Python : expressions régulières

Pascal Vanier

Python — M1 Informatique, FST, UPEC, 2015/2016

Exercices

Exercice Écrivez un générateur qui génère tous les nombres premiers.

Exercice Écrivez une fonction `tri_bulle(liste)` qui trie `liste`. Vous vous servirez des index.

Expressions régulières

Le module `re` fait partie des modules de base de python.

Imaginons que l'on veuille trouver un mot ne commençant pas par Z dans un texte :

```
1  import re
2
3  z=re.compile(r"[bB][a-zA-Z]*") # chaines commençant par b ou B
4                                 # suivies de n'importe quelles
5                                 # lettres de a-z ou A-Z
6  w=re.compile(r"[\w]+") # mots d'au moins une lettre de \w toutes
7                         # ie. toutes les lettres (y compris accents)
8
9  phrase = "Zorro zozotte contrairement à Bernardo"
10
11 w.findall(phrase) # on cherche tous les w dans phrase
12                  # et renvoie une liste de ceux-ci
13
14 z.search(phrase) # on cherche la premiere occurence de z
15                 # dans phrase
16
17 z.match("Bernardo") # renvoie None si aucun prefixe n'est
18                     # reconnaissable par l'expression
19
20 h=w.finditer(phrase) # comme findall mais renvoie un itérateur
```

Expressions régulières

- `match` et `search` renvoient des **match objects**.
- `findall()` renvoie toutes les sous chaines qui vérifient l'expression régulière sous forme d'une liste.
- `finditer()` fait comme `findall()` mais renvoie un itérateur.

On peut appliquer les méthodes suivantes à un **match object**:

```
1  e = re.compile(r"P[\w]+")
2  m = e.search("hakuna Patata")
3
4  if m:
5      m.start() # 7
6      m.group() # "Patata"
7      m.end()   # 13
8      m.span()  # (7,13)
9  else:
10     print("Pas de correspondance")
```

Expressions régulières

Les `[ ]` déterminent un sous ensemble de lettres :

```
[ski] # les lettres s k et i
[a-z] # les lettres de a à z = [abcdefghijklmnopqrtuvxyz]
[^a-z] # tout sauf [a-z]
```

On a des groupes de caractères :

```
.  # tous les caractères sauf \n
\w # les lettres (unicode) qui peuvent être dans un mot,
   # y compris les chiffres et _
\W # les caractères qui ne sont pas dans \w
\s # les caractères d'espacement (espace, tabulation, retour
   # à la ligne...)
\S # les caractères qui ne sont pas dans \s
\d # les chiffres (unicode)
\D # les caractères qui ne sont pas dans \d
\b # caractère vide mais uniquement au début ou à la fin
   # d'un mot
\B # caractère vide mais pas au début et à la fin d'un mot.
\A # début de la chaine de caractères = ^
\Z # fin de la chaine de caractères = $
```

On peut composer :

```
[\d\s] # les chiffres et les espaces
```

Expressions régulières : construction

- `expr1|expr2` on matche soit `expr1` soit `expr2`
- `expr*` on matche une répétition de `expr` (potentiellement 0 fois)
- `expr+` on matche une répétition de `expr` (au moins 1 fois)
- `expr?` on matche `expr` au plus une fois
- `+`, `?` et `*` font référence au caractère précédent : on peut utiliser des `(?:expr)` pour englober une expression.
- `^` permet de matcher le début de ligne et
- `$` la fin de ligne.

Par défaut, `search` ne cherche que sur la première ligne, il faut specifier l'option `re.MULTILINE` (= `re.M`) pour qu'il cherche sur toutes les lignes.

Expressions régulières : `\b` et `\B`

On veut trouver toutes les occurences du mot **tache** dans le texte suivant :

```
texte = "Qui mange avec une moustache se tache."

p = re.compile(r"tache")
p.search(texte).span() # retourne 23,28
```

Le problème est que l'on trouve la première occurence, même si elle est dans un autre mot.

Pour éviter ce problème on peut utiliser `\b` :

```
p = re.compile(r"\btache\b")
p.search(texte).span() # retourne 32,37
```

`\B` sert à s'assurer que l'on est pas un début ou une fin de mot :

```
texte = "Une moustache on s'y attache, même si ça tache."

p = re.compile(r"\w*\Btache\w*|\w*tache\B\w*")
p.findall(texte) # ['moustache', 'attache']
```

Exercices

Exercice Quels mots vérifient l'expression `^a(?:ba)*a$` ?

abababa
abaa
aabbaa
aba
abababba

Exercice Quels mots vérifient l'expression `^a(?:ab|a|c)+a$`

aabcca
aba
? aca
aa
abca

Exercice Construisez une expression régulière qui reconnait un tag html sans attribut.

Expressions régulières : gloutonnerie

`*`, `+` et `?` sont **gloutons**, ce qui veut dire qu'ils essayent de "manger" le plus possible de fois l'expression précedente.

Ils ont tous une version non-gloutonne : `*?`, `+?` et `??`, qui tentera d'en manger le moins possible.

Par exemple (avec `search`):

`b.*b`

baababbabab
cc**btghtbaab**a

`b.*?b`

baababbabab
cc**btghtb**aaba

Expressions régulières : nombre d'occurences

- `l{n,m}` reconnait entre n et m occurences de l
- `l{,n}` reconnait au plus n occurences de l
- `l{n,}` reconnait au moins n occurences de l

On peut bien entendu étendre ça à une expression : `(?:expr){n,m}`.

Exercice Ecrire une expression régulière qui reconnait entre 3 et 6 occurences de `matou` et `chat`.

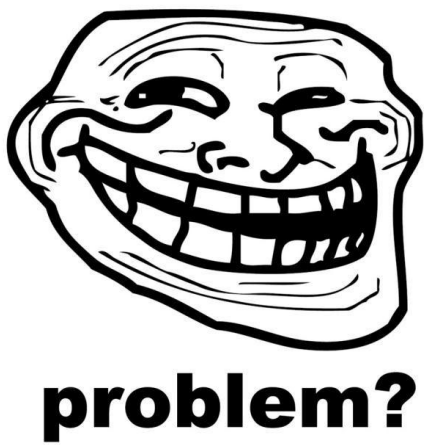
`{n,m}?` reconnait en mode non-glouton.

Expressions régulières : groupes

Les parenthèses `( )` permettent de définir des groupes : ce qui a été matché par un groupe peut ensuite être référencé par `\nombre` où nombre est le numéro du groupe.

Si par exemple on veut matcher la première balise xml (imaginons que l'on ait aucun attribut) avec la première balise fermante correspondant :

```
<(\w+?)>.*?</\0>
```

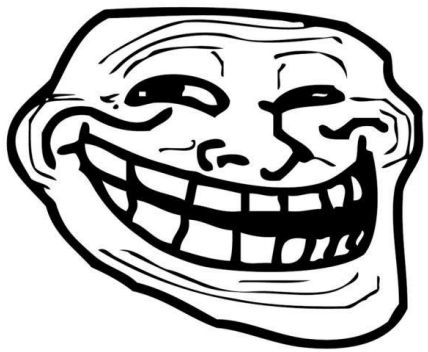


Expressions régulières : groupes

Les parenthèses `( )` permettent de définir des groupes : ce qui a été matché par un groupe peut ensuite être référencé par `\nombre` où nombre est le numéro du groupe.

Si par exemple on veut matcher la première balise xml (imaginons que l'on ait aucun attribut) avec la **première** balise fermante correspondant :

```
<(\w+?)>.*?</\0>
```



Piège : on ne peut pas parser du XML avec des regexps!

Expressions régulières : groupes

Plutôt que de compter pour trouver à quel groupe on veut faire référence, on peut leur donner des noms :

```
<(?P<nom>\w+?)>.*?</(?P=nom)>
```

On peut accéder au contenu d'un groupe dans un match object :

```
m.group("nom") # accède au groupe nom,
m.group(1) # pour le groupe 1
```

On peut faire des choses avancées avec les groupes :

```
(<)?\w+(?(1)>|$)
(?P<truc><)?\w+(?(truc)>|$)
```

Ces deux regexp **équivalentes** reconnaissent les choses de la forme `<blabla>` et `blabla`

`(?(groupe)oui|non)` matche `oui` si le groupe `groupe` existe et `non` sinon.

Expressions régulières : remplacements

Les groupes sont très utiles pour les **substitutions** :

```
re.sub("(\d+)","\g<0>0","10 100 1000 25")
# "100 1000 10000 250"
```

Syntaxe :

```
re.sub(regexp,remplacement,texte)
```

Ici `\g<0>` signifie la même chose que `\0` (pratique quand on veut rajouter un chiffre derrière car `\10` n'est pas la même chose que `\g<1>0`.

Ressources

- https://docs.python.org/3/reference
- https://www.jeffknupp.com/
- http://www-igm.univ-mlv.fr/~jyt/IMAC/
- http://www.pythonchallenge.com/ à ne pas confondre avec http://www.pythonchallenge.org