Exemples:

Regex	Contribution	Allumettes?
`\Bb\B`	`abc`	**Oui** , puisque `b` n'est pas entouré de limites de mots.
`\Ba\B`	`abc`	**Non** , `a` a une limite de mot sur son côté gauche.
`a\B`	`abc`	**Oui** , `a` n'a pas de limite de mot sur son côté droit.
`\B,\B`	`a,,,b`	**Oui** , il correspond à la deuxième virgule car `\B` correspondra également à l'espace entre deux caractères non-mots (il convient de noter qu'il existe une limite de mots à gauche de la première virgule et à droite du second).

Rendez le texte plus court mais ne brisez pas le dernier mot

Pour créer un texte long avec le maximum de N caractères mais laisser le dernier mot intact, utilisez le pattern `.{0,N}\b` :

```
^(.{0,N})\b.*
```

Lire Frontière de mot en ligne: https://riptutorial.com/fr/regex/topic/1539/frontiere-de-mot

Chapitre 6: Groupement Atomique

Introduction

Les groupes non capturés régulièrement permettent au moteur de ré-entrer dans le groupe et d'essayer de faire correspondre quelque chose de différent (comme une alternance différente ou de faire correspondre moins de caractères lorsqu'un quantificateur est utilisé).

Les groupes atomiques diffèrent des groupes non capturés réguliers en ce sens que le retour en arrière est interdit. Une fois que le groupe est sorti, toutes les informations de retour en arrière sont supprimées, donc aucune autre correspondance ne peut être tentée.

Remarques

Un quantificateur possessif se comporte comme un groupe atomique en ce sens que le moteur ne pourra pas revenir en arrière sur un jeton ou un groupe.

Les fonctionnalités suivantes sont équivalentes, bien que certaines soient plus rapides que d'autres:

```
a*+abc
(?>a*)abc
(?:a+)*+abc
(?:a)*+abc
(?:a*)*+abc
(?:a*)++abc
```

Examples

Grouper avec (?>)

Utiliser un groupe atomique

Les groupes atomiques ont le format `(?>...)` avec un `?>` Après le paren ouvert.

Considérez l'exemple de texte suivant:

```
ABC
```

Le regex va tenter de faire correspondre à partir de la position 0 du texte, qui est avant le `A` dans `ABC` .

Si une expression insensible à la casse `(?>a*)abc` était utilisée, le `(?>a*)` correspondrait à 1 caractère `A` , laissant

```
BC
```

comme le reste du texte correspondant. Le groupe `(?>a*)` est quitté et `abc` est tenté sur le texte restant, qui ne correspond pas.

Le moteur ne peut pas revenir en arrière dans le groupe atomique et la passe en cours échoue. Le moteur passe à la position suivante dans le texte, qui serait à la position 1, qui est après le `A` et avant le `B` de `ABC` .

Le regex `(?>a*)abc` est tenté à nouveau, et `(?>a*)` correspond à `A` 0 fois, laissant

```
BC
```

comme le reste du texte correspondant. Le groupe `(?>a*)` est quitté et `abc` est tenté, ce qui échoue.

Encore une fois, le moteur ne peut pas revenir en arrière dans le groupe atomique et la passe en cours échoue. Le regex continuera d'échouer jusqu'à ce que toutes les positions dans le texte aient été épuisées.

Utilisation d'un groupe non atomique

Les groupes réguliers non capturants ont le format `(?:...)` avec un `?:` Après le paren ouvert.

Étant donné le même exemple de texte, mais avec l'expression insensible à la casse `(?:a*)abc` place, une correspondance aurait lieu car le retour en arrière est autorisé à se produire.

Au début, `(?:a*)` consommera la lettre `A` dans le texte

```
ABC
```

en quittant

```
BC
```

comme le reste du texte correspondant. Le groupe `(?:a*)` est quitté et `abc` est tenté sur le texte restant qui ne correspond pas.

Le moteur revient en arrière dans le groupe `(?:a*)` et tente de faire correspondre 1 caractère de moins: au lieu de faire correspondre le caractère 1 `A` , il tente de faire correspondre les caractères 0 `A` et le groupe `(?:a*)` est quitté. Cela laisse

```
ABC
```

comme le reste du texte correspondant. Le regex `abc` est maintenant capable de correspondre avec succès au texte restant.

Autre exemple de texte

Considérez cet exemple de texte, avec à la fois des groupes atomiques et non atomiques (encore une fois, insensible à la casse):

```
AAAABC
```

Le regex tentera de faire correspondre à partir de la position 0 du texte, qui est avant le premier `A` de `AAAABC` .

Le motif utilisant le groupe atomique `(?>a*)abc` ne pourra **pas** correspondre, se comportant presque de façon identique à l'exemple `ABC` ci-dessus: tous les 4 caractères `A` sont d'abord associés à `(?>a*)` (laissant `BC` comme reste du texte à faire correspondre) et `abc` ne peut pas correspondre à ce texte. Le groupe **ne** peut pas être ré-entré, la correspondance échoue.

Le pattern utilisant le groupe non-atomique `(?:a*)abc` **pourra** correspondre, se comportant de la même façon que l'exemple `ABC` non-atomique ci-dessus: tous les 4 caractères `A` sont d'abord associés à `(?:a*) BC` tant que texte restant à faire correspondre) et `abc` ne peut pas correspondre à ce texte. Le groupe **est** capable d'être rentré, donc une moins `A` est tentée: 3 `A` caractères sont mis en correspondance à la place de 4 (en laissant `ABC` comme le reste du texte pour correspondre), et `abc` est en mesure de correspondre avec succès sur ce texte.

Lire Groupement Atomique en ligne: https://riptutorial.com/fr/regex/topic/8770/groupement-atomique

Chapitre 7: Groupes de capture

Examples

Groupes de capture de base

Un *groupe* est une section d'une expression régulière entre parenthèses `()` . Ceci est communément appelé "sous-expression" et répond à deux objectifs:

- Il rend la sous-expression atomique, c'est-à-dire qu'elle correspondra, échouera ou se répètera dans son ensemble.
- La partie du texte correspondante est accessible dans le reste de l'expression et dans le reste du programme.

Les groupes sont numérotés dans les moteurs de regex, en commençant par 1. Traditionnellement, le nombre maximal de groupes est de 9, mais de nombreuses variantes de regex modernes prennent en charge des nombres de groupes plus élevés. Le groupe 0 correspond toujours au modèle entier, de la même manière que l'entière expression entière entre parenthèses.

Le nombre ordinal augmente à chaque parenthèse d'ouverture, que les groupes soient placés l'un après l'autre ou imbriqués:

```
foo(bar(baz)?) (qux)+|(bla)
   1   2       3      4
```

les groupes et leurs nombres

Après qu'une expression ait atteint une correspondance globale, tous ses groupes seront utilisés - qu'un groupe particulier ait réussi ou non à correspondre.

Un groupe peut être facultatif, comme `(baz)?` ci-dessus, ou dans une autre partie de l'expression qui n'a pas été utilisée pour la correspondance, comme `(bla)` ci-dessus. Dans ces cas, les groupes qui ne correspondent pas ne contiennent tout simplement aucune information.

Si un quantificateur est placé derrière un groupe, comme dans `(qux)+` ci-dessus, le nombre total de groupes de l'expression reste le même. Si un groupe correspond à plusieurs fois, son contenu sera la dernière occurrence de correspondance. Cependant, les saveurs regex modernes permettent d'accéder à toutes les occurrences de sous-correspondance.

Si vous souhaitez récupérer la date et le niveau d'erreur d'une entrée de journal comme celle-ci:

```
2012-06-06 12:12.014 ERROR: Failed to connect to remote end
```

Vous pourriez utiliser quelque chose comme ceci:

```
^(\d{4}-\d{2}-\d{2}) \d{2}:\d{2}.\d{3} (\w*): .*$
```

Cela extraira la date de l'entrée de journal `2012-06-06` tant que groupe de capture 1 et le niveau d' `ERROR` tant que groupe de capture 2.

Références et groupes non capturés

Étant donné que les groupes sont "numérotés", certains moteurs prennent également en charge la correspondance avec ce qu'un groupe a précédemment mis en correspondance.

En supposant que vous vouliez correspondre à quelque chose où deux chaînes de longueur égale trois sont divisées par un `$` vous utiliseriez:

```
(.{3})\$\1
```

Cela correspondrait à l'une des chaînes suivantes:

```
"abc$abc"
"a b$a b"
"af $af "
"   $   "
```

Si vous souhaitez qu'un groupe ne soit pas numéroté par le moteur, vous pouvez le déclarer non capturé. Un groupe non capturant ressemble à ceci:

```
(?:)
```

Ils sont particulièrement utiles pour répéter un certain nombre de fois, car un groupe peut également être utilisé comme "atome". Considérer:

```
(\d{4}(?:-\d{2}){2} \d{2}:\d{2}.\d{3}) (.*)[\r\n]+\1 \2
```

Cela correspondra à deux entrées de journalisation dans les lignes adjacentes ayant le même horodatage et la même entrée.

Groupes de capture nommés

Certaines variantes d'expression rationnelle permettent des *groupes de capture nommés* . Au lieu d'un index numérique, vous pouvez vous référer à ces groupes par leur nom dans le code suivant, c'est-à-dire dans les backreferences, dans le pattern replace et dans les lignes suivantes du programme.

Les index numériques changent au fur et à mesure que le nombre ou la disposition des groupes dans une expression change, de sorte qu'ils sont plus fragiles en comparaison.

Par exemple, pour faire correspondre un mot (`\w+`) entre guillemets simples ou doubles (`['"]`), nous pourrions utiliser:

```
(?<quote>['"])\w+\k{quote}
```

Ce qui équivaut à:

```
(['"])\w+\1
```

Dans une situation simple comme celle-ci, un groupe de capture numéroté régulier ne présente aucun inconvénient.

Dans des situations plus complexes, l'utilisation de groupes nommés rendra la structure de l'expression plus évidente pour le lecteur, ce qui améliorera la maintenabilité.

L'analyse de fichier journal est un exemple d'une situation plus complexe qui tire parti des noms de groupe. Voici le format de journal commun Apache (CLF):

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
```

L'expression suivante capture les parties dans des groupes nommés:

```
(?<ip>\S+) (?<logname>\S+) (?<user>\S+) (?<time>\[[^]]+\]) (?<request>"[^"]+") (?<status>\S+)
(?<bytes>\S+)
```

La syntaxe dépend de la saveur, les plus courantes sont:

- `(?<name>...)`
- `(?'name'...)`
- `(?P<name>...)`

Références:

- `\k<name>`
- `\k{name}`
- `\k'name'`
- `\g{name}`
- `(?P=name)`

Dans l'arôme .NET, plusieurs groupes peuvent partager le même nom, ils utiliseront des piles de capture .

Dans PCRE, vous devez l'activer explicitement en utilisant le modificateur `(?J)` (`PCRE_DUPNAMES`) ou en utilisant le groupe de réinitialisation de branche `(?|)` . Seule la dernière valeur capturée sera accessible.

```
(?J)(?<a>...)(?<a>...)
(?|(?<a>...)|(?<a>...))
```

Lire Groupes de capture en ligne: https://riptutorial.com/fr/regex/topic/660/groupes-de-capture

Chapitre 8: Groupes de capture nommés

Syntaxe

- Créez un groupe de capture nommé (`x` étant le modèle que vous souhaitez capturer):

  (? 'nom'X) (? X) (? PX)

- Référencez un groupe de capture nommé:

  $ {nom} \ {nom} g \ {nom}

Remarques

Python et Java n'autorisent pas plusieurs groupes à utiliser le même nom.

Examples

À quoi ressemble un groupe de capture nommé

Compte tenu des saveurs, le groupe de capture nommé peut ressembler à ceci:

```
(?'name'X)
(?<name>X)
(?P<name>X)
```

Avec `x` étant le motif que vous souhaitez capturer. Considérons la chaîne suivante:

> Il était une fois une *jolie petite fille* ...
>
> Il était une fois une *licorne avec un chapeau* ...
>
> Il était une fois un *bateau avec un drapeau de pirate* ...

Dans lequel je veux capturer le sujet (en *italique)* de chaque ligne. J'utiliserai l'expression suivante `.* was a (?<subject>[\w ]+)[.]{3}.`

Le résultat correspondant tiendra:

```
MATCH 1
subject    [29-47]    `pretty little girl`
MATCH 2
subject    [80-99]    `unicorn with an hat`
MATCH 3
subject    [132-155]    `boat with a pirate flag`
```

Référence à un groupe de capture nommé

Comme vous le savez (ou non), vous pouvez référencer un groupe de capture avec:

```
$1
```

`1` étant le numéro de groupe.

De la même manière, vous pouvez référencer un groupe de capture nommé avec:

```
${name}
\{name}
g\{name}
```

Prenons l'exemple précédent et remplaçons les correspondances avec

```
The hero of the story is a ${subject}.
```

Le résultat que nous obtiendrons est:

```
The hero of the story is a pretty little girl.
The hero of the story is a unicorn with an hat.
The hero of the story is a boat with a pirate flag.
```

Lire Groupes de capture nommés en ligne: https://riptutorial.com/fr/regex/topic/744/groupes-de-capture-nommes

Chapitre 9: Lookahead et Lookbehind

Syntaxe

- **Regard positif:** `(?=pattern)`
- **Lookahead négatif:** `(?!pattern)`
- **Lookbehind positif** : `(?<=pattern)`
- **Lookbehind négatif** : `(?<!pattern)`

Remarques

Non pris en charge par tous les moteurs regex.

En outre, de nombreux moteurs de regex limitent les modèles à l'intérieur des styles à des chaînes de longueur fixe. Par exemple, le modèle `(?<=a+)b` doit correspondre au `b` dans `aaab` mais génère une erreur dans Python.

Les groupes de capture sont autorisés et fonctionnent comme prévu, y compris les références arrière. Le lookahead / lookbehind lui-même n'est pas un groupe de capture, cependant.

Examples

Les bases

Un **lookahead positif** `(?=123)` affirme que le texte est suivi par le motif donné, sans inclure le motif dans la correspondance. De même, un **lookbehind positif** `(?<=123)` affirme que le texte est précédé du motif donné. Remplacer le `=` avec `!` nie l'assertion.

Entrée : `123456`

- `123(?=456)` correspond à `123` (*résultat positif*)
- `(?<=123)456` correspond à `456` (*lookbehind positif*)
- `123(?!456)` échoue (*tête de lecture négative*)
- `(?<!123)456` échoue (*lookbehind négatif*)

Entrée : `456`

- `123(?=456)` échoue
- `(?<=123)456` échoue
- `123(?!456)` échoue
- `(?<!123)456` correspondances `456`

Utiliser lookbehind pour tester les fins

Un lookbehind peut être utilisé à la fin d'un pattern pour s'assurer qu'il se termine ou non d'une certaine manière.

`([az ]+|[AZ ]+)(?<! )` correspond uniquement aux séquences de mots minuscules ou uniquement de mots en majuscule tout en excluant les espaces de fin.

Simulation d'une longueur variable avec \ K

Certaines versions de regex (Perl, PCRE, Oniguruma, Boost) ne supportent que les lookbehinds de longueur fixe, mais offrent la fonctionnalité `\K` , qui peut être utilisée pour simuler une apparence de longueur variable au début d'un motif. En rencontrant un `\K` , le texte correspondant jusqu'à ce point est ignoré et seul le texte correspondant à la partie du motif *suivant* `\K` est conservé dans le résultat final.

```
ab+\Kc
```

Est équivalent à:

```
(?<=ab+)c
```

En général, un motif de la forme:

```
(subpattern A)\K(subpattern B)
```

Finit par être similaire à:

```
(?<=subpattern A)(subpattern B)
```

Sauf si le sous-modèle B peut correspondre au même texte que le sous-modèle A - vous pourriez vous retrouver avec des résultats subtilement différents, car le sous-modèle A consomme toujours le texte, contrairement à un vrai lookback.

Lire Lookahead et Lookbehind en ligne: https://riptutorial.com/fr/regex/topic/639/lookahead-et-lookbehind

Chapitre 10: Lorsque vous ne devez PAS utiliser les expressions régulières

Remarques

Les expressions régulières étant limitées à une grammaire régulière ou à une grammaire sans contexte, il existe de nombreuses utilisations abusives des expressions régulières. Donc, dans cette rubrique, il y a quelques exemples où vous ne devriez *PAS* utiliser des expressions régulières, mais plutôt utiliser votre langue préférée.

> *Certaines personnes, confrontées à un problème, pensent:*
> *"Je sais, je vais utiliser des expressions régulières."*
> *Maintenant, ils ont deux problèmes.*
> - Jamie Zawinski

Examples

Les paires correspondantes (comme les parenthèses, les parenthèses…)

Certains moteurs de regex (tels que .NET) peuvent gérer des expressions sans contexte, et vont les résoudre. Mais ce n'est pas le cas pour la plupart des moteurs standard. Et même s'ils le font, vous finirez par avoir une expression complexe difficile à lire, tandis que l'utilisation d'une bibliothèque d'analyse syntaxique pourrait vous faciliter la tâche.

- Comment trouver toutes les correspondances regex possibles en python?

Opérations de chaîne simples

Les expressions régulières pouvant faire beaucoup, il est tentant de les utiliser pour les opérations les plus simples. Mais utiliser un moteur de regex a un coût en mémoire et en utilisation de processeur: vous devez compiler l'expression, stocker l'automate en mémoire, l'initialiser puis le nourrir avec la chaîne pour l'exécuter.

Et il y a beaucoup de cas où il n'est tout simplement pas nécessaire de l'utiliser! Quelle que soit votre langue de prédilection, elle dispose toujours des outils de manipulation de chaînes de base. Donc, en règle générale, lorsqu'un outil permet d'effectuer une action dans votre bibliothèque standard, utilisez cet outil, et non une expression régulière:

- fendre une corde?

Par exemple, l'extrait suivant fonctionne en Python, Ruby et Javascript:

```
'foo.bar'.split('.')
```

Ce qui est plus facile à lire et à comprendre, et beaucoup plus efficace que l'expression régulière (en quelque sorte) équivalente:

```
(\w+)\.(\w+)
```

- Supprimer les espaces de fin?

La même chose s'applique aux espaces de fuite!

```
'foobar     '.strip() # python or ruby
'foobar     '.trim() // javascript
```

Ce qui serait équivalent à l'expression suivante:

```
([^\n]*)\s*$ # keeping \1 in the substitution
```

Analyse HTML (ou XML, JSON ou C, ou…)

Si vous souhaitez extraire quelque chose d'une page Web (ou de tout langage de représentation / programmation), une regex est le mauvais outil pour la tâche. Vous devriez plutôt utiliser les bibliothèques de votre langue pour accomplir la tâche.

Si vous voulez lire du code HTML, XML ou JSON, utilisez simplement la bibliothèque qui l'analyse correctement et l'utilise comme objet utilisable dans votre langue préférée! Vous vous retrouverez avec du code lisible et plus maintenable, et vous ne vous retrouverez pas

- RegEx correspondent à des balises ouvertes à l'exception des balises autonomes XHTML
- Analyse Python HTML à l'aide d'expressions régulières
- existe-t-il une regex pour générer tous les entiers pour un certain langage de programmation

Lire Lorsque vous ne devez PAS utiliser les expressions régulières en ligne: https://riptutorial.com/fr/regex/topic/4527/lorsque-vous-ne-devez-pas-utiliser-les-expressions-regulieres

Chapitre 11: Match Reset: \ K

Remarques

Regex101 définit la fonctionnalité \ K comme:

> `\K` réinitialise le point de départ de la correspondance signalée. Tous les personnages précédemment consommés ne sont plus inclus dans le match final

La séquence d'échappement `\K` est supportée par plusieurs moteurs, langages ou outils, tels que:

- boost (depuis ???)
- grep -P ← *utilise PCRE*
- Oniguruma (depuis 5.13.3)
- PCRE (depuis 7.2)
- Perl (depuis 5.10.0)
- PHP (depuis 5.2.4)
- Ruby (depuis 2.0.0)

... et (jusqu'à présent) non pris en charge par:

- .NET
- awk
- frapper
- GNOU
- ICU
- Java
- Javascript
- Bloc-notes ++
- Objectif c
- POSIX
- Python
- Qt / QRegExp
- sed
- Tcl
- vim
- XML
- XPath

Examples

Rechercher et remplacer en utilisant l'opérateur \ K

Vu le texte:

foo: bar

Je voudrais remplacer tout ce qui suit "foo:" par "baz", mais je veux garder "foo:". Cela pourrait être fait avec un groupe de capture comme celui-ci:

```
s/(foo: ).*/$1baz/
```

Qui résulte dans le texte:

foo: baz

Exemple 1

ou nous pourrions utiliser `\K` , qui "oublie" tout ce qu'il a précédemment trouvé, avec un motif comme celui-ci:

```
s/foo: \K.*/baz/
```

Le regex correspond à "foo:" et rencontre ensuite le `\K` , les caractères de correspondance précédents sont pris pour acquis et laissés par le regex, ce qui signifie que seule la chaîne correspondant à `.*` Sera remplacée par "baz", résultant dans le texte:

foo: baz

Exemple 2

Lire Match Reset: \ K en ligne: https://riptutorial.com/fr/regex/topic/1338/match-reset----k

Chapitre 12: Matchers UTF-8: Lettres, Marques, Ponctuation etc.

Examples

Correspondance des lettres dans différents alphabets

Les exemples ci-dessous sont donnés en Ruby, mais les mêmes appariements devraient être disponibles dans toutes les langues modernes.

Disons que nous avons la chaîne `"A𐐨Naïve"` , produite par Messy Artificial Intelligence. Il est composé de lettres, mais `\w` matcher générique ne correspondra pas beaucoup:

```
▶ "A𐐨Naïve"[/\w+/]
#⇒ "A"
```

La manière correcte de faire correspondre une lettre Unicode avec des marques combinées consiste à utiliser `\X` pour spécifier un cluster grapheme. Il y a une mise en garde pour Ruby, cependant. Onigmo, le moteur de regex pour Ruby, utilise toujours l'ancienne définition d'un cluster grapheme . Il n'a pas encore été mis à jour dans Extended Grapheme Cluster, comme défini dans l' Annexe 29 de la norme Unicode .

Donc, pour Ruby, nous pourrions avoir une solution de contournement: `\p{L}` ira presque bien, sauf que cela échoue sur l'accent diacritique combiné sur `i` :

```
▶ "A𐐨Naïve"[/\p{L}+/]
#⇒ "A𐐨Nai"
```

En ajoutant les «symboles de marque» à l'expression, nous pouvons enfin tout faire correspondre:

```
▶ "A𐐨Naïve"[/[\p{L}\p{M}]+/]
#⇒ "A𐐨Naïve"
```

Lire Matchers UTF-8: Lettres, Marques, Ponctuation etc. en ligne: https://riptutorial.com/fr/regex/topic/1527/matchers-utf-8--lettres--marques--ponctuation-etc-

Chapitre 13: Modèles simples assortis

Examples

Faire correspondre un caractère à un chiffre avec [0-9] ou \ d (Java)

`[0-9]` et `\d` sont des modèles équivalents (à moins que votre moteur Regex ne soit unicode et que `\d` corresponde également à des choses comme ②). Ils correspondent tous deux à un caractère à un seul chiffre afin que vous puissiez utiliser la notation que vous trouvez plus lisible.

Créez une chaîne du motif que vous souhaitez faire correspondre. Si vous utilisez la notation \ d, vous devrez ajouter une deuxième barre oblique inverse pour échapper à la première barre oblique inverse.

```
String pattern = "\\d";
```

Créez un objet Pattern. Transmettez la chaîne de modèle dans la méthode compile ().

```
Pattern p = Pattern.compile(pattern);
```

Créez un objet Matcher. Passez la chaîne que vous cherchez à trouver le modèle dans la méthode matcher (). Vérifiez si le motif est trouvé.

```
Matcher m1 = p.matcher("0");
m1.matches(); //will return true

Matcher m2 = p.matcher("5");
m2.matches(); //will return true

Matcher m3 = p.matcher("12345");
m3.matches(); //will return false since your pattern is only for a single integer
```

Matching différents numéros

`[ab]` où a et b sont des chiffres compris entre `0` et `9`

```
[3-7] will match a single digit in the range 3 to 7.
```

Faire correspondre plusieurs chiffres

```
\d\d       will match 2 consecutive digits
\d+        will match 1 or more consecutive digits
\d*        will match 0 or more consecutive digits
\d{3}      will match 3 consecutive digits
\d{3,6}    will match 3 to 6 consecutive digits
\d{3,}     will match 3 or more consecutive digits
```

Le `\d` dans les exemples ci-dessus peut être remplacé par une plage de numéros:

```
[3-7][3-7]    will match 2 consecutive digits that are in the range 3 to 7
[3-7]+        will match 1 or more consecutive digits that are in the range 3 to 7
[3-7]*        will match 0 or more consecutive digits that are in the range 3 to 7
[3-7]{3}      will match 3 consecutive digits that are in the range 3 to 7
[3-7]{3,6}    will match 3 to 6 consecutive digits that are in the range 3 to 7
[3-7]{3,}     will match 3 or more consecutive digits that are in the range 3 to 7
```

Vous pouvez également sélectionner des chiffres spécifiques:

```
[13579]       will only match "odd" digits
[02468]       will only match "even" digits
1|3|5|7|9     another way of matching "odd" digits - the | symbol means OR
```

Numéros correspondants dans des plages contenant plus d'un chiffre:

```
\d|10        matches 0 to 10    single digit OR 10.  The | symbol means OR
[1-9]|10     matches 1 to 10    digit in range 1 to 9 OR 10
[1-9]|1[0-5] matches 1 to 15    digit in range 1 to 9 OR 1 followed by digit 1 to 5
\d{1,2}|100  matches 0 to 100   one to two digits OR 100
```

Numéros correspondants qui divisent par d'autres numéros:

```
\d*0         matches any number that divides by 10  - any number ending in 0
\d*00        matches any number that divides by 100 - any number ending in 00
\d*[05]      matches any number that divides by 5   - any number ending in 0 or 5
\d*[02468]   matches any number that divides by 2   - any number ending in 0,2,4,6 or 8
```

faire correspondre les nombres qui divisent par 4 - tout nombre égal à 0, 4 ou 8 ou se terminant par 00, 04, 08, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60 , 64, 68, 72, 76, 80, 84, 88, 92 ou 96

```
[048]|\d*(00|04|08|12|16|20|24|28|32|36|40|44|48|52|56|60|64|68|72|76|80|84|88|92|96)
```

Cela peut être raccourci. Par exemple, au lieu d'utiliser `20|24|28` nous pouvons utiliser `2[048]` . De même, comme les années 40, 60 et 80 ont le même schéma, nous pouvons les inclure: `[02468][048]` et les autres ont aussi un motif `[13579][26]` . Donc, toute la séquence peut être réduite à:

```
[048]|\d*([02468][048]|[13579][26])    - numbers divisible by 4
```

Les nombres correspondants qui n'ont pas de motif comme ceux divisibles par 2,4,5,10 etc. ne peuvent pas toujours être utilisés de manière succincte et vous devez généralement recourir à une série de chiffres. Par exemple, faire correspondre tous les nombres qui se divisent par 7 dans la plage de 1 à 50 peut être fait simplement en énumérant tous ces nombres:

```
7|14|21|28|35|42|49

or you could do it this way
```

```
7|14|2[18]|35|4[29]
```

Correspondant aux espaces de début / fin

Espaces de fuite

`\s*$` : cela correspond à tout espace (`*`) (`\s`) à la fin (`$`) du texte

Espaces principaux

`^\s*` : Cela correspondra à tout espace (`*`) (`\s`) au début (`^`) du texte

Remarques

`\s` est un métacaractère commun à plusieurs moteurs RegExp, et est destiné à capturer des caractères d'espaces (espaces, nouvelles lignes et onglets par exemple). **Remarque** : il *ne* capturera probablement *pas* tous les caractères d'espace Unicode . Vérifiez la documentation de vos moteurs pour en être sûr.

Correspond à n'importe quel flotteur

```
[\+\-]?\d+(\.\d*)?
```

Cela correspond à tout flottant signé, si vous ne voulez pas de signes ou si vous analysez une équation, supprimez `[\+\-]?` vous avez donc `\d+(\.\d+)?`

Explication:

- `\d+` correspond à tout nombre entier
- `()?` signifie que le contenu des parenthèses est facultatif mais doit toujours apparaître ensemble
- '\.' allumettes "., nous devons y échapper depuis". correspond normalement à n'importe quel caractère

Donc cette expression correspondra

```
5
+5
-5
5.5
+5.5
-5.5
```

Sélection d'une certaine ligne dans une liste basée sur un mot à un certain

endroit

J'ai la liste suivante:

```
1. Alon Cohen
2. Elad Yaron
3. Yaron Amrani
4. Yogev Yaron
```

Je veux choisir le prénom des gars avec le nom de famille Yaron.

Comme je ne me soucie pas du nombre, je le mets juste comme n'importe quel chiffre et un point et un espace correspondants après le début de la ligne, comme ceci: `^[\d]+\.\s` .

Maintenant, nous devons faire correspondre l'espace et le prénom, car nous ne pouvons pas dire s'il s'agit de lettres majuscules ou minuscules, nous allons simplement faire correspondre les deux: `[a-zA-Z]+\s` ou `[aZ]+\s` et peut aussi être `[\w]+\s` .

Maintenant, nous allons spécifier le nom de famille requis pour obtenir uniquement les lignes contenant Yaron comme nom de famille (à la fin de la ligne): `\sYaron$` .

Tout cela ensemble `^[\d]+\.\s[\w]+\sYaron$` .

Exemple en direct: https://regex101.com/r/nW4fH8/1

Lire Modèles simples assortis en ligne: https://riptutorial.com/fr/regex/topic/343/modeles-simples-assortis

Chapitre 14: Modificateurs de regex (flags)

Introduction

Les modèles d'expression régulière sont souvent utilisés avec des *modificateurs* (également appelés *flags*) qui redéfinissent le comportement des regex. Les modificateurs de regex peuvent être *réguliers* (par exemple `/abc/i`) et *inline* (ou *incorporés*) (par exemple `(?i)abc`). Les modificateurs les plus courants sont les modificateurs globaux, insensibles à la casse, multilignes et dotall. Cependant, les versions de regex diffèrent par le nombre de modificateurs de regex pris en charge et leurs types.

Remarques

Modificateurs PCRE

Modificateur	En ligne	La description
PCRE_CASELESS	(?je)	Correspondance insensible à la casse
PCRE_MULTILINE	(? m)	Correspondance de lignes multiples
PCRE_DOTALL	(? s)	. correspond à de nouvelles lignes
PCRE_ANCHORED	(?UNE)	Meta-character `^` correspond uniquement au début
PCRE_EXTENDED	(?X)	Les espaces blancs sont ignorés
PCRE_DOLLAR_ENDONLY	n / a	Meta-character `$` correspond seulement à la fin
PCRE_EXTRA	(?X)	Analyse stricte de l'évasion
PCRE_UTF8		Gère les `UTF-8`
PCRE_UTF16		Gère les `UTF-16`
PCRE_UTF32		Gère les `UTF-32`
PCRE_UNGREEDY	(? U)	Définit le moteur pour une correspondance lente
PCRE_NO_AUTO_CAPTURE	(? :)	Désactive les groupes de capture automatique

Modificateurs Java

Modifier (`Pattern.###`)	Valeur	La description
UNIX_LINES	1	Active le mode lignes Unix .
CASE_INSENSITIVE	2	Active la correspondance insensible à la casse.
COMMENTAIRES	4	Autorise les espaces et les commentaires dans un motif.
MULTILINE	8	Active le mode multiligne.
LITTÉRAL	16	Permet l'analyse littérale du motif.
DOTALL	32	Active le mode dotall.
UNICODE_CASE	64	Active le pliage de casse compatible Unicode.
CANON_EQ	128	Permet l'équivalence canonique.
UNICODE_CHARACTER_CLASS	256	Active la version Unicode des classes de caractères prédéfinies et des classes de caractères POSIX.

Examples

Modificateur DOTALL

Un modèle de regex où un modificateur DOTALL (dans la plupart des expressions de regex exprimées avec `s`) modifie le comportement de `.` lui permettant de correspondre à un symbole de nouvelle ligne (LF):

```
/cat (.*?) dog/s
```

Cette regex de style Perl correspondra à une chaîne telle que `"cat fled from\na dog"` capturant `"fled from\na"` vers le groupe 1.

Une version en ligne: `(?s)` (par exemple `(?s)cat (.*?) dog`)

Note : En Ruby, l'équivalent du modificateur DOTALL est `m` , le modificateur `Regexp::MULTILINE` (par exemple `/a.*b/m`).

Remarque : JavaScript ne fournit pas de modificateur DOTALL, donc a `.` ne peut jamais être autorisé à correspondre à un caractère de nouvelle ligne. Pour obtenir le même effet, une solution de contournement est nécessaire, par exemple en remplaçant tout le `.` s avec une classe de

caractère catch-all comme `[\S\s]` , ou une classe de caractère *non rien* `[^]` (cependant, cette construction sera traitée comme une erreur par tous les autres moteurs et n'est donc pas portable).

Modificateur MULTILINE

Un autre exemple est un modificateur MULTILINE (généralement exprimé avec `m` flag (pas dans Oniguruma (par exemple Ruby) qui utilise `m` pour désigner un modificateur DOTALL)) qui fait que les ancres `^` et `$` correspondent au début / à la fin d'une *ligne* , pas au début / à la fin de la chaîne entière.

```
/^My Line \d+$/gm
```

trouvera toutes les *lignes* commençant par `My Line` , puis contiendra un espace et 1+ chiffres jusqu'à la fin de la ligne.

Une version en ligne: `(?m)` (par exemple `(?m)^My Line \d+$`)

NOTE : Dans Oniguruma (par exemple en Ruby), et aussi dans presque tous les éditeurs de texte prenant en charge les expressions rationnelles, les ancres `^` et `$` indiquent les positions de début / fin de *ligne par défaut* . Vous devez utiliser `\A` pour définir le document / la chaîne de début et `\z` pour indiquer la fin du document / de la chaîne. La différence entre `\Z` et `\z` est que le premier peut correspondre au symbole de nouvelle ligne (LF) à la fin de la chaîne (par exemple `/\Astring\Z/` trouvera une correspondance dans `"string\n"`) (sauf Python, où le comportement de `\Z` est égal à `\z` et `\z` anchor n'est pas pris en charge).

IGNORE CASE modificateur

Le modificateur commun pour ignorer la casse est `i` :

```
/fog/i
```

va correspondre à `Fog` , `foG` , etc.

La version en ligne du modificateur ressemble à `(?i)` .

Remarques:

En Java, par défaut, la *correspondance insensible à la casse suppose que seuls les caractères du jeu de caractères US-ASCII sont mis en correspondance. La correspondance insensible à la casse compatible Unicode peut être activée en spécifiant l'indicateur* `UNICODE_CASE` *conjointement avec cet* `CASE_INSENSITIVE` (`CASE_INSENSITIVE`) . (par ex. `Pattern p = Pattern.compile("YOUR_REGEX", Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);`). Vous trouverez plus d'informations à ce sujet *dans la correspondance insensible* à la *casse dans Java RegEx* . De même, `UNICODE_CHARACTER_CLASS` peut être utilisé pour rendre la correspondance Unicode compatible.

Modificateur VERBOSE / COMMENT / IgnorePatternWhitespace

Le modificateur qui permet d'utiliser des espaces à l'intérieur de certaines parties du motif pour le formater pour une meilleure lisibilité et pour permettre des commentaires commençant par `#` :

```
/(?x)^          # start of string
  (?=\D*\d)     # the string should contain at least 1 digit
  (?!\d+$)      # the string cannot consist of digits only
  \#            # the string starts with a hash symbol
  [a-zA-Z0-9]+ # the string should have 1 or more alphanumeric symbols
  $             # end of string
/
```

Exemple de chaîne: `#word1here` . Notez que le symbole `#` est échappé pour indiquer un littéral `#` qui fait partie d'un modèle.

L'espace blanc non échappé dans le modèle d'expression régulière est ignoré, y échappe pour en faire une partie du motif.

Généralement, les espaces à l'intérieur des classes de caractères (`[...]`) sont traités comme des espaces littéraux, sauf en Java.

En outre, il est important de mentionner que dans PCRE, .NET, Python, Ruby Oniguruma, ICU, Boost regex peut utiliser les commentaires `(?#:...)` dans le modèle regex.

Modificateur de capture explicite

Ceci est un modificateur spécifique à .ge regex exprimé avec `n` . Lorsqu'ils sont utilisés, les groupes non nommés (comme `(\d+)`) ne sont pas capturés. Seules les captures valides sont des groupes explicitement nommés (par exemple `(?<name> subexpression)`).

```
(?n)(\d+)-(\w+)-(?<id>\w+)
```

correspondra à l'ensemble `123-1_abc-00098` , mais `(\d+)` et `(\w+)` ne créeront pas de groupes dans l'objet de correspondance résultant. Le seul groupe sera `${id}` . Voir la démo .

Modificateur UNICODE

Le modificateur UNICODE, généralement exprimé sous la forme `u` (PHP, Python) ou `U` (Java), fait que le moteur regex traite le modèle et la chaîne d'entrée comme des chaînes et des motifs Unicode, ce qui rend les classes comme `\w` , `\d` , `\s` , etc. compatible Unicode.

```
/\A\p{L}+\z/u
```

est une regex PHP pour correspondre à des chaînes composées d'au moins une lettre Unicode. Voir la démo regex .

Notez qu'en **PHP** , le modificateur `/u` permet au moteur PCRE de gérer les chaînes en tant que chaînes UTF8 (en `PCRE_UTF8` verbe `PCRE_UTF8`) et de rendre les classes de caractères abrégées du modèle Unicode (en activant le verbe `PCRE_UCP` , voir plus sur *pcre.org*) .

> **Les chaînes de motif et de sujet sont traitées comme UTF-8.** Ce modificateur est disponible depuis PHP 4.1.0 ou supérieur sous Unix et depuis PHP 4.2.3 sous win32. La validité UTF-8 du modèle et du sujet est vérifiée depuis PHP 4.3.5. Un sujet invalide fera que la fonction preg_ * ne correspondra à rien; un motif invalide déclenchera une erreur de niveau E_WARNING. Les séquences UTF-8 de cinq et six octets sont considérées comme non valides depuis PHP 5.3.4 (resp. PCRE 7.3 2007-08-28); anciennement ceux-ci ont été considérés comme valables UTF-8.

Dans Python 2.x, le `re.UNICODE` n'affecte que le motif lui-même: *Faites en* `re.UNICODE` que `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` *et* `\S` *dépendent de la base de données des propriétés de caractère Unicode.*

Une version en ligne: `(?u)` en Python, `(?U)` en Java. Par exemple:

```
print(re.findall(ur"(?u)\w+", u"Dąb")) # [u'D\u0105b']
print(re.findall(r"\w+", u"Dąb"))      # [u'D', u'b']

System.out.println("Dąb".matches("(?U)\\w+")); // true
System.out.println("Dąb".matches("\\w+"));     // false
```

Modificateur PCRE_DOLLAR_ENDONLY

Le modificateur *PCRE_DOLLAR_ENDONLY* compatible *PCRE* qui établit la correspondance `$` anchor à la *toute fin de la chaîne* (en excluant la position avant le dernier changement de ligne dans la chaîne).

```
/^\d+$/D
```

est égal à

```
/^\d+\z/
```

et correspond à une chaîne entière composée de 1 chiffre ou plus et ne correspondra pas à `"123\n"` , mais correspondra à `"123"` .

Modificateur PCRE_ANCHORED

Un autre modificateur conforme à PCRE exprimé avec le modificateur `/A` Si ce modificateur est défini, le motif est obligé d'être "ancré", c'est-à-dire qu'il est contraint de ne correspondre qu'au début de la chaîne recherchée (la "chaîne sujet"). Cet effet peut également être obtenu par des constructions appropriées dans le modèle lui-même, ce qui est le seul moyen de le faire en Perl.

```
/man/A
```

est le même que

```
/^man/
```

Modificateur PCRE_UNGREEDY

L'indicateur PCRE_UNGREEDY conforme à PCRE exprimé avec `/U` Il fait basculer la gourmandise dans un pattern: `/a.*?b/U` = `/a.*b/` et vice versa.

Modificateur PCRE_INFO_JCHANGED

Un autre modificateur PCRE permettant l'utilisation de groupes nommés en double.

REMARQUE : seule la version en *ligne* est prise en charge - `(?J)` et doit être placée au début du modèle.

Si tu utilises

```
/(?J)\w+-(?:new-(?<val>\w+)|\d+-empty-(?<val>[^-]+)-collection)/
```

les valeurs du groupe "val" ne seront jamais vides (seront toujours définies). Un effet similaire peut être obtenu avec la réinitialisation de branche.

Modificateur PCRE_EXTRA

Un modificateur PCRE qui provoque une erreur si une barre oblique inverse dans un motif est suivie d'une lettre sans signification particulière. Par défaut, une barre oblique suivie d'une lettre sans signification particulière est traitée comme un littéral.

Par exemple

```
/big\y/
```

va correspondre à `bigy` , mais

```
/big\y/X
```

jettera une exception.

Version en ligne: `(?X)`

Lire Modificateurs de regex (flags) en ligne:
https://riptutorial.com/fr/regex/topic/5138/modificateurs-de-regex--flags-

Chapitre 15: Personnages d'ancre: Caret (^)

Remarques

Terminologie

Le caractère Caret (^) est également désigné par les termes suivants:

- chapeau
- contrôle
- flèche vers le haut
- chevron
- accent circonflexe

Usage

Il a deux utilisations dans les expressions régulières:

- Pour indiquer le début de la ligne
- Si elle est utilisée immédiatement après un crochet (`[^`], elle annule l'ensemble des caractères autorisés (c.-à-d. `[123]` signifie que le caractère 1, 2 ou 3 est autorisé, tandis que l'instruction `[^123]` signifie tout caractère autre que 1 , 2 ou 3 est autorisé.

Évasion de personnage

Pour exprimer un caret sans signification particulière, il faut l'éviter en le précédant par une barre oblique inverse; c'est à dire `\^` .

Examples

Début de ligne

Lorsque le modificateur multiligne `(?m)` est désactivé , `^` ne correspond qu'au début de la chaîne d'entrée:

Pour le regex

```
^He
```

Les chaînes d'entrée suivantes correspondent à:

- `Hedgehog\nFirst line\nLast line`
- `Help me, please`
- `He`

Et les chaînes d'entrée suivantes **ne** correspondent **pas** :

- `First line\nHedgehog\nLast line`
- `IHedgehog`
- `Hedgehog` (dû aux espaces blancs)

Lorsque plusieurs lignes `(?m)` modificateur est activé, `^` correspond au début de chaque ligne:

```
^He
```

Ce qui précède correspondrait à toute chaîne d'entrée contenant une ligne commençant par `He` .

Considérant `\n` comme le nouveau caractère de ligne, les lignes suivantes correspondent:

- `Hello`
- `First line\nHedgehog\nLast line` (deuxième ligne seulement)
- `My\nText\nIs\nHere` (dernière ligne seulement)

Et les chaînes d'entrée suivantes **ne** correspondent **pas** :

- `Camden Hells Brewery`
- `Helmet` (dû aux espaces blancs)

Faire correspondre les lignes vides en utilisant `^`

Un autre cas d'utilisation typique de caret est la correspondance des lignes vides (ou une chaîne vide si le modificateur multi-lignes est désactivé).

Afin de faire correspondre une ligne vide (multi-line **on**), un caret est utilisé à côté d'un `$` qui est un autre caractère d'ancrage représentant la position en fin de ligne (caractères d'ancre: Dollar ($)). Par conséquent, l'expression régulière suivante correspondra à une ligne vide:

```
 ^$
```

Echapper au personnage du caret

Si vous devez utiliser le caractère `^` dans une classe de caractères (classes de caractères), placez-le ailleurs que dans le début de la classe:

```
[12^3]
```

Ou échapper à la `^` utilisant une barre oblique inverse `\` :

```
[\^123]
```

Si vous souhaitez faire correspondre le caractère du caret lui-même en dehors d'une classe de personnage, vous devez y échapper:

```
\^
```

Cela évite que le `^` soit interprété comme le caractère d'ancrage représentant le début de la chaîne / ligne.

Comparaison du début de la ligne d'ancrage et du début de la chaîne d'ancrage

Alors que beaucoup de gens pensent que `^` signifie le début d'une chaîne, cela signifie en fait le début d'une ligne. Pour un début réel d'utilisation d'ancre de chaîne, `\A`

La chaîne `hello\nworld` (ou plus clairement)

```
hello
world
```

Serait égalé par les expressions régulières `^h` , `^w` et `\Ah` mais pas par `\Aw`

Modificateur multiligne

Par défaut, le caret `^` metacharacter correspond à la **position** avant le premier caractère de la chaîne.

Compte tenu de la chaîne " **charsequence** " appliquée aux modèles suivants: `/^char/` & `/^sequence/` , le moteur essaiera de faire correspondre les éléments suivants:

- `/^char/`
    - **^** - charsequence
    - **c** - `c` harsequence
    - **h** - `ch` arsequence
    - **a** - `cha` rsequence
    - **r** - `char` séquence

  Match trouvé

- `/^sequence/`
    - **^** - charsequence
    - **s** - charsequence

  Match non trouvé

Le même comportement sera appliqué même si la chaîne contient des *terminateurs de ligne* , tels que `\r?\n` Seule la position au début de la chaîne sera associée.

Par exemple:

```
/^/g
```

Harchar \ r \ n
\ r \ n
séquence

Cependant, si vous devez faire correspondre après chaque terminaison de ligne, vous devrez définir le mode **multiligne** (`//m` , `(?m)`) dans votre modèle. Ce faisant, le caret `^` correspondra « au début de chaque ligne », ce qui correspond à la position au début de la chaîne et les positions **immédiatement après** [1] Les terminaisons de ligne.

[1] Dans certaines versions (Java, PCRE, ...), `^` ne correspondra pas après le terminateur de ligne, si le terminateur de ligne est le dernier de la chaîne.

Par exemple:

```
/^/gm
```

Harchar \ r \ n
⋮ \ r \ n
Séquence

Certains des moteurs d'expression régulière prenant en charge le modificateur Multiline:

- Java

```
Pattern pattern = Pattern.compile("(?m)^abc");
Pattern pattern = Pattern.compile("^abc", Pattern.MULTILINE);
```

- .NET

```
var abcRegex = new Regex("(?m)^abc");
var abdRegex = new Regex("^abc", RegexOptions.Multiline)
```

- PCRE

```
/(?m)^abc/
/^abc/m
```

- Python 2 & 3 (module `re` intégré)

```
abc_regex = re.compile("(?m)^abc");
abc_regex = re.compile("^abc", re.MULTILINE);
```

Lire Personnages d'ancre: Caret (^) en ligne:
https://riptutorial.com/fr/regex/topic/452/personnages-d-ancre--caret----

Chapitre 16: Quantificateurs gourmands et paresseux

Paramètres

Quantificateurs	La description
`?`	Faites correspondre le caractère ou la sous-expression précédent 0 ou 1 fois (de préférence 1).
`*`	Faites correspondre le caractère précédent ou la sous-expression 0 ou plusieurs fois (autant que possible).
`+`	Faites correspondre le caractère précédent ou la sous-expression 1 ou plusieurs fois (autant que possible).
`{n}`	Faites correspondre le caractère ou la sous-expression précédent exactement *n* fois.
`{min,}`	Faites correspondre le caractère précédent ou la sous-expression *min* ou plusieurs fois (autant que possible).
`{0,max}`	Faites correspondre le caractère précédent ou la sous-expression *max* ou moins (le plus près possible de *max*).
`{min,max}`	Correspondre au caractère précédent ou sous - expression d' au moins *min* fois , mais pas plus de fois *maximum* (aussi près que possible *max).*
Quantificateurs paresseux	**La description**
`??`	Faites correspondre le caractère ou la sous-expression précédent 0 ou 1 fois (de préférence 0).
`*?`	Faites correspondre le caractère précédent ou la sous-expression 0 ou plusieurs fois (le moins possible).
`+?`	Faites correspondre le caractère précédent ou la sous-expression 1 fois ou plus (le moins possible).
`{n}?`	Faites correspondre le caractère ou la sous-expression précédent exactement *n* fois. Aucune différence entre les versions gourmandes et paresseuses.
`{min,}?`	Faites correspondre le caractère précédent ou la sous-expression *min*

Quantificateurs	La description
	ou plusieurs fois (aussi près que possible de *min*).
`{0,max}?`	Faites correspondre le caractère précédent ou la sous-expression *max* ou moins (le moins possible).
`{min,max}?`	Faites correspondre le caractère ou la sous-expression précédente avec les temps *minimum* , mais pas plus que les durées *maximales* (aussi proches que possible de *min*).

Remarques

Cupidité

Un quantificateur *gourmand* essaie toujours de répéter le sous-modèle autant de fois que possible avant d'explorer des correspondances plus courtes par retour en arrière.

Généralement, un motif gourmand correspondra à la plus longue chaîne possible.

Par défaut, tous les quantificateurs sont gourmands.

Paresse

Un quantificateur *paresseux* (également appelé *non-gourmand* ou *réticent*) tente toujours de répéter le sous-modèle aussi *peu de* fois que possible, avant d'explorer les correspondances plus longues par expansion.

Généralement, un modèle paresseux correspond à la chaîne la plus courte possible.

Pour rendre les quantificateurs paresseux, ajoutez-les simplement `?` au quantificateur existant, par exemple `+?` , `{0,5}?` .

Le concept de gourmandise et de paresse n'existe que dans les moteurs de retour en arrière

La notion de quantificateur gourmand / paresseux n'existe que dans le backtracking des moteurs de regex. Dans les moteurs de regex non-backtracking ou les moteurs regex conformes à POSIX, les quantifiers spécifient uniquement la limite supérieure et la limite inférieure de la répétition, sans spécifier comment trouver la correspondance - ces moteurs correspondront toujours à la plus longue chaîne de gauche.

Examples

La cupidité et la paresse

Compte tenu de l'entrée suivante:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
```

Nous allons utiliser deux modèles: un gourmand: `A.*Z` , et un paresseux: `A.*?Z` Ces modèles donnent les résultats suivants:

- `A.*Z` donne 1 correspondance: `AlazyZgreeedyAlaaazyZ` (exemples: Regex101 , Rubular)
- `A.*?Z` donne 2 correspondances: `AlazyZ` et `AlaaazyZ` (exemples: Regex101 , Rubular)

Commencez par vous concentrer sur ce que fait `A.*Z` Quand il correspond au premier `A` , le `.*` , Étant gourmand, essaie alors de correspondre autant `.` comme possible.

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_________________________/
      A.* matched, Z can't match
```

Étant donné que le `Z` ne correspond pas, les retours en arrière du moteur et `.*` Doivent alors correspondre à un de moins `.` :

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \________________________/
      A.* matched, Z can't match
```

Cela se produit encore quelques fois, jusqu'à ce qu'il arrive à ceci:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____________________/
      A.* matched, Z can now match
```

Maintenant, `Z` peut correspondre, donc le motif global correspond à:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____________________/
      A.*Z matched
```

En revanche, la répétition (paresseuse) réticente dans `A.*?Z` correspond au premier peu `.` que possible, puis en prendre plus `.` le cas échéant. Cela explique pourquoi il trouve deux correspondances dans l'entrée.

Voici une représentation visuelle de la correspondance entre les deux motifs:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____/l      \______/l      l = lazy
     \_________g_________/       g = greedy
```

Exemple basé sur la réponse apportée par les polygenélubrifiants .

Le standard POSIX n'inclut pas le `?` opérateur, tant de moteurs de regex POSIX n'ont pas de correspondance paresseuse. Bien que le refactoring, en particulier avec le "plus grand tour de

passe-passe" , puisse aider dans certains cas, le seul moyen d'avoir une correspondance parfaite est d'utiliser un moteur qui le supporte.

Limites avec plusieurs correspondances

Lorsque vous avez une entrée avec des limites bien définies et que vous attendez plus d'une correspondance dans votre chaîne, vous avez deux options:

- Utiliser des quantificateurs paresseux;
- Utiliser une classe de caractère négatif.

Considérer ce qui suit:

Vous avez un moteur de template simple, vous voulez remplacer des sous-chaînes comme `$[foo]` où `foo` peut être n'importe quelle chaîne. Vous voulez remplacer cette sous-chaîne par celle qui est basée sur la partie comprise entre `[]` .

Vous pouvez essayer quelque chose comme `\$\[(.*)\]` , Puis utiliser le premier groupe de capture.

Le problème avec ceci est si vous avez une chaîne comme `something $[foo] lalala $[bar] something else` votre match sera

```
something $[foo] lalala $[bar] something else
          | \_______CG1_______/|
          \________Match_______/
```

Le groupe de capture étant `foo] lalala $[bar` qui peut être ou ne pas être valide.

Vous avez deux solutions

1. Utiliser la paresse: Dans ce cas, faire `*` lazy est une façon de trouver les bonnes choses. Donc, vous changez votre expression en `\$\[(.*?)\]`
2. En utilisant la classe de caractère négatif: `[^\]]` vous modifiez votre expression en `\$\[([^\]]*)\]` .

Dans les deux solutions, le résultat sera le même:

```
something $[foo] lalala $[bar] something else
          | \_/|         | \_/|
          \____/         \____/
```

Le groupe de capture étant respectivement `foo` et `bar` .

L'utilisation de la classe de caractères nuls réduit le problème de retour en arrière et peut économiser beaucoup de temps à votre processeur lorsqu'il s'agit de grandes entrées.

Lire Quantificateurs gourmands et paresseux en ligne:

https://riptutorial.com/fr/regex/topic/429/quantificateurs-gourmands-et-paresseux

Chapitre 17: Quantificateurs Possessifs

Remarques

NB Émulation des quantificateurs possessifs

Examples

Utilisation de base des quantificateurs possessifs

Les quantificateurs possessifs sont une autre classe de quantificateurs dans de nombreuses variantes de regex qui permettent de désactiver efficacement le retour en arrière pour un jeton donné. Cela peut aider à améliorer les performances, tout en empêchant les correspondances dans certains cas.

La classe des quantificateurs possessifs peut être distinguée des quantificateurs paresseux ou gourmands par l'ajout d'un `+` après le quantificateur, comme indiqué ci-dessous:

Quantificateur	Glouton	Paresseux	Possessif
Zéro ou plus	`*`	`*?`	`*+`
Un ou plus	`+`	`+?`	`++`
Zéro ou un	`?`	`??`	`?+`

Considérons, par exemple, les deux modèles `".*"` Et `".*+"` , Agissant sur la chaîne `"abc"d` . Dans les deux cas, le `"` au début de la chaîne correspond, mais après cela, les deux modèles auront des comportements et des résultats différents.

Le quantificateur gourmand va alors slurp le reste de la chaîne, `abc"d` . Parce que cela ne correspond pas au modèle, il va alors revenir en arrière et supprimer le `d` , en laissant le **quantificateur** contenant `abc"` . Étant donné que cela ne correspond toujours pas au modèle, le quantificateur supprimera le `"` , ne contenant que l' `abc` . Cela correspond au modèle (car le `"` correspond à un littéral plutôt qu'au quantificateur), et l'expression rationnelle indique un succès.

Le quantificateur possessif slurp également le reste de la chaîne, mais contrairement au quantificateur gourmand, il ne fera pas marche arrière. Étant donné que son contenu, `abc"d` , ne permet pas le reste du motif de la correspondance, le regex s'arrêtera et signalera l'échec de la correspondance.

Étant donné que les quantificateurs possessifs ne font pas de retour en arrière, ils peuvent entraîner une augmentation significative des performances sur les modèles longs ou complexes. Ils peuvent cependant être dangereux (comme illustré ci-dessus) si l'on ne sait pas exactement comment fonctionnent les quantificateurs en interne.

Lire Quantificateurs Possessifs en ligne: https://riptutorial.com/fr/regex/topic/5916/quantificateurs-possessifs

Chapitre 18: Récursivité

Remarques

La récursivité est principalement disponible en versions compatibles Perl, telles que:

- Perl
- PCRE
- Oniguruma
- Renforcer

Examples

Recurser le motif entier

La construction `(?R)` est équivalente à `(?0)` (ou `\g<0>`) - elle vous permet de recréer tout le motif:

```
<(?>[^<>]+|(?R))+>
```

Cela correspondra entre crochets correctement équilibrés avec n'importe quel texte entre les crochets, comme `<a<b>c<d>e>` .

Recurser dans un sous-modèle

Vous pouvez Recurse dans un sous - motif en utilisant les constructions suivantes (en fonction de l'arôme), en supposant que `n` est un numéro de groupe de capture, et `name` le nom d'un groupe de capture.

- `(?n)`
- `\g<n>`
- `\g'0'`
- `(?&name)`
- `\g<name>`
- `\g'name'`
- `(?P>name)`

Le modèle suivant:

```
\[(?<angle><(?&angle)*+>)*\]
```

Correspondra au texte tel que: `[<<><>><>]` - les crochets bien placés entre crochets. La récursivité est souvent utilisée pour les constructions équilibrées.

Définitions de Subpattern

La construction `(?(DEFINE)...)` vous permet de définir des sous-modèles que vous pouvez

référencer ultérieurement via la récursivité. Lorsque rencontré dans le modèle, il *ne* sera *pas* mis en correspondance.

Ce groupe doit contenir des définitions de sous-modèle nommées, qui ne seront accessibles que par récursivité. Vous pouvez définir les grammaires de cette façon:

```
(?x) # ignore pattern whitespace
(?(DEFINE)
  (?<string> ".*?" )
  (?<number> \d+ )
  (?<value>
    \s* (?:
        (?&string)
      | (?&number)
      | (?&list)
    ) \s*
  )
  (?<list> \[ (?&value) (?: , (?&value) )* \] )
)
^(?&value)$
```

Ce modèle valide le texte comme suit:

```
[42, "abc", ["foo", "bar"], 10]
```

Notez comment une liste peut contenir une ou plusieurs valeurs, et une valeur peut elle-même être une liste.

Références de groupe relatives

Les sous-modèles peuvent être référencés avec leur numéro de groupe *relatif* :

- `(?-1)` rentrera dans le groupe *précédent*
- `(?+1)` rentrera dans le groupe *suivant*

Aussi utilisable avec la syntaxe `\g<N>` .

Backreferences en récurrences (PCRE)

Dans PCRE, les groupes correspondants utilisés pour les références avant une récursivité sont conservés dans la récursivité. Mais après la récursivité, ils ont tous ramené à ce qu'ils étaient avant d'y entrer. En d'autres termes, les groupes correspondants dans la récursivité sont tous oubliés.

Par exemple:

```
(?J)(?(DEFINE)(\g{a}(?<a>b)\g{a}))(?<a>a)\g{a}(?1)\g{a}
```

allumettes

```
aaabba
```

Les récurrences sont atomiques (PCRE)

Dans PCRE, il n'y a pas de rétrolien après la première correspondance pour une récursivité. Alors

```
(?(DEFINE)(aaa|aa|a))(?1)ab
```

ne correspond pas

```
aab
```

car après la correspondance avec `aa` dans la récursivité, il n'essaie plus jamais de faire correspondre seulement `a` .

Lire Récursivité en ligne: https://riptutorial.com/fr/regex/topic/739/recursivite

Chapitre 19: Référence arrière

Examples

Les bases

Les références arrière sont utilisées pour correspondre au même texte précédemment associé à un groupe de capture. Cela permet à la fois de réutiliser les parties précédentes de votre modèle et d'assurer deux parties d'une chaîne.

Par exemple, si vous essayez de vérifier qu'une chaîne a un chiffre compris entre zéro et neuf, un séparateur, tel que des traits d'union, des barres obliques ou même des espaces, une lettre minuscule, un autre séparateur, utilisez une regex comme ceci:

```
[0-9][-/ ][a-z][-/ ][0-9]
```

Cela correspondrait à `1-a-4` , mais cela correspondrait *aussi* à `1-a/4` ou `1 a-4` . Si nous voulons que les séparateurs correspondent, nous pouvons utiliser un groupe de capture et une référence arrière. La référence arrière examine la correspondance trouvée dans le groupe de capture indiqué et vérifie que l'emplacement de la référence arrière correspond exactement.

En utilisant notre même exemple, le regex deviendrait:

```
[0-9]([-/ ])[a-z]\1[0-9]
```

Le `\1` indique le premier groupe de capture du motif. Avec ce petit changement, le regex correspond maintenant à `1-a-4` ou `1 a 4` mais pas `1 a-4` ou `1-a/4` .

Le nombre à utiliser pour votre référence arrière dépend de l'emplacement de votre groupe de capture. Le nombre peut être compris entre un et neuf et peut être trouvé en comptant vos groupes de capture.

```
([0-9])([-/ ])[a-z][-/ ]([0-9])
|--1--||--2--|           |--3--|
```

Les groupes de capture imbriqués modifient légèrement ce nombre. Vous comptez d'abord le groupe de capture extérieur, puis le niveau suivant, et continuez jusqu'à ce que vous quittiez le nid:

```
(([0-9])([-/ ]))([a-z])
 |--2--||--3--|
|-------1------||--4--|
```

Références ambiguës

Problème: Vous devez faire correspondre le texte d'un certain format, par exemple:

```
1-a-0
6/p/0
4 g 0
```

C'est un chiffre, un séparateur (un des `-` , `/` ou un espace), une lettre, le même séparateur et un zéro.

Solution naïve: en adaptant la regex à l' exemple de base , vous obtenez cette regex:

```
[0-9]([-/ ])[a-z]\10
```

Mais cela ne fonctionnera probablement pas. La plupart des versions de regex prennent en charge plus de neuf groupes de capture, et très peu d'entre elles sont suffisamment intelligentes pour comprendre que, puisqu'il n'y a qu'un groupe de capture, `\10` doit être une référence au groupe 1 suivi d'un littéral `0` . La plupart des saveurs le traiteront comme une référence rétroactive au groupe 10. Quelques-unes d'entre elles lanceront une exception car il n'ya pas de groupe 10; le reste échouera tout simplement.

Il y a plusieurs façons d'éviter ce problème. L'une consiste à utiliser des groupes nommés (et des références nommées):

```
[0-9](?<sep>[-/ ])[a-z]\k<sep>0
```

Si votre langage regex le prend en charge, le format `\g{n}` (où `n` est un nombre) peut contenir le numéro de référence arrière entre accolades pour le séparer des chiffres suivants:

```
[0-9]([-/ ])[a-z]\g{1}0
```

Une autre méthode consiste à utiliser un format regex étendu, en séparant les éléments avec des espaces non significatifs (en Java, vous devrez échapper de l'espace entre crochets):

```
(?x) [0-9] ([-/ ]) [a-z] \1 0
```

Si votre version regex ne prend pas en charge ces fonctionnalités, vous pouvez ajouter une syntaxe inutile mais inoffensive, comme un groupe non capturant:

```
[0-9]([-/ ])[a-z](?:\1)0
```

... ou un quantificateur factice (c'est peut-être la seule circonstance où `{1}` est utile):

```
[0-9]([-/ ])[a-z]\1{1}0
```

Lire Référence arrière en ligne: https://riptutorial.com/fr/regex/topic/4072/reference-arriere

Chapitre 20: Regex de validation du mot de passe

Examples

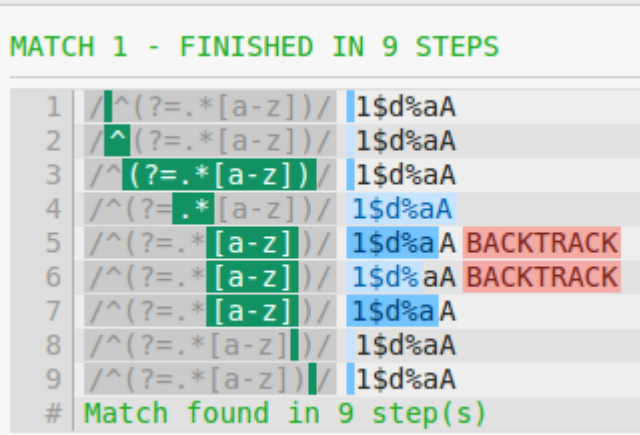
Un mot de passe contenant au moins 1 majuscule, 1 minuscule, 1 chiffre, 1 caractère spécial et une longueur d'au moins 10

Comme les caractères / chiffres peuvent être n'importe où dans la chaîne, nous avons besoin de points de référence. Les lookaheads sont de `zero width` ce qui signifie qu'ils ne consomment aucune chaîne. En termes simples, la position de la vérification se réinitialise à la position d'origine après chaque condition de recherche.

Hypothèse : - *Considérer les caractères non-verbaux comme spéciaux*

```
^(?=.{10,}$)(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*\W).*$
```

Avant de procéder à l'explication, jetons un coup d'oeil à la façon dont l'expression rationnelle `^(?=.*[az])` fonctionne (la *longueur n'est pas prise en compte ici*) sur la chaîne `1$d%aA`



Crédit image : - **https://regex101.com/**

Choses à remarquer

- La vérification est lancée depuis le début de la chaîne en raison de la balise d'ancrage `^` .
- La position de vérification est réinitialisée au démarrage après que la condition de recherche est satisfaite.

Ventilation des regex

```
^ #Starting of string
 (?=.{10,}$) #Check there is at least 10 characters in the string.
             #As this is lookahead the position of checking will reset to starting again
 (?=.*[a-z]) #Check if there is at least one lowercase in string.
```

```
            #As this is lookahead the position of checking will reset to starting again
 (?=.*[A-Z]) #Check if there is at least one uppercase in string.
            #As this is lookahead the position of checking will reset to starting again
 (?=.*[0-9]) #Check if there is at least one digit in string.
            #As this is lookahead the position of checking will reset to starting again
 (?=.*\W) #Check if there is at least one special character in string.
          #As this is lookahead the position of checking will reset to starting again
.*$ #Capture the entire string if all the condition of lookahead is met. This is not required
if only validation is needed
```

Nous pouvons également utiliser la version *non gourmande* de la regex ci-dessus

```
^(?=.{10,}$)(?=.*?[a-z])(?=.*?[A-Z])(?=.*?[0-9])(?=.*?\W).*$
```

Un mot de passe contenant au moins 2 majuscules, 1 minuscule, 2 chiffres et d'une longueur d'au moins 10

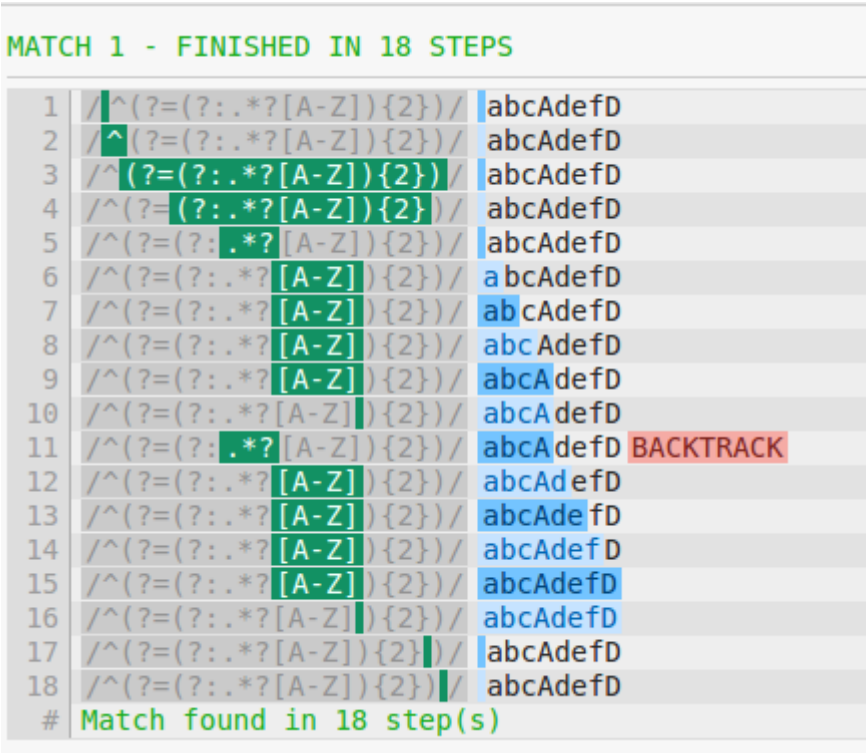
Cela peut être fait avec un peu de modification dans la regex ci-dessus

```
 ^(?=.{10,}$)(?=(?:.*?[A-Z]){2})(?=.*?[a-z])(?=(?:.*?[0-9]){2}).*$
```

ou

```
 ^(?=.{10,}$)(?=(?:.*[A-Z]){2})(?=.*[a-z])(?=(?:.*[0-9]){2}).*
```

Voyons comment une expression rationnelle simple `^(?=(?:.*?[AZ]){2})` fonctionne sur la chaîne `abcAdefD`



Crédit image : - **https://regex101.com/**

Lire Regex de validation du mot de passe en ligne:
https://riptutorial.com/fr/regex/topic/5340/regex-de-validation-du-mot-de-passe

Chapitre 21: Regex Pitfalls

Examples

Pourquoi le point (.) Ne correspond-il pas au caractère de nouvelle ligne ("\n")?

`.*` dans regex signifie en gros "attraper **tout** jusqu'à la fin de la saisie".

Donc, pour les chaînes simples, comme `hello world .*` Fonctionne parfaitement. Mais si vous avez une chaîne représentant, par exemple, des lignes dans un fichier, ces lignes seraient séparées par un *séparateur de ligne* , tel que `\n` (nouvelle ligne) sur les systèmes de type Unix et `\r\n` (retour chariot et nouvelle ligne) sur Les fenêtres.

Par défaut , dans la plupart des moteurs regex, `.` **ne** correspond **pas** aux caractères de nouvelle ligne, donc la correspondance s'arrête à la fin de chaque *ligne logique* . Si vous voulez `.` pour correspondre **vraiment** tout, y compris les nouvelles lignes, vous devez activer « dot-Correspondances- tous » mode dans votre moteur de regex de choix (par exemple, ajouter `re.DOTALL` drapeau en Python ou `/s` dans PCRE.

Pourquoi une regex ignore-t-elle certaines parenthèses / parenthèses et les fait correspondre par la suite?

Considérez cet exemple:

> Il entra dans le café "Dostoevski" et dit: "Bonsoir".

Nous avons ici deux jeux de citations. Supposons que nous voulions faire correspondre les deux, de sorte que notre regex corresponde à `"Dostoevski"` **et** `"Good evening."`

Au début, vous pourriez être tenté de rester simple:

```
".*"  # matches a quote, then any characters until the next quote
```

Mais ça ne marche pas: cela correspond à la première citation dans `"Dostoevski"` et **jusqu'à** la citation finale dans `"Good evening."` , y compris le `and said:` part. Regex101 démo

Pourquoi est-ce arrivé?

Cela se produit parce que le moteur de regex, quand il rencontre `.*` , "Mange" toutes les entrées à la fin. Ensuite, il doit correspondre à la finale `"` . Donc, il" recule "à partir de la fin du match, lâchant le texte correspondant jusqu'à ce que le premier `"` soit trouvé - et, bien sûr, le dernier `"` du match. , à la fin de la partie `"Good evening."` .

Comment empêcher cela et correspondre exactement aux premiers guillemets?

Utilisez `[^"]*` . Il ne mange pas toutes les entrées - seulement jusqu'au premier `"` , juste au besoin.
Regex101 démo

Lire Regex Pitfalls en ligne: https://riptutorial.com/fr/regex/topic/10747/regex-pitfalls

Chapitre 22: Retour en arrière

Examples

Qu'est-ce qui cause le retour en arrière?

Pour trouver une correspondance, le moteur regex consommera des caractères un par un. Lorsqu'un match partiel commence, le moteur se souviendra de la position de départ afin de pouvoir revenir en arrière si les personnages suivants ne terminent pas le match.

- Si le match est terminé, il n'y a pas de retour en arrière
- Si le match n'est pas terminé, le moteur va revenir en arrière sur la chaîne (comme lorsque vous rembobinez une vieille bande) pour essayer de trouver une correspondance complète.

Par exemple: `\d{3}[az]{2}` contre la chaîne `abc123def` sera parcouru comme tel:

```
abc123def
^ Does not match \d
abc123def
 ^ Does not match \d
abc123def
  ^ Does not match \d
abc123def
   ^ Does match \d (first one)
abc123def
    ^ Does match \d (second one)
abc123def
     ^ Does match \d (third one)
abc123def
      ^ Does match [a-z] (first one)
abc123def
       ^ Does match [a-z] (second one)
          MATCH FOUND
```

Modifions maintenant le regex à `\d{2}[az]{2}` contre la même chaîne (`abc123def`):

```
abc123def
^ Does not match \d
abc123def
 ^ Does not match \d
abc123def
  ^ Does not match \d
abc123def
   ^ Does match \d (first one)
abc123def
    ^ Does match \d (second one)
abc123def
     ^ Does not match [a-z]
abc123def
    ^ BACKTRACK to catch \d{2} => (23)
abc123def
      ^ Does match [a-z] (first one)
abc123def
```

```
      ^ Does match [a-z] (second one)
        MATCH FOUND
```

Pourquoi le retour en arrière peut-il être un piège?

Le backtracking peut être provoqué par des quantificateurs optionnels ou des constructions alternées, car le moteur regex essaiera d'explorer tous les chemins. Si vous exécutez la regex `a+b` contre `aaaaaaaaaaaaa` il n'y a pas de correspondance et le moteur le trouvera assez rapidement.

Mais si vous modifiez le regex à `(aa*)+b` le nombre de combinaisons augmentera assez rapidement et la plupart des moteurs (non optimisés) tenteront d'explorer tous les chemins et prendront une éternité pour essayer de trouver une correspondance ou exception de délai d'attente. Cela s'appelle **un retour en arrière catastrophique** .

Bien sûr, `(aa*)+b` semble être une erreur de débutant mais il est là pour illustrer ce point et parfois vous vous retrouverez avec le même problème mais avec des modèles plus compliqués.

Un cas plus extrême de retour en arrière catastrophique se produit avec le regex `(x+x+)+y` (vous l'avez probablement déjà vu ici et ici), qui nécessite un temps exponentiel pour déterminer qu'une chaîne contenant `x` s et rien d'autre (par exemple, `xxxxxxxxxxxxxxxxxxx`) ne correspondent pas.

Comment l'éviter

Soyez aussi précis que possible, réduisez autant que possible les chemins possibles. Notez que certains appariements de regex ne sont pas vulnérables au backtracking, comme ceux inclus dans `awk` ou `grep` car ils sont basés sur Thompson NFA .

Lire Retour en arrière en ligne: https://riptutorial.com/fr/regex/topic/977/retour-en-arriere

Chapitre 23: Substitutions avec des expressions régulières

Paramètres

En ligne	La description
nombre $	Remplace la sous-chaîne par numéro de groupe.
$ {name}	Remplace la sous-chaîne par un nom de groupe nommé .
$$	Caractère échappé '$' dans la chaîne de résultat (remplacement).
$ & *(0 $)*	Remplace par la chaîne complète correspondante.
$ + *($ &)*	Remplace le texte correspondant par le dernier groupe capturé.
$ `	Remplace tout le texte correspondant par chaque texte non apparié avant le match.
$ '	Remplace tout le texte correspondant par chaque texte non apparié après le match.
$ _	Remplace tout le texte correspondant à la chaîne entière.
Remarque:	*Les* termes en *italique* signifient que les chaînes sont volatiles (peuvent varier en fonction de votre saveur regex).

Examples

Bases de la substitution

L'une des méthodes les plus courantes et les plus utiles pour remplacer le texte par regex consiste à utiliser des groupes de capture .
Ou même un groupe de capture nommé , comme référence pour stocker ou remplacer les données.

Il y a deux termes très similaires dans les documents de regex, il peut donc être important de ne jamais mélanger les **substitutions** (c'est- `$1` dire `$1`) avec les **backreferences** (c.-à-d. `\1`). Les termes de substitution sont utilisés dans un texte de remplacement; Backreferences, dans l'expression pure Regex. Même si certains langages de programmation acceptent les deux pour les substitutions, ce n'est pas encourageant.

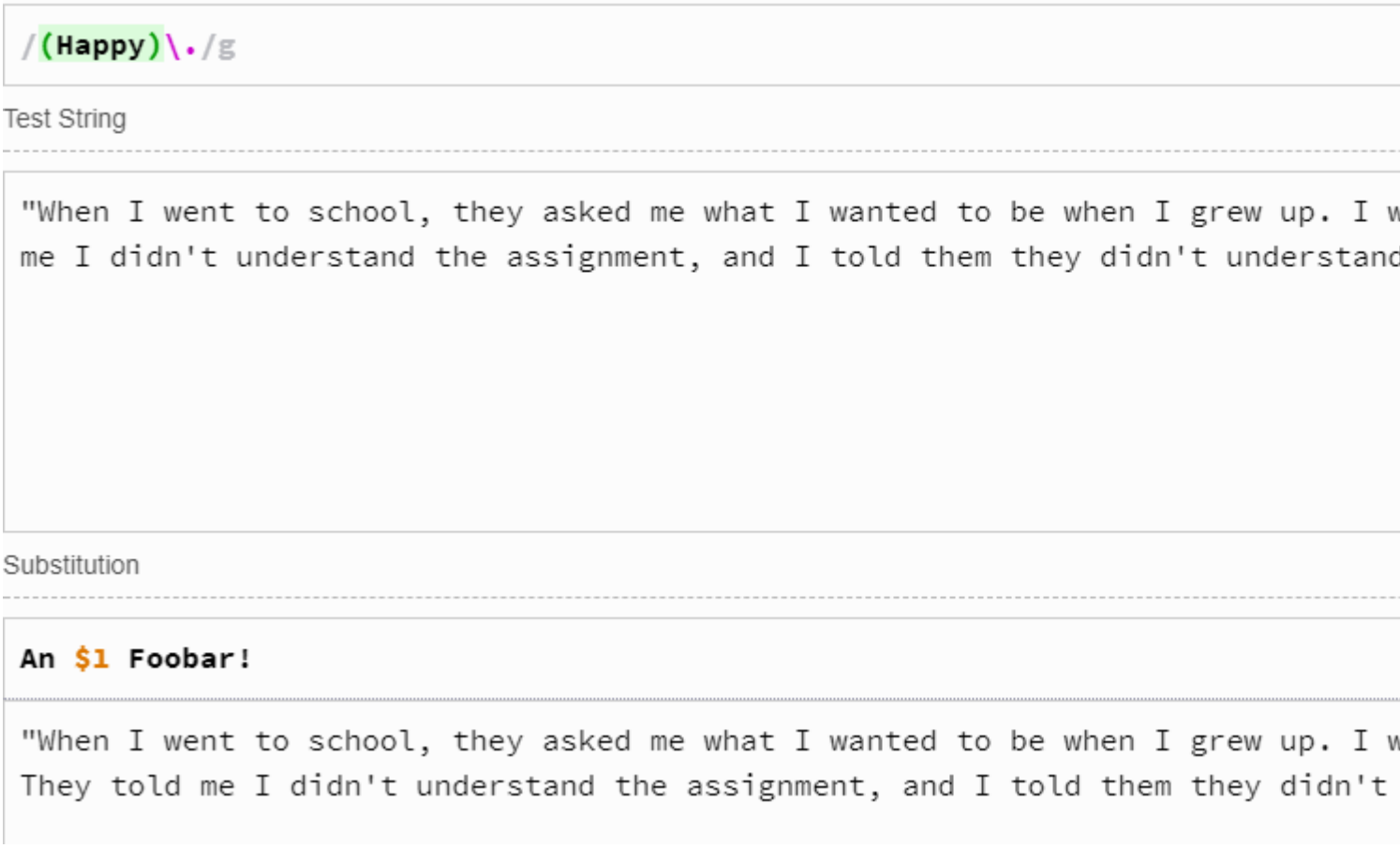
Disons que nous avons cette regex: `/hello(\s+)world/i` . Chaque fois que `$number` est référencé (dans ce cas, `$1`), les espaces blancs correspondant à `\s+` seront remplacés à la place.

Le même résultat sera exposé avec le regex: `/hello(?<spaces>\s+)world/i` . Et comme nous avons un groupe nommé ici, nous pouvons également utiliser `${spaces}` .

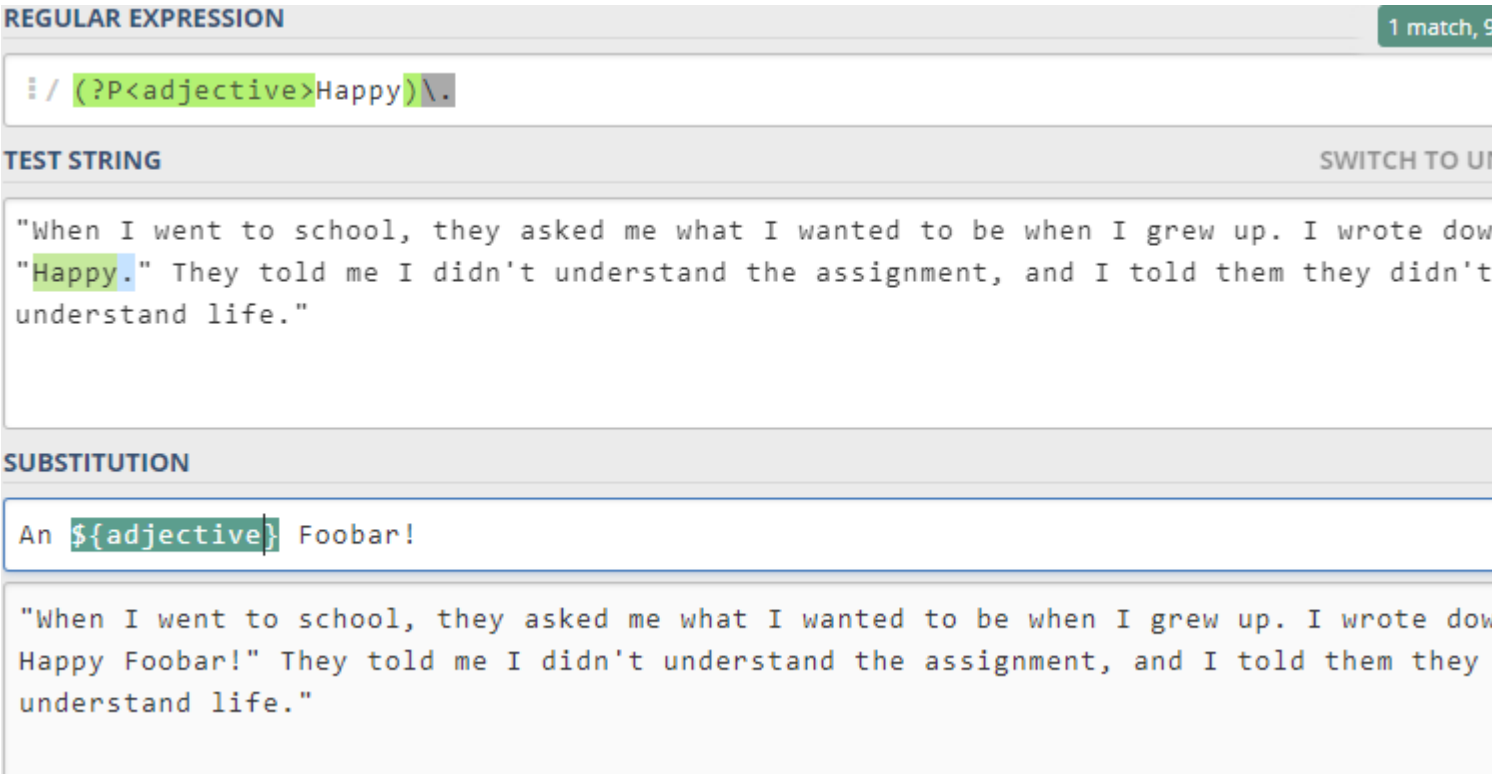
Dans ce même exemple, nous pouvons également utiliser `$0` ou `$&` (**Remarque:** `$&` peut être utilisé comme `$+` place, ce qui signifie récupérer le groupe de capture **LAST** dans d'autres moteurs regex), en fonction du type de regex avec lequel vous travaillez, pour obtenir le texte entier correspondant. (c.- `$&` d. `$&` renverra `hEllo woRld` pour la chaîne: `hEllo woRld of Regex!`)

Jetez un coup d'oeil à cet exemple simple de substitution en utilisant le devis adapté de John Lennon en utilisant la syntaxe `$number` et la syntaxe `${name}` :

Exemple de groupe de capture simple:



Exemple de groupe de capture nommé:



Remplacement Avancé

Certains langages de programmation ont leurs propres particularités Regex, par exemple le terme `$+` (en C #, Perl, VB etc.) qui remplace le texte correspondant au dernier groupe capturé.

Exemple:

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)\s\1\b";
      string substitution = "$+";
      string input = "The the dog jumped over the fence fence.";
      Console.WriteLine(Regex.Replace(input, pattern, substitution,
                        RegexOptions.IgnoreCase));
   }
}
// The example displays the following output:
//       The dog jumped over the fence.
```

Exemple du réseau de développeurs de Microsoft Official [1]

Les autres termes de substitution rares sont `$`` et `$'` :

`$`` = Remplace les correspondances au texte **avant** la chaîne correspondante
`$'` = Remplace les correspondances au texte **après** la chaîne correspondante

De ce fait, ces chaînes de remplacement devraient faire leur travail comme ceci:

```
Regex: /part2/
Input: "part1part2part3"
Replacement: "$`"
Output: "part1part1part3" //Note that part2 was replaced with part1, due &` term
--------------------------------------------------------------------------------
Regex: /part2/
Input: "part1part2part3"
Replacement: "$'"
Output: "part1part3part3" //Note that part2 was replaced with part3, due &' term
```

Voici un exemple de ces substitutions travaillant sur un morceau de javascript:

```
var rgx = /middle/;
var text = "Your story must have a beginning, middle, and end"
console.log(text.replace(rgx, "$`"));
//Logs: "Your story must have a beginning, Your story must have a beginning, , and end"
console.log(text.replace(rgx, "$'"))
//Logs: "Your story must have a beginning, , and end, and end"
```

Il y a aussi le terme `$_` qui récupère tout le texte correspondant à la place:

```
Regex: /part2/
Input: "part1part2part3"
Replacement: "$_"
Output: "part1part1part2part3part3" //Note that part2 was replaced with part1part2part3,
                                                                    // due $_ term
```

Convertir ceci en VB nous donnerait ceci:

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "ABC123DEF456"
      Dim pattern As String = "\d+"
      Dim substitution As String = "$_"
      Console.WriteLine("Original string:          {0}", input)
      Console.WriteLine("String with substitution: {0}", _
                        Regex.Replace(input, pattern, substitution))
   End Sub
End Module
' The example displays the following output:
'       Original string:          ABC123DEF456
'       String with substitution: ABCABC123DEF456DEFABC123DEF456
```

Exemple du réseau de développeurs de Microsoft Official [2]

Et le dernier terme de substitution, mais non le moindre, est `$$` , qui traduit en une expression rationnelle serait le même que `\$` (version échappée du littéral **$**).

Si vous voulez faire correspondre une chaîne comme ceci: `USD: $3.99` par exemple, et que vous voulez stocker le `3.99` , mais le remplacer par `$3.99` avec une seule regex, vous pouvez utiliser:

```
Regex: /USD:\s+\$([\d.]+)/
```

```
Input: "USD: $3.99"
Replacement: "$$$1"
To Store: "$1"
Output: "$3.99"
Stored: "3.99"
```

Si vous souhaitez tester cela avec Javascript, vous pouvez utiliser le code:

```
var rgx = /USD:\s+\$([\d.]+)/;
var text = "USD: $3.99";
var stored = parseFloat(rgx.exec(text)[1]);
console.log(stored); //Logs 3.99
console.log(text.replace(rgx, "$$$1")); //Logs $3.99
```

Les références

[1]: substituer le dernier groupe capturé
[2]: substituer la chaîne de saisie entière

Lire Substitutions avec des expressions régulières en ligne:
https://riptutorial.com/fr/regex/topic/9852/substitutions-avec-des-expressions-regulieres

Chapitre 24: Types de moteur d'expression régulière

Examples

NFA

Un moteur NFA (Nondeterministic Finite Automaton) est *entraîné par le motif* .

Principe

Le motif de regex est analysé dans un arbre.

Le pointeur de la *position actuelle* est défini sur le début de la chaîne d'entrée et une correspondance est tentée à cette position. Si la correspondance est faisable, la position est incrémentée au prochain caractère de la chaîne et une autre correspondance est tentée à partir de cette position. Ce processus est répété jusqu'à ce qu'une correspondance soit trouvée ou que la fin de la chaîne d'entrée soit atteinte.

Pour chaque tentative de match

L'algorithme fonctionne en effectuant une traversée de l'arbre de configuration pour une position de départ donnée. Au fur et à mesure de sa progression dans l'arborescence, il met à jour la *position actuelle en* utilisant les caractères correspondants.

Si l'algorithme rencontre un nœud d'arbre qui ne correspond pas à la chaîne d'entrée à la position actuelle, il devra *revenir* en *arrière* . Ceci est effectué en revenant au nœud parent dans l'arborescence, en réinitialisant la position d'entrée actuelle à la valeur qu'il avait lors de la saisie du nœud parent et en essayant la branche suivante.

Si l'algorithme parvient à quitter l'arborescence, il signale une correspondance réussie. Sinon, lorsque toutes les possibilités ont été essayées, le match échoue.

Optimisations

Les moteurs Regex appliquent généralement certaines optimisations pour de meilleures performances. Par exemple, s'ils déterminent qu'une correspondance doit commencer par un caractère donné, ils tenteront une correspondance uniquement aux positions de la chaîne d'entrée où ce caractère apparaît.

Exemple

`abeacab` correspondre `a(b|c)a` avec la chaîne d'entrée `abeacab` :

L'arbre de modèle pourrait ressembler à:

```
CONCATENATION
    EXACT: a
    ALTERNATION
        EXACT: b
        EXACT: c
    EXACT: a
```

Les processus de correspondance sont les suivants:

```
a(b|c)a      abeacab
^            ^
```

`a` se trouve dans la chaîne d'entrée, consommez-le et passez à l'élément suivant dans l'arbre des modèles: l'alternance. Essayez la première possibilité: un `b` exact.

```
a(b|c)a      abeacab
  ^            ^
```

`b` se trouve, si l'alternance réussit, consommer et passer à l'élément suivant de la concaténation: une exacte `a` :

```
a(b|c)a      abeacab
      ^        ^
```

`a` n'est *pas* trouvé à la position attendue. Retour à l'alternance, réinitialiser la position d'entrée à la valeur qu'il avait à l'entrée de l'alternance pour la première fois, et essayez la *deuxième* alternative:

```
a(b|c)a      abeacab
    ^         ^
```

`c` n'est *pas* trouvé à cette position. Retour à la concaténation. Il n'y a pas d'autres possibilités à essayer à ce stade, donc il n'y a pas de correspondance au début de la chaîne.

Tentez une deuxième correspondance à la position de saisie suivante:

```
a(b|c)a      abeacab
^             ^
```

`a` ne correspond *pas* là. Tentez un autre match à la position suivante:

```
a(b|c)a      abeacab
```

```
^             ^
```

Pas de chance non plus. Avance à la position suivante.

```
a(b|c)a      abeacab
^               ^
```

`a` match, alors consommez-le et entrez l'alternance:

```
a(b|c)a      abeacab
  ^               ^
```

`b` ne correspond pas. Essayez la deuxième alternative:

```
a(b|c)a      abeacab
    ^               ^
```

`c` correspond, consommez-le et passez à l'élément suivant dans la concaténation:

```
a(b|c)a      abeacab
      ^              ^
```

`a` correspond, et la fin de l'arbre a été atteint. Signaler un match réussi:

```
a(b|c)a      abeacab
               \_/
```

DFA

Un moteur DFA (Déterministic Finite Automaton) est *piloté par l'entrée* .

Principe

L'algorithme analyse la chaîne d'entrée une *fois* et se souvient de tous les chemins possibles dans l'expression régulière. Par exemple, lorsqu'une alternance est rencontrée dans le motif, deux nouveaux chemins sont créés et tentés indépendamment. Lorsqu'un chemin donné ne correspond pas, il est supprimé des possibilités.

Implications

Le temps de correspondance est limité par la taille de la chaîne en entrée. Il n'y a pas de retour en arrière et le moteur peut trouver plusieurs correspondances simultanément, même des correspondances qui se chevauchent.

Le principal inconvénient de cette méthode est la réduction du nombre de fonctionnalités pouvant

être prises en charge par le moteur, par rapport au type de moteur NFA.

Exemple

Match `a(b|c)a` contre `abadaca` :

```
abadaca      a(b|c)a
^            ^        Attempt 1      ==> CONTINUE

abadaca      a(b|c)a
 ^           ^        Attempt 2      ==> FAIL
               ^      Attempt 1.1    ==> CONTINUE
                 ^    Attempt 1.2    ==> FAIL

abadaca      a(b|c)a
  ^          ^        Attempt 3      ==> CONTINUE
                 ^    Attempt 1.1    ==> MATCH

abadaca      a(b|c)a
   ^         ^        Attempt 4      ==> FAIL
               ^      Attempt 3.1    ==> FAIL
                 ^    Attempt 3.2    ==> FAIL

abadaca      a(b|c)a
    ^        ^        Attempt 5      ==> CONTINUE

abadaca      a(b|c)a
     ^       ^        Attempt 6      ==> FAIL
               ^      Attempt 5.1    ==> FAIL
                 ^    Attempt 5.2    ==> CONTINUE

abadaca      a(b|c)a
      ^      ^        Attempt 7      ==> CONTINUE
                 ^    Attempt 5.2    ==> MATCH

abadaca      a(b|c)a
       ^       ^      Attempt 7.1    ==> FAIL
                 ^    Attempt 7.2    ==> FAIL
```

Lire Types de moteur d'expression régulière en ligne:
https://riptutorial.com/fr/regex/topic/2861/types-de-moteur-d-expression-reguliere

Chapitre 25: Vitrine utile de regex

Examples

Faire correspondre une date

Vous devez vous rappeler que regex a été conçu pour correspondre à une date (ou non). Dire qu'une date est *valide* est une tâche beaucoup plus complexe, car elle nécessitera beaucoup de gestion des exceptions (voir conditions des années bissextiles).

Commençons par faire correspondre le mois (1 - 12) avec un 0 optionnel:

```
0?[1-9]|1[0-2]
```

Pour correspondre au jour, également avec un 0 optionnel:

```
0?[1-9]|[12][0-9]|3[01]
```

Et pour correspondre à l'année (supposons juste la gamme 1900 - 2999):

```
(?:19|20)[0-9]{2}
```

Le séparateur peut être un espace, un tiret, une barre oblique, un espace vide, etc. N'hésitez pas à ajouter tout ce qui pourrait être utilisé comme séparateur:

```
[-\\/ ]?
```

Maintenant, vous concaténez le tout et obtenez:

```
(0?[1-9]|1[0-2])[-\\/ ]?(0?[1-9]|[12][0-9]|3[01])[-/ ]?(?:19|20)[0-9]{2} // MMDDYYYY
(0?[1-9]|[12][0-9]|3[01])[-\\/ ]?(0?[1-9]|1[0-2])[-/ ]?(?:19|20)[0-9]{2} // DDMMYYYY
(?:19|20)[0-9]{2}[-\\/ ]?(0?[1-9]|1[0-2])[-/ ]?(0?[1-9]|[12][0-9]|3[01]) // YYYYMMDD
```

Si vous voulez être un peu plus pédant, vous pouvez utiliser une référence arrière pour vous assurer que les deux séparateurs seront identiques:

```
(0?[1-9]|1[0-2])([-\\/ ]?)(0?[1-9]|[12][0-9]|3[01])\2(?:19|20)[0-9]{2} // MMDDYYYY
                                                   ^ refer to [-/ ]
(0?[1-9]|[12][0-9]|3[01])([-\\/ ]?)(0?[1-9]|1[0-2])\2(?:19|20)[0-9]{2} // DDMMYYYY
(?:19|20)[0-9]{2}([-\\/ ]?)(0?[1-9]|1[0-2])\2(0?[1-9]|[12][0-9]|3[01]) // YYYYMMDD
```

Correspond à une adresse e-mail

> Faire correspondre une adresse e-mail dans une chaîne est une tâche difficile, car la spécification qui la définit, la RFC2822 , est complexe, ce qui la rend difficile à implémenter en tant que regex. Pour plus de détails pourquoi ce n'est pas une bonne

> idée de faire correspondre un email avec une regex, veuillez vous référer à l'exemple d'antipattern pour ne pas utiliser une regex: pour les emails correspondants . Le meilleur conseil à noter sur cette page est d'utiliser une bibliothèque évaluée par des pairs et largement répandue dans votre langue préférée pour l'implémenter.

Valider un format d'adresse e-mail

Lorsque vous devez valider rapidement une entrée pour vous assurer qu'elle *ressemble à* un courrier électronique, la meilleure solution est de rester simple:

```
^\S{1,}@\S{2,}\.\S{2,}$
```

Cette regex vérifiera que l'adresse mail est une séquence de caractères de longueur supérieure à un, séparés par un espace, suivie d'un `@` , suivie de deux séquences de caractères non-espaces de longueur supérieure ou égale à deux `.` . Ce n'est pas parfait et peut valider des adresses non valides (selon le format), mais surtout, cela n'invalide pas les adresses valides.

Vérifiez l'adresse existe

Le seul moyen fiable de vérifier la validité d'un email est de vérifier son existence. `VRFY` commande `VRFY` SMTP avait été conçue à cette fin, mais malheureusement, après avoir été abusée par des spammeurs, elle n'est plus disponible .

Donc, la seule façon de vérifier que le courrier est valide et qu'il existe, c'est d'envoyer un courrier électronique à cette adresse.

Énormes alternatives Regex

Cependant, il n'est pas impossible de valider une adresse email en utilisant une regex. Le seul problème est que plus les spécifications seront proches de la spécification, plus elles seront grosses et, par conséquent, difficiles à lire et à gérer. Vous trouverez ci-dessous des exemples de regex plus précises utilisées dans certaines bibliothèques.

> **X** Les regex suivantes sont données à des fins de documentation et d'apprentissage, les copier-coller dans votre code est une mauvaise idée. Au lieu de cela, utilisez directement cette bibliothèque, de sorte que vous pouvez vous fier au code en amont et aux développeurs homologues pour maintenir votre code d'analyse des e-mails à jour et maintenu.

Module de correspondance d'adresses Perl

Les meilleurs exemples de telles regex se trouvent dans certaines bibliothèques standard de langages. Par exemple, il y en a un du module `RFC::RFC822::Address` dans la bibliothèque Perl qui essaie d'être aussi précis que possible selon la RFC. Pour votre curiosité, vous pouvez trouver une version de cette regex à cette URL , qui a été générée à partir de la grammaire, et si vous êtes tenté de copier-coller, voici une citation de l'auteur de regex:

" *Je ne maintiens pas l'expression régulière [linked]. Il peut y avoir des bogues déjà corrigés dans le module Perl.* "

Module de correspondance d'adresse .Net

Une autre variante, plus courte, est celle utilisée par la bibliothèque standard .Net dans le module `EmailAddressAttribute` :

```
^((([a-z]|\d|[!#\$%&'\*\+\-\/=\?\^_`{\|}~]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])+(\.([a-
z]|\d|[!#\$%&'\*\+\-\/=\?\^_`{\|}~]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])+)*)|((\x22)((((\x20|\x09)*(\x0d\x0a))?(\x20|\x09)+)?(([\x01-\x08\x0b\x0c\x0e-
\x1f\x7f]|\x21|[\x23-\x5b]|[\x5d-\x7e]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])|(\\([\x01-
\x09\x0b\x0c\x0d-\x7f]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])))*(((\x20|\x09)*(\x0d\x0a))?(\x20|\x09)+)?(\x22)))@((([a-z]|\d|[\u00A0-\uD7FF\uF900-
\uFDCF\uFDF0-\uFFEF])|(([a-z]|\d|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])([a-z]|\d|-
|\.|_|~|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])*([a-z]|\d|[\u00A0-\uD7FF\uF900-
\uFDCF\uFDF0-\uFFEF])))\.)+(([a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])|(([a-z]|[\u00A0-
\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])([a-z]|\d|-|\.|_|~|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])*([a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])))\.?$
```

Mais même s'il est *plus court,* il est encore trop gros pour être lisible et facile à maintenir.

Module de correspondance d'adresse Ruby

Dans Ruby, une composition de regex est utilisée dans le module rfc822 pour correspondre à une adresse. C'est une bonne idée, car si des bogues sont détectés, il sera plus facile de trouver la partie regex à modifier et de la réparer.

Module de correspondance d'adresse Python

Par exemple, le module d'analyse de courrier électronique python n'utilise pas de regex, mais l'implémente à l'aide d'un analyseur.

Faire correspondre un numéro de téléphone

Voici comment faire correspondre un code de préfixe (a `+` ou (00), puis un nombre compris entre 1 et 1939, avec un espace facultatif):
Cela ne cherche pas un préfixe *valide* mais quelque chose qui pourrait être un préfixe. Voir la liste complète des préfixes

```
(?:00|\+)?[0-9]{4}
```

Puis, comme la longueur totale du numéro de téléphone est, au maximum, 15, nous pouvons rechercher jusqu'à 14 chiffres:
Au moins un chiffre est dépensé pour le préfixe

```
[0-9]{1,14}
```

Les numéros peuvent contenir des espaces, des points ou des tirets et peuvent être groupés par 2 ou 3.

```
(?:[ .-][0-9]{3}){1,5}
```

Avec le préfixe optionnel:

```
(?:(?:00|\+)?[0-9]{4})?(?:[ .-][0-9]{3}){1,5}
```

Si vous souhaitez faire correspondre un format de pays spécifique, vous pouvez utiliser cette requête de recherche et ajouter le pays, la question a certainement déjà été posée.

Faire correspondre une adresse IP

IPv4

Pour correspondre au format d'adresse IPv4, vous devez vérifier les nombres `[0-9]{1,3}` trois fois `{3}` séparés par des points `\.` et se terminant par un autre numéro.

```
^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$
```

Cette expression régulière est trop simple - si vous voulez qu'elle soit précise, vous devez vérifier que les nombres sont compris entre `0` et `255` , avec le regex ci-dessus acceptant `444` dans n'importe quelle position. Vous voulez vérifier 250-255 avec `25[0-5]` , ou toute autre valeur 200 `2[0-4][0-9]` , ou toute valeur 100 ou moins avec `[01]?[0-9][0-9]` . Vous voulez vérifier qu'il est suivi d'un point `\.` trois fois `{3}` et une fois sans période.

```
^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

IPv6

Les adresses IPv6 prennent la forme de mots de 8 hexagonaux de 16 bits délimitées par les deux points (`:` caractère). Dans ce cas, nous vérifions 7 mots suivis de deux-points, suivis d'un autre qui ne l'est pas. Si un mot a des zéros non significatifs, ils *peuvent* être tronqués, ce qui signifie que chaque mot peut contenir entre 1 et 4 chiffres hexadécimaux.

```
^(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$
```

Ceci est cependant insuffisant. Comme les adresses IPv6 peuvent devenir très "verbeuses", la norme spécifie que les mots contenant uniquement des zéros peuvent être remplacés par `::` . Cela ne peut être fait qu'une seule fois dans une adresse (pour n'importe où entre 1 et 7 mots consécutifs), car il serait autrement indéterminé. Cela produit un certain nombre de variations (plutôt désagréables):

```
^::(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}$
^[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}$
^[0-9a-fA-F]{1,4}:[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:)?[0-9a-fA-F]{1,4}$
```

```
^(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}::[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}::$
```

Maintenant, mettre tout cela ensemble (en utilisant l'alternance) donne:

```
^(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$|
^::(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}$|
^[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}$|
^[0-9a-fA-F]{1,4}:[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:)?[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}::[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}::$
```

Assurez-vous de l'écrire en mode multiligne et avec une pile de commentaires afin que quiconque est inévitablement chargé de déterminer ce que cela signifie ne vous poursuit pas avec un objet contondant.

Valider une chaîne de temps de 12 heures et 24 heures

Pour un format de 12 heures, on peut utiliser:

```
^(?:0?[0-9]|1[0-2])[-:][0-5][0-9]\s*[ap]m$
```

Où

- `(?:0?[0-9]|1[0-2])` est l'heure
- `[-:]` est le séparateur qui peut être ajusté pour répondre à vos besoins
- `[0-5][0-9]` est la minute
- `\s*[ap]m` suivi un nombre quelconque de caractères d'espacement et `am` ou `pm`

Si vous avez besoin des secondes:

```
^(?:0?[0-9]|1[0-2])[-:][0-5][0-9][-:][0-5][0-9]\s*[ap]m$
```

Pour un format de 24 heures:

```
^(?:[01][0-9]|2[0-3])[-:h][0-5][0-9]$
```

Où:

- `(?:[01][0-9]|2[0-3])` est l'heure
- `[-:h]` le séparateur, qui peut être ajusté pour répondre à vos besoins
- `[0-5][0-9]` est la minute

Avec les secondes:

```
^(?:[01][0-9]|2[0-3])[-:h][0-5][0-9][-:m][0-5][0-9]$
```

Où `[-:m]` est un deuxième séparateur, en remplaçant le `h` pour les heures par un `m` pour les minutes et `[0-5][0-9]` pour le second.

Code postal du Royaume-Uni

Regex pour faire correspondre les codes postaux au Royaume-Uni

Le format est le suivant, où A signifie une lettre et 9 un chiffre:

Format	Couverture	Exemple
Cellule	Cellule	
AA9A 9AA	Zone de code postal WC; EC1 – EC4, NW1W, SE1P, SW1	EC1A 1BB
A9A 9AA	E1W, N1C, N1P	W1A 0AX
A9 9AA, A99 9AA	B, E, G, L, M, N, S, W	M1 1AE, B33 8TH
AA9 9AA, AA99 9AA	Tous les autres codes postaux	CR2 6XH, DN55 1PT

```
(GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[QVX]][0-9][A-HJKPSTUW])|([A-Z-[QVX]][A-Z-[IJZ]][0-9][ABEHMNPRVWXY])))) [0-9][A-Z-[CIKMOV]]{2})
```

Où première partie:

```
 (GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[QVX]][0-9][A-HJKPSTUW])|([A-Z-[QVX]][A-Z-[IJZ]][0-9][ABEHMNPRVWXY]))))
```

Seconde:

```
[0-9][A-Z-[CIKMOV]]{2})
```

Lire Vitrine utile de regex en ligne: https://riptutorial.com/fr/regex/topic/3605/vitrine-utile-de-regex

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec les expressions régulières	0rkan, Addison, balpha, Community, Confiqure, Ibrahim, J F, JelmerS, JohnLBevan, Kendra, Laurel, Maria Deleva, Mariano, Mateus, mnoronha, Rudy M, Stephen Leppik, Tot Zam, TylerH, Wolf, Yaron, zmo
2	Caractères d'ancre: Dollar ($)	ArtOfCode, CPHPython, hjpotter92, Kendra, rubayet.R, Tom Lord, UNagaswamy, Wiktor Stribiżew
3	Classes de caractères	Acey, CPHPython, Dmitry Bychenko, HamZa, kdhp, Lucas Trzesniewski, Maria Deleva, RamenChef, rgoliveira, rock321987, Wiktor Stribiżew
4	Échapper	CPHPython, David Knipe, Laurel
5	Frontière de mot	cdm, jonathanking, kdhp, Maria Deleva, Peter G, rgoliveira, Tushar
6	Groupement Atomique	OnlineCop
7	Groupes de capture	Addison, Alan Moore, Lucas Trzesniewski, Tomalak, Vogel612
8	Groupes de capture nommés	Thomas Ayoub
9	Lookahead et Lookbehind	BoppreH, hwnd, Lucas Trzesniewski, Maria Deleva, Wiktor Stribiżew
10	Lorsque vous ne devez PAS utiliser les expressions régulières	dorukayhan, Kendra, zmo
11	Match Reset: \ K	nhahtdh, Wiktor Stribiżew, Will Barnwell
12	Matchers UTF-8: Lettres, Marques, Ponctuation etc.	mudasobwa
13	Modèles simples assortis	balpha, GradientByte, Graham, Joe, Mariano, rgoliveira, Tot Zam, Yaron

14	Modificateurs de regex (flags)	Eder, Mateus, Tim Pietzcker, Wiktor Stribiżew
15	Personnages d'ancre: Caret (^)	CPHPython, Eder, J F, JohnLBevan, Jojodmo, knut, Mateus, Mike H-R, Mr. Deathless, nhahtdh, revo, rgoliveira, Tom Lord, zb226
16	Quantificateurs gourmands et paresseux	0rkan, Confiqure, David Knipe, GradientByte, Laurel, Mario, Mark Stewart, Nathan Arthur, nhahtdh, phatfingers, sweaver2112, Thomas Ayoub, Tim Pietzcker
17	Quantificateurs Possessifs	Mark Hurd, Sebastian Lenartowicz
18	Récursivité	Keith Hall, Laurel, Lucas Trzesniewski, user23013
19	Référence arrière	Alan Moore, Kendra, OnlineCop
20	Regex de validation du mot de passe	rock321987
21	Regex Pitfalls	BrightOne
22	Retour en arrière	dorukayhan, Mike, Miljen Mikic, SQB, Thomas Ayoub, Vituel
23	Substitutions avec des expressions régulières	Mateus
24	Types de moteur d'expression régulière	Lucas Trzesniewski, Markus Jarderot
25	Vitrine utile de regex	depperm, Devid Farinelli, Echelon, Herb, Kendra, Matas Vaitkevicius, nhahtdh, Sebastian Lenartowicz, Steve Chambers, Thomas Ayoub, Tomasz Jakub Rup, zmo