Guide des expressions régulières

Version 3.11.8

Guido van Rossum and the Python development team

avril 02, 2024

Python Software Foundation
Email : docs@python.org

Table des matières

Auteur
A.M. Kuchling <amk@amk.ca>

Résumé

Ce document constitue un guide d'introduction à l'utilisation des expressions régulières en Python avec le module `re`. Il fournit une introduction plus abordable que la section correspondante dans le guide de référence de la bibliothèque.

1 Introduction

Les expressions régulières (notées RE ou motifs *regex* dans ce document) sont essentiellement un petit langage de programmation hautement spécialisé embarqué dans Python et dont la manipulation est rendue possible par l'utilisation du module `re`. En utilisant ce petit langage, vous définissez des règles pour spécifier une correspondance avec un ensemble souhaité de chaînes de caractères ; ces chaînes peuvent être des phrases, des adresses de courriel, des commandes *TeX* ou tout ce que vous voulez. Vous pouvez ensuite poser des questions telles que « Est-ce que cette chaîne de caractères correspond au motif ? » ou « Y a-t-il une correspondance pour ce motif à l'intérieur de la chaîne de caractères ? ». Vous pouvez aussi utiliser les RE pour modifier une chaîne de caractères ou la découper de différentes façons.

Un motif d'expression régulière est compilé en code intermédiaire (*bytecode* en anglais) qui est ensuite exécuté par un moteur de correspondance écrit en C. Pour une utilisation plus poussée, il peut s'avérer nécessaire de s'intéresser à la manière dont le moteur exécute la RE afin d'écrire une expression dont le code intermédiaire est plus rapide. L'optimisation n'est pas traitée dans ce document, parce qu'elle nécessite d'avoir une bonne compréhension des mécanismes internes du moteur de correspondance.

Le langage des expressions régulières est relativement petit et restreint, donc toutes les tâches de manipulation de chaînes de caractères ne peuvent pas être réalisées à l'aide d'expressions régulières. Il existe aussi des tâches qui *peuvent* être réalisées à l'aide d'expressions régulières mais qui ont tendance à produire des expressions régulières très compliquées. Dans ces cas, il est plus utile d'écrire du code Python pour réaliser le traitement ; même si le code Python est plus lent qu'une expression régulière élaborée, il sera probablement plus compréhensible.

2 Motifs simples

Nous commençons par étudier les expressions régulières les plus simples. Dans la mesure où les expressions régulières sont utilisées pour manipuler des chaînes de caractères, nous commençons par l'action la plus courante : la correspondance de caractères.

Pour une explication détaillée sur le concept informatique sous-jacent aux expressions régulières (automate à états déterministe ou non-déterministe), vous pouvez vous référer à n'importe quel manuel sur l'écriture de compilateurs.

2.1 Correspondance de caractères

La plupart des lettres ou caractères correspondent simplement à eux-mêmes. Par exemple, l'expression régulière `test` correspond à la chaîne de caractères `test`, précisément. Vous pouvez activer le mode non-sensible à la casse qui permet à cette RE de correspondre également à `Test` ou `TEST` (ce sujet est traité par la suite).

Il existe des exceptions à cette règle ; certains caractères sont des *métacaractères* spéciaux et ne correspondent pas à eux-mêmes. Au lieu de cela, ils signalent que certaines choses non ordinaires doivent correspondre, ou ils affectent d'autre portions de la RE en les répétant ou en changeant leur sens. Une grande partie de ce document est consacrée au fonctionnement de ces métacaractères.

Voici une liste complète des métacaractères ; leur sens est décrit dans la suite de ce guide.

```
. ^ $ * + ? { } [ ] \ | ( )
```

Les premiers métacaractères que nous étudions sont `[` et `]`. Ils sont utilisés pour spécifier une classe de caractères, qui forme un ensemble de caractères dont vous souhaitez trouver la correspondance. Les caractères peuvent être listés

individuellement, ou une plage de caractères peut être indiquée en fournissant deux caractères séparés par un `'-'`. Par exemple, `[abc]` correspond à n'importe quel caractère parmi `a`, `b` ou `c` ; c'est équivalent à `[a-c]`, qui utilise une plage pour exprimer le même ensemble de caractères. Si vous voulez trouver une chaîne qui ne contient que des lettres en minuscules, la RE est `[a-z]`.

Les métacaractères (à l'exception de `\`) ne sont pas actifs dans les classes. Par exemple, `[akm$]` correspond à n'importe quel caractère parmi `'a'`, `'k'`, `'m'` ou `'$'` ; `'$'` est habituellement un métacaractère mais dans une classe de caractères, il est dépourvu de sa signification spéciale.

Vous pouvez trouver une correspondance avec les caractères non listés dans une classe en spécifiant le *complément* de l'ensemble. Ceci est indiqué en plaçant un `'^'` en tant que premier caractère de la classe. Par exemple, `[^5]` correspond à tous les caractères, sauf `'5'`. Si le caret se trouve ailleurs dans la classe de caractères, il ne possède pas de signification spéciale. Ainsi, `[5^]` correspond au `'5'` ou au caractère `'^'`.

Le métacaractère le plus important est probablement la barre oblique inverse (*backslash* en anglais), `\`. Comme dans les chaînes de caractères en Python, la barre oblique inverse peut être suivie par différents caractères pour signaler différentes séquences spéciales. Elle est aussi utilisée pour échapper tous les métacaractères afin d'en trouver les correspondances dans les motifs ; par exemple, si vous devez trouver une correspondance pour `[` ou `\`, vous pouvez les précéder avec une barre oblique inverse pour annuler leur signification spéciale : `\[` ou `\\`.

Certaines séquences spéciales commençant par `'\'` représentent des ensembles de caractères prédéfinis qui sont souvent utiles, tels que l'ensemble des chiffres, l'ensemble des lettres ou l'ensemble des caractères qui ne sont pas des « blancs ».

Prenons un exemple : `\w` correspond à n'importe quel caractère alphanumérique. Si l'expression régulière est exprimée en *bytes*, c'est équivalent à la classe `[a-zA-Z0-9_]`. Si l'expression régulière est une chaîne de caractères, `\w` correspond à tous les caractères identifiés comme lettre dans la base de données Unicode fournie par le module `unicodedata`. Vous pouvez utiliser la définition plus restrictive de `\w` dans un motif exprimé en chaîne de caractères en spécifiant l'option `re.ASCII` lors de la compilation de l'expression régulière.

La liste de séquences spéciales suivante est incomplète. Pour une liste complète des séquences et définitions de classes étendues relatives aux motifs de chaînes de caractères Unicode, reportez-vous à la dernière partie de la référence Syntaxe d'Expressions Régulières de la bibliothèque standard. En général, les versions Unicode trouvent une correspondance avec n'importe quel caractère présent dans la catégorie appropriée de la base de données Unicode.

`\d`
: Correspond à n'importe quel caractère numérique ; équivalent à la classe `[0-9]`.

`\D`
: Correspond à n'importe quel caractère non numérique ; équivalent à la classe `[^0-9]`.

`\s`
: Correspond à n'importe quel caractère « blanc » ; équivalent à la classe `[ \t\n\r\f\v]`.

`\S`
: Correspond à n'importe quel caractère autre que « blanc » ; équivalent à la classe `[^ \t\n\r\f\v]`.

`\w`
: Correspond à n'importe quel caractère alphanumérique ; équivalent à la classe `[a-zA-Z0-9_]`.

`\W`
: Correspond à n'importe quel caractère non-alphanumérique ; équivalent à la classe `[^a-zA-Z0-9_]`.

Ces séquences peuvent être incluses dans une classe de caractères. Par exemple, `[\s,.]` est une classe de caractères qui correspond à tous les caractères « blanc » ou `','` ou `'.'`.

Le dernier métacaractère de cette section est `.`. Il correspond à tous les caractères, à l'exception du caractère de retour à la ligne ; il existe un mode alternatif (`re.DOTALL`) dans lequel il correspond également au caractère de retour à la ligne. `.` est souvent utilisé lorsque l'on veut trouver une correspondance avec « n'importe quel caractère ».

2.2 Répétitions

Trouver des correspondances de divers ensembles de caractères est la première utilisation des expressions régulières, ce que l'on ne peut pas faire avec les méthodes des chaînes. Cependant, si c'était la seule possibilité des expressions régulières, le gain ne serait pas significatif. Une autre utilisation consiste à spécifier des portions d'une RE qui peuvent être répétées un certain nombre de fois.

Le premier métacaractère pour la répétition que nous abordons est `*`. `*` ne correspond pas au caractère littéral `'*'` ; à la place, il spécifie que le caractère précédent peut correspondre zéro, une ou plusieurs fois (au lieu d'une seule fois).

Par exemple, `cha*t` correspond à `'cht'` (0 caractère `'a'`), `'chat'` (1 `'a'`), `'chaaat'` (3 caractères `'a'`) et ainsi de suite.

Les répétitions telles que `*` sont *gloutonnes* ; quand vous répétez une RE, le moteur de correspondance essaie de trouver la correspondance la plus longue en répétant le motif tant qu'il le peut. Si la suite du motif ne correspond pas, le moteur de correspondance revient en arrière et essaie avec moins de répétitions.

Un exemple étape par étape mettra les choses au clair. Considérons l'expression `a[bcd]*b`. Elle correspond à la lettre `'a'`, suivi d'aucune ou plusieurs lettres de la classe `[bcd]` et finit par un `'b'`. Maintenant, supposons que nous cherchons une correspondance de cette RE avec la chaîne de caractères `'abcbd'`.

Étape	Correspond	Explication
1	`a`	Le `a` correspond dans la RE.
2	`abcbd`	Le moteur de correspondance trouve `[bcd]*`, va aussi loin qu'il le peut, c.-à-d. la fin de la chaîne.
3	*échec*	Le moteur essaie de trouver une correspondance avec `b` mais la position courante est à la fin de la chaîne de caractères, donc il échoue.
4	`abcb`	Retour en arrière, de manière à ce que `[bcd]*` corresponde avec un caractère de moins.
5	*échec*	Essaie encore `b`, mais la position courante est le dernier caractère, qui est `'d'`.
6	`abc`	Encore un retour en arrière, de manière à ce que `[bcd]*` ne corresponde qu'à `bc`.
6	`abcb`	Essaie `b` encore une fois. Cette fois, le caractère à la position courante est `'b'`, donc cela fonctionne.

La fin de la RE est maintenant atteinte et la correspondance trouvée est `'abcb'`. Ceci démontre comment le moteur de correspondance essaie d'aller le plus loin possible en premier et, si la correspondance échoue, il revient progressivement en arrière et ré-essaie avec le reste de la RE encore et encore. Il revient en arrière jusqu'à qu'il n'y ait aucune correspondance pour `[bcd]*` et, si cela échoue toujours, le moteur conclut que la chaîne de caractères et la RE ne correspondent pas du tout.

Un autre métacaractère de répétition est +, qui fait correspondre une ou plusieurs fois. Faites bien attention à la différence entre `*` et + ; `*` fait correspondre *zéro* fois ou plus, ainsi ce qui doit être répété peut ne pas être présent du tout, alors que + requiert au moins *une* occurrence. Pour continuer avec le même exemple, `cha+t` correspond avec `'chat'` (1 `'a'`), `'chaaat'` (3 `'a'`) mais ne correspond pas avec `'cht'`.

Il existe deux autres quantificateurs pour les répétitions. Le point d'interrogation, `?`, fait correspondre zéro ou une fois ; vous pouvez vous le représenter comme indiquant une option. Par exemple, `méta-?caractère` fait correspondre soit `'métacaractère'`, soit `'méta-caractère'`.

Le plus compliqué des quantificateurs est `{m,n}` où *m* et *n* sont des entiers décimaux. Ce quantificateur indique qu'il faut au moins *m* répétitions et au plus *n*. Par exemple, `a/{1,3}b` fait correspondre `'a/b'`, `'a//b'` et `'a///b'`. Elle ne fait pas correspondre `'ab'` (pas de barre oblique) ni `'a////b'` (quatre barres obliques).

Vous pouvez omettre soit *m*, soit *n* ; dans ce cas, une valeur raisonnable est prise pour la valeur manquante. Omettre *m* considère que la borne basse est 0 alors qu'omettre *n* signifie qu'il n'y a pas de borne supérieure.

The simplest case `{m}` matches the preceding item exactly *m* times. For example, `a/{2}b` will only match `'a//b'`.

Le lecteur attentif aura noté que les trois premiers quantificateurs peuvent être exprimés en utilisant cette notation. `{0,}` est la même chose que `*`, `{1,}` est équivalent à + et `{0,1}` se comporte comme `?`. Il est préférable d'utiliser `*`, + ou `?` quand vous le pouvez, simplement parce qu'ils sont plus courts et plus faciles à lire.

3 Utilisation des expressions régulières

Maintenant que nous avons vu quelques expressions régulières simples, utilisons-les concrètement. Le module `re` fournit une interface pour le moteur de correspondance, ce qui permet de compiler les RE en objets et d'effectuer des correspondances avec.

3.1 Compilation des expressions régulières

Les expressions régulières sont compilées en objets motifs, qui possèdent des méthodes pour diverses opérations telles que la recherche de correspondances ou les substitutions dans les chaînes.

```
>>> import re
>>> p = re.compile('ab*')
>>> p
re.compile('ab*')
```

`re.compile()` accepte aussi une option *flags*, utilisée pour activer des fonctionnalités particulières et des variations de syntaxe. Nous étudierons les options disponibles plus tard et, pour l'instant, un petit exemple suffit :

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

La RE passée à `re.compile()` est une chaîne. Les RE sont des chaînes car les expressions régulières ne font pas partie intrinsèque du langage Python et aucune syntaxe particulière n'a été créée pour les exprimer (il existe des applications qui ne nécessitent aucune RE et il n'a donc aucune raison de grossir les spécifications du langage en incluant les RE). Ainsi, le module `re` est simplement un module d'extension en C inclus dans Python, tout comme les modules `socket` ou `zlib`.

Exprimer les RE comme des chaînes de caractères garde le langage Python plus simple mais introduit un inconvénient qui fait l'objet de la section suivante.

3.2 La maudite barre oblique inverse

Comme indiqué précédemment, les expressions régulières utilisent la barre oblique inverse (*backslash* en anglais) pour indiquer des constructions particulières ou pour autoriser des caractères spéciaux sans que leur signification spéciale ne soit invoquée. C'est en contradiction avec l'usage de Python qui est qu'un caractère doit avoir la même signification dans les littéraux de chaînes de caractères.

Considérons que vous voulez écrire une RE qui fait correspondre la chaîne de caractères `\section` (on en trouve dans un fichier *LaTeX*). Pour savoir ce qu'il faut coder dans votre programme, commençons par la chaîne de caractères cherchée. Ensuite, nous devons échapper chaque barre oblique inverse et tout autre métacaractère en les précédant d'une barre oblique inverse, ce qui donne la chaîne de caractères `\\section`. La chaîne résultante qui doit être passée à `re.compile()` est donc `\\section`. Comme nous devons l'exprimer sous la forme d'une chaîne littérale Python, nous devons échapper les deux barres obliques inverses *encore une fois*.

Caractères	Niveau
`\section`	Chaîne de caractère à chercher
`\\section`	Barre oblique inverse échappée pour `re.compile()`
`"\\\\section"`	Barres obliques inverses échappées pour un littéral de chaîne de caractères

Pour faire court, si vous cherchez une correspondance pour une barre oblique inverse littérale, écrivez `'\\\\'` dans votre chaîne RE, car l'expression régulière doit être `\\` et que chaque barre oblique inverse doit être exprimée comme `\\` dans un littéral chaîne de caractères Python. Dans les RE qui comportent plusieurs barres obliques inverses, cela conduit à beaucoup de barres obliques inverses et rend la chaîne résultante difficile à comprendre.

La solution consiste à utiliser les chaînes brutes Python pour les expressions régulières ; les barres obliques inverses ne sont pas gérées d'une manière particulière dans les chaînes littérales préfixées avec `'r'`. Ainsi, `r"\n"` est la chaîne

de deux caractères contenant `'\'` et `'n'` alors que `"\n"` est la chaîne contenant uniquement le caractère retour à la ligne. Les expressions régulières sont souvent écrites dans le code Python en utilisant la notation « chaînes brutes ».

En complément, les séquences d'échappement valides dans les expressions régulières, mais non valides dans les littéraux chaînes classiques, produisent dorénavant un `DeprecationWarning` et, possiblement, deviendront une `SyntaxError`, ce qui signifie que les séquences seront invalides si la notation « chaîne brute » ou l'échappement des barres obliques inverses ne sont pas utilisés.

Chaîne normale	Chaîne de caractères brute
`"ab*"`	`r"ab*"`
`"\\\\section"`	`r"\\section"`
`"\\w+\\s+\\1"`	`r"\w+\s+\1"`

3.3 Recherche de correspondances

Une fois que nous avons un objet représentant une expression régulière compilée, qu'en faisons-nous ? Les objets motifs ont plusieurs méthodes et attributs. Seuls les plus significatifs seront couverts ici ; consultez la documentation `re` pour la liste complète.

Méthode/Attribut	Objectif
`match()`	Détermine si la RE fait correspond dès le début de la chaîne.
`search()`	Analyse la chaîne à la recherche d'une position où la RE correspond.
`findall()`	Trouve toutes les sous-chaînes qui correspondent à la RE et les renvoie sous la forme d'une liste.
`finditer()`	Trouve toutes les sous-chaînes qui correspondent à la RE et les renvoie sous la forme d'un itérateur.

`match()` et `search()` renvoient `None` si aucune correspondance ne peut être trouvée. Si elles trouvent une correspondance, une instance d'objet correspondance est renvoyée, contenant les informations relatives à la correspondance : position de départ et de fin, la sous-chaîne qui correspond et d'autres informations.

Vous pouvez apprendre leur fonctionnement en expérimentant de manière interactive avec le module `re`. Si vous disposez de `tkinter`, vous pouvez aussi regarder les sources de Tools/demo/redemo.py, un programme de démonstration inclus dans la distribution Python. Ce programme vous permet de rentrer des RE et des chaînes, affichant si la RE correspond ou pas. `redemo.py` peut s'avérer particulièrement utile quand vous devez déboguer une RE compliquée.

Ce guide utilise l'interpréteur standard de Python pour ses exemples. Commencez par lancer l'interpréteur Python, importez le module `re` et compilez une RE :

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
re.compile('[a-z]+')
```

Maintenant, vous pouvez tester des correspondances de la RE `[a-z]+` avec différentes chaînes. Une chaîne vide ne doit pas correspondre, puisque + indique « une ou plusieurs occurrences ». `match()` doit renvoyer `None` dans ce cas, ce qui fait que l'interpréteur n'affiche rien. Vous pouvez afficher le résultat de `match()` explicitement pour que ce soit clair.

```
>>> p.match("")
>>> print(p.match(""))
None
```

Maintenant, essayons sur une chaîne qui doit correspondre, par exemple `tempo`. Dans ce cas, `match()` renvoie un objet correspondance, vous pouvez ainsi stocker le résultat dans une variable pour une utilisation ultérieure.

```
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

Maintenant, vous pouvez interroger l'objet correspondance pour obtenir des informations sur la chaîne qui correspond. Les instances d'objets correspondances possèdent plusieurs méthodes et attributs ; les plus importants sont :

Méthode/Attribut	Objectif
`group()`	Renvoie la chaîne de caractères correspondant à la RE
`start()`	Renvoie la position de début de la correspondance
`end()`	Renvoie la position de fin de la correspondance
`span()`	Renvoie un *n*-uplet contenant les positions (début, fin) de la correspondance

Essayons ces méthodes pour clarifier leur signification :

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

`group()` renvoie la sous-chaîne qui correspond à la RE. `start()` et `end()` renvoient les indices de début et de fin de la correspondance. `span()` renvoie les indices de début et de fin dans un seul couple. Comme la méthode `match()` ne fait que vérifier si la RE correspond au début de la chaîne, `start()` vaut toujours zéro. Cependant, la méthode `search()` d'un motif analyse toute la chaîne, afin de trouver une correspondance potentielle qui ne commence pas à zéro.

```
>>> print(p.match('::: message'))
None
>>> m = p.search('::: message'); print(m)
<re.Match object; span=(4, 11), match='message'>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

Dans les programmes réels, la façon de faire la plus courante consiste à stocker l'objet correspondance dans une variable, puis à vérifier s'il vaut `None`. Généralement, cela ressemble à ceci :

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print('Match found: ', m.group())
else:
    print('No match')
```

Deux méthodes de motifs renvoient toutes les correspondances pour un motif. `findall()` renvoie une liste des chaînes qui correspondent :

```
>>> p = re.compile(r'\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
```

Le préfixe `r`, qui indique une chaîne brute littérale, est nécessaire dans cet exemple car les séquences d'échappement dans une chaîne littérale qui ne sont pas reconnues par Python, alors qu'elles le sont par les expressions régulières, produisent maintenant un `DeprecationWarning` et deviendront possiblement des `SyntaxError`. Reportez-vous à *La maudite barre oblique inverse*.

`findall()` doit créer la liste entière avant de la renvoyer comme résultat. La méthode `finditer()` renvoie une séquence d'instances d'objets correspondances en tant qu'itérateur :

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
...     print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

3.4 Fonctions de niveau module

Vous n'avez pas besoin de créer un objet motif et d'appeler ses méthodes ; le module `re` fournit des fonctions à son niveau, ce sont `match()`, `search()`, `findall()`, `sub()` et ainsi de suite. Ces fonctions prennent les mêmes arguments que les méthodes correspondantes des objets motifs, avec la chaîne RE ajoutée en tant que premier argument. Elles renvoient toujours `None` ou une instance d'objet correspondance.

```
>>> print(re.match(r'From\s+', 'Fromage amk'))
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<re.Match object; span=(0, 5), match='From '>
```

En interne, ces fonctions créent simplement un objet motif pour vous et appellent la méthode appropriée de cet objet. Elles stockent également l'objet compilé dans un cache afin que les appels suivants qui utilisent la même RE n'aient pas besoin d'analyser le motif une nouvelle fois.

Devez-vous utiliser ces fonctions au niveau des modules ou devez-vous calculer le motif et appeler vous-même ses méthodes ? Si vous utilisez l'expression régulière à l'intérieur d'une boucle, la pré-compilation permet d'économiser quelques appels de fonctions. En dehors des boucles, il n'y a pas beaucoup de différence grâce au cache interne.

3.5 Options de compilation

Les options de compilation vous permettent de modifier le comportement des expressions régulières. Ces options sont accessibles dans le module `re` par deux noms, un long du type `IGNORECASE` et un court (une seule lettre) tel que `I` (si vous êtes habitués aux modificateurs de motifs Perl, la version courte utilise les mêmes lettres que Perl, par exemple la version courte de `re.VERBOSE` est `re.X`). Plusieurs options peuvent être spécifiées en appliquant l'opérateur bit-à-bit *OR* ; par exemple, `re.I | re.M` active à la fois les options `I` et `M`.

Vous trouvez ci-dessous le tableau des options disponibles, suivies d'explications détaillées.

Option	Signification
`ASCII`, `A`	Transforme plusieurs échappements tels que `\w`, `\b`, `\s` et `\d` de manière à ce qu'ils ne correspondent qu'à des caractères ASCII ayant la propriété demandée.
`DOTALL`, `S`	Fait en sorte que `.` corresponde à n'importe quel caractère, caractère de retour à la ligne inclus.
`IGNORECASE`, `I`	Recherche une correspondance sans tenir compte de la casse.
`LOCALE`, `L`	Recherche une correspondance en tenant compte de la configuration de la région.
`MULTILINE`, `M`	Correspondance multi-lignes, affecte `^` et `$`.
`VERBOSE`, `X` (pour *extended*, c.-à-d. étendu en anglais)	Active les RE verbeuses, qui peuvent être organisées de manière plus propre et compréhensible.

`re.I`

re.**IGNORECASE**

Perform case-insensitive matching ; character class and literal strings will match letters by ignoring case. For example, `[A-Z]` will match lowercase letters, too. Full Unicode matching also works unless the `ASCII` flag is used to disable non-ASCII matches. When the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the `IGNORECASE` flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters : 'İ' (U+0130, Latin capital letter I with dot above), 'ı' (U+0131, Latin small letter dotless i), 'ſ' (U+017F, Latin small letter long s) and 'K' (U+212A, Kelvin sign). `Spam` will match `'Spam'`, `'spam'`, `'spAM'`, or `'ſpam'` (the latter is matched only in Unicode mode). This lowercasing doesn't take the current locale into account ; it will if you also set the `LOCALE` flag.

re.**L**

re.**LOCALE**

Rend `\w`, `\W`, `\b`, `\B` et les correspondances non sensibles à la casse dépendants de la configuration de la région courante au lieu de la base de données Unicode.

Les *locales* sont une caractéristique de la bibliothèque C destinées à favoriser une programmation qui tient compte des différences linguistiques (NdT : nous utilisons *configuration de région* dans cette page pour désigner ce concept de la bibliothèque C). Par exemple, si vous traitez du texte français, vous souhaitez pouvoir écrire `\w+` pour faire correspondre les mots, mais `\w` ne correspond qu'à la classe de caractères `[A-Za-za-z]` en octets ; cette classe ne correspond pas avec les octets codant `é` ou `ç`. Si votre système est configuré correctement et que la configuration de région est définie sur 'français', certaines fonctions C diront à votre programme que l'octet codant `é` doit être considéré comme une lettre. Définir l'option `LOCALE` lors de la compilation d'une expression régulière fait que l'objet compilé résultant utilise ces fonctions C pour `\w` ; c'est plus lent mais cela permet à `\w+` de correspondre avec les mots français tel qu'attendu. L'utilisation de cette option est déconseillée en Python 3 car le mécanisme de locale est très peu fiable, il ne gère qu'une seule « culture » à la fois et il ne fonctionne qu'avec des locales 8 bits. La correspondance Unicode est déjà activée par défaut dans Python 3 pour les motifs Unicode (type *str*) et elle est capable de gérer différentes configurations de régions.

re.**M**

re.**MULTILINE**

Nota : `^` et `$` n'ont pas encore été expliqués ; ils sont introduits dans la section *Plus de métacaractères*.

Normalement, `^` correspond uniquement au début de la chaîne, et `$` correspond uniquement à la fin de la chaîne et immédiatement avant la nouvelle ligne (s'il y en a une) à la fin de la chaîne. Lorsque cette option est spécifiée, `^` correspond au début de la chaîne de caractères et au début de chaque ligne de la chaîne de caractères, immédiatement après le début de la nouvelle ligne. De même, le métacaractère `$` correspond à la fin de la chaîne de caractères ou à la fin de chaque ligne (précédant immédiatement chaque nouvelle ligne).

re.**S**

re.**DOTALL**

Fait que le caractère spécial `'.'` corresponde avec n'importe quel caractère, y compris le retour à la ligne ; sans cette option, `'.'` correspond avec tout, *sauf* le retour à la ligne.

re.**A**

re.**ASCII**

Fait que `\w`, `\W`, `\b`, `\B`, `\s` et `\S` ne correspondent qu'avec des caractères ASCII au lieu de l'ensemble des caractères Unicode. Cette option n'a de sens que pour des motifs Unicode, elle est ignorée pour les motifs *bytes*.

re.**X**

re.**VERBOSE**

Cette option vous permet d'écrire des expressions régulières plus lisibles en vous permettant plus de flexibilité pour le formatage. Lorsque cette option est activée, les « blancs » dans la chaîne RE sont ignorés, sauf lorsque le « blancs » se trouve dans une classe de caractères ou est précédé d'une barre oblique inverse ; ceci vous permet d'organiser et d'indenter vos RE plus clairement. Cette option vous permet également de placer des commentaires dans une RE, ils seront ignorés par le moteur ; les commentaires commencent par un `'#'` qui n'est ni dans une classe de caractères, ni précédé d'une barre oblique inverse.

Par exemple, voici une RE qui utilise `re.VERBOSE` ; vous pouvez constater qu'elle est beaucoup plus facile à lire

```
charref = re.compile(r"""
 &[#]                # Start of a numeric entity reference
 (
     0[0-7]+         # Octal form
   | [0-9]+          # Decimal form
   | x[0-9a-fA-F]+   # Hexadecimal form
 )
 ;                   # Trailing semicolon
""", re.VERBOSE)
```

Sans l'option verbeuse, cette RE ressemble à ceci :

```
charref = re.compile("&#(0[0-7]+"
                     "|[0-9]+"
                     "|x[0-9a-fA-F]+);")
```

Dans l'exemple ci-dessus, Python concatène automatiquement les littéraux chaînes de caractères qui ont été utilisés pour séparer la RE en petits morceaux, mais la RE reste plus difficile à comprendre que sa version utilisant `re.VERBOSE`.

4 Des motifs plus puissants

Jusqu'à présent nous avons seulement couvert une partie des fonctionnalités des expressions régulières. Dans cette section, nous couvrirons quelques nouveaux métacaractères et l'utilisation des groupes pour récupérer des portions de textes correspondantes.

4.1 Plus de métacaractères

Nous n'avons pas encore couvert tous les métacaractères. Cette section traite de la plupart de ceux que nous n'avons pas abordés.

Certains métacaractères restants sont des *assertions de largeur zéro* (*zero-width assertions* en anglais). Ils ne font pas avancer le moteur dans la chaîne de caractères ; ils ne consomment aucun caractère et ils réussissent ou échouent tout simplement. Par exemple, `\b` est une assertion selon laquelle la position actuelle est située à la limite d'un mot ; la position n'est pas modifiée par le " b ". Cela signifie que les assertions de largeur zéro ne doivent pas être répétées car, si elles correspondent à un endroit donné, elles correspondent automatiquement un nombre infini de fois.

|

Union ensembliste ou opérateur « ou ». Si *A* et *B* sont des expressions régulières, `A|B` correspond à toute chaîne qui correspond à A ou à B. La priorité de `|` est très faible afin de pouvoir effectuer simplement des unions de chaînes de plusieurs caractères. `Crow|Servo` correspond avec `'Crow'` ou `'Servo'`, mais pas avec `'Cro'`, un `'w'` ou un `'S'`, ou encore `'ervo'`.

Pour correspondre avec un `'|'` littéral, utilisez `\|` ou placez-le dans une classe de caractères, comme ceci `[|]`.

^

Correspond à un début de ligne. À moins que l'option `MULTILINE` ne soit activée, cela ne fait correspondre que le début de la chaîne. Dans le mode `MULTILINE`, cela fait aussi correspondre immédiatement après chaque nouvelle ligne à l'intérieur de la chaîne.

Par exemple, si vous voulez trouver le mot `From` uniquement quand il est en début de ligne, la RE à utiliser est `^From`.

```
>>> print(re.search('^From', 'From Here to Eternity'))
<re.Match object; span=(0, 4), match='From'>
>>> print(re.search('^From', 'Reciting From Memory'))
None
```

Pour trouver un `'^'` littéral, utilisez `\^`.

$

Correspond à une fin de ligne, ce qui veut dire soit la fin de la chaîne, soit tout emplacement qui est suivi du caractère de nouvelle ligne.

```
>>> print(re.search('}$', '{block}'))
<re.Match object; span=(6, 7), match='}'>
>>> print(re.search('}$', '{block} '))
None
>>> print(re.search('}$', '{block}\n'))
<re.Match object; span=(6, 7), match='}'>
```

Pour trouver un `'$'` littéral, utilisez `\$` ou placez-le à l'intérieur d'une classe de caractères, comme ceci `[$]`.

\A

Correspond au début de la chaîne de caractères, uniquement. Si l'option `MULTILINE` n'est pas activée, `\A` et `^` sont équivalents. Dans le mode `MULTILINE`, ils sont différents : `\A` ne correspond toujours qu'au début de la chaîne alors que `^` correspond aussi aux emplacements situés immédiatement après une nouvelle ligne à l'intérieur de la chaîne.

\Z

Correspond uniquement à la fin d'une chaîne de caractères.

\b

Limite de mot. C'est une assertion de largeur zéro qui correspond uniquement aux positions de début et de fin de mot. Un mot est défini comme une séquence de caractères alphanumériques ; ainsi, la fin d'un mot est indiquée par un « blanc » ou un caractère non-alphanumérique.

L'exemple suivant fait correspondre `class` seulement si c'est un mot complet ; il n'y a pas de correspondance quand il est à l'intérieur d'un autre mot.

```
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<re.Match object; span=(3, 8), match='class'>
>>> print(p.search('the declassified algorithm'))
None
>>> print(p.search('one subclass is'))
None
```

Quand vous utilisez cette séquence spéciale, gardez deux choses à l'esprit. Tout d'abord, c'est la pire collision entre les littéraux des chaînes Python et les séquences d'expressions régulières. Dans les littéraux de chaîne de caractères Python, `\b` est le caractère de retour-arrière (*backspace* en anglais), dont la valeur ASCII est 8. Si vous n'utilisez pas les chaînes de caractères brutes, alors Python convertit le `\b` en retour-arrière, et votre RE ne correspond pas à ce que vous attendez. L'exemple suivant ressemble à notre RE précédente, mais nous avons omis le `'r'` devant la chaîne RE.

```
>>> p = re.compile('\bclass\b')
>>> print(p.search('no class at all'))
None
>>> print(p.search('\b' + 'class' + '\b'))
<re.Match object; span=(0, 7), match='\x08class\x08'>
```

Ensuite, dans une classe de caractères, où cette assertion n'a pas lieu d'être, `\b` représente le caractère retour-arrière, afin d'être compatible avec les littéraux de chaînes de caractères.

\B

Encore une assertion de largeur zéro, qui est l'opposée de `\b`, c'est-à-dire qu'elle fait correspondre uniquement les emplacements qui ne sont pas à la limite d'un mot.

4.2 Regroupement

Souvent, vous avez besoin d'obtenir plus d'informations que le simple fait que la RE corresponde ou non. Ainsi, les expressions régulières sont souvent utilisées pour analyser des chaînes de caractères en écrivant une RE qui divise une chaîne en plusieurs sous-groupes, correspondant chacun à une information particulière. Par exemple, une ligne d'en-tête RFC-822 peut se diviser en un nom d'en-tête et une valeur associée, séparés par un `':'`, comme ceci :

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
MIME-Version: 1.0
To: editor@example.com
```

Vous pouvez alors écrire une expression régulière qui fait correspondre une ligne d'en-tête entière et qui comporte un groupe correspondant au nom de l'en-tête, et un autre groupe correspondant à la valeur de l'en-tête.

Les groupes sont délimités par les métacaractères marqueurs `'('` et `')'`. `'('` et `')'` ont à peu près le même sens que dans les expressions mathématiques ; ils forment un groupe à partir des expressions qu'ils encadrent ; vous pouvez répéter le contenu d'un groupe à l'aide d'un quantificateur, comme `*`, `+`, `?` ou `{m,n}`. Par exemple, `(ab)*` correspond à zéro, une ou plusieurs fois `ab`.

```
>>> p = re.compile('(ab)*')
>>> print(p.match('abababababab').span())
(0, 10)
```

Les groupes délimités par `'('` et `')'` capturent également les indices de début et de fin du texte avec lequel ils correspondent ; ces indices peuvent être récupérés en passant un argument à `group()`, `start()`, `end()` ou `span()`. Les groupes sont numérotés à partir de 0, le groupe 0 étant toujours présent ; c'est l'ensemble de la RE, donc les méthodes de l'objet correspondance ont le groupe 0 comme argument par défaut. Plus loin, nous voyons comment exprimer les groupes qui ne capturent pas l'étendue du texte avec lequel ils correspondent.

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

Les sous-groupes sont numérotés de la gauche vers la droite, à partir de 1. Les groupes peuvent être imbriqués ; pour déterminer le numéro, il vous suffit de compter le nombre de parenthèses ouvrantes de la gauche vers la droite.

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

Vous pouvez passer plusieurs numéros de groupes à `group()` en même temps, elle vous renvoie alors un *n*-uplet contenant les valeurs correspondantes pour ces groupes.

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

La méthode `groups()` renvoie un *n*-uplet contenant les chaînes pour tous les sous-groupes, en commençant par le numéro 1 jusqu'au dernier.

```
>>> m.groups()
('abc', 'b')
```

Les renvois dans un motif vous permettent de spécifier que le contenu d'un groupe précédent doit aussi être trouvé à l'emplacement actuel dans la chaîne. Par exemple, `\1` réussit si le contenu du premier groupe se trouve aussi à la position courante, sinon il échoue. Rappelez-vous que les littéraux de chaînes Python utilisent aussi la barre oblique inverse suivie d'un nombre pour insérer des caractères arbitraires dans une chaîne ; soyez sûr d'utiliser une chaîne brute quand vous faites des renvois dans une RE.

Par exemple, la RE suivante détecte les mots doublés dans une chaîne.

```
>>> p = re.compile(r'\b(\w+)\s+\1\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

Les renvois tels que celui-ci ne sont pas très utiles pour effectuer une simple recherche dans une chaîne — il n'y a que peu de formats de textes qui répètent des données ainsi — mais vous verrez bientôt qu'ils sont *très* utiles pour effectuer des substitutions dans les chaînes.

4.3 Groupes non de capture et groupes nommés

Les RE élaborées peuvent utiliser de nombreux groupes, à la fois pour capturer des sous-chaînes intéressantes ainsi que pour regrouper et structurer la RE elle-même. Dans les RE complexes, il devient difficile de garder la trace des numéros de groupes. Deux caractéristiques aident à résoudre ce problème, toutes deux utilisant la même syntaxe d'extension des expressions régulières. Nous allons donc commencer en examinant cette syntaxe.

Les puissantes extensions des expressions régulières de Perl 5 sont réputées. Pour les mettre en œuvre, les développeurs Perl ne pouvaient pas utiliser de nouveaux métacaractères simples ou de nouvelles séquences commençant par `\` sans que les RE Perl ne deviennent trop différentes des RE standards au point de créer de la confusion. S'ils avaient choisi `&` comme nouveau métacaractère, par exemple, les expressions déjà écrites auraient considéré que `&` était un caractère standard et ne l'aurait pas échappé en écrivant `\&` ou `[&]`.

La solution adoptée par les développeurs Perl a été d'utiliser `(?...)` comme syntaxe d'extension. Placer `?` immédiatement après une parenthèse était une erreur de syntaxe, parce que le `?` n'a alors rien à répéter. Ainsi, cela n'a pas introduit de problème de compatibilité. Les caractères qui suivent immédiatement le `?` indiquent quelle extension est utilisée, donc `(?=truc)` est une chose (une assertion positive anticipée) et `(?:truc)` est une autre chose (la sous-expression `truc` que l'on groupe).

Python gère plusieurs des extensions Perl et rajoute une extension à la syntaxe des extensions Perl. Si le premier caractère après le point d'interrogation est `P`, cela signifie que c'est une extension spécifique à Python.

Après avoir vu la syntaxe générale d'extension des RE, nous pouvons revenir aux fonctionnalités qui simplifient le travail avec les groupes dans des RE complexes.

Parfois, vous souhaitez utiliser un groupe pour marquer une partie de l'expression régulière mais le contenu de ce groupe ne vous intéresse pas vraiment. Vous pouvez l'indiquer explicitement en utilisant la syntaxe de groupe, mais sans indiquer que vous voulez en capturer le contenu : `(?:...)`, où vous remplacez les `...` par n'importe quelle expression régulière.

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

À part le fait que vous n'avez pas accès au contenu du groupe, un groupe se comporte exactement de la même manière qu'un groupe de capture ; vous pouvez placer n'importe quoi dedans, spécifier une répétition avec un métacaractère tel que `*` et l'imbriquer dans un autre groupe (de capture ou pas). `(?:...)` est particulièrement utile quand vous modifiez des motifs existants, puisque vous pouvez ajouter de nouveaux groupes sans changer la façon dont les autres groupes sont numérotés. Nous devons mentionner ici qu'il n'y a aucune différence de performance dans la recherche de groupes, de capture ou non ; les deux formes travaillent à la même vitesse.

Une fonctionnalité plus importante est le nommage des groupes : au lieu d'y faire référence par des nombres, vous pouvez référencer des groupes par leur nom.

La syntaxe pour nommer les groupes est l'une des extensions spécifiques à Python : `(?P<nom>....)`. *nom* est, vous vous en doutez, le nom du groupe. Les groupes nommés se comportent exactement comme des groupes de capture, sauf qu'ils associent en plus un nom à un groupe. Les méthodes des objets correspondances qui gèrent les groupes de capture acceptent soit des entiers qui font référence aux numéros des groupes, soit des chaînes de caractères qui désignent les noms des groupes désirés. Les groupes nommés se voient toujours attribuer un numéro, vous pouvez ainsi récupérer les informations d'un groupe de deux façons :

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search( '(((( Lots of punctuation )))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

De plus, vous pouvez récupérer les groupes nommés comme dictionnaire avec `groupdict()` :

```
>>> m = re.match(r'(?P<first>\w+) (?P<last>\w+)', 'Jane Doe')
>>> m.groupdict()
{'first': 'Jane', 'last': 'Doe'}
```

Les groupes nommés sont pratiques car il est plus facile de se rappeler un nom qu'un numéro. Voici un exemple de RE tirée du module `imaplib` :

```
InternalDate = re.compile(r'INTERNALDATE "'
        r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
        r'(?P<year>[0-9][0-9][0-9][0-9])'
        r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9])'
        r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9][0-9])'
        r'"')
```

Il est évidemment plus facile de récupérer `m.group('zonem')` que de se rappeler de récupérer le groupe 9.

La syntaxe des renvois dans une expression telle que `(....)\1` fait référence au numéro du groupe. Il y a naturellement une variante qui utilise le nom du groupe au lieu du numéro. C'est une autre extension Python : `(?P=nom)` indique que le contenu du groupe appelé *nom* doit correspondre à nouveau avec l'emplacement courant. L'expression régulière pour trouver des mots doublés, `\b(\w+)\s+\1\b` peut ainsi être ré-écrite en `\b(?P<mot>\w+)\s+(?P=mot)\b` :

```
>>> p = re.compile(r'\b(?P<word>\w+)\s+(?P=word)\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

4.4 Assertions prédictives

Une autre assertion de largeur zéro est l'assertion prédictive. Une assertion prédictive peut s'exprimer sous deux formes, la positive et la négative, comme ceci :

`(?=...)`
: Assertion prédictive positive. Elle réussit si l'expression régulière contenue, représentée ici par `...`, correspond effectivement à l'emplacement courant ; dans le cas contraire, elle échoue. Mais, une fois que l'expression contenue a été essayée, le moteur de correspondance n'avance pas ; le reste du motif est testé à l'endroit même où l'assertion a commencé.

`(?!...)`
: Assertion prédictive négative. C'est l'opposée de l'assertion positive ; elle réussit si l'expression régulière contenue *ne* correspond *pas* à l'emplacement courant dans la chaîne.

Pour rendre ceci plus concret, regardons le cas où une prédiction est utile. Considérons un motif simple qui doit faire correspondre un nom de fichier et le diviser en un nom de base et une extension, séparés par un `.`. Par exemple, dans `news.rc`, `news` est le nom de base et `rc` est l'extension du nom de fichier.

Le motif de correspondance est plutôt simple :

`.*[.].*$`

Notez que le `.` doit être traité spécialement car c'est un métacaractère, nous le plaçons donc à l'intérieur d'une classe de caractères pour ne faire correspondre que ce caractère spécifique. Notez également le `$` en fin ; nous l'avons ajouté pour nous assurer que tout le reste de la chaîne est bien inclus dans l'extension. Cette expression régulière fait correspondre `truc.bar`, `autoexec.bat`, `sendmail.cf` et `printers.conf`.

Maintenant, compliquons un peu le problème ; si nous voulons faire correspondre les noms de fichiers dont l'extension n'est pas `bat` ? voici quelques tentatives incorrectes :

`.*[.][^b].*$` Le premier essai ci-dessus tente d'exclure `bat` en spécifiant que le premier caractère de l'extension ne doit pas être `b`. Cela ne fonctionne pas, car le motif n'accepte pas `truc.bar`.

`.*[.]([^b]..|.[^a].|..[^t])$`

L'expression devient plus confuse si nous essayons de réparer la première solution en spécifiant l'un des cas suivants : le premier caractère de l'extension n'est pas `b` ; le deuxième caractère n'est pas `a` ; ou le troisième caractère n'est pas `t`. Ce motif accepte `truc.bar` et rejette `autoexec.bat`, mais elle nécessite une extension de trois lettres et n'accepte pas un nom de fichier avec une extension de deux lettres comme `sendmail.cf`. Compliquons encore une fois le motif pour essayer de le réparer.

`.*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$`

Pour cette troisième tentative, les deuxième et troisième lettres sont devenues facultatives afin de permettre la correspondance avec des extensions plus courtes que trois caractères, comme `sendmail.cf`.

Le motif devient vraiment compliqué maintenant, ce qui le rend difficile à lire et à comprendre. Pire, si le problème change et que vous voulez exclure à la fois `bat` et `exe` en tant qu'extensions, le modèle deviendra encore plus compliqué et confus.

Une assertion prédictive négative supprime toute cette confusion :

`.*[.](?!bat$)[^.]*$` Cette assertion prédictive négative signifie : si l'expression `bat` ne correspond pas à cet emplacement, essaie le reste du motif ; si `bat$` correspond, tout le motif échoue. Le `$` est nécessaire pour s'assurer que quelque chose comme `sample.batch`, où c'est seulement le début de l'extension qui vaut `bat`, est autorisé. Le `[^...]*` s'assure que le motif fonctionne lorsqu'il y a plusieurs points dans le nom de fichier.

Exclure une autre extension de nom de fichier est maintenant facile ; il suffit de l'ajouter comme alternative à l'intérieur de l'assertion. Le motif suivant exclut les noms de fichiers qui se terminent par `bat` ou `exe` :

`.*[.](?!bat$|exe$)[^.]*$`

5 Modification de chaînes

Jusqu'à présent, nous avons simplement effectué des recherches dans une chaîne statique. Les expressions régulières sont aussi couramment utilisées pour modifier les chaînes de caractères de diverses manières, en utilisant les méthodes suivantes des motifs :

Méthode/Attribut	Objectif
`split()`	Découpe la chaîne de caractère en liste, la découpant partout où la RE correspond
`sub()`	Recherche toutes les sous-chaînes de caractères où la RE correspond et les substitue par une chaîne de caractères différente
`subn()`	Fait la même chose que `sub()`, mais renvoie la nouvelle chaîne et le nombre de remplacements effectués

5.1 Découpage de chaînes

La méthode `split()` d'un motif divise une chaîne de caractères à chaque fois que la RE correspond, retournant la liste des morceaux. C'est similaire à la méthode `split()` mais est beaucoup plus générale par les délimiteurs que vous pouvez spécifier ; la méthode `split()` du type *string* ne gère que les découpages en suivant les « blancs » ou suivant une chaîne définie. Comme vous pouvez vous y attendre, il y a aussi une fonction `re.split()` de niveau module.

.split (*string*[, *maxsplit=0*])

Découpe *string* en suivant les correspondances de l'expression régulière. Si des parenthèses de capture sont utilisées dans la RE, leur contenu est également renvoyé dans la liste résultante. Si *maxsplit* n'est pas nul, au plus *maxsplit* découpages sont effectués.

Vous pouvez limiter le nombre de découpages effectués en passant une valeur pour *maxsplit*. Quand *maxsplit* n'est pas nul, au plus *maxsplit* découpages sont effectués et le reste de la chaîne est renvoyé comme dernier élément de la liste. Dans l'exemple suivant, le délimiteur est toute séquence de caractères non alphanumériques.

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

Parfois, vous voulez récupérer le texte entre les délimiteurs mais aussi quel était le délimiteur. Si des parenthèses de capture sont utilisées dans la RE, leurs valeurs sont également renvoyées dans la liste. Comparons les appels suivants :

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '... ', 'is', ' ', 'a', ' ', 'test', '.', '']
```

La fonction de niveau module `re.split()` ajoute la RE à utiliser comme premier argument, mais est par ailleurs identique.

```
>>> re.split(r'[\W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'([\W]+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

5.2 Recherche et substitution

Une autre tâche classique est de trouver toutes les correspondances d'un motif et de les remplacer par une autre chaîne. La méthode `sub()` prend une valeur de substitution, qui peut être une chaîne de caractères ou une fonction, et la chaîne à traiter.

.sub (*replacement*, *string*[, *count=0*])

Renvoie la chaîne obtenue en remplaçant les occurrences sans chevauchement les plus à gauche de la RE dans *string* par la substitution *replacement*. Si le motif n'est pas trouvé, *string* est renvoyée inchangée.

L'argument optionnel *count* est le nombre maximum d'occurrences du motif à remplacer ; *count* doit être un entier positif ou nul. La valeur par défaut de 0 signifie qu'il faut remplacer toutes les occurrences.

Voici un exemple simple utilisant la méthode `sub()`. Nous remplaçons les noms des couleurs par le mot `colour` :

```
>>> p = re.compile('(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

La méthode `subn()` fait la même chose mais renvoie un couple contenant la nouvelle valeur de la chaîne de caractères et le nombre de remplacements effectués :

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

Les correspondances vides ne sont remplacées que lorsqu'elles ne sont pas adjacentes à une correspondance vide précédente.

```
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b--d-'
```

Si *replacement* est une chaîne de caractères, toute barre oblique inverse d'échappement est traitée. C'est-à-dire que `\n` est converti en caractère de nouvelle ligne, `\r` est converti en retour chariot, et ainsi de suite. Les échappements inconnus comme `\&` sont laissés tels quels. Les renvois, tels que `\6`, sont remplacés par la sous-chaîne correspondante au groupe dans le RE. Ceci vous permet d'incorporer des parties du texte original dans la chaîne de remplacement résultante.

Cet exemple fait correspondre le mot `section` suivi par une chaîne encadrée par `{` et `}`, et modifie `section` en `subsection` :

```
>>> p = re.compile('section{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First} section{second}')
'subsection{First} subsection{second}'
```

Il existe aussi une syntaxe pour faire référence aux groupes nommés définis par la syntaxe `(?P<nom>....)`. `\g<nom>` utilise la sous-chaîne correspondante au groupe nommé `nom` et `\g<numéro>` utilise le numéro de groupe correspondant. `\g<2>` est donc l'équivalent de `\2`, mais n'est pas ambigu dans une chaîne de substitution telle que `\g<2>0` (`\20` serait interprété comme une référence au groupe 20 et non comme une référence au groupe 2 suivie du caractère littéral `'0'`). Les substitutions suivantes sont toutes équivalentes mais utilisent les trois variantes de la chaîne de remplacement.

```
>>> p = re.compile('section{ (?P<name> [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}','section{First}')
'subsection{First}'
```

replacement peut aussi être une fonction, ce qui vous donne encore plus de contrôle. Si *replacement* est une fonction, la fonction est appelée pour chaque occurrence non chevauchante de *pattern*. À chaque appel, un argument objet correspondance est passé à la fonction, qui peut utiliser cette information pour calculer la chaîne de remplacement désirée et la renvoyer.

Dans l'exemple suivant, la fonction de substitution convertit un nombre décimal en hexadécimal :

```
>>> def hexrepl(match):
...     "Return the hex string for a decimal number"
...     value = int(match.group())
...     return hex(value)
...
>>> p = re.compile(r'\d+')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
'Call 0xffd2 for printing, 0xc000 for user code.'
```

Quand vous utilisez la fonction de niveau module `re.sub()`, le motif est passé comme premier argument. Vous pouvez fournir le motif sous forme d'objet ou de chaîne de caractères ; si vous avez besoin de spécifier des options pour l'expression régulière, vous devez soit utiliser un objet motif comme premier paramètre, soit utiliser des modificateurs intégrés dans la chaîne de caractères, par exemple `sub("(?i)b+", "x", "bbbb BBBBB")` renvoie `'x x'`.

6 Problèmes classiques

Les expressions régulières constituent un outil puissant pour certaines applications mais, à certains égards, leur comportement n'est pas intuitif et, parfois, elles ne se comportent pas comme vous pouvez vous y attendre. Cette section met en évidence certains des pièges les plus courants.

6.1 Utilisez les méthodes du type *string*

Parfois, il ne faut pas utiliser le module `re`. Si vous faites correspondre une chaîne fixe, ou une classe à un seul caractère, et que vous n'utilisez pas les fonctionnalités de `re` telles que l'option `IGNORECASE`, alors la puissance des expressions régulières n'est pas nécessaire. Les chaînes de caractères ont plusieurs méthodes pour opérer sur des chaînes fixes et elles sont généralement beaucoup plus rapides, parce que l'implémentation est une seule petite boucle C qui a été optimisée, au lieu du gros moteur d'expressions régulières plus généraliste.

Nous pouvons prendre l'exemple du remplacement d'une seule chaîne fixe par une autre ; vous souhaitez remplacer `mot` par `acte`. `re.sub()` semble être faite pour cela, mais regardons la méthode `replace()`. Notons que `replace()` remplace aussi `mot` à l'intérieur des mots, transformant `moteur` en `acteeur`, mais la RE naïve `mot` aurait aussi fait cela (pour éviter d'effectuer la substitution sur des parties de mots, le motif doit être `\bmot\b`, qui exige que `mot` soit en limite de mot d'un côté et de l'autre ; c'est au-delà des capacités de la méthode `replace()`).

Une autre tâche classique est de supprimer chaque occurrence d'un seul caractère d'une chaîne de caractères ou de le remplacer par un autre caractère unique. Vous pouvez le faire avec quelque chose comme `re.sub('\n', ' ', S)`, mais `translate()` en est capable et est plus rapide que n'importe quelle opération d'expression régulière.

Bref, avant de passer au module `re`, évaluez d'abord si votre problème peut être résolu avec une méthode de chaîne plus rapide et plus simple.

6.2 *match()* contre *search()*

La fonction `match()` vérifie seulement si la RE correspond au début de la chaîne de caractères tandis que `search()` parcourt la chaîne de caractères pour trouver une correspondance. Il est important de garder cette distinction à l'esprit. Rappelez-vous, `match()` ne trouve qu'une correspondance qui commence à 0 ; si la correspondance commence plus loin, `match()` *ne la trouve pas*.

```
>>> print(re.match('super', 'superstition').span())
(0, 5)
>>> print(re.match('super', 'insuperable'))
None
```

D'un autre côté, `search()` balaie la chaîne de caractères, rapportant la première correspondance qu'elle trouve.

```
>>> print(re.search('super', 'superstition').span())
(0, 5)
>>> print(re.search('super', 'insuperable').span())
(2, 7)
```

Vous pouvez être tenté d'utiliser `re.match()` en ajoutant simplement `.*` au début de votre RE. Ce n'est pas une bonne idée, utilisez plutôt `re.search()`. Le compilateur d'expressions régulières analyse les RE pour optimiser

le processus de recherche d'une correspondance. Cette analyse permet de déterminer ce que doit être le premier caractère d'une correspondance ; par exemple, un motif commençant par `Corbeau` doit faire correspondre un `'C'` en tête. L'analyse permet au moteur de parcourir rapidement la chaîne de caractères à la recherche du caractère de départ, n'essayant la correspondance complète que si un `'C'` a déjà été trouvé.

Ajouter `.*` annihile cette optimisation, nécessitant un balayage jusqu'à la fin de la chaîne de caractères, puis un retour en arrière pour trouver une correspondance pour le reste de la RE. Préférez l'utilisation `re.search()`.

6.3 Glouton contre non-glouton

Si vous répétez un motif dans une expression régulière, comme `a*`, l'action résultante est de consommer autant de motifs que possible. C'est un problème lorsque vous essayez de faire correspondre une paire de délimiteurs, comme des chevrons encadrant une balise HTML. Le motif naïf pour faire correspondre une seule balise HTML ne fonctionne pas en raison de la nature gloutonne de `.*`.

```
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print(re.match('<.*>', s).span())
(0, 32)
>>> print(re.match('<.*>', s).group())
<html><head><title>Title</title>
```

La RE correspond au `'<'` de `'<html>'` et le `.*` consomme le reste de la chaîne. Mais, comme il reste des éléments du motif dans la RE et que le > ne peut pas correspondre à la fin de la chaîne de caractères, le moteur d'expression régulière doit faire marche arrière caractère par caractère jusqu'à ce qu'il trouve une correspondance pour le >. La correspondance finale s'étend du `'<'` de `'<html>'` au `'>'` de `'</title>'`, ce qui n'est pas ce que vous voulez.

Dans ce cas, la solution consiste à utiliser des quantificateurs non gloutons tels que `*?`, `+?`, `??` ou `{m,n}?`, qui effectuent une correspondance aussi *petite* que possible. Dans l'exemple ci-dessus, le `'>'` est essayé immédiatement après que le `'<'` corresponde et, s'il échoue, le moteur avance caractère par caractère, ré-essayant `'>'` à chaque pas. Nous obtenons alors le bon résultat :

```
>>> print(re.match('<.*?>', s).group())
<html>
```

Note : l'analyse du HTML ou du XML avec des expressions régulières est tout sauf une sinécure. Les motifs écrits à la va-vite traiteront les cas communs, mais HTML et XML ont des cas spéciaux qui font planter l'expression régulière évidente ; quand vous aurez écrit une expression régulière qui traite tous les cas possibles, les motifs seront *très* compliqués. Utilisez un module d'analyse HTML ou XML pour de telles tâches.

6.4 Utilisez *re.VERBOSE*

À présent, vous vous êtes rendu compte que les expressions régulières sont une notation très compacte, mais qu'elles ne sont pas très lisibles. Une RE modérément complexe peut rapidement devenir une longue collection de barres obliques inverses, de parenthèses et de métacaractères, ce qui la rend difficile à lire et à comprendre.

Pour de telles RE, activer l'option `re.VERBOSE` à la compilation de l'expression régulière peut être utile ; cela vous permet de formater l'expression régulière de manière plus claire.

L'option `re.VERBOSE` a plusieurs effets. Les espaces dans l'expression régulière qui *ne sont pas* à l'intérieur d'une classe de caractères sont ignorées. Cela signifie qu'une expression comme `chien | chat` est équivalente à `chien|chat` qui est moins lisible, mais `[a b]` correspond toujours aux caractères `'a'`, `'b'` ou à une espace. En outre, vous avez la possibilité de mettre des commentaires à l'intérieur d'une RE ; les commentaires s'étendent du caractère # à la nouvelle ligne suivante. Lorsque vous l'utilisez avec des chaînes à triple guillemets, cela permet aux RE d'être formatées plus proprement :

```
pat = re.compile(r"""
 \s*                 # Skip leading whitespace
 (?P<header>[^:]+)   # Header name
 \s* :               # Whitespace, and a colon
 (?P<value>.*?)      # The header's value -- *? used to
                     # lose the following trailing whitespace
 \s*$                # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

Ceci est beaucoup plus lisible que :

```
pat = re.compile(r"\s*(?P<header>[^:]+)\s*:(?P<value>.*?)\s*$")
```

7 Vos commentaires

Les expressions régulières sont un sujet compliqué. Est-ce que ce document vous a aidé à les comprendre ? Des parties ne sont pas claires, ou des problèmes que vous avez rencontrés ne sont pas traités ici ? Si tel est le cas, merci d'envoyer vos suggestions d'améliorations à l'auteur.

Le livre le plus complet sur les expressions régulières est certainement *Mastering Regular Expressions* de Jeffrey Friedl, publié par O'Reilly. Malheureusement, il se concentre sur les déclinaisons Perl et Java des expressions régulières et ne contient aucun contenu pour Python ; il n'est donc pas utile d'en faire référence pour la programmation Python. (La première édition traitait du module Python `regex`, maintenant supprimé, ce qui ne vous aidera pas beaucoup.) Pensez à le retirer de votre bibliothèque.