W6-S6-C1-Complement-expressions-regulieres

December 15, 2014

1 Expressions régulières et le module `re`

1.1 Complément - niveau intermédiaire

Une expression régulière est un objet mathématique permettant de décrire un ensemble de textes qui possèdent des propriétés communes ; par exemple, s'il vous arrive d'utiliser un terminal, et que vous tapez :

```
$ dir *.txt
```

l'expression régulière `*.txt` *filtre* toutes les chaînes qui se terminent par `.txt`

Le langage **Perl** a été le premier à populariser l'utilisation des expressions régulières en les supportant nativement dans le langage, et non au travers d'une librairie.

En python, les expressions régulières sont disponibles de manière plus traditionnelle, via le module `re` de la librairie standard, que nous allons voir maintenant.

Dans la commande ci-dessus, `*.txt` est une expression régulière très simple. Le module `re` fournit le moyen de construire des expressions régulières très élaborées et plus puissantes que ce que supporte le shell. C'est pourquoi la syntaxe des regexps de `re` est un peu différente ; par exemple pour rechercher (on dit encore filtrer, de l'anglais *pattern matching*) la même famille de chaînes que `*.txt` avec le module `re`, il nous faudra écrire l'expression régulière sous une forme légèrement différente.

Le propos de ce complément est de vous donner une première introduction au module `re`.

```
In []: import re
```

Je vous conseille d'avoir sous la main la documentation du module `re` pendant que vous lisez ce complément.

1.1.1 Avertissement

Dans ce complément nous serons amenés à utiliser des traits qui dépendent du LOCALE, c'est-à-dire, pour faire simple, de la configuration de l'ordinateur vis-à-vis de la langue.

Tant que vous exécutez ceci dans le notebook sur la plateforme, en principe tout le monde verra exactement la même chose. Par contre, si vous faites tourner le même code sur votre ordinateur, il se peut que vous obteniez des résultats différents.

1.1.2 Un exemple simple

`findall` On se donne deux exemples de chaînes :

```
In []: sentences = ['Lacus a donec, + vitae gravida proin sociis.',
                    'Neque ipsum! rhoncus cras quam.']
```

On peut **chercher tous** les mots se terminant par a ou m dans une chaîne avec `findall` :

```
In []: for sentence in sentences:
           print 4*'-','dans >{}<'.format(sentence)
           print re.findall(r"\w*[am]\W", sentence)
```

Ce code permet de chercher toutes (`findall`) les occurrences de l'expression régulière, qui ici est définie par le *raw-string* :

```
r"\w*[am]\W"
```

Nous verrons tout à l'heure comment fabriquer des expressions régulières plus en détail, mais pour démystifier au moins celle-ci, on a mis bout à bout les morceaux suivants : * `\w*` : il nous faut trouver une sous-chaîne qui commence par un nombre quelconque, y compris nul (`*`) de caractères alphanumériques (`\w`) ; ceci est défini en fonction de votre LOCALE, on y reviendra ; * `[am]` : immédiatement après, il nous faut trouver un caratère `a` ou `m`; * `\W` : et enfin, il nous faut un caractère qui ne soit pas alphanumérique. Ceci est important puisqu'on cherche les mots qui **se terminent** par un `a` ou un `m`; si on ne le mettait pas on obtiendrait ceci :

```
In []: # le \W final est important
       # voici ce qu'on obtient si on l'omet
       for sentence in sentences:
           print 4*'-','dans >{}<'.format(sentence)
           print re.findall(r"\w*[am]", sentence)
```

`split` Une autre forme simple d'utilisation des regexps est `re.split`, qui fournit une **fonctionnalité voisine de** `str.split` qu'on a vue en semaine 2, mais ou les séparateurs sont exprimés comme une expression régulière ; ou encore on peut le voir un peu comme la négation de `findall` :

```
In []: for sentence in sentences:
           print 4*'-','dans >{}<'.format(sentence)
           print re.split(r"\W+", sentence)
```

Ici l'expression régulière, qui donc décrit le séparateur, est simplement `\W+` c'est-à-dire toute suite d'au moins un caractère non alphanumérique.

Nous avons donc là un moyen simple, et plus puissant que `str.split`, de couper un texte en mots.

`sub` Une troisième méthode utilitaire est `re.sub` qui permet de remplacer les occurrences d'une *regexp*, comme par exemple :

```
In []: for sentence in sentences:
           print 4*'-','dans >{}<'.format(sentence)
           print re.sub(r"(\w+)", r"X\1X", sentence)
```

Ici, l'expression régulière (le premier argument) contient un **groupe** : on a utilisé des parenthèses autour du `\w+`. Le second argument est la **chaîne de remplacement**, dans laquelle on a fait **référence au groupe** en écrivant `\1`, qui veut dire tout simplement "le premier groupe".

Donc au final l'effet de cet appel est d'entourer toutes les suites de caractères alphanumériques par une paire de `X`.

Pourquoi un *raw-string* ? En guise de digression, il n'y a aucune obligation à utiliser un *raw-string* ; d'ailleurs on rappelle qu'il n'y a pas de différence de nature entre un *raw-string* et une chaîne usuelle :

```
In []: raw = r'abc'
       regular = 'abc'
       # comme on a pris une 'petite' chaîne ce sont les mêmes objets
       print 'is', raw is regular
       # et donc a fortiori
       print '==', raw == regular
```

Il se trouve que, comme dans notre premier exemple, le *backslash* \ à l'intérieur des expressions régulières est d'un usage assez courant. C'est pourquoi on **utilise fréquemment un *raw-string*** pour décrire une expression régulière, et en général à chaque fois qu'elle comporte un *backslash*.

1.1.3 (Digression : petits outils d'affichage)

Avant de voir un deuxième exemple, nous allons digresser à nouveau, et pour améliorer la présentation, nous allons nous écrire deux petits outils de mise en page, qui n'ont rien à voir avec les expressions régulières *per se* :

```
In []: # mettre en colonnes les inputs et les regexps
       def i_r(input, cols_in, regexp=None, cols_re=0):
           if regexp:
               return "IN={} - RE={} ->".format(input.rjust(cols_in),
                                                regexp.ljust(cols_re))
           else:
               return "IN={} ->".format(input.rjust(cols_in))

       # afficher 'Match' ou 'None' plutôt que l'objet Match
       def m(match):
           return 'MATCH' if match else 'Nope'
```

Nous utiliserons `i_r` avec des tailles de colonnes choisies à la main, ce qui donnera quelque chose comme :

```
In []: input = 'aaabbb'
       regexp = 'a*b+'
       print i_r(input, 6, regexp, 4), m(re.match(regexp, input))
```

Ici 6 et 4 représentent la largeur des colonnes pour afficher `aaabbb` et `a*b+` respectivement. Nous allons voir plus bas pourquoi cette expressions régulière filtre effectivement cette entrée.

1.1.4 Un deuxième exemple

Reprenons, nous allons maintenant voir comment on peut d'abord vérifier si une chaîne est conforme au critère défini par l'expression régulière, mais aussi *extraire* les morceaux de la chaîne qui correspondent aux différentes parties de l'expression.

Pour cela, supposons qu'on s'intéresse aux chaînes qui comportent 5 parties, une suite de chiffres, une suite de lettres, des chiffres à nouveau, des lettres, et enfin de nouveau des chiffres.

Pour cela on considère ces trois chaines en entrée :

```
In []: inputs = ['890hj000nnm890',    # cette entrée convient
                 '123abc456def789',   # celle-ci aussi
                 '8090abababab879',   # celle-ci non
                 ]
```

`match` Pour commencer, voyons que l'on peut facilement **vérifier si une chaîne vérifie** ou non le critère :

```
In []: regexp1 = "[0-9]+[A-Za-z]+[0-9]+[A-Za-z]+[0-9]+"
```

Ce qui nous donne, en utilisant les helpers `i_r` et `m` pour la mise en page :

```
In []: for input in inputs:
           print i_r(input, 15), m(re.match(regexp1, input))
```

Ici plutôt que d'utiliser les raccourcis comme `\w` j'ai préféré écrire explicitement les ensembles de caractères en jeu ; de cette façon, on rend son code indépendant du LOCALE si c'est ce qu'on veut faire. Il y a deux morceaux qui interviennent tour à tour : * `[0-9]+` signifie une suite de au moins un caractère dans l'intervalle `[0-9]`, * `[A-Za-z]+` pour une suite d'au moins un caractère dans l'intervalle `[A-Z]` ou dans l'intervalle `[a-z]`.

Et comme tout à l'heure on a simplement juxtaposé les morceaux dans le bon ordre pour construire l'expression régulière complète.

Nommer un morceau (un groupe)

```
In []: # on se concentre sur une entrée correcte
       haystack = inputs[1]
       haystack
```

Maintenant, on va même pouvoir **donner un nom** à un morceau de la regexp, ici `needle` :

```
In []: # la même regexp, mais on donne un nom au groupe de chiffre central
       regexp2 = "[0-9]+[A-Za-z]+(?P<needle>[0-9]+)[A-Za-z]+[0-9]+"
```

Et une fois que c'est fait, on peut demander à l'outil de nous **retrouver la partie correspondante** dans la chaine initiale :

```
In []: print re.match(regexp2, haystack).group('needle')
```

Dans cette expression on a utilisé un **groupe nommé** `(?P<needle>[0-9]+)`, dans lequel : * les parenthèses définissent un groupe, * `?P<needle>` spécifie que ce groupe pourra être référencé sous le nom `needle`. J'aimerais pouvoir vous donner une explication plus convaincante au sujet de cette syntaxe, mais je n'en connais pas.

1.1.5 Un troisième exemple

Enfin, et c'est un trait qui n'est pas présent dans tous les langages, on peut restreindre un morceau de chaîne à être identique à un groupe déjà vu plus tôt dans la chaîne ; dans l'exemple ci-dessus on pourrait ajouter comme contrainte que le premier et le dernier groupes de chiffres soient identiques, comme ceci :

```
In []: regexp3 = "(?P<id>[0-9]+)[A-Za-z]+(?P<needle>[0-9]+)[A-Za-z]+(?P=id)"
```

Si bien que maintenant, avec les mêmes entrées que tout à l'heure :

```
In []: for input in inputs:
           print i_r(input,15), m(re.match(regexp3, input))
```

Comme précédemment on a défini le groupe nommé `id` comme étant la première suite de chiffres. La nouveauté ici est la **contrainte** qu'on a imposée sur le dernier groupe avec `(?P=id)`. Comme vous le voyez, on n'obtient un *match* qu'avec les entrées dans lesquelles le dernier groupe de chiffres est identique au premier.

1.1.6 Comment utiliser la librairie

Avant d'apprendre à écrire une expression régulière, disons quelques mots du mode d'emploi de la librairie.

Fonctions de commodité et *workflow* Comme vous le savez peut-être, une expression régulière décrite sous forme de chaîne, comme par exemple `"\w*[am]\W"`, peut être traduite dans un **automate fini** qui permet de faire le filtrage avec une chaîne. C'est ce qui explique le *workflow* que nous avons résumé dans cette figure :

La méthode recommandée pour utiliser la librairie, lorsque vous avez le même *pattern* à appliquer à un grand nombre de chaînes, est de : * compiler **une seule fois** votre chaîne en un automate, qui est matérialisé par un objet de la classe `re.RegexObject`, en utilisant `re.compile`, * puis d'**utiliser directement cet objet** autant de fois que vous avez de chaînes.

Nous avons utilisé dans les exemples plus haut (et nous continuerons plus bas pour une meilleure lisibilité) des **fonctions de commodité** du module, qui sont pratiques, par exemple, pour mettre au point une expression régulière en mode interactif, mais qui ne **sont pas forcément** adaptées dans tous les cas.

Ces fonctions de commodité fonctionnent toutes sur le même principe :

`re.match(regexp, input)` $\Longleftrightarrow$ `re.compile(regexp).match(input)`

Donc à chaque fois qu'on utilise une fonction de commodité, on recompile la chaîne en automate, ce qui, dès qu'on a plus d'une chaîne à traiter, représente un surcoût.

C'est ainsi qu'au lieu de faire comme plus haut :

```
In []: # imaginez 10**4 chaînes dans inputs
       for input in inputs:
           print i_r(input,15), m(re.match(regexp3, input))
```

dans du vrai code on fera plutôt :

```
In []: # on compile la chaîne en automate une seule fois
       re_obj3 = re.compile(regexp3)

       # ensuite on part directement de l'automate
       for input in inputs:
           print i_r(input,15), m(re_obj3.match(input))
```

qui ne compile qu'une fois la chaîne en automate, et donc est plus efficace.

Les méthodes sur la classe `RegexObject` Les objets de la classe `RegexObject` représentent donc l'automate à état fini qui est le résultat de la compilation de l'expression régulière. Pour résumer ce qu'on a déjà vu, les méthodes les plus utiles sur un objet `RegexObject` sont : * `match` et `search`, qui cherchent un *match* soit uniquement au début (`match`) ou n'importe où dans la chaîne (`search`); * `findall` et `split` pour chercher toutes les occurences (`findall`) ou leur négatif (`split`); * `sub` (qui aurait pu sans doute s'appeler `replace`, mais c'est comme ça) pour remplacer les occurrences de pattern.

Exploiter le résultat Les **méthodes** disponibles sur la classe `re.MatchObject` sont documentées en détail ici. On en a déjà rencontré quelques-unes, en voici à nouveau un aperçu rapide sur cet exemple très simple :

```
In []: # exemple
       input = "    Isaac Newton, physicist"
       match = re.search(r"(\w+) (?P<name>\w+)", input)
```

- `re` et `string` pour retrouver les données d'entrée du match :

```
In []: match.string
```

```
In []: match.re
```

- `group`, `groups`, `groupdict` pour retrouver les morceaux de la chaîne d'entrée qui correspondent aux **groupes** de la regexp; on peut y accéder par rang, ou par nom (comme on l'a vu plus haut avec `needle`);

```
In []: match.groups()
```

```
In []: match.group(1)
```

```
In []: match.group('name')
```

```
In []: match.group(2)
```

```
In []: match.groupdict()
```

Comme on le voit pour l'accès par rang **les indices commencent à 1** pour des raisons historiques (on peut déjà référencer \1 en sed depuis la fin des années 70).

On peut aussi accéder au **groupe 0** comme étant la partie de la chaîne de départ qui a effectivement été filtrée par l'expression régulière, et qui peut tout à fait être au beau milieu de la chaîne de départ, comme dans notre exemple :

```
In []: match.group(0)
```

- `expand` permet de faire une espèce de `str.format` avec les valeurs des groupes;

```
In []: match.expand(r"last_name \g<name> first_name \1")
```

- `span` pour connaître les index dans la chaîne d'entrée pour un groupe donné :

```
In []: begin, end = match.span('name')
       input[begin:end]
```

Les différents modes (*flags*) Enfin il faut noter qu'on peut passer à `re.compile` un certain nombre de *flags* qui modifient globalement l'interprétation de la chaîne, et qui peuvent rendre service.

Vous trouverez une liste exhaustive de ces *flags* ici ; ils ont en général un nom long et parlant, et un alias court sur un seul caractère. Les plus utiles sont sans doute : * `IGNORECASE` (*alias* `I`): pour, eh bien, ne pas faire la différence entre minuscules et majuscules, * `UNICODE` (*alias* `U`): pour rendre les séquences `\w` et autres basées sur les propriétés des caractères dans la norme Unicode, * `LOCALE` (*alias* `L`): cette fois `\w` dépend du `locale` courant. * `MULTILINE` (*alias* `M`), et * `DOTALL` (*alias* S): pour ces deux flags voir la discussion à la fin du complément.

Comme c'est souvent le cas, on doit passer à `re.compile` un **ou logique** (caractère |) des différents flags que l'on veut utiliser, c'est-à-dire qu'on fera par exemple :

```
In []: re_obj = re.compile("a*b+", flags=re.IGNORECASE | re.DEBUG)

In []: print m(re_obj.match("AabB"))
```

1.1.7 Comment construire une expression régulière

Nous pouvons à présent passer en revue les constructions qui permettent d'élaborer une expression régulière, en tentant de rester synthétique puisque la documentation du module `re` en donne une version exhaustive.

La brique de base : le caractère Au commencement il faut spécifier des caractères : * **un seul** caractère : * vous le citez tel quel, en le précédent d'un backslash \ s'il a par ailleurs un sens spécial dans le micro-langage de regexps (comme `+`, `*`, `[`, etc.) ; * le **wildcard** : * un point `.` signifie "n'importe quel caractère" ; * **un ensemble** de caractères avec la notation `[...]` qui permet de décrire par exemple : * `[a1=]` un ensemble in extenso, ici un caractère parmi `a`, `1`, ou `=`, * `[a-z]` un intervalle de caractères, ici de `a` à `z`, * `[15e-g]` un mélange des deux, ici un ensemble qui contiendrait `1`, `5`, `e`, `f` et `g`, * `[^15e-g]` une **négation**, qui a `^` comme premier caractère dans les `[]`, ici tout sauf l'ensemble précédent ; * un **ensemble prédéfini** de caractères, qui peuvent alors dépendre de l'environnement (UNICODE et LOCALE) avec entre autres les notations : * `\w` les caractères alphanumériques, et `\W` (les autres), * `\s` les caractères "blancs" - espace, tabulation, saut de ligne, etc., et `\S` (les autres), * `\d` pour les chiffres, et `\D` (les autres).

```
In []: input = "abcd"

       for regexp in ['abcd', 'ab[cd][cd]', r'abc.', r'abc\.']:
           print i_r(input, 4, regexp,10), m(re.match(regexp, input))
```

Pour ce dernier exemple, comme on a backslashé le `.` il faut que la chaîne en entrée contienne vraiment un `.` :

```
In []: print m(re.match (r"abc\.", "abc."))
```

En série ou en parallèle Si je fais une analogie avec les montages électriques, jusqu'ici on a vu le montage en série : on met des expressions régulières bout à bout, qui filtrent (`match`) la chaine en entrée séquentiellement du début à la fin. On a *un peu* de marge pour spécifier des alternatives, lorsqu'on fait par exemple

```
"ab[cd]ef"
```

mais c'est limité à **un seul** caractère. Si on veut reconnaitre deux mots qui n'ont pas grand-chose à voir comme `abc` **ou** `def`, il faut en quelque sorte mettre deux regexps en parallèle, et c'est ce que permet l'opérateur |

```
In []: regexp = "abc|def"

       for input in ['abc', 'def', 'aef']:
           print i_r(input, 3, regexp,7), m(re.match(regexp, input))
```

Fin(s) de chaîne Selon que vous utilisez `match` ou `search`, vous précisez si vous vous intéressez uniquement à un match en début (`match`) ou n'importe où (`search`) dans la chaîne.

Mais indépendamment de cela, il peut être intéressant de "coller" l'expression en début ou en fin de ligne, et pour ça il existe des caractères spéciaux : * `^` lorsqu'il est utilisé comme un caractère (c'est à dire pas en début de `[]`) signifie un début de chaîne ; * `\A` a le même sens (sauf en mode MULTILINE), et je le recommande de préférence à `^` qui est déjà pas mal surchargé ; * `$` matche une fin de ligne ; * `\Z` est voisin mais pas tout à fait identique.

Reportez-vous à la documentation pour le détails des différences. Attention aussi à entrer le `^` correctement, il vous faut le caractère ASCII et non un voisin dans la ménagerie Unicode.

```
In []: input = 'abcd'

       for regexp in [ 'bc', r'\Aabc', '^abc', r'\Abc', '^bc', r'bcd\Z', 'bcd$', r'bc\Z', 'bc$' ]:
           print i_r(input, 4, regexp, 5), m(re.search(regexp,input))
```

On a en effet bien le pattern `bc` dans la chaine en entrée, mais il n'est ni au début ni à la fin.

Parenthéser - (grouper) Pour pouvoir faire des montages élaborés, il faut pouvoir parenthéser :

```
In []: # une parenthése dans une RE pour mettre en ligne
       # un début 'a', deux possibilités pour le milieu 'bc' ou 'de'
       # et une fin 'f'
       regexp = "a(bc|de)f"

       for input in ['abcf', 'adef', 'abf']:
           print i_r(input, 4, regexp, 9), m(re.match(regexp, input))
```

Les parenthèses jouent un rôle additionel de **groupe**, ce qui signifie qu'on **peut retrouver** le texte correspondant à l'expression régulière comprise dans les `()`. Par exemple, pour le premier match :

```
In []: input = 'abcf'
       print i_r(input, 4, regexp, 9), re.match(regexp, input).groups()
```

dans cet exemple, on n'a utilisé qu'un seul groupe `()`, et le morceau de chaîne qui correspond à ce groupe se trouve donc être le seul groupe retourné par `MatchObject.group`.

Compter les répétitions Vous disposez des opérateurs suivants : * `*` l'étoile qui signifie n'importe quel nombre, même nul, d'occurrences - par exemple, `(ab)*` pour indiquer `''` ou `'ab'` ou `'abab'` ou etc., * `+` le plus qui signifie au moins une occurrence - e.g. `(ab)+` pour `ab` ou `abab` ou `ababab` ou etc. * `?` qui indique une option, c'est-à-dire 0 ou 1 occurence - e.g. `(ab)?` pour `''` ou `ab`, * {`n`} pour exactement n occurrences de `(ab)` - e.g. `(ab)`{`3`} qui serait exactement équivalent à `ababab`; * {`m,n`} entre m et n fois inclusivement, c'est-à-dire que `*<re>*?` est exactement équivalent à `*<re>*`{`0,1`}.

```
In []: inputs = [n*'ab'for n in [0, 1, 3, 4]] + ['foo']

       for regexp in ['(ab)*', '(ab)+', '(ab){3}', '(ab){3,4}']:
```

```
    # on ajoute \A \Z pour matcher toute la chaine
    line_regexp = r"\A{}\Z".format(regexp)
    for input in inputs:
        print i_r(input, 8, line_regexp, 13), m(re.match(line_regexp, input))
```

Groupes et contraintes Nous avons déjà vu un exemple de groupe nommé (voir `needle` plus haut) ; les opérateurs que l'on peut citer dans cette catégorie sont : * `(...)` les parenthèses définissent un groupe anonyme, * `(?P<name>...)` définit un groupe nommé, * `(?:...)` permet de mettre des parenthèses mais sans créer un groupe, pour optimiser l'exécution puisqu'on n'a pas besoin de conserver les liens vers la chaîne d'entrée, * `(?P=name)` qui ne matche que si l'on retrouve à cet endroit de l'entrée la même sous-chaîne que celle trouvée pour le groupe `name` en amont, * enfin `(?=...)`, `(?!...)`et `(?<=...)` permettent des contraintes encore plus élaborées, nous vous laissons le soin d'expérimenter avec elles si vous êtes intéressés ; sachez toutefois que l'utilisation de telles constructions peut en théorie rendre l'interprétation de votre expression régulière beaucoup moins efficace.

Greedy *vs* non-greedy Lorsqu'on stipule une répétition un nombre indéfini de fois, il se peut qu'il existe **plusieurs** façons de filtrer l'entrée avec l'expression régulière. Que ce soit avec `*`, ou `+`, ou `?`, l'algorithme va toujours essayer de trouver la **séquence la plus longue**, c'est pourquoi on qualifie l'approche de *greedy* - quelque chose comme glouton en français :

In []:
```
# un fragment d'HTML
line='<h1>Title</h1>'

# si on cherche un texte quelconque entre crochets
# c'est-à-dire l'expression régulière "<.*>"
re_greedy = '<.*>'

# on obtient ceci
# on rappelle que group(0) montre la partie du fragment
# HTML qui matche l'expression régulière
match = re.match(re_greedy, line)
match.group(0)
```

Ça n'est pas forcément ce qu'on voulait faire ; aussi on peut spécifier l'approche inverse, c'est-à-dire de trouver la **plus-petite** chaîne qui matche, dans une approche dite *non-greedy*, avec les opérateurs suivants : * `*?` : `*` mais *non-greedy*, * `+?` : `+` mais *non-greedy*, * `??` : `?` mais *non-greedy*,

In []:
```
# ici on va remplacer * par *? pour rendre l'opérateur * non-greedy
re_non_greedy = re_greedy = '<.*?>'

# mais on continue à cherche un texte entre <> naturellement
# si bien que cette fois, on obtient
match = re.match(re_non_greedy, line)
match.group(0)
```

S'agissant du traitement des fins de ligne Il peut être utile, pour conclure cette présentation, de préciser un peu le comportement de la librairie vis-à-vis des fins de ligne.

Historiquement, les expressions régulières telles qu'on les trouve dans les librairies C, donc dans `sed`, `grep` et autre utilitaires Unix, sont associées au modèle mental où on filtre les entrées ligne par ligne.

Le module `re` en garde des traces, puisque :

In []:
```
# un exemple de traitement des 'newline'
input = u"""une entrée
sur
plusieurs
```

```
lignes
"""
```

```
In []: match = re.compile("(.*)").match(input)
       match.groups()
```

Vous voyez donc que l'attrape-tout `'.'` en fait n'attrape pas le caractère de fin de ligne `\n`, puisque si c'était le cas et compte tenu du coté *greedy* de l'algorithme on devrait voir ici tout le contenu de `input`. Il existe un *flag* `re.DOTALL` qui permet de faire de `.` un vrai attrape-tout qui capture aussi les *newline* :

```
In []: match = re.compile("(.*)", flags=re.DOTALL).match(input)
       match.groups()
```

Cela dit, le caractère *newline* est par ailleurs considéré comme un caractère comme un autre, on peut le mentionner **dans une regexp** comme les autres ; voici quelques exemples pour illustrer tout ceci :

```
In []: # sans mettre le flag unicode \w ne matche que l'ASCII
       match = re.compile("([\w ]*)").match(input)
       match.groups()
```

```
In []: # sans mettre le flag unicode \w ne matche que l'ASCII
       match = re.compile("([\w ]*)", flags=re.U).match(input)
       match.groups()
```

```
In []: # si on ajoute \n à la liste des caractères attendus
       # on obtient bien tout le contenu initial

       # attention ici il ne FAUT PAS utiliser un raw string,
       # car on veut vraiment écrire un newline dans la regexp

       match = re.compile("([\w \n]*)", flags=re.UNICODE).match(input)
       match.groups()
```

1.1.8 Conclusion

La mise au point d'expressions régulières est certes un peu exigeante, et demande pas mal de pratique, mais permet d'écrire en quelques lignes des fonctionnalités très puissantes, c'est un investissement très rentable :)

Je vous signale enfin l'existence de **sites web** qui évaluent une expression régulière **de manière interactive** et qui peuvent rendre la mise au point moins fastidieuse. Pour être honnête je n'ai pas d'expérience préalable avec ces sites, mais en préparant le cours j'ai essayé rapidement regex101.com qui semble à première vue très bien fait; prenez garde toutefois à bien choisir le mode python dans la bannière de gauche car chaque langage de regexps a ses petites différences (ce ne serait pas drôle sinon:).

1.1.9 Pour en savoir plus

Pour ceux qui ont quelques rudiments de la théorie des langages, vous savez qu'on distingue en général * l'**analyse lexicale**, qui découpe le texte en morceaux (qu'on appelle des *tokens*), * et l'**analyse syntaxique** qui décrit pour simplifier à l'extrême l'ordre dans lequel on peut trouver les tokens.

Avec les expression régulières, on adresse le niveau de l'analyse lexicale. Pour l'analyse syntaxique, qui est franchement au delà des objectifs de ce cours, il existe de nombreuses alternatives, parmi lesquelles : * `pyparsing` * `PLY` (Python Lex-Yacc) * `ANTLR` qui est un outil écrit en Java mais qui peut générer des parsers en python, * ...