regexp

October 15, 2021

1 Les expressions régulières ou expressions rationnelles

Elles sont aussi souvent désignées par regex ou regexp. L'idée générale esst de capturer via un langage de motif un ensemble de mots ayant une certaine forme, comme par exemple les mots qui finissent en `er` ou ceux qui ne contiennent pas de `x`, ou encore ceux qui ne contiennent qu'un seul `y`, pas de `z`commencent par `t e` et ne finissent pas par `q`.

> Ces ensembles de mots reconnaissables forment des structures mathématiques particulières (on en dira rien ici), mais il existe un pan entier de travaux mathématico-informatiques sur la question de la structure mathématique des langages formels. Peut-être, en tant que linguiste, avez-vous entendu parler de la «hiérarchie de Chomsky-Schützenberger» qui catégorise les langages formels ? Note : M.-P. Schützenberger était un chercheur français pionnier de l'informatique fondamentale et qui était professeur à Paris 7. Les langages que l'on peut reconnaitre à l'aide des expressions régulières sont les langages réguliers et qui sont tout en bas de la hiérarchie. Cette hiérarchie donne des informations sur le type de calcul nécessaire afin de reconnaître ces différents langages. Reconnaître signifie ici qu'étant donné un mot on puisse dire s'il appartient au langage donné (attention il s'agit de langages dont le nombre de mots est infini).

Chaque langage (ou presque) propose dans l'une de ses bibliothèque de quoi manipuler des expressions régulières car leur usage est très répandu en informatique : reconnaître si une entrée correspond à l'écriture d'un nombre par exemple. Les langages permettant l'écriture des expressions régulières diffèrent parfois dans la syntaxe, mais jamais dans le fond.

Tout d'abord un petit exemple

[1]:
```
import re

automate = re.compile("ab");
ok = automate.search("bonjour"); print(ok)
ok = automate.search("ab"); print(ok)
ok = automate.search("c'est avec abnégation que blablabla"); print(ok)
```

```
None
<re.Match object; span=(0, 2), match='ab'>
<re.Match object; span=(11, 13), match='ab'>
```

La première ligne permet d'importer le module (la bibliothèque) `re` des fonctions de manipulation des expressions régulières.

La seconde permet d'obtenir la «machinerie» interne qui permettra de tenter de déterminer plus

tard si des mots font partie du langage (on appelle langage un ensemble de mots). Le langage décrit ici est réduit au mot `ab`! C'est un exemple…

La troisième ligne permet de déterminer si un mot (`bonjour`) appartient au langage correspondant à la «machinerie» puis affiche le résultat. `None` indique donc que le mot `bonjour` n'appartient pas au langage ne contenant que le mot `ab`.

La quatrième ligne permet elle de déterminer si le mot `ab` appartient au langage. L'affichage indique que oui et donne des informations techniques sur la reconnaissance obtenue (ce n'est pas très important ici).

La cinquième ligne illustre le fait que la fonction `search` permet non pas de déterminer si le mot donné est exactement un mot du langage, mais si le texte donné contient un mot du langage. En effet, `ab` est bien contenu dans le texte donné en argument.

La question est donc maintenant : «puis exprimer des langages plus compliqués ?». Oui!

> Note : la «machinerie» interne dont on parle s'apelle un automate fini (finite automaton).

Soit l'exemple suivant :

[2]:
```
regexp = "^a*$"
automate = re.compile(regexp)
```

Le langage capturé ici est l'ensemble des mots ne contenant que des `a`. Pour le décrire on a utilisé les caractères `^`, `*`, `$` qui ont donc un pouvoir spécial. On parle de *wildcards* ou de *jokers*, c'est-à-dire que ces caractères ne représentent pas eux-mêmes mais une fonctionnalité particulière.

- le caractère `*` désigne la répétition du motif précédent un nombre quelconque de fois (0 fois, 1 fois, 2 fois, etc). Ainsi `a*` designe le langage contenant le mot vide (noté), le mot `a`, le mot `aa`, le mot `aaa`, etc. On a donc obtenu un langage infini (même si sa structure est simple).
- le caractère `^` désigne le début du mot à reconnaître.
- le caractère `$` désigne la fin du mot à reconnaître.

Les deux derniers servent à reconnaître les mots du langage avec exactitude, en effet on veut parfois reconnaître dans un texte l'apparition d'un mot du langage (on verra plus loin).

Essayons de reconnaître des mots :

[3]:
```
mots = [ "zorglub", "asterix", "aaaaa", "baaaa", "", "aaaaaaaaaaaaaaaaa",␣
 ↪"assez", "avancez", "plein" ]
for m in mots:
    if automate.search(m)!=None:
        print("Le mot '"+m+"' appartient au langage "+regexp)
```

```
Le mot 'aaaaa' appartient au langage ^a*$
Le mot '' appartient au langage ^a*$
Le mot 'aaaaaaaaaaaaaaaaa' appartient au langage ^a*$
```

Un autre caractère *magique* est la caractère `.` qui permet de désigner un caractère quelconque :

[4]:
```
regexp = "^a.*z$"
automate = re.compile(regexp)
for m in mots:
    if automate.search(m)!=None:
        print("Le mot '"+m+"' appartient au langage "+regexp)
```

```
Le mot 'assez' appartient au langage ^a.*z$
Le mot 'avancez' appartient au langage ^a.*z$
```

Il est temps d'introduire une autre écriture Python pour les listes. Il s'agit des listes exprimées en intention (comprehensive lists). En voici un exemple :

[5]:
```
t = [m for m in mots if automate.search(m)]
print("Les mots %s appartiennent au langage %s" % (t,regexp))
```

```
Les mots ['assez', 'avancez'] appartiennent au langage ^a.*z$
```

L'idée est de fabriquer la liste des mots reconnus, c'est donc la liste d'origine mais filtrée (si le mot est reconnu on le garde, sinon non). La première ligne est donc assez parlante : `t` est la liste des mots `m` tels que pout tout `m` de la liste des `mots` il vérifie la condition de reconnaissance `automate.search(m)`.

Une autre construction d'expression régulière est l'expression d'un choix dans un ensemble de caractères. Par exemple :

[6]:
```
regexp = "^[zp]"
automate = re.compile(regexp)
t = [m for m in mots if automate.search(m)]
print("Les textes %s contiennent un mot du langage %s" % (t,regexp))
```

```
Les textes ['zorglub', 'plein'] contiennent un mot du langage ^[zp]
```

Tout d'abord `[zp]` qui désigne un caractère parmi `z` ou `p`.

Ensuite, vous l'avez peut-être remarqué nous avons supprimé `$` et changé la formulation de l'affichage du résultat. En effet, cette fois en enlevant `$` nous n'imposons plus que la reconnaissance se termine exactement à la fin du texte, mais simplement quelle se termine. Ainsi notre regexp désigne l'ensemble des textes qui commencent par `z` ou `p`.

Le Joker `^` a aussi un autre sens, car placé juste derrière `[` il exprime que la liste des caractères est définie par négation, ainsi `[^zp]` signifie ni `z`, ni `p` :

[7]:
```
regexp = "^[^zp]"
automate = re.compile(regexp)
t = [m for m in mots if automate.search(m)]
print("Les textes %s contiennent un mot du langage %s" % (t,regexp))
```

```
Les textes ['asterix', 'aaaaa', 'baaaa', 'aaaaaaaaaaaaaaaaa', 'assez',
'avancez'] contiennent un mot du langage ^[^zp]
```

Il existe de nombreuses autres constructions, mais vous les découvrirez en lisant la documentation des expressions régulières en Python (obtenir l'ensemble des mots reconnus dans un texte, les

positions où ils sont reconnus, la reconnaissance gloutonne, etc). Pour l'instant cela suffit déjà à expérimenter de nombreuses choses.

1.0.1 Exercice

- Prendre une liste de mots français (que vous pouvez récupérer ici et en extraire l'ensemble des mots finissant en `er` (donc potentiellement des verbes du premier groupe).
- Sauriez-vous modifier tous ces mots, en considérant qu'ils sont tous des verbes du premier groupe même si ce n'est pas le cas, de sorte à obtenir leur conjugaison à la première personne du pluriel du subjonctif ?
- Sauriez-vous prendre ces mots comme des verbes et les conjuquer à tous les temps et toutes les personnes ?

Pour toutes ces questions, oubliez les cas particuliers! Il s'agit d'un exercice, mais si vous voulez vous pouvez tentez votre chance en considérant les cas des verbes en `yer`, `eyer` et autres joyeuses exceptions… On voit bien ici la différence entre les langages formels qui ont une régularité stricte et les langues naturelles qui sont pleines d'exceptions et étrangetés difficiles à capter par calcul.

[ ]: