

Beste de savoir

Les expressions régulières

19 août 2020

Table des matières

1.	Les différents ensembles	1
1.1.	Les ensembles	2
1.2.	Les raccourcis d'ensembles	3
1.3.	Caractères de regex	4
1.4.	Caractères unicode	4
2.	Les opérateurs	4
2.1.	L'opérateur d'alternative	5
2.2.	Les opérateurs de quantité	5
2.3.	Quelques exemples	5
3.	Les groupes	5
3.1.	Les groupes capturant	5
3.2.	Les groupes non capturant	6
3.3.	Les groupes nommés	6
3.4.	Les groupes spéciaux	7
4.	Utilisation en milieu professionnel	8
4.1.	Où utiliser les regex?	8
4.2.	Exemples en programmation	9
4.3.	Exemples en ligne de commande	10
4.4.	Exemple de différences	11
4.5.	La substitution	12
5.	Entraînement	13
	Contenu masqué	14

Une expression régulière est une chaîne de caractères qui décrit les ensembles possible d'une autre chaîne de caractères.

En suivant ce tutoriel, vous serez capable d'écrire vous-même vos expressions régulières en fonction de vos besoins. On peut les utiliser aussi bien pour rechercher des "motifs" dans des fichiers texte, appliquer des transformations dans des cellules de tableurs, valider la conformité d'une donnée ou encore extraire des informations.

Nous commencerons avec les bases des expressions régulières en s'exerçant sur des exemples simples, puis nous verrons ensuite quelques cas d'utilisations dans un milieu professionnel (en programmation et avec des outils en ligne de commande).

Le cours sera séparé en cinq parties :

1. Les différents ensembles

Quand on écrit sur un clavier, on peut utiliser un ensemble défini de caractères. On y trouve les **lettres** minuscules/majuscules, les **chiffres**, les **espaces** ou **caractères "blancs"** et des

1. Les différents ensembles

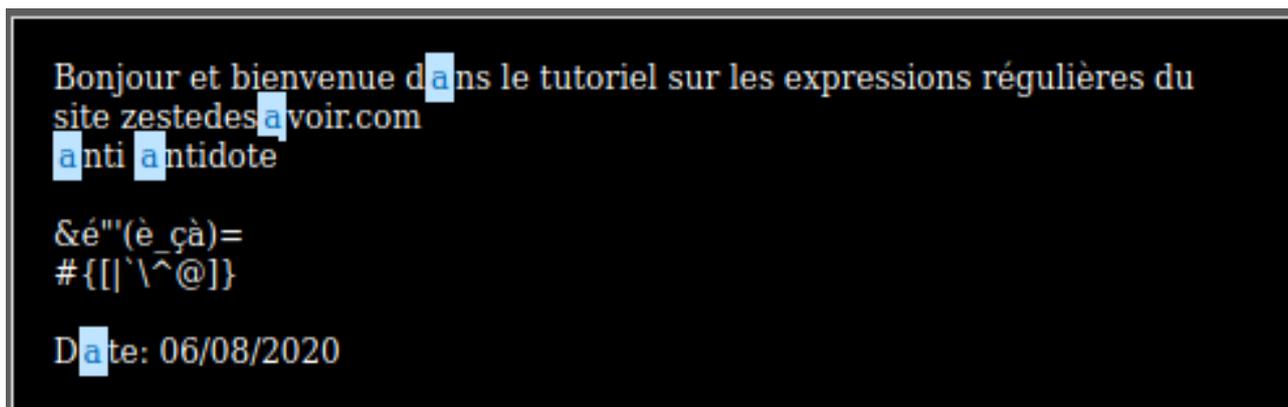
caractères "spéciaux" (& " ' ...).

Chacun de ces types de caractères appartient à un ensemble et nous allons tout de suite commencer à jouer avec.

Pour commencer à explorer le monde des *regex*, ouvrez le site [Rubular](#) et dans la partie *Your test string*, collez ceci :

```
1 Bonjour et bienvenue dans le tutoriel sur les expressions
   régulières du site zestedesavoir.com
2 anti antidote
3
4 &é"'(è_çà)=
5 #{[|\`^\@]}
6
7 Date: 06/08/2020
```

Si dans la partie *Your regular expression* on écrit juste `a`, on voit dans la partie *Match result* que tous les `a` présents dans le texte se sont allumés (sauf le `à` qui est un caractère spécial).



Faire une recherche sur une seule lettre n'a pas vraiment d'intérêt. On pourrait alors vouloir chercher d'un seul coup l'ensemble des lettres de l'alphabet, ou bien l'ensemble des voyelles dans le texte... Et c'est là qu'interviennent les ensembles! 🍊

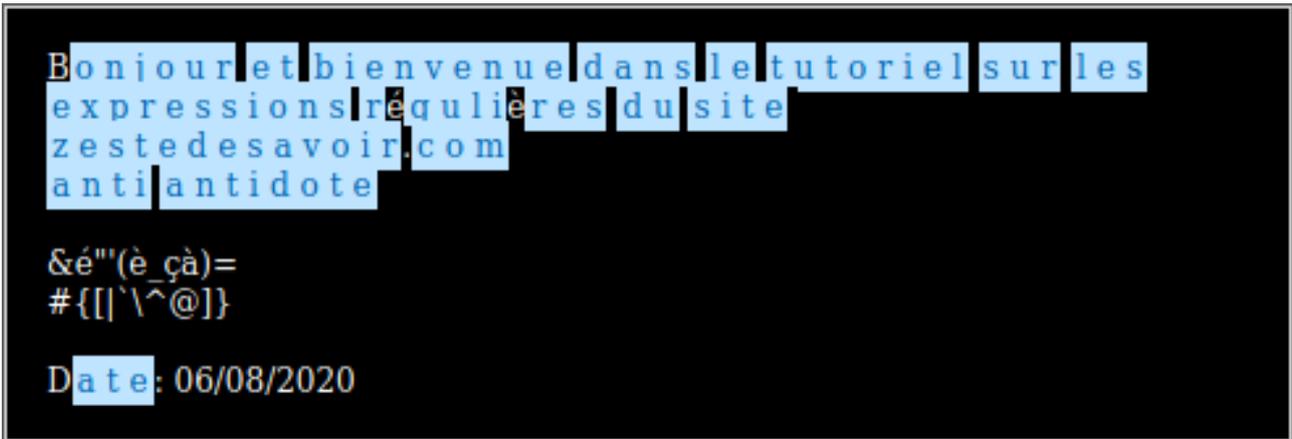
1.1. Les ensembles

Dans les expressions régulières, un ensemble se représente entre crochets `[]` :

- les lettres `[a-z]`;
- les chiffres `[0-9]`;
- les caractères blanc `[\t\n]`:
 - `\t` est la manière textuelle de représenter une tabulation;
 - `\n` est la manière textuelle de représenter un retour à la ligne.
- les caractères spéciaux `&é"'(è_çà)=` (à compléter en fonction des besoins);
- la négation (trouver ce qui n'est pas compris dans mon ensemble) `[^a]` (tout ce qui n'est pas un `a`).

1. Les différents ensembles

Maintenant écrivons dans la partie *Your regular expression* l'ensemble des lettres (`[a-z]`), nous devrions obtenir ceci :



On peut observer que toutes les **lettres minuscules** se sont allumées mais pas les **lettres majuscules**. C'est pour la simple et bonne raison que ce sont 2 ensembles différents. Pour avoir l'ensemble des lettres du texte, on va pouvoir écrire 2 ensembles à l'intérieur de notre ensemble : `[a-zA-Z]`.

Ainsi on comprend bien que `[a-z]` signifie: "Je veux les lettres allant de `a` à `z`", ce qu'il ne sera pas possible de faire avec les voyelles puisqu'elles ne se suivent pas, il faudra écrire `[aeiou]`.

i

Si vous utilisez `-` en dehors d'un ensemble (`[]`) il sera interprété comme le caractère `-`, ce qui sera également le cas dans un ensemble **si** et seulement **si** il n'est pas entouré. Ex. `[-aeiou]` trouve les voyelles et le tiret.

`[aei-ou]` trouve les `a`, `e`, ce qui se trouve entre `i` et `o` dans la table ascii, et le `u`.

1.2. Les raccourcis d'ensembles

Il peut vite être fastidieux d'écrire un grand ensemble et de réécrire toujours la même chose. Il existe des ensembles déjà définis pour ceux présentés plus haut:

- les lettres + les chiffres + l'underscore `\w`;
- les chiffres `\d`;
- les caractères blancs `\s`;
- les caractères spéciaux: désolé je n'irai pas plus loin pour celui ci;
- n'importe quel caractère `.`.

i

Si on veut détecter le caractère "point" `.`, il va falloir écrire `\.`, sinon, l'expression régulière pensera que l'on cherche n'importe quel caractère.

Très pratique, chaque raccourci a également son contraire:

2. Les opérateurs

- ce qui n'est pas une lettre ou un chiffre ou un underscore `\W`;
- ce qui n'est pas un chiffre `\D`;
- ce qui n'est pas un caractère blanc `\S`.

L'ensemble en majuscule indique l'inverse de l'ensemble minuscule. Dans ce cas il n'est pas utile d'utiliser les `[]` sauf si on veut combiner différents ensembles.

1.3. Caractères de regex

Dans certains cas, on peut vouloir détecter des éléments qu'on ne peut pas écrire au clavier, c'est le cas d'un début de ligne, une fin de ligne, mais également d'un début ou une fin de mot.

Pour les détecter avec une regex, il existe ceci:

- début de ligne: `^`;
- fin de ligne: `$`;
- début/fin de mot: `\b`.

Pour comprendre l'utilisation de ces caractères, voici des exemples:

- si on veut récupérer le premier mot de chaque ligne : `^\w+`;
- si on veut récupérer le dernier mot de chaque ligne : `\w+$`;
- si on veut récupérer le mot anti:
 - en utilisant juste `anti`, on en obtiendrait 2 (celui de `anti` et de `antidote`);
 - en utilisant `\banti\b` on obtient bien le mot seul `anti` mais pas celui de `antidote`.

1.4. Caractères unicode

Il est aussi possible d'utiliser des regex pour trouver des caractères [Unicode](#) :

- le caractère `!` en unicode : `\x21` ou `\u0021`;
- un marqueur unicode : `\p{M}`;
- n'importe quelle lettre de n'importe quel langage : `\p{L}\p{M}*`;
- n'importe quel graphème unicode : `\X` (équivalent de `\P{M}\p{M}*`).

i

Un graphème unicode est un caractère potentiellement enrichi de marqueurs (comme des signes diacritiques) représenté comme une seule unité graphique. Par exemple `a` (`\u0061`) et `à` (qui peut être encodé comme `\u0224` ou bien `\u0061\u0300`) sont des graphèmes.

2. Les opérateurs

Dans la partie précédente, on a vu comment chercher des éléments d'un ensemble mais cela fonctionnait caractère par caractère. Nous allons maintenant voir comment faire pour trouver un mot au lieu d'une succession de caractères.

3. Les groupes

2.1. L'opérateur d'alternative

Il est possible de dire que l'on veut un caractère **OU** un autre. Pour cela on va simplement écrire `a|b` qui signifie "Je veux seulement les `a` ou les `b`" (aussi équivalent à l'ensemble `[ab]`).

2.2. Les opérateurs de quantité

- 0 ou 1 élément ?
- 0 ou plusieurs éléments *
- Au moins 1 élément +
- Un nombre défini d'élément `{n,m}`
 - `{0,}` = *
 - `{1,}` = +
 - `{,1}` = ?
 - `\b\w{4,6}\b` (les mots qui font de 4 à 6 lettres)

L'intérêt est par exemple de ne plus chercher un chiffre mais un nombre. Avec la partie précédente on peut écrire `\d+` qui va allumer tous les nombres (dans une date par exemple).

Petite précision

Le chiffre est au nombre ce que la lettre est au mot.

2.3. Quelques exemples

Pour les exemples suivants, nous prendrons comme référence le texte "*Bonjour 2020*".

- L'expression `[a-zA-Z]` doit trouver la liste suivante `['B', 'o', 'n', 'j', 'o', 'u', 'r']` alors que l'expression `[a-zA-Z]+` doit trouver `['Bonjour']`.
- L'expression `\d` doit trouver la liste suivante `['2', '0', '2', '0']` alors que l'expression `\d+` doit trouver `['2020']`.
- L'expression `on|ou` doit trouver la liste suivante `['on', 'ou']`.
- L'expression `\d{2}` doit trouver la liste suivante `['20', '20']`.

3. Les groupes

3.1. Les groupes capturant

Dans certains cas, on peut vouloir capturer seulement une partie de l'expression régulière que l'on a écrite. Par exemple, quand on sait comment une phrase est formée, on peut vouloir récupérer une information précise.

Ex. "*Bonjour, je m'appelle Toto*"

3. Les groupes

Si on veut récupérer le prénom, on ne peut pas écrire `[a-zA-Z]+` car nous aurions tous les mots même ceux qui ne nous intéressent pas. On sait que le prénom se trouve généralement après avoir dit "je m'appelle".

On peut alors écrire `je m'appelle ([a-zA-Z]+)` et on obtient ceci :

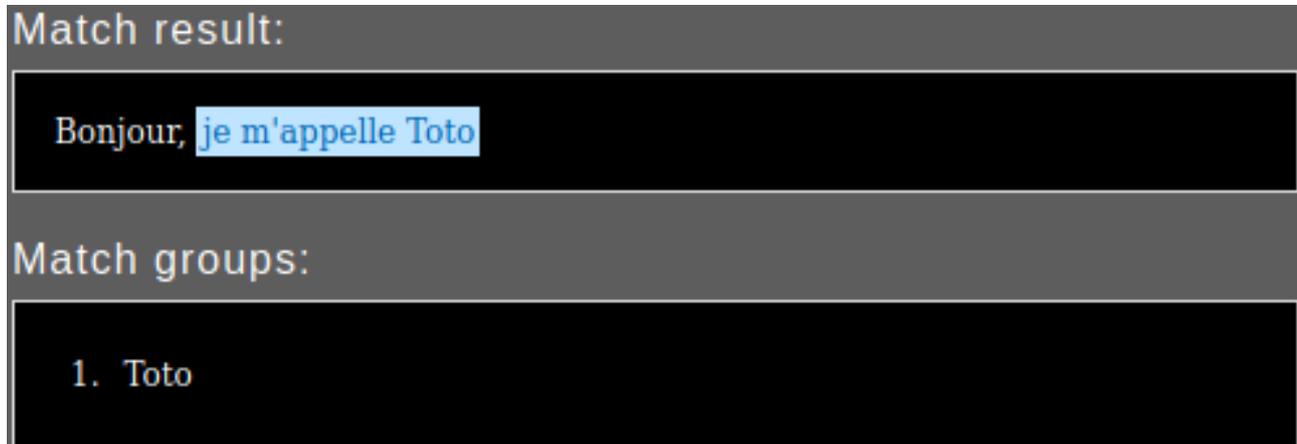


FIGURE 3.1. – groupe capturant

Le match complet est `je m'appelle Toto` mais on a précisé que la capture intéressante était la partie après "je m'appelle".

3.2. Les groupes non capturant

Dans certains cas, il est possible que nous devions repérer une information importante mais dont la capture finale nous importe peu. Dans ce cas on peut utiliser un groupe qui ne va pas capturer ce qui est entre parenthèses: `(?:noncapturant)`.

Pas d'exemple pour celui là, vous en trouverez sûrement une utilité en pratiquant par vous même 🍊 .

3.3. Les groupes nommés

Quand on veut récupérer des données ordonnées d'une certaine manière comme le format `jj/dd/yyyy` d'une date par exemple, ce type de groupe devient très intéressant. Un groupe nommé s'écrit de cette manière : `(?<name>selection)`.

Donc si on veut récupérer le jour, le mois et l'année, on peut écrire :

`(?<day>\d+)\/(?<month>\d+)\/(?<year>\d+)`

The image shows a terminal window with a dark background. At the top, it says "Match result:" followed by the text "Bonjour et bienvenue dans le tutoriel sur les expressions régulières du site zestedesavoir.com". Below this, a regex pattern is shown: `&é"(è_çà)=#[[\\`^@]]`. The date "06/08/2020" is highlighted in blue. Below that, it says "Match groups:" followed by a table with three rows: "day" with value "06", "month" with value "08", and "year" with value "2020".

FIGURE 3.2. – groupe nommé

i

Le caractère `/` dans une expression régulière est un caractère spécial. Pour l'utiliser, il faut "échapper" le caractère avec un `\`, exactement comme pour les tabulations et les retours à la ligne : `\/`.

3.4. Les groupes spéciaux

Ces types de groupes vont être utilisés pour faire des recherches plus avancées dans le texte.

— Positive lookahead: trouver l'élément qui précède

Ex. `a(?=b)` -> "Les `a` qui précèdent un `b`".

— Negative lookahead: trouver l'élément qui ne précède pas

Ex. `a(?!b)` -> "Les `a` qui ne précèdent pas un `b`".

— Positive lookbehind: trouver l'élément qui succède

Ex. `(?<=b)a` -> "Les `a` qui succèdent un `b`".

— Negative lookbehind: trouver l'élément qui ne succède pas

4. Utilisation en milieu professionnel

Ex. `(?!b)a` -> "Les `a` qui ne succèdent pas un `b`".

Ex. "Je veux le mot qui se trouve avant *zestedesavoir*"
`\w+(?=\s*zestedesavoir)` -> doit allumer "site".

La liste complète des groupes est accessible [ici](#) (en anglais).

4. Utilisation en milieu professionnel

Depuis le début du tutoriel, vous faites vos tests sur le site **Rubular** qui interprète des expressions régulières basées sur le langage Ruby. Si je vous précise ça, c'est que ce détail a son importance.

En effet, en fonction des langages ou des outils utilisés, le moteur de regex ne sera pas forcément le même et vous pouvez vous trouver dans un cas où une expression régulière fonctionne dans un langage/outil et pas dans un autre.

i

En dehors des langages et des outils, il existe également d'autres sites que Rubular pour tester des expressions régulières.

Ex. [Regex101](#) , [RegExr](#) , [RegexTester](#) ...

!

Ce n'est pas parce que l'on maîtrise l'utilisation des regex qu'il faut en abuser. Même si dans certains cas cela peut sembler très pratique, il faut garder à l'esprit que les utiliser abusivement peut parfois mener à des problèmes de performances et de maintenabilité.

4.1. Où utiliser les regex ?

On peut les utiliser dans les différents langages de programmation, notamment en C++, Java, Python, Ruby, Perl, Php, SQL (en fonction du SGBD), Javascript, etc. Ce qui changera en fonction du langage, c'est le moteur qui interprète la regex.

Il est également possible de les utiliser dans des outils pour développeurs (éditeur de code, IDE...) comme Notepad++, Visual studio code, Eclipse... La liste pouvant être assez longue contentons nous de ces exemples 🍌 .

Et il est aussi possible de s'en servir dans des outils en ligne de commande comme [grep](#) , [sed](#) ou encore [awk](#) .

?

Puisque tu nous a dis qu'il ne fallait pas abuser des regex, existe-t-il un exemple où l'utilisation des expressions régulières est idéale?

Oui, les regex peuvent être exactement ce qu'il nous faut pour par exemple donner le format précis d'une carte de crédit en fonction du type de carte:

4. Utilisation en milieu professionnel

- la Visa: `^4[0-9]{12}(:[0-9]{3})?$;`
- la MasterCard: `^(?:5[1-5][0-9]{2}|222[1-9]|22[3-9][0-9]|2[3-6][0-9]{2}|27[01][0-9]|2720)[0-9]{12}$;`
- l’American Express: `^3[47][0-9]{13}$.`

C’est d’ailleurs grâce à ça que quand vous entrez le numéro de votre carte sur un site en ligne, il est capable de vous afficher le type de carte que vous utilisez.

4.2. Exemples en programmation

Prenons un exemple simple, on veut savoir si une chaîne de caractères est une [couleur hexadécimale](#) ou non.

4.2.1. JavaScript

```
1 let re = /^#[0-9a-fA-F]{6}$/
2
3 re.test('#80e0f5') // true
4 re.test('#F39580') // true
5 re.test('#8cj380') // false
```

4.2.2. Python

```
1 import re
2 p = re.compile('^#[0-9a-fA-F]{6}$')
3
4 bool(p.match('#80e0f5')) # True
5 bool(p.match('#F39580')) # True
6 bool(p.match('#8cj380')) # False
```

4.2.3. C++

```
1 #include <iostream>
2 #include <regex>
3
4 int main()
5 {
6     std::regex re("^#[0-9a-fA-F]{6}$");
7     std::cmatch m;
```

4. Utilisation en milieu professionnel

```
8
9     std::cout << std::regex_match("#80e0f5", m, re) << std::endl;
    // 1
10    std::cout << std::regex_match("#F39580", m, re) << std::endl;
    // 1
11    std::cout << std::regex_match("#8cj380", m, re) << std::endl;
    // 0
12 }
```

4.3. Exemples en ligne de commande



grep, sed et awk ont été portés sur Windows, vous pouvez les trouver [ici](#) . Ces outils en ligne de commande sont également accessible sur Windows grâce à [WSL](#) , je considère donc que ce qui suit est valable aussi bien pour Windows que Linux.

Pour nos tests, on va créer un fichier de log fictif, copier/coller le texte suivant et l'enregistrer dans un fichier:

```
1 192.168.0.11 [01 Feb 2019] - GET data
2 192.168.0.12 [02 Feb 2019] - POST data
3 192.168.0.12 [02 Apr 2019] - DELETE data
4 192.168.0.12 [24 Jun 2020] - POST data
5 192.168.0.14 [12 Jul 2020] - GET data
6 192.168.0.14 [14 Jul 2020] - POST data
7 192.168.0.14 [17 Aug 2020] - DELETE data
```

Imaginons que pour X raisons, on vous demande de trouver tout ce qui s'est passé à des dates précises sur un système ou une machine. Vous n'allez évidemment pas lire ligne par ligne et extraire à la main les lignes souhaitées.

4.3.1. Avec grep

Ex 1. *"Je veux savoir tout ce qui s'est passé en avril 2019 et juillet 2020"*.

```
1 grep -P "[\d{2} (Apr 2019|Jul 2020)\]" log_file.txt
```

La commande précédente devrait vous retourner ceci:

```
192.168.0.12 [02 Apr 2019] - DELETE data
192.168.0.14 [12 Jul 2020] - GET data
192.168.0.14 [14 Jul 2020] - POST data
```

4. Utilisation en milieu professionnel

4.3.2. Avec sed

Ex 2. "Je veux connaître seulement les IP de ceux qui ont fait quelque chose en avril 2019 et juillet 2020".

```
1 sed -nE 's/^(.*) \[[0-9]{2} (Apr 2019|Jul 2020)\].*/\1/p'  
log_file.txt | uniq
```

La commande précédente devrait vous retourner ceci:

```
192.168.0.12  
192.168.0.14
```

4.3.3. Avec awk

Même exemple qu'avec `sed`:

```
1 awk -e '/(.*) \[[0-9]{2} (Apr 2019|Jul 2020)\]/ {print $1}'  
log_file.txt | uniq
```

4.4. Exemple de différences

En fonction du langage ou de l'outil utilisé, il peut y avoir des différences, le but ici n'est pas de toutes les détailler mais de savoir que ça peut arriver pour que vous sachiez que dans certains cas, il faut faire une recherche dans les documentations pour voir comment contourner le problème.

4.4.1. Les groupes nommés

L'utilisation des groupes nommés fait partie des choses qui peuvent changer.

En Python par exemple, en utilisant le module `re`, si on souhaite utiliser un groupe nommé, on ne va pas écrire `(?<group>blabla)` mais `(?P<group>blabla)`.

En javascript, il se peut qu'en fonction du navigateur, il ne soit pas assez récent pour les gérer (de plus en plus rare).

On peut également trouver cette syntaxe : `(?'group'blabla)`, bref vous avez compris, ce sont des choses qui peuvent arriver.

4. Utilisation en milieu professionnel

4.4.2. L'utilisation des ensembles

Il se peut que dans certains cas, l'utilisation des ensembles se fasse autrement que présenté précédemment.

On peut trouver `[:digit:]` qui équivaut à `\d`, `[:alpha:]` équivalent de `[a-zA-Z]`...

Encore une fois, vous l'aurez compris, le but ici est uniquement de vous informer qu'il peut y avoir des différences en fonction de ce que l'on utilise.

4.5. La substitution

Une des fonctionnalités très intéressante et puissante des regex, c'est la substitution. Imaginez un fichier texte qui contient des dates au format `jj/mm/aaaa` et que vous devez toutes les passer au format `aaaa-mm-jj`. Si votre fichier fait quelques lignes, le faire à la main sera assez rapide, mais dans un fichier de plusieurs centaines de lignes, les regex peuvent nous faire gagner un temps fou.

i

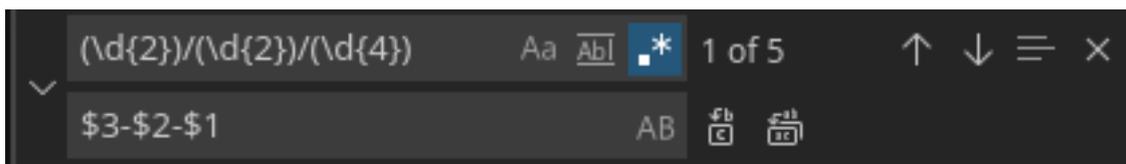
Dans mon cas j'utilise Visual Studio Code pour l'exemple qui suit, mais vous pouvez utiliser l'éditeur que vous préférez.

```
1 01/01/2000
2 07/08/2019
3 13/09/2020
4 21/12/2012
5 07/11/2022
```

La première étape consiste à écrire le format actuel de nos dates sans oublier les groupes capturant:

- le jour: `(\d{2})` (groupe 1);
- le mois: `(\d{2})` (groupe 2);
- l'année: `(\d{4})` (groupe 3).

Au complet nous avons donc: `(\d{2})\(/(\d{2})\(/(\d{4})`. Il suffit maintenant de faire un `Ctrl`+`h` pour écrire cette regex dans la partie haute et d'écrire `$3-$2-$1` dans la partie basse pour faire référence au groupe 3, 2 et 1.



Et là, magie! Vous avez bien changé le format de toutes les dates de votre fichier.

5. Entraînement

i

Il se peut que dans certains éditeurs, la référence à un groupe se fasse avec `\1` au lieu de `$1`.

i

Cet exemple est un travail taillé pour sed.

```
1 sed -E 's/([0-9]{2})\/([0-9]{2})\/([0-9]{4})\/\3-\2-\1/'
   date_file.txt
```

Vous pouvez également directement modifier le fichier et ne pas "échapper" le `/` avec la commande suivante:

```
1 sed -Ei 's!([0-9]{2})\/([0-9]{2})\/([0-9]{4})!\3-\2-\1!'
   date_file.txt
```

5. Entraînement

Pour nous entraîner, nous allons utiliser une fable de **La Fontaine**: *Le Corbeau et le Renard*.

Pour chaque question, essayez de faire une version sans Unicode et une avec.

```
1 Maître Corbeau, sur un arbre perché,
2 Tenait en son bec un fromage.
3 Maître Renard, par l'odeur alléché,
4 Lui tint à peu près ce langage :
5 Et bonjour, Monsieur du Corbeau,
6 Que vous êtes joli ! que vous me semblez beau !
7 Sans mentir, si votre ramage
8 Se rapporte à votre plumage,
9 Vous êtes le Phénix des hôtes de ces bois.
10 À ces mots le Corbeau ne se sent pas de joie,
11 Et pour montrer sa belle voix,
12 Il ouvre un large bec, laisse tomber sa proie.
13 Le Renard s'en saisit, et dit : Mon bon Monsieur,
14 Apprenez que tout flatteur
15 Vit aux dépens de celui qui l'écoute.
16 Cette leçon vaut bien un fromage sans doute.
17 Le Corbeau honteux et confus
18 Jura, mais un peu tard, qu'on ne l'y prendrait plus.
```

Exercices

Contenu masqué

1. Le premier mot de chaque ligne.
2. Les mots qui commencent par une majuscule mais qui ne sont pas en début de ligne.
3. Les mots de plus de 8 lettres.
4. Les mots qui ont au moins 3 voyelles d'affilées.
5. Les mots qui n'ont pas d'accent.

i

Pour chaque cas, il existe plusieurs manières de faire. Je ne donnerai donc qu'une solution pour chaque point.

👁️ Contenu masqué n°1

Maîtriser les expressions régulières permet de faire des tâches de tri ou de filtre plus efficacement et de manière plus complexe.

Maintenant que vous avez les bases, l'important est d'être curieux et d'essayer vous même de modifier certains exemples, voir ce qu'il se passe et vraiment bien comprendre le fonctionnement.

5.0.1. Remerciements

Merci à @kayou, @Yarflam, @SpaceFox, @QuentinC et @adril pour leurs remarques 🍌

Contenu masqué

Contenu masqué n°1

1. `^[a-zA-ZÀÉèèàîï]+`, Unicode: `^\p{L}+`.
2. `(?:.)([A-Z][a-zA-ZÀÉèèàîï]+)`, Unicode: `(?:.)(\p{Lu}\p{L}+)`.
3. `[a-zA-ZÀÉèèàîï]{8,}`, Unicode: `\p{L}{8,}`.
4. `[a-zA-ZÀÉèèàîï]*[aeiou]{3,}[a-zA-ZÀÉèèàîï]*`, Unicode: `\p{L}*[aeiou]{3,}\p{L}*`.
5. `\b[a-zA-Zç]+\b`.

[Retourner au texte.](#)