EXPRESSIONS RÉGULIÈRES

Table des matières

fiche 0 : trouver et remplacer dans une chaîne.

introduction.

Avant d'aborder le module des expressions régulières, on va examiner quelques méthodes relatives aux chaînes de caractères ou qui leur sont apparentées.

1. Poser le problème.

Problème : On dispose d'un **texte**, c'est à dire pour Python d'une chaîne de caractères ; par ailleurs, on propose un **motif**, qui est lui aussi une chaîne de caractères. Le problème consiste à détecter la présence du motif dans le texte, et éventuellement de remplacer ce motif par un autre..

Le texte se présente sous forme d'une chaîne de caractères en mémoire vive (variable, chaîne littérale) ; on ne se pose pas de problème d'encodage (UNICODE) puisque l'on ne cherche pas à transformer la chaîne en tableau binaire. Attention cependant, si on dispose d'un fichier de texte, le problème de l'encodage se pose lors du chargement (voir la fiche sur les fichiers de texte en Python) !

Le motif est une chaîne sans problème particulier : les sauts de lignes sont codés **`\n`**, l'antislash est codé **`\\`**. L'antislash protège les quotes contenues dans la chaîne. Il n'y a pas d'autre utilisation de l'antislash.

rappel : un antislash en fin de ligne annule la fin de ligne. Il permet par exemple, d'écrire une chaîne de caractères un peu longue sur plusieurs lignes physiques.

2. Trouver.

2.1. ce que dit la documentation.

`str.find(sub[, start[, end]])`

retourne le premier indice de la chaîne où la sous-chaîne **`sub`** est trouvée ; si **`start`** et **`end`** sont précisés, le domaine de recherche est la sous-chaîne formée des caractères dont l'indice est entre les valeurs de **`start`** et de **`end`** (la tranche commence avec **`start`** et se termine **avant `end`**). La méthode retourne -1 en cas d'échec de la recherche.

`str.index(sub[, start[, end]])`

comme **`find()`**, mais lève une exception **`ValueError`** si la sous-chaîne n'est pas trouvée. Cela permet de programmer les ruptures de séquences avec des exceptions.

`str.rfind(sub[, start[, end]])`

comme **`find()`**, mais commence par la fin de la chaîne.

`str.rindex(sub[, start[, end]])`

comme **`rfind()`** mais lève une exception **`ValueError`** si la sous-chaîne n'est pas trouvée.

`str.count(sub[, start[, end]])`

retourne le nombre d'occurrences de la sous-chaîne **`sub`** dans la tranche **`[start,end]`**

2.2. un exemple.

```
#!/usr/bin/python3
# rechercher dans une chaîne
texte = "Un IDE ou \"environnement de développement\" est un logiciel \
constitué d'outils qui facilitent l'écriture et les tests dans un \
langage défini, voire plusieurs.\
\nCet IDE comporte en général un éditeur avec coloration syntaxique,\
un système de gestion de fichiers (sauvegarde/chargement),\
un compilateur, un exécuteur de programme, un système d'aide en ligne,\
des indicateurs de syntaxe etc. \
\nLe plus connu est peut être Éclipse."

# première occurence du motif / numérotation qui commence à 0
motif = "IDE"
resultat = texte.find(motif)
print ("première occurence du motif :", resultat)

# nombre d'occurrences du motif
resultat= texte.count (motif)
print ("nombre d'occurrences du motif :",resultat)

# recherche d'un motif avec problème de casse
motif = "éclipse"
resultat = (texte.upper()).find(motif.upper())
print (motif, "est présent comme",texte[resultat:resultat+len(motif)])
```

résultat :

```
>>>
première occurrence du motif : 3
nombre d'occurrences du motif : 2
éclipse est présent comme Éclipse
>>>
```

3. Remplacer.

3.1. ce que dit la documentation.

str.replace(*old*, *new*[, *count*])

retourne une copie de la chaîne où toutes les occurrences de la sous-chaîne **old** sont remplacées par la chaîne **new**. Si la valeur **count** est précisée, seules les **count** premières occurrences sont remplacées.

3.2. un exemple.

```
#!/usr/bin/python3
# remplacer dans une chaîne
texte = "Un IDE ou \"environnement de développement\" est un logiciel \
```

```
constitué d'outils qui facilitent l'écriture et les tests dans un \
langage défini, voire plusieurs.\
\nCet IDE comporte en général un éditeur avec coloration syntaxique,\
un système de gestion de fichiers (sauvegarde/chargement),\
un compilateur, un exécuteur de programme, un système d'aide en ligne,\
des indicateurs de syntaxe etc. \
\nLe plus connu est peut être Éclipse."
# remplacement
motif = "\n"
resultat = texte.replace(motif, " ")
print (resultat)
```

résultat :

```
>>>
Un IDE ou "environnement de développement" est un logiciel constitué
 d'outils qui facilitent l'écriture et les tests dans un langage défini,
 voire plusieurs. Cet IDE comporte en général un éditeur avec coloration
 syntaxique,un système de gestion de fichiers (sauvegarde/chargement),un
 compilateur, un exécuteur de programme, un système d'aide en ligne,des
 indicateurs de syntaxe etc.  Le plus connu est peut être Éclipse.
>>>
```

4. Découper.

4.1. ce que dit la documentation.

`str.split([sep[, maxsplit]])`

retourne une liste de mots de la chaîne, en utilisant sep comme délimiteur. Si **`maxsplit`** est donné, il y a au plus **`maxsplit`** coupures (et la liste a au plus **`maxsplit+1`** éléments).

En l'absence de séparateur spécifié, ou avec le séparateur None, l'espace est considéré comme séparateur.

`str.splitlines([keepends])`

retourne une liste des lignes de la chaîne. En principe, le séparateur de ligne n'est pas gardé, sauf si **`keepend`** est posé à **`True`**.

4.2. exemples.

```
#!/usr/bin/python3
# couper une chaîne
texte = "123,,456,"
# split
liste = texte.split (",")
print ("texte :",texte, "\nliste :", liste,"\n\n")

# splitline
texte = "\nABC\nDEF\nGHI"
liste = texte.splitlines()
print ("texte :",texte, "\nliste :", liste,"\n\n")
```

```
# splitline avec keepend
texte = "\nABC\nDEF\nGHI"
liste = texte.splitlines(True)
print ("texte :",texte, "\nliste :", liste,"\n\n")
```

résultats :

```
>>>
texte : 123,,456,
liste : ['123', '', '456', '']


texte :
ABC
DEF
GHI
liste : ['', 'ABC', 'DEF', 'GHI']


texte :
ABC
DEF
GHI
liste : ['\n', 'ABC\n', 'DEF\n', 'GHI']


>>>
```

fiche 1 : le module Python RE

introduction

Les «expressions régulières » en Python 3 peuvent être abordées à l'aide du module re :

```
import re  #  module des expressions régulières
```

Les fichiers du module **re.py, re.pyc, re.pyo,** respectivement le fichier source, le fichier compilé, le fichier optimisé se trouvent dans le répertoire des librairies **Lib**, à la racine. Il n'y a donc aucun problème particulier pour son utilisation. C'est là que la machine Python cherche les modules (après le répertoire actuel, évidemment).

Dans les fiches qui suivent, on utilise l'importation sous la forme **import** ; il faut donc qualifier systématiquement les éléments constitutifs du module. En production, il se peut qu'il soit intéressant, pour éviter des lourdeurs de code, d'utiliser la forme **from... import** et de définir des alias. C'est évidemment tout à fait possible, mais ce n'est pas ici le lieu d'en discuter.

1. Matching et Searching.

1.1. vocabulaire.

Pour la clarté de l'exposé, nous traduirons le verbe **to match** par ***correspondre à***, et **to search** par ***rechercher*** et donc **matching** par ***correspondance*** et **searching** par ***recherche***. Il ne faut pas chercher de signification "intuitive" à ces concepts qui ont dans chacun des langages comme C, Java ou Python un sens précis, mais malheureusement pas le même d'un langage à l'autre.

1.2. poser le problème.

Problème : On dispose d'un **texte**, c'est à dire pour Python une chaîne de caractères ; par ailleurs, on propose un **patron (ou motif)**, qui est lui aussi une chaîne de caractères. Le problème consiste à détecter une sous-chaîne respectant le patron dans le texte et éventuellement à remplacer par une chaîne prédéfinie la (ou les) sous-chaînes(s) trouvées. D'autres opérations sont possibles, qui seront vues en leur temps.

Le texte se présente sous forme d'une chaîne de caractères en mémoire vive (variable, chaîne littérale) ; on ne se pose pas de problème d'encodage (UNICODE par défaut en version 3 ; en versions 2.x, le code est le code local. On peut définir la norme de l'encodage par une clause **-code-**). Si on dispose d'un fichier de texte, le problème de l'encodage se pose lors du chargement !

Le patron est pour l'instant une chaîne, avec quelques problèmes : comme d'habitude, les sauts de lignes sont codés **\n**, l'antislash est codé ****. L'antislash protège les quotes. Mais il doit aussi protéger un certain nombre de caractères usuels comme le point, ^, $, + , ?, * . Cela sera explicité plus complètement dans les fiches qui suivent.

1.3. compiler le patron.

La syntaxe est la suivante :

```
cpatron = re.compile(patron, directives)
```

Le **type** (classe) de **cpatron** est : **'_sre.SRE_Pattern'**

_sre est le module appelé dans le module **re** ; **SRE_Pattern** est le nom de la classe.

Les directives se présentent sous forme d'un entier : chaque directive est une puissance de deux, et le paramètre ***directives*** est **la somme de directives particulières**.

constante	abrégé	valeur	signification
IGNORECASE	**I**	**2**	ignorer la casse ; fonctionne avec les lettres accentuées
LOCALE	**L**	**4**	définir comme "lettre" ce que la langue locale définit comme tel dans la variable système.;
MULTILINE	**M**	**8**	considèrer le texte comme décomposé en lignes (le caractère **\n** est le début de chaque ligne)
DOTALL	**S**	**16**	considèrer le saut de ligne comme un caractère ordinaire.
UNICODE	**U**	**32**	*obsolète dans Python 3*
VERBOSE	**X**	**64**	permettre d'écrire des commentaires dans les patrons.
ASCII	**A**	**256**	permettre de travailler en ASCII

On peut écrire indifféremment **re.IGNORECASE+re.MULTILINE** ou **re.I+re.M**

En pratique, dans le début du travail, seules les directives **IGNORECASE, MULTILINE** et **DOTALL** sont à prendre en compte. On verra plus tard la directive **VERBOSE**.

1.4. comparer et chercher.

* La syntaxe des fonctions est la suivante :

```
cpatron.search (texte)
cpatron.match (texte)
```

* La recherche (**search()**) consiste à parcourir le texte depuis le début ; si le patron n'est pas identifiable dans le texte, la fonction retourne **None**. Sinon, elle retourne une instance de l'objet **MatchObject,** appartenant lui aussi au module **_sre**.

La comparaison (**match()**) consiste à identifier le début du texte au patron. Elle retourne **None** si aucune comparaison n'est possible. Sinon elle retourne une instance **MatchObject**.

*** l'objet : MatchObject**

Cet objet peut être interrogé par ses méthodes. Pour l'instant, les méthodes **start()** (premier caractère reconnu) et **stop()** (position après le dernier caractère reconnu)

```
#!/usr/bin/python3
import re

texte = "Un IDE ou \"environnement de développement\" est un logiciel \
constitué d'outils qui facilitent l'écriture et les tests dans un \
langage défini, voire plusieurs.\
\nCet IDE comporte en général un éditeur avec coloration syntaxique,\
un système de gestion de fichiers (sauvegarde/chargement),\
un compilateur, un exécuteur de programme, un système d'aide en ligne,\
des indicateurs de syntaxe etc. \
\nLe plus connu est peut être Éclipse."

# recherche du patron dans le texte ; résultat affiché : 158 165
patron = "Cet IDE"
cpatron = re.compile(patron)
resultat = cpatron.search(texte)
```

```
if resultat :
    print (resultat.start(), resultat.end())
else:
    print (resultat)
# comparaison du patron au texte ; résultat affiché : None
resultat = cpatron.match(texte)
if resultat :
    print (resultat.start(), resultat.end())
else :
    print (resultat)

# comparaison du patron au texte ; résultat affiché : 0  6
patron = "Un IDE"
cpatron = re.compile (patron)
resultat = cpatron.match(texte)
if resultat :
    print (resultat.start(), resultat.end())
else :
    print (resultat)

# recherche en ignorant la casse ; résultat affiché : 413 420
patron = "éclipse"
cpatron = re.compile (patron, re.IGNORECASE)
resultat = cpatron.search(texte)
if resultat :
    print (resultat.start(), resultat.end())
else:
    print (resultat)
```

note. Il existe plusieurs présentations syntaxiques pour les méthodes comme **`search()`**. Dans le script ci-dessus, **`search()`** est une méthode de l'objet **`SRE_Pattern`**. Mais Python l'a également redéfini comme une fonction du module **`re :`**

exemple : **`re.search(patron, texte, re.IGNORECASE))`**...

Pour la clarté de l'exposé, on s'en tient à une seule syntaxe, celle du script ci-dessus, qui a l'avantage de ressembler à ce qui se fait en java.

2. Expression régulière.

2.1. remarques sur la section précédente.

La section qui précède a pour objectif de spécifier une syntaxe : celle requise par le module **`re`** pour faire des opérations assez semblables à celles réalisées avec plus de simplicité par les méthodes de la classe **`str`**.

Quel est l'apport du module **`re`** pour les opérations de recherche, de comparaison, de substitution, de découpage de chaînes ? Lorsque l'on fait une recherche ou une substitution dans un traitement de texte ou un éditeur, la chaîne recherchée doit être explicite. On peut imposer de ne pas différencier majuscules et minuscules, ou de ne rechercher que des mots entiers ; ces aménagements sont insuffisants.

note : les éditeurs et traitements de texte actuels peuvent travailler différemment !

On peut souhaiter disposer, pour écrire les patrons, de **jokers** (comme le * ou le ? dans les shells des

systèmes d'exploitation) ou de **conditions** (exemple : trouver les virgules qui sont suivies d'un caractère alphabétique ; trouver les espaces multiples ; se limiter aux caractères alphanumériques). **Les expressions régulières** sont des chaînes comportant des jokers et des éléments de description de chaînes de caractères.

2.2. les jokers et marqueurs fondamentaux.

signe	signification générale	signification sous directive
.	le point remplace tout caractère sauf le saut de ligne (\n).	sous **DOTALL**, le point représente tout caractère, y compris le saut de ligne.
^	Le chevron (circonflexe) représente le début de la chaîne analysée.	sous **MULTILINE**, représente **en plus** la position après le saut de ligne
$	Le dollar représente la fin de la chaîne analysée.	sous **MULTILINE**, représente en plus la position avant le saut de ligne.

exemples d'expressions régulières :

"azer.tyiop" : cette **expression régulière** correspond à : **azer** suivi de **n'importe quel caractère**, suivi de **tyiop** sur une ligne. En mode **DOTALL**, le caractère peut être le saut de ligne.

En recherche, **azeratyiop, azerAtyiop, azer(tyiop, azer.tyiop** satisfont au modèle proposé.

"^aze" : le patron "**aze**" doit être recherché en début de texte uniquement en début de texte.

En mode **MULTILINE**, il est aussi recherché en début de chaque ligne.

"tyiop$" : le patron "**tyiop**" est recherché en fin de texte.

En mode **MULTILINE**, il est aussi recherché en fin de chaque ligne.

"^$" : recherche d'un texte vide ou en mode **MULTILINE** d'une ligne vide.

2.3. répétitions d'un caractère.

itérateur	interprétation
*	zéro, une ou plusieurs fois le caractère qui précède
+	une ou plusieurs fois le caractère qui précède
?	zéro ou une fois le caractère qui précède
{m}	exactement m fois le caractère qui précède
{m,}	au moins m fois le caractère qui précède
{m,n}	au moins m fois et au plus n fois le caractère qui précède

exemples :

exp. reg.	chaînes reconnues
ab*c	ac, abc, abbc, abbbc, ...
ab+c	abc, abbc, abbbc, ...

`ab?c`	`ac, abc`
`ab{2,}c`	`abbc, abbbc, abbbbc, ....`
`ab{2,4}c`	`abbc, abbbc, abbbbc`

2.4. consommation.

Lorsqu'une recherche est couronnée de succès, la partie de la chaîne comprise entre le début de celle-ci et la fin de la sous-chaîne trouvée est consommée. Ce qui signifie par exemple que la fonction `findall()` commence la recherche à partir du caractère suivant le dernier caractère consommé. La recherche est séquentielle et elle ne fait aucun retour en arrière.

2.5. gourmandise.

les itérateurs `*, +, ?` sont gourmands. C'est-à-dire que dans une recherche, ils ont un comportement qui conduit à reconnaître la plus grande chaîne possible. Ils consomment la chaîne au maximum, même si une correspondance a déjà été trouvée sur le patron proposé.

Par exemple, supposons que l'on ait :

```
texte = "<h1>Expressions Régulières</h1>"
patron = "<.*>"
```

Le sous-motif `.*` conduit à avoir le plus grand texte possible : l'expression régulière s'identifie à tout le texte. Il faut lire `patron` comme :

> le caractère **<** pour commencer ; **un maximum de caractères** ; le caractère **>** pour finir.

On aurait pu souhaiter au contraire une identification à **<h1>**, c'est-à-dire :

> le caractère **<** pour commencer ; **un minimum de caractères** ; le caractère **>**.

Une nouvelle classe d'itérateurs réalisent ces opérations :

itérateur	interprétation
`*?`	zéro, une ou plusieurs fois le caractère qui précède ; non gourmand. Dans une expression, `*?` s'identifie à la plus petite chaîne possible.
`+?`	une ou plusieurs fois le caractère qui précède ; non gourmand.
`??`	zéro ou une fois le caractère qui précède ; non gourmand.
`{m,n}?`	le caractère qui précède m fois.

exemple :

```
import re

texte = "<<aaAAabbbccaaannn"

patron = "a{2,3}b?"
cpatron = re.compile(patron, re.I)
res = cpatron.findall(texte)
print ("patron :",patron,"résultat : ",res)

patron = "a{2,3}?b?"
cpatron = re.compile(patron, re.I)
```

```
res = cpatron.findall(texte)
print ("patron :",patron,"résultat : ",res)

patron = "a{2,3}b??"
cpatron = re.compile(patron, re.I)
res = cpatron.findall(texte)
print ("patron :",patron,"résultat : ",res)
```

résultat :

```
>>>
patron : a{2,3}b? résultat :  ['aaA', 'Aab', 'aaa']
patron : a{2,3}?b? résultat :  ['aa', 'AA', 'aa']
patron : a{2,3}b?? résultat :  ['aaA', 'Aa', 'aaa']
>>>
```

Pour bien comprendre, il faut simuler la consommation !!!

2.5. échappement.

Les jokers et les opérateurs sont des caractères courants. Comment, dans une expression régulière peut-on distinguer ce caractère en tant que caractère et le même en tant qu'opérateur ou composant d'un opérateur ? On utilise le même système que Python dans ce cas : pour considérer un caractère comme *se représentant lui-même*, il faut l'échapper par un antislash. On aura donc pour le point `\.`, pour l'astérisque `\*`, pour le signe plus `\+` et ainsi de suite. Cela peut se compliquer ; un chapitre spécial sera consacré à cette question.

3. Des méthodes à recherches multiples.

3.1. la méthode findall().

itération de la recherche.

La méthode **`search()`** permet de trouver la première sous-chaîne qui correspond à l'expression régulière donnée comme patron. On peut itérer la recherche grâce à la fonction **`findall()`**, qui retourne tous les éléments qui correspondent dans une liste de chaînes.

Exemple.

problème : on dispose d'une liste de noms de localités et l'on se propose de rechercher celles dont le nom de termine par ville. On ne distingue pas les majuscules et les minuscules dans la recherche.

Ablon	**non**
Acqueville	**oui**
Agy	**non**
Aigner-Ville	**oui / majuscules**
Airan	**non**
Amayé-sur-Orne	**non**
Amblie	**non**
Amfreville	**oui**
Angervillers	**non / ville non terminal**
Angoville	**oui**

Arganchy	non
Argences	non
Arroville-les-Bains	non / ville non terminal
Asnelles	non
Asnières-Surville	oui

le source :

```
#!/usr/bin/python3
import re

texte ="Ablon\nAcqueville\nAgy\nAigner-Ville\nAiran\nAmayé-sur-Orne\
\nAmblie\nAmfreville\nAngervillers\nAngoville\nArganchy\nArgences\
\nArromanches-les-Bains\nAsnelles\nAsnières-Surville"

patron = ".*ville$"
cpatron = re.compile (patron)
resultat = cpatron.findall(texte)
print (resultat, "\n")

cpatron = re.compile (patron, re.MULTILINE)
resultat = cpatron.findall(texte)
print (resultat, "\n")

cpatron = re.compile (patron, re.MULTILINE+re.IGNORECASE)
resultat = cpatron.findall(texte)
print (resultat, "\n")
```

`".*ville$"` : une suite gourmande de caractères, suivie de ville, suivie de la fin de texte.

Les fins de chaîne ne sont pas incluses. Seule la dernière ligne peut être prise en compte si on ne met pas de directives !

résultats :

```
>>>
['Asnières-Surville']

['Acqueville', 'Amfreville', 'Angoville', 'Asnières-Surville']

['Acqueville', 'Aigner-Ville', 'Amfreville', 'Angoville', 'Asnières-
Surville']

>>>
```

3.2. remplacer : la méthode sub().

La syntaxe est la suivante :

```
cpatron.sub (remplacement, texte, [count])
```

Retourne le texte avec ses remplacements. La valeur **remplacement** est une chaîne (qui peut être obtenue par application d'une fonction, appelée à chaque remplacement). **count** fixe le nombre maximum de remplacements ; 0 est la valeur par défaut et signifie (paradoxalement) que tous les remplacements possibles doivent être effectués.

On a aussi la syntaxe :

```
cpatron.subn (remplacement, texte, count)
```

Renvoie un tuple comportant la nouvelle chaîne et le nombre de remplacements.

```
#!/usr/bin/python3
# substitution dans une chaîne
import re

texte = "Retourne le texte avec ses remplacements. La \
valeur remplacement est une chaîne (qui  peut être obtenue \
par application d'une fonction, appelée à chaque \
remplacement). count donne le maximum de remplacements ; \
0 est la valeur par défaut et signifie que tous les remplacements\
 possibles doivent être effectués."

patron= "\. " # antislash de protection du point
remplacement ="\n\n"
cpatron = re.compile(patron)
nouveau = cpatron.sub(remplacement, texte)
print (nouveau)
print ("\n************************************\n")

retTuple = cpatron.subn (remplacement, texte)
print (retTuple)
```

résultat :

```
>>>
Retourne le texte avec ses remplacements

La valeur remplacement est une chaîne (qui  peut être obtenue
 par application d'une fonction, appelée à chaque remplacement)

count donne le maximum de remplacements ; 0 est la valeur par défaut
et signifie que tous les remplacements possibles doivent être effectués.

************************************

("Retourne le texte avec ses remplacements\n\nLa valeur remplacement
est une chaîne (qui  peut être obtenue par application d'une
```

```
fonction, appelée à chaque remplacement)\n\ncount donne le maximum
de remplacements ; 0 est la valeur par défaut et signifie que tous
les remplacements possibles doivent être effectués.", 2)
>>>
```

3.3. la méthode split()

La syntaxe est la suivante :

`cpatron.split(texte, [maxsplit])`

Pour mémoire, reprenons l'exemple précédent avec **`split()`** au lieu de **`findall()`**

```
>>>
['Ablon\nAcqueville\nAgy\nAigner-Ville\nAiran\nAmayé-sur-
Orne\nAmblie\nAmfreville\nAngervillers\nAngoville\nArganchy\nArgences\n
Arromanches-les-Bains\nAsnelles\n', '']

['Ablon\n', '\nAgy\nAigner-Ville\nAiran\nAmayé-sur-Orne\nAmblie\n',
 '\nAngervillers\n', '\nArganchy\nArgences\nArromanches-les-
Bains\nAsnelles\n', '']

['Ablon\n', '\nAgy\n', '\nAiran\nAmayé-sur-Orne\nAmblie\n',
 '\nAngervillers\n', '\nArganchy\nArgences\nArromanches-les-
Bains\nAsnelles\n', '']

>>>
```

fiche 2 : E.R. les fondamentaux.

introduction.

Les opérateurs vus dans la fiche qui précède permettent de répondre aux problèmes liés à la répétition d'un caractère. Le seul joker dont on dispose est le point (sous deux acceptions, avec ou sans fin de ligne). La première extension souhaitable est de diversifier les jokers.

1. Diversifier les jokers et marqueurs.

1.1. jeux de caractères définis ou classes.

Au lieu de dire "tout caractère" comme avec le joker "point", on peut définir en extension **un jeu** de caractères. La syntaxe est simple :

[*caractères du jeu*]

Les caractères sont écrits en suivant, dans une suite **sans espacements** : **[abcABC$*+]** est le jeu (la classe) des 9 caractères **a,b,c,A,B,C,$,*,+**

Il n'est pas utile d'échapper **les caractères litigieux des expressions régulières comme *, +, . etc.** C'est une bonne méthode de protection : on écrit **[*]** au lieu de *****

Il existe une **forme abrégée pour les séquences** : premier et dernier caractère séparés par un trait d'union : **[a-z]** pour une minuscule, [**A-Z]** pour une majuscule, [**0-9]** pour un chiffre.

C'est ainsi que **[a-zA-Z]** équivaut à "toute lettre ASCII". Le jeu **[À-ÿ]** est tout à fait valide.

Une définition de jeu peut être définie par exclusion : il suffit de **commencer par ^**. Par exemple le jeu **[^5]** définit tout caractère sauf 5. **[^0-9**] : tout caractère sauf un chiffre.

Pour éviter certaines protections lourdes, on peut définir des jeux à un caractère : **[*]**, ou **[+]**, ou **[\\] ; cela remplace avantageusement * \+** ou ****

note. Le caractère **^** doit être protégé lorsque son utilisation est littérale, soit en début d'expression, soit en début de jeu de caractère : **[\^abcd]; [\^]**. **On ne peut pas écrire [^].**

1.2. classes et marqueurs usuels prédéfinis.

Certaines classes sont prédéfinies, ainsi que certains marqueurs :

jeu prédéfini	signification
\A	trouver uniquement au début de la chaîne. Ressemble à ^ mais n'est pas affecté par la directive **MULTILINE**.
\b	**\b** est réservé en python (c'est le backspace) ; il faut donc écrire **\\b** pour utiliser dans une chaîne Python. Le caractère qui suit est en début ou fin de mot.
\B	le caractère qui suit n'est ni en début de mot, ni en fin de mot.
\d	équivalent de [0-9]
\D	tout caractère sauf un chiffre
\s	équivaut à **[\b\t\n\r\f\v],** espace, backspace, tabulation, fin de ligne(LF), début de ligne (CR), à la ligne (FF), tabulation verticale. On désigne parfois ces caractères comme des **blancs**.
\S	tout caractère sauf ceux de **\s**

\w	équivalent à **[a-zA-Z0-9_]**, caractères alphanumériques. Peut être affecté par la directive LOCALE.
\W	tout caractère non alphanumérique
\Z	trouve uniquement en fin de la chaîne. Ressemble à $ mais est insensible à la directive **MULTILINE**.

2. Parenthésage.

2.1. analogie mathématique.

En calcul numérique, on utilise les parenthèses pour hiérarchiser les opérations. Il en est de même pour les expressions régulières. On peut parenthéser une sous-expression régulière et lui **appliquer les opérateurs d'itération vus pour les caractères**.

ATTENTION !

Le parenthésage a une syntaxe peu intuitive et déroutante :

(?:*expression régulière*)

2.2. un exemple.

problème : examiner si une chaîne est une adresse mail valide. Une adresse mail comporte une ou deux chaînes alphanumériques ; dans ce second cas, un point les sépare. Le caractère @ suit. La seconde partie est constituée d'une suite d'au moins deux chaînes alphanumériques séparées par un point.

```
#!/usr/bin/python3
import re
textes1 = ['zorlub@free.fr', "zor.lub@free.fr",
           "zor.lub.ego@free.fr",'zorlubfree.fr']
textes2 = ['zorlub@free', "zor.lub@free.domaine.fr",
 "zor.lub@free.domaine. fr","zor.lub@free.domaine.fr   "]

g_patron = "^\w+(?:[.]\w+)?" # partie gauche
d_patron = "\w+(?:[.]\w+)+$  "# partie droite
cpatron = re.compile(g_patron+"@"+d_patron)

for x in textes1 + textes2:
    res = cpatron.search(x)
    if res:
        print (x, "est correct")
    else:
        print (x, "n'est pas une adresse valide")
```

```
>>>
zorlub@free.fr est correct
zor.lub@free.fr est correct
zor.lub.ego@free.fr n'est pas une adresse valide
zorlubfree.fr n'est pas une adresse valide
zorlub@free n'est pas une adresse valide
zor.lub@free.domaine.fr est correct
```

```
zor.lub@free.domaine. fr n'est pas une adresse valide
zor.lub@free.domaine.fr    n'est pas une adresse valide
>>>
```

3. La double analyse lexicale.

Un analyseur lexical prend un flot de caractères sur son entrée standard, isole les unités lexicales, et envoie un flot d'unités lexicales sur sa sortie standard. Une unité lexicale peut être un identificateur, un mot réservé du langage, un signe (+ - * / ! etc), un délimiteur (parenthèse, accolade, crochet, **/***, ***/** etc), c'est-à-dire toute unité pertinente pour l'analyse syntaxique.

3.1. deux analyses en pipeline.

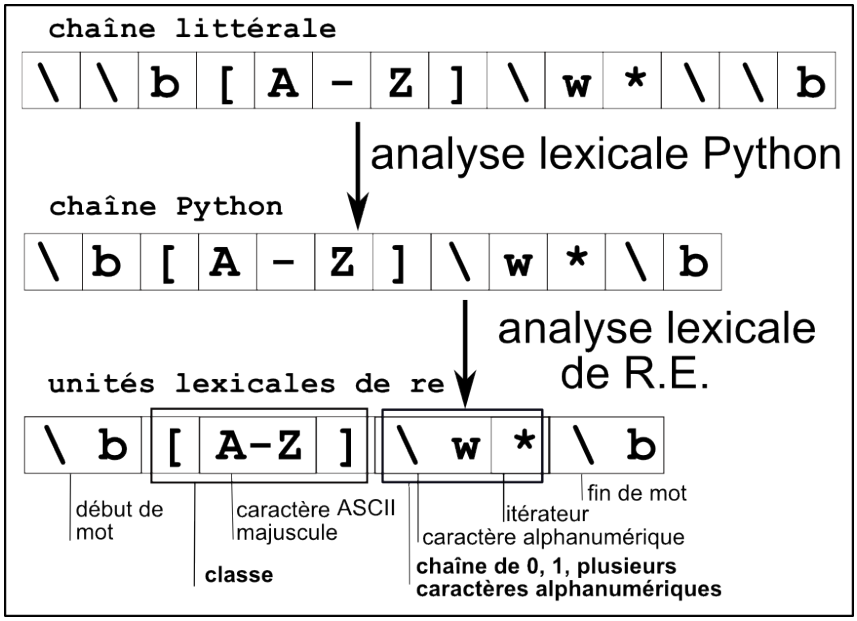
Que se passe-t-il entre la lecture du source d'une expression régulière et l'édition d'un patron compilé qui est utilisé par les méthodes **search(), match(), findall(), sub()** et **split()** ?

patron = ***chaîne_source***

cpatron = compile(patron)

Il y a deux étapes au niveau lexical : l'analyseur de chaîne Python lit la chaîne source et la traite : il en sort une chaîne interne dans le format de stockage de Python. C'est à ce niveau que sont traités les caractères spéciaux comme le backslash (**\b**) et la fin de ligne (**\n**).

Puis l'analyseur lexical de la machine d'analyse des expressions régulières prend le relai et isole les unités lexicales que comprend l'analyseur d'E.R.. Les jokers, les marqueurs, les expressions sont de telles unités lexicales.



On a figuré deux éléments qui relèvent de l'analyse syntaxique (classe, chaîne).

3.2. l'échappement dans les chaînes en Python.

Lorsque Python réalise l'analyse lexicale d'une chaîne littérale pour en créer la représentation interne, il lit la représentation littérale (flot d'entrée) de gauche à droite pour en dégager les unités lexicales, qui seront ensuite codées comme un caractère (flot de sortie) :

- l'antislash en fin de ligne est ignoré (chaînes sur plusieurs lignes). Le changement de ligne qui suit est ignoré également.

- les unités lexicales tiennent en général sur un seul caractère ; c'est le cas des caractères alphanumériques ou des caractères accentués ou de la ponctuation par exemple.

- certaines unités ont besoin de deux caractères dans la chaîne littérale : par exemple la fin de ligne est notée **\n**, le backspace **\b**, la tabulation **\t**. Ces couples correspondent à une seule unité lexicale (un caractère dans la chaîne interne).

- certaines peuvent tenir sur un ou deux caractères sans poser de difficulté : **\"** et **\'**. En effet, les caractères **"** et **'** étant des délimiteurs, leur échappement par l'antislash permet de les considérer comme des littéraux. Ainsi si on pose :

```
maChaine ='papa "Noël" s\'est envolé'
taChaine = 'papa \"Noël\" s\'est envolé'
saChaine ='papa \"Noël" s\'est envolé'
print (maChaine==taChaine,maChaine==saChaine)
```

on peut vérifier que les trois chaînes sont similaires.

- la difficulté apparaît lorsque l'on utilise l'antislash. L'antislash peut s'échapper lui-même : c'est sans ambigüité ; même chose pour **"** et **'**. Mais qu'en est-il s'il précède un caractère qui ne peut être échappé. Comme par exemple un espace,les caractères c, d, w, ... L'antislash est alors considéré comme littéral !

exemple :

```
print("\c\d\w")
```

donne : **\c\d\w**

- enfin, plus délicat encore certaines séquences antislashées introduisent des séquences spécifiques, comme par exemple des séquences unicodes ou numériques (cas de **\u \U \x \N)**

Voici ce que dit la documentation officielle :

Escape Sequence	Meaning
\newline	Backslash and newline ignored
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Escape sequences only recognized in string literals are:

Escape Sequence	Meaning
`\N{name}`	`Character named name in the Unicode database`
`\uxxxx`	`Character with 16-bit hex value xxxx`
`\Uxxxxxxxx`	`Character with 32-bit hex value xxxxxxxx`

quelque cas :

La chaîne `"\123"` donne le caractère de code octal 123 en Unicode, soit S (code 64+2x8+3 = 83) :

```
print (ord("\123"), "123")
```

La chaîne "\u00b5" donne le µ de code 181 :

```
print (ord("\u00b5"), "\u00b5")
```

Chaque caractère unicode a un nom (importer le module unicode):

```
print (ord("\u00b5"), "\u00b5", unicodedata.name("\u00b5"))
```

donne : `181 µ MICRO SIGN`

D'où la liste des caractères alphanumériques pour lesquelles l'antislash n'est pas un littéral

```
caractères particuliers pour l'échappement
N U u x  0 1 2 3 4 5 6 7  a b f n r v
```

On insiste bien sur le fait que l'analyse lexicale de Python s'applique à tout ce qui se trouve sur le flot d'entrée de l'analyseur, texte comme patron.

3.3. l'échappement des expressions régulières.

Lors de la compilation, c'est la chaîne interne de Python qui est examinée par l'analyseur lexical du moteur d'analyse des expressions régulières. Il y a alors le second niveau d'analyse, qui lui aussi utilise l'antislash pour caractère d'échappement ! Cela ne pose pas de problème avec `^, ( , ) , { , }, w, W, B, 8, 9, $.` Il y en a un avec les caractères particuliers qu'il faut échapper à l'aide d'un `\\` lors de la première analyse lexicale. Il faut prêter une attention particulière aux caractères `b, 0, 1, 2, 3, 4, 5, 6, 7`. Quand à l'antislash, comme il est doublement caractère d'échappement, pour l'utiliser comme littéral, il vaut mieux l'échapper pour Python, puis pour l'analyse re, ce qui fait quatre antislash !

4. Donner de l'air aux expressions régulières.

le problème : les expressions régulières sont particulièrement indigestes. Parmi les faits qui concourent à une lecture difficile on peut citer :

- l'absence de caractères d'aération, ce que l'on appelle parfois "les blancs" : espaces, tabulation, saut de ligne ;
- l'absence de commentaires explicatifs insérés dans l'expression ;
- les difficultés liées au fait que l'on cumule deux niveaux d'analyse : l'analyse lexicale de Python qui donne une première représentation de l'expression, puis l'analyse lexicale du moteur d'expressions régulières. La multiplication des échappements est liée à cet aspect de la question.

Cette difficulté de lecture peut être en partie levée par l'utilisation de techniques appartenant à l'un ou l'autre des analyseurs lexicaux.

note : dans les exemples qui suivent, la signification des expressions régulières n'a pas à être examinée. Ces exemples illustrent l'aération du patron.

4.1. la saisie multiligne des chaînes

En Python, on peut saisir une chaîne sur plusieurs lignes physiques ; il suffit de terminer chaque élément significatif du texte par un antislash suivi d'un passage à la ligne. Attention, après l'antislash il ne doit rien y avoir !Il y a alors concaténation des segments de texte à l'analyse lexicale.

exemple :

```
#!/usr/bin/python3
import re

textes = "jean@free.fr,marie.jeanne@free.fr,phil@laposte.net,\
bob.robert@dom.com,yann.caro@yahoo.uk,richard.hernu@asso.net"

patron = "\\b\
\w+\
(?:[.]\w+)?\
@\
\w+\
(?:[.]\w+)*\
[.](?:fr|com)\
\\b"
print (patron,"\n")

cpatron = re.compile(patron)

print (cpatron.findall(textes))
```

```
>>>
\b\w+(?:[.]\w+)?@\w+(?:[.]\w+)*[.](?:fr|com)\b

['jean@free.fr', 'marie.jeanne@free.fr', 'bob.robert@dom.com']
>>>
```

4.2. le commentaire en ligne.

La syntaxe est **(?# *texte*)**

Le texte est un commentaire ; il est ignoré. On doit prendre garde à ce que le commentaire en ligne est un élément de l'expression régulière ; il ne faut pas introduire d'espaces qui seraient pris comme des caractères constitutifs de l'expression.

exemple :

```
#!/usr/bin/python3
import re

textes = "jean@free.fr,marie.jeanne@free.fr,phil@laposte.net,\
bob.robert@dom.com,yann.caro@yahoo.uk,richard.hernu@asso.net"

patron = "\\b(?# début de mot)\
\w+(?:[.]\w+)?(?# un ou deux mots séparés par un point)\
```

```
@\
\w+(?:[.]\w+)*(?# un, ou plusieurs mots séparés par un point)\
[.]\
(?:fr|com)(?# le dernier étant soit fr soit com)\
\\b(?# fin de mot)"
print (patron,"\n")

cpatron = re.compile(patron)

print (cpatron.findall(textes))
```

donne :

```
>>
\b(?# dé> but de mot)\w+(?:[.]\w+)?(?# un ou deux mots séparés par
 un point)@\w+(?:[.]\w+)*(?# un, ou plusieurs mots séparés
 par un point)[.](?:fr|com)(?# le dernier étant soit fr soit com)
\b(?# fin de mot)

['jean@free.fr', 'marie.jeanne@free.fr', 'bob.robert@dom.com']
>>>
```

* On a ici combiné les deux méthodes : le gain de lisibilité est réel mais ce n'est pas encore le grand confort : il manque une aération par des espaces par exemple. D'autre part, la méthode de l'antislash terminal peut introduire des erreurs peu visibles si on ajoute malencontreusement des blancs.

4.3. la directive VERBOSE.

La directive **VERBOSE** a comme effet d'occulter tous les blancs (espaces, tabulations, fin de ligne...) de la chaîne de saisie. De plus, l'introduction dans une ligne du symbole # a le même effet qu'en Python : il supprime la fin de ligne physique à partir du dièse. Encore faut-il qu'il y ait fin de ligne ! La triple quote s'impose alors ; la chaîne délimitée ainsi est prise littéralement, avec ses fins de lignes.

On dispose de l'aération souhaitée, mais au prix d'une perte des blancs et du dièse. Il suffit en général d'utiliser le joker **\s** pour résoudre le problème, ou alors de protéger le caractère dans une classe ou avec un antislash.

exemple.

```
#!/usr/bin/python3
import re

textes = "jean@free.fr,marie.jeanne@free.fr,phil@laposte.net,\
bob.robert@dom.com,yann.caro@yahoo.uk,richard.hernu@asso.net"

patron = """
  \\b                 # début de mot)
    \w+(?:[.]\w+)?    # un ou deux mots séparés par un point
     @\
     \w+(?:[.]\w+)*   # un, ou plusieurs mots mots séparés par un point
     [.]\
```

```
    (?:fr|com)      # le dernier étant soit fr soit com
  \\b               # fin de mot
"""
print (patron,"\n")

cpatron = re.compile(patron, re.VERBOSE)

print (cpatron.findall(textes))
```

```
>>>

  \b                # début de mot)
    \w+(?:[.]\w+)?   # un ou deux mots séparés par un point
     @     \w+(?:[.]\w+)*  # un, ou plusieurs mots mots séparés par un
point
     [.]     (?:fr|com)      # le dernier étant soit fr soit com
  \b                # fin de mot


['jean@free.fr', 'marie.jeanne@free.fr', 'bob.robert@dom.com']
>>>
```

4.4. jouer sur l'implicite.

```
#!/usr/bin/python3
import re

textes = "jean@free.fr, marie.jeanne@free.fr, phil@laposte.net,\
bob.robert@dom.com, yann.caro@yahoo.uk, richard.hernu@asso.net"

cpatron = re.compile(
    "\\b"           # début de mot
    "\w+"           # suite de caractères alphamériques
    "(?:[.]\w+)?"   # 0 ou 1 point et suite de caractères alphanumériques
    "@"             #
    "\w+"           # suite de caractères alphamériques
    "(?:[.]\w+)*"   # x point et suite de caractères alphanumériques
    "[.](?:fr|com)" # suffixe en fr ou en com
    "\\b"           # fin de mot
    )

print (cpatron.findall(textes))
```

On joue ici triplement sur l'implicite : dans une parenthèse, les nouvelles lignes ne sont pas prises en compte ; les commentaires sont ignorés à l'analyse lexicale ; deux chaînes littérales juxtaposées sont concaténées... Cela fonctionne, mais c'est dangereux !

4.5. l'opérateur de chaîne r.

Il existe un opérateur de chaîne, **r**, qui permet de court-circuiter l'analyse lexicale de Python. Il suffit

alors de fournir comme patron la chaîne d'entrée de l'analyseur du moteur d'évaluation des expressions régulières. On y gagne quelques antislashes ???

```
#!/usr/bin/python3
import re

textes = "jean@free.fr,marie.jeanne@free.fr,phil@laposte.net,\
bob.robert@dom.com,yann.caro@yahoo.uk,richard.hernu@asso.net"

patron = r"\b\w+(?:[.]\w+)?@\w+(?:[.]\w+)*[.](?:fr|com)\b"
cpatron = re.compile(patron)

print (cpatron.findall(textes))
```

5. Expression avec un ou logique.

5.1. le ou logique.

Jusque là, on n'a rencontré que des expressions régulières "additives" : tout l'ensemble décrit doit correspondre à une sous-chaîne pour que la correspondance soit reconnue. On peut avoir besoin de chercher une correspondance avec l'une ou l'autre de deux expressions régulières. Dans ce cas, on les sépare par le symbole `|` (barre verticale, le **`Alt gr 6`** du clavier / valeur ASCII 124). On écrit **`A|B`** avec A et B comme expressions régulières.

Plusieurs questions se posent avec le **ou** logique :

- les deux conditions peuvent-elles être satisfaites en même temps ? Cela n'est pas interdit.
- dans quel ordre se fait l'évaluation des expressions régulières ? De gauche à droite.
- lorsque l'évaluation de la première expression est satisfaite, la seconde est-elle testée ? La réponse est non. Cette absence de gourmandise peut parfois causer des surprises, puisque la seconde évaluation aurait pu apporter une solution plus "complète".
- où s'arrête l'analyse lorsqu'une correspondance est détectée ? l'analyse s'arrête sur le premier caractère non reconnu (ou la fin de chaîne).
- peut-on avoir cet opérateur en série (succession de **ou** logiques) ou en cascade (insertion d'une expression avec un ou plusieurs ou dans une expression complexe. La réponse est oui. Dans la série, la règle "de gauche à droite" s'applique.

note. le caractère `|` doit être protégé `\|` ou constituer un jeu de caractère `[|]` s'il est littéral

5.2. un exemple sans parenthésage.

problème : On dispose d'un texte comportant des noms de communes. On veut en extraire la liste des noms qui comportent le mot ville (ou Ville) en tant que chaîne terminale, c'est-à-dire suivi par un trait d'union ou en fin de nom. Par exemple Villers n'est pas admis ; mais Maville, Pont-Ville, Hauteville-Cajun, Ville-Sur-Cher le sont.

```
#!/usr/bin/python3
import re

texte ="Ablon\nAcqueville\nAgy\nAigner-Ville\nAiran\nAmaville-sur-Ville\
\nAmblie\nTrouville\nAngervillers\nAngoville\nArganchy\
\nAsnelles\nAsnières-Surville\nVille\nVilleneuve"
```

```
patron = "^.*ville-.*|^.*ville$"
cpatron = re.compile (patron,re.MULTILINE+re.IGNORECASE)
resultat = cpatron.findall(texte)
print (resultat)
```

```
['Acqueville', 'Aigner-Ville', 'Amaville-sur-Ville', 'Trouville',
 'Angoville', 'Asnières-Surville', 'Ville']
```

* **attention :** le point ne peut être un `\n`. En faisant `.*` on s'arrête automatiquement en fin de ligne si on l'atteint. Par contre dans la seconde partie de l'alternative, le $ est nécessaire.

* dans le cas de `Amaville-sur-Ville`, seule la première expression est évaluée.

5.2. exemples avec parenthésage.

exemple 1.

Rechercher dans une liste Python des adresses mail correctes qui ont pour extension terminale `.fr` ou `.com` ; sont donc rejetées les adresses incorrectes et celles qui n'ont pas la bonne extension.

```
#!/usr/bin/python3
import re

textes = ["jean@free.fr","marie.jeanne@free.fr","phil@laposte.net",
          "bob.robert@dom.com", "richard.hernu@asso.net" ]

patron = "^\w+(?:[.]\w+)?@\w+(?:[.]\w+)*[.](?:fr|com)$"
cpatron = re.compile(patron)

for x in textes:
    res = cpatron.search(x)
    if res:
        print (x)
```

```
>>>
jean@free.fr
marie.jeanne@free.fr
bob.robert@dom.com
>>>
```

exemple 2 : même chose avec `findall()`

```
#!/usr/bin/python3
import re

textes = "jean@free.fr,marie.jeanne@free.fr,phil@laposte.net,\
bob.robert@dom.com,richard.hernu@asso.net"

patron = "\\b\w+(?:[.]\w+)?@\w+(?:[.]\w+)*[.](?:fr|com)\\b"
cpatron = re.compile(patron)
```

```
print (cpatron.findall(textes))
```

```
>>>
['jean@free.fr', 'marie.jeanne@free.fr', 'bob.robert@dom.com']
>>>
```

6. Les directives insérées dans les expressions

On peut insérer les directives dans le patron. La syntaxe est la suivante :

(?iLmsuxa)

où une ou plusieurs lettres peuvent être absentes : **i** pour **IGNORECASE**, **L** pour **LOCALE**, **m** pour **MULTILINE**, **s** pour **DOTALL**, **u** pour **UNICODE**, **x** pour **VERBOSE**, **a** pour **ASCII**.

Étant donné le processus que nous avons recommandé, l'usage en est rare ; il n'en est pas de même si on avait choisi une autre méthodologie, par exemple celle des fonctions du module re (fonction **re.search()**, **re.findall()** etc).

fiche 3 : les groupes

introduction.

On a déjà rencontré un mode de groupement : le "parenthésage" ; il existe un second système appelé groupe, qui a, en plus des propriétés du parenthésage, des propriétés très spécifiques. On prendra garde que le "groupe" n'est pas l'équivalent du parenthésage mathématique.

1. Les groupes.

1.1. groupement de sous-expressions régulières.

Un patron est un groupe ; c'est le groupe de niveau le plus bas, le niveau 0. Si on veut définir des sous-groupes, il suffit de parenthéser des sous-expressions régulières constituant l'expression globale. Les groupes sont numérotés : 0 pour le patron, 1, 2,... pour les sous-groupes. La numérotation se fait de gauche à droite, et pour un groupe, elle est égale **au nombre total de parenthèses ouvrantes qui le précède**.

On récupère les groupes en analysant l'objet **`SRE_Pattern`** retourné par **`search()`** ; quant à la méthode **`findall()`** elle retourne une liste groupes ou de tuples (n-uplets) de groupes (voir exemple ci-dessous).

note. Les littéraux **`(`** et **`)`** doivent être protégés. Mais **attention** ! Le point d'interrogation ne peut suivre la parenthèse ouvrante ; il faut aussi le protéger : **`(?`** se code **`\(\?`**.

Les opérateurs comme *, +, ?, |, qui s'appliquent à un caractère ou une expression paranthésée s'appliquent à un groupe.

1.2. exemples avec search().

exemple 1.

```
#!/usr/bin/python3
import re
texte ="Ablon\nAcqueville\nAgy\nAigner-Ville\nAiran\nAmayé-sur-Orne\
\nAmblie\nAmfreville\nAngervillers\nAngoville\nArganchy\nArgences\
\nArromanches-les-Bains\nAsnelles\nAsnières-Surville"

patron = "(.*)ville$"
cpatron = re.compile (patron,re.MULTILINE)
resultat = cpatron.search(texte)
print (resultat.start(0),resultat.end(0), resultat.group(0))
print (resultat.start(1),resultat.end(1), resultat.group(1))
```

```
>>>
6 16 Acqueville
6 11 Acque
>>>
```

exemple 2.

```
#!/usr/bin/python3
import re
texte ="Ablon\nAcqueville\nAgy\nAigner-Ville\nAiran\nAmayé-sur-Orne\
\nAmblie\nAmfreville\nAngervillers\nAngoville\nArganchy\nArgences\
```

```
\nArromanches-les-Bains\nAsnelles\nAsnières-Surville"

patron = "(.*)(vi\w*)$"
cpatron = re.compile (patron,re.MULTILINE)
resultat = cpatron.search(texte)
print (resultat.start(0),resultat.end(0), resultat.group(0))
print (resultat.start(1),resultat.end(1), resultat.group(1))
print (resultat.start(2),resultat.end(2), resultat.group(2))
```

```
>>>
6 16 Acqueville
6 11 Acque
11 16 ville
>>>
```

1.3. exemples avec findall().

`findall()` ne reprend pas ce qu'on a trouvé avec le patron, mais ses sous-groupes.

exemple 1.

```
#!/usr/bin/python3
import re
texte ="Ablon\nAcqueville\nAgy\nAigner-Ville\nAiran\nAmayé-sur-Orne\
\nAmblie\nAmfreville\nAngervillers\nAngoville\nArganchy\nArgences\
\nArromanches-les-Bains\nAsnelles\nAsnières-Surville"

patron = "(.*)vi\w*$"
cpatron = re.compile (patron, re.MULTILINE)
resultat = cpatron.findall(texte)
print (resultat)
```

```
>>>
['Acque', 'Amfre', 'Anger', 'Ango', 'Asnières-Sur']
>>>
```

exemple 2, avec deux sous-groupes.

```
#!/usr/bin/python3
import re
texte ="Ablon\nAcqueville\nAgy\nAigner-Ville\nAiran\nAmayé-sur-Orne\
\nAmblie\nAmfreville\nAngervillers\nAngoville\nArganchy\nArgences\
\nArromanches-les-Bains\nAsnelles\nAsnières-Surville"

patron = "(.*)(vi\w*)$"
cpatron = re.compile (patron, re.MULTILINE+re.IGNORECASE)
resultat = cpatron.findall(texte)
print (resultat)
```

```
>>>
[('Acque', 'ville'), ('Aigner-', 'Ville'), ('Amfre', 'ville'),
('Anger', 'villers'), ('Ango', 'ville'), ('Asnières-Sur', 'ville')]
>>>
```

exemple 3

On reprend l'exemple précédent, mais en transformant chaque alternative en groupe. La méthode **`findall()`** continue à donner l'évaluation des deux groupes. Cependant, si c'est le premier groupe qui est reconnu, le second n'est pas évalué (absence de gourmandise) : il est vide ; si c'est le second qui est reconnu, le premier est vide parce que non reconnu.

```
#!/usr/bin/python3
import re

texte ="Ablon\nAcqueville\nAgy\nAigner-Ville\nAiran\nAmaville-sur-Ville\
\nAmblie\nTrouville\nAngervillers\nAngoville\nArganchy\
\nAsnelles\nAsnières-Surville\nVille\nVilleneuve"

patron = "(^.*ville-.*)|(^.*ville$)"
cpatron = re.compile (patron,re.MULTILINE+re.IGNORECASE)
resultat = cpatron.findall(texte)
print (resultat)
```

```
>>>
[('', 'Acqueville'), ('', 'Aigner-Ville'), ('Amaville-sur-Ville', ''),
 ('', 'Trouville'), ('', 'Angoville'), ('', 'Asnières-Surville'), ('',
 'Ville')]
>>>
```

2. Numéroter et nommer les groupes.

2.1. règle de numérotation.

rappel : La numérotation se fait de gauche à droite, et pour un groupe, elle est égale au nombre total de parenthèses ouvrantes qui le précède.

```
#!/usr/bin/python3
import re
texte = "   abcdefghijklmnopqrstuvwxyz   "

patron = "abc(def(ghi)(jkl))(mno)pqr"
cpatron = re.compile(patron)

res = cpatron.search(texte)
if res :
    try:
        for i in range (99) : # maximum de 99 groupes
            print ("groupe", i , ":", res.group(i))
    except:
        pass
```

donne :

```
>>>
groupe 0 : abcdefghijklmnopqr
groupe 1 : defghijkl
groupe 2 : ghi
groupe 3 : jkl
groupe 4 : mno
>>>
```

attention : Si le patron comporte un ou logique la règle de nmérotage continue à s'appliquer. Par exemple, le patron "ab(c)d|xy(z)" comporte deux groupes : numérotés 1 (c) et 2 (z).

2.2. Nommer les groupes.

Il est possible de donner un nom à un groupe au moment où on le définit. Le nom doit être un identificateur Python valide. Il n'y a aucune interférence avec la numérotation.

La syntaxe est

(?P<*nom du groupe*>*expression régulière*)

exemple :

```
#!/usr/bin/python3
import re
texte = "   abcdefghijklmnopqrstuvwxyz   "

patron = "abc(?P<gr1>def(?P<gr2>ghi)(?P<gr3>jkl))(?P<gr4>mno)pqr"
cpatron = re.compile(patron)

res = cpatron.search(texte)
if res :
    for i in range (1, 99) :
        try :
            print ("groupe "+str(i), res.group("gr"+str(i)))
        except :
            pass
```

donne :

```
>>>
groupe 1 defghijkl
groupe 2 ghi
groupe 3 jkl
groupe 4 mno
>>>
```

3. Capturer les groupes et utiliser la capture.

3.1. capture des groupes.

Lorsqu'un groupe est identifié à un segment de chaîne, la valeur de la sous-chaîne identifiée peut être

"capturée", et utilisée dans l'expression régulière et dans la fonction de remplacement **`sub()`**.

La valeur reconnue se désigne dans l'expression régulière par un antislash suivi de son numéro (exemple : `\5` pour la chaîne du groupe 5) ou par la convention de nommage :

`(?P=`*`identificateur`*`)`

Mais l'antislash suivi d'un chiffre peut avoir une signification en Python, **il faut donc doubler l'antislash dans la chaîne Python**. Par ailleurs, cette notation devient ambiguë avec les valeurs au delà de 10. Comment interpréter **`\14`** ? groupe 1 suivi de 4 ou groupe 14 ? Dans le patron le nombre à deux chiffres est correctement interprété comme "groupe 14" ; le groupe 1 suivi de 4 doit s'écrire par exemple **`\\1[4]`**. Pour la fonction **`sub()`**, voir le paragraphe 3.4.

3.2. exemple d'usage du numéro

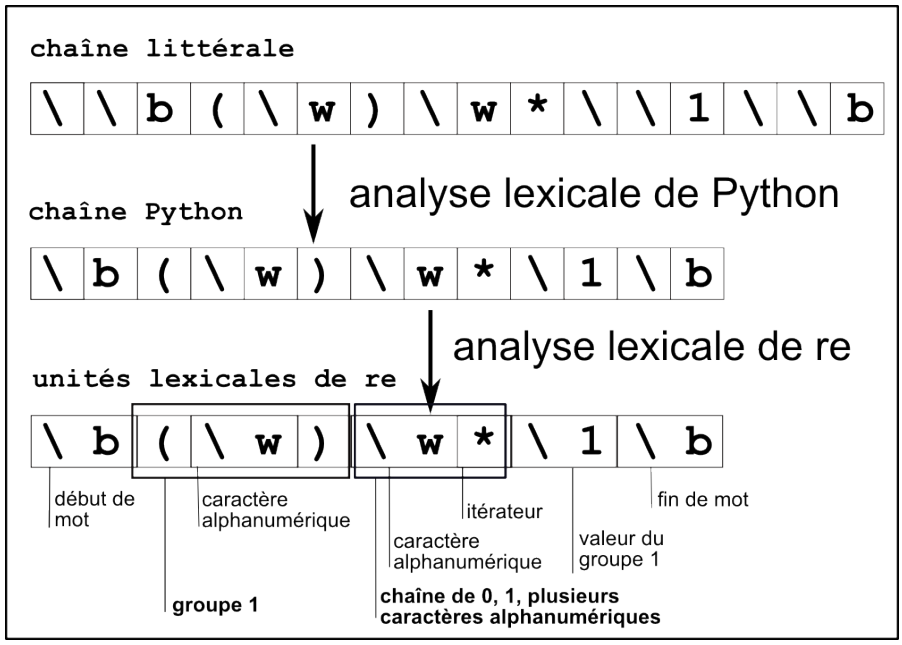
problème : trouver les mots d'un texte qui commencent et se terminent par la même lettre.

```
#!/usr/bin/python3
import re

texte = "J'ai été en sursis et j'irai bien désormais"

patron = "\\b(\w).*?\\1\\b"
cpatron = re.compile(patron)

while True :
    print (texte)
    res = cpatron.search(texte)
    if res :
        print (res.group(0))
        texte = texte [res.end():]
    else:
        break
```

donne :

```
>>>
J'ai été en sursis et j'irai bien désormais
été
 en sursis et j'irai bien désormais
sursis
 et j'irai bien désormais
irai
 bien désormais
>>>
```

schéma de la double analyse syntaxique avec groupe.



On a figuré le groupe et la chaîne dont l'identification relève de l'analyse syntaxique.

3.3. usage de la convention de nommage.

exemple

```
#!/usr/bin/python3
import re

texte = "J'ai été en sursis et j'irai bien désormais"

patron = "\\b(?P<premiereLettre>\w)\w*(?P=premiereLettre)\\b"
cpatron = re.compile(patron)

while True :
    res = cpatron.search(texte)
    if res :
        print (res.group(0))
        texte = texte [res.end():]
    else:
        break
```

donne :

```
>>>
été
sursis
irai
>>>
```

3.4. la fonction de remplacement sub() avec capture.

On peut utiliser les éléments capturés dans la chaîne de remplacement. Les conventions de nommage sont alors :

\g<*identificateur*>

\g<*numéro*>

Les symboles < et > ne sont pas indispensables si le numéro n'a qu'un chiffre ; mais cette facilité est à éviter à cause de l'ambiguïté qui naît avec les valeurs à deux chiffres.

Voici un **problème d'illustration** : on dispose d'un texte html, et on veut remplacer l'expression align=center par une autre chaîne, mais uniquement dans les balises de bloc de texte, c'est-à-dire commençant par h0,h1 ...h9 ou p. De plus, on veut éliminer tout le reste des attributs de ces balises.

```
import re

texte = """<h1 name="titre"
align=CENTER>align=center</h1>
<img src="kw154.png" align = center>
<p align = center id="zazozi"  >wawawa</p>
<h2 align= center>fin de texte</h2>"""

patron = """
(<h\d|<p)                    # balise commençant par <h avec chiffre ou <p
.*?                          # suite non gourmande de caractères
align\s*=\s*center           # suivie du motif recherché
.*?                          # suite non gourmande de caractères
(>)                          # terminée par >
"""
# le patron comporte deux groupes

cpatron = re.compile(patron, re.I+re.S+re.X) # IGNORECASE+DOTALL+VERBOSE
print (cpatron.findall(texte),"\n")

# les deux groupes sont utilisés dans la chaîne de remplacement.
res = cpatron.sub('\g<1> style=\'align:center;\' \g<2>', texte)

print (res)
```

```
>>>
[('<h1', '>'), ('<p', '>'), ('<h2', '>')]

<h1 style='text-align:center;' >align=center</h1>
<img src="kw154.png" align = center>
<p style='text-align:center;' >wawawa</p>
<h2 style='text-align:center;' >fin de texte</h2>
>>>
```

note : au lieu de numéros, on aurait pu utiliser des identificateurs...

fiche 4 : lookahead et lookbehind

1. Lookahead.

1.1. lookahead positif ou négatif.

le problème :

Supposons que l'on dispose de l'expression régulière **patron** mais que l'on veuille chercher une correspondance conditionnelle : seulement si l'expression à trouver est "suivie" (ou n'est pas suivie) d'une expression trouvée avec une autre expression régulière **hpatron**.

Par exemple on cherche la chaîne "**Isaac**" suivie par **"Newton" ou "Asimov"**. Ou le contraire. Évidemment, on peut compliquer un peu.

Une telle recherche s'appelle un **lookahead**. Sa caractéristique essentielle est qu'elle ne consomme que la chaîne recherchée, pas la chaîne conditionnelle.

La syntaxe est la suivante :

(?=*expression lookahead recherchée*)

(?!*expression lookehaed rejetée*)

1.2. lookahead postposé

exemple :

```
#!/usr/bin/python3
import re
texte = "isaac jacob, Isaac Newton, Isaac Asimov,\
 isaac isaac\n, isaac  Isaac"
patron ="Isaac\s(?=Asimov|Newton)"
cpatron = re.compile(patron, re.IGNORECASE)
print (cpatron.findall(texte))

patron ="Isaac(?:\s|$)(?!Asimov|Newton)"
cpatron = re.compile(patron, re.IGNORECASE)
print (cpatron.findall(texte))
```

```
>>>
['Isaac ', 'Isaac ']
['isaac ', 'isaac ', 'isaac\n', 'isaac ', 'Isaac']
>>>
```

* rappel **\s** pour un blanc (espace, fin de ligne).

* on voit bien sur le second cas **qu'il n'y a pas consommation du lookahead.**

1.3. lookahead préposé.

exemple 2 :

```
#!/usr/bin/python3
import re
texte = "isaac jacob, IsaacNewton, Isaac Asimov, bellisaac"
patron ="(?=\\bisa\w*\\b)isaac"
cpatron = re.compile(patron, re.IGNORECASE)
```

```
print (cpatron.findall(texte))
```

```
>>>
['isaac', 'Isaac', 'Isaac']
>>>
```

* le patron du lookahead est d'abord recherché ; s'il est trouvé, la recherche de la seconde partie du patron se fait **à partir du premier élément trouvé**, qui est ici un début de mot. On dit qu'**il y a chevauchement.**

2. Lookbehind.

2.1. lookbehind positif ou négatif.

Une recherche de correspondance avec comme condition de suivre (ou non) une correspondance définie par une autre expression régulière est un **lookbehind**. Le lookbehind consomme l'expression précédente, mais ne la retourne pas. Il n'y a pas de chevauchement avec le lookbehind, car la chaîne conditionnelle est avant la chaîne recherché.

Restriction : l'expression qui sert de condition doit avoir une longueur préfixée ; on ne peut avoir de `.*` ou `\w*` ; mais on peut avoir `toto|papa`. **Sinon, Il y a erreur à la compilation.**

syntaxe :

```
(?<=expression lookbehind recherchée)
(?<!expression lookbehind rejetée)
```

2.2. lookbehind préposé

exemple 1 :

```
#!/usr/bin/python3
import re
texte = "isaac jacob, IsaacNewton, IsaacAsimov, isaac"

patron ="(?<=isaac)[a-z]+"
cpatron = re.compile(patron, re.IGNORECASE)
print (cpatron.findall(texte))
```

donne :

```
>>>
['Newton', 'Asimov']
>>>
```

exemple 2

```
#!/usr/bin/python3
import re
texte = "isaac jacob, IsaacNewton, IsaacAsimov, mimieDany"

patron ="(?<=isaac|m[a-z]{4})[a-z]+"
cpatron = re.compile(patron, re.IGNORECASE)
print (cpatron.findall(texte))
```

donne :

```
>>>
['Newton', 'Asimov', 'Dany']
>>>
```

2.2. lookbehind postposé.

problème : dans une liste de noms/prénoms, sortir les noms de 6 lettres précédées par un prénom de 5 lettres.

```
#!/usr/bin/python3
import re
texte = "isaac jacob, Isaac Newton, Isaac Léa, Isaac Asimov"

motif ="\w{6}(?<=\w{5}\s\w{6})"
cmotif = re.compile(motif, re.IGNORECASE)
print (cmotif.findall(texte))
```

```
>>>
['Newton', 'Asimov']
>>>
```

Le mécanisme est le suivant : il y a d'abord recherche d'un mot de 6 lettres ; puis, à partir du dernier élément consommé, évaluation de la condition **lookbehind** qui doit se terminer sur ce dernier élément consommé. Il n'y a pas chevauchement puisque tout se passe avant le dernier élément consommé, pas après comme dans le **lookahead**.

on a donc les 4 schémas possibles suivants :