Guide des expressions régulières

Version 2.7.16

Guido van Rossum
and the Python development team

octobre 07, 2019

Python Software Foundation
Email : docs@python.org

Table des matières

Author 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

The `re` module was added in Python 1.5, and provides Perl-style regular expression patterns. Earlier versions of Python came with the `regex` module, which provided Emacs-style patterns. The `regex` module was removed completely in Python 2.5.

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 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.

You can match the characters not listed within the class by *complementing* the set. This is indicated by including a `'^'` as the first character of the class. For example, `[^5]` will match any character except `'5'`. If the caret appears elsewhere in a character class, it does not have special meaning. For example : `[5^]` will match either a `'5'` or a `'^'`.

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 `\\`.

Some of the special sequences beginning with `'\'` represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that isn't whitespace. The following predefined special sequences are a subset of those available. The equivalent classes are for byte string patterns. For a complete list of sequences and expanded class definitions for Unicode string patterns, see the last part of Regular Expression Syntax.

- **`\d`** Correspond à n'importe quel caractère numérique ; équivalent à la classe `[0-9]`.
- **`\D`** Correspond à n'importe 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 caractère autre que « blanc » ; équivalent à la classe `[^ \t\n\r\f\v]`.
- **`\w`** Correspond à n'importe caractère alphanumérique ; équivalent à la classe `[a-zA-Z0-9_]`.
- **`\W`** Correspond à n'importe 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 « blancs » ou `','` ou `'.'`.

The final metacharacter in this section is `.`. It matches anything except a newline character, and there's an alternate mode (`re.DOTALL`) where it will match even a newline. `'.'` is often used where you want to match « any character ».

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.

The first metacharacter for repeating things that we'll look at is `*`. `*` doesn't match the literal character `*` ; instead, it specifies that the previous character can be matched zero or more times, instead of exactly once.

For example, `ca*t` will match `ct` (0 `a` characters), `cat` (1 `a`), `caaat` (3 `a` characters), and so forth. The RE engine has various internal limitations stemming from the size of C's `int` type that will prevent it from matching over 2 billion `a` characters ; you probably don't have enough memory to construct a string that large, so you shouldn't run into that limit.

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.

A step-by-step example will make this more obvious. Let's consider the expression `a[bcd]*b`. This matches the letter `'a'`, zero or more letters from the class `[bcd]`, and finally ends with a `'b'`. Now imagine matching this RE against the string `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, ce qui n'est pas 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.

The end of the RE has now been reached, and it has matched `abcb`. This demonstrates how the matching engine goes as far as it can at first, and if no match is found it will then progressively back up and retry the rest of the RE again and again. It will back up until it has tried zero matches for `[bcd]*`, and if that subsequently fails, the engine will conclude that the string doesn't match the RE at all.

Another repeating metacharacter is `+`, which matches one or more times. Pay careful attention to the difference between `*` and `+`; `*` matches *zero* or more times, so whatever's being repeated may not be present at all, while `+` requires at least *one* occurrence. To use a similar example, `ca+t` will match `cat` (1 `a`), `caaat` (3 `a`'s), but won't match `ct`.

There are two more repeating qualifiers. The question mark character, `?`, matches either once or zero times ; you can think of it as marking something as being optional. For example, `home-?brew` matches either `homebrew` or `home-brew`.

The most complicated repeated qualifier is `{m,n}`, where *m* and *n* are decimal integers. This qualifier means there must be at least *m* repetitions, and at most *n*. For example, `a/{1,3}b` will match `a/b`, `a//b`, and `a///b`. It won't match `ab`, which has no slashes, or `a////b`, which has four.

You can omit either *m* or *n* ; in that case, a reasonable value is assumed for the missing value. Omitting *m* is interpreted as a lower limit of 0, while omitting *n* results in an upper bound of infinity — actually, the upper bound is the 2-billion limit mentioned earlier, but that might as well be infinity.

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
<_sre.SRE_Pattern object at 0x...>
```

`re.compile()` accepte aussi une option *flags*, utilisée pour activer des fonctionnalités particulières et des variations de syntaxe. Nous étudierons la définition des variables plus tard et, pour l'instant, un seul 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 ».

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()` and `search()` return `None` if no match can be found. If they're successful, a match object instance is returned, containing information about the match : where it starts and ends, the substring it matched, and more.

You can learn about this by interactively experimenting with the `re` module. If you have Tkinter available, you may also want to look at Tools/scripts/redemo.py, a demonstration program included with the Python distribution. It allows you to enter REs and strings, and displays whether the RE matches or fails. `redemo.py` can be quite useful when trying to debug a complicated RE. Phil Schwartz's Kodos is also an interactive tool for developing and testing RE patterns.

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 :

```
Python 2.2.2 (#1, Feb 10 2003, 12:57:01)
>>> import re
>>> p = re.compile('[a-z]+')
>>> p  #doctest: +ELLIPSIS
<_sre.SRE_Pattern object at 0x...>
```

Now, you can try matching various strings against the RE `[a-z]+`. An empty string shouldn't match at all, since + means "one or more repetitions". `match()` should return `None` in this case, which will cause the interpreter to print no output. You can explicitly print the result of `match()` to make this clear.

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

Now, let's try it on a string that it should match, such as `tempo`. In this case, `match()` will return a match object, so you should store the result in a variable for later use.

```
>>> m = p.match('tempo')
>>> m
<_sre.SRE_Match object at 0x...>
```

Now you can query the match object for information about the matching string. match object instances also have several methods and attributes ; the most important ones are :

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 *tuple* 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()` returns the substring that was matched by the RE. `start()` and `end()` return the starting and ending index of the match. `span()` returns both start and end indexes in a single tuple. Since the `match()` method only checks if the RE matches at the start of a string, `start()` will always be zero. However, the `search()` method of patterns scans through the string, so the match may not start at zero in that case.

```
>>> print p.match('::: message')
None
>>> m = p.search('::: message'); print m
<_sre.SRE_Match object at 0x...>
>>> 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'
```

Two pattern methods return all of the matches for a pattern. `findall()` returns a list of matching strings :

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

`findall()` has to create the entire list before it can be returned as the result. The `finditer()` method returns a sequence of match object instances as an iterator. [1]

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable-iterator object at 0x...>
>>> for match in iterator:
...     print match.span()
...
```

(suite sur la page suivante)

1. Introduced in Python 2.2.2.

(suite de la page précédente)

```
(0, 2)
(22, 24)
(29, 31)
```

3.4 Fonctions de niveau module

You don't have to create a pattern object and call its methods; the `re` module also provides top-level functions called `match()`, `search()`, `findall()`, `sub()`, and so forth. These functions take the same arguments as the corresponding pattern method, with the RE string added as the first argument, and still return either `None` or a match object instance.

```
>>> print re.match(r'From\s+', 'Fromage amk')
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<_sre.SRE_Match object at 0x...>
```

Under the hood, these functions simply create a pattern object for you and call the appropriate method on it. They also store the compiled object in a cache, so future calls using the same RE are faster.

Should you use these module-level functions, or should you get the pattern and call its methods yourself ? That choice depends on how frequently the RE will be used, and on your personal coding style. If the RE is being used at only one point in the code, then the module functions are probably more convenient. If a program contains a lot of regular expressions, or re-uses the same ones in several locations, then it might be worthwhile to collect all the definitions in one place, in a section of code that compiles all the REs ahead of time. To take an example from the standard library, here's an extract from the deprecated `xmllib` module :

```
ref = re.compile( ... )
entityref = re.compile( ... )
charref = re.compile( ... )
starttagopen = re.compile( ... )
```

I generally prefer to work with the compiled object, even for one-time uses, but few people will be as much of a purist about this as I am.

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
`DOTALL`, `S`	Make `.` match any character, including newlines
`IGNORECASE`, `I`	Do case-insensitive matches
`LOCALE`, `L`	Do a locale-aware match
`MULTILINE`, `M`	Multi-line matching, affecting `^` and `$`
`VERBOSE`, `X`	Active les RE verbeuses, qui peuvent être organisées de manière plus propre et compréhensible.
`UNICODE`, `U`	Makes several escapes like `\w`, `\b`, `\s` and `\d` dependent on the Unicode character database.

`I`

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, and `Spam` will match `Spam`, `spam`, or `spAM`. This lowercasing doesn't take the current locale into account ; it will if you also set the `LOCALE` flag.

L
LOCALE
Make `\w`, `\W`, `\b`, and `\B`, dependent on the current locale.
Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing French text, you'd want to be able to write `\w+` to match words, but `\w` only matches the character class `[A-Za-z]` ; it won't match `'é'` or `'ç'`. If your system is configured properly and a French locale is selected, certain C functions will tell the program that `'é'` should also be considered a letter. Setting the `LOCALE` flag when compiling a regular expression will cause the resulting compiled object to use these C functions for `\w` ; this is slower, but also enables `\w+` to match French words as you'd expect.

M
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).

S
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.

U
UNICODE
Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` dependent on the Unicode character properties database.

X
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 « blanc » 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.

| Alternation, or the « or » operator. If A and B are regular expressions, `A|B` will match any string that matches either `A` or `B`. `|` has very low precedence in order to make it work reasonably when you're alternating multi-character strings. `Crow|Servo` will match either `Crow` or `Servo`, not `Cro`, a `'w'` or an `'S'`, and `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')
<_sre.SRE_Match object at 0x...>
>>> print re.search('^From', 'Reciting From Memory')
None
```

$ 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('}$', '{block}')
<_sre.SRE_Match object at 0x...>
>>> print re.search('}$', '{block} ')
None
>>> print re.search('}$', '{block}\n')
<_sre.SRE_Match object at 0x...>
```

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 caractère, 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')
<_sre.SRE_Match object at 0x...>
>>> 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')
<_sre.SRE_Match object at 0x...>
```

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

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a `':'`, like this :

```
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)
```

Groups indicated with `'('`, `')'` also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`. Groups are numbered starting with

0. Group 0 is always present ; it's the whole RE, so match object methods all have group 0 as their default argument. Later we'll see how to express groups that don't capture the span of text that they match.

```
>>> 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'
```

`group()` can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

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

The `groups()` method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```
>>> 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.

Perl 5 added several additional features to standard regular expressions, and the Python `re` module supports most of them. It would have been difficult to choose new single-keystroke metacharacters or new special sequences beginning with `\` to represent the new features without making Perl's regular expressions confusingly different from standard REs. If you chose `&` as a new metacharacter, for example, old expressions would be assuming that `&` was a regular character and wouldn't have escaped it by writing `\&` or `[&]`.

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 adds an extension syntax to Perl's extension syntax. If the first character after the question mark is a `P`, you know that it's an extension that's specific to Python. Currently there are two such extensions : `(?P<name>...)` defines a named group, and `(?P=name)` is a backreference to a named group. If future versions of Perl 5 add similar features using a different syntax, the `re` module will be changed to support the new syntax, while preserving the Python-specific syntax for compatibility's sake.

Now that we've looked at the general extension syntax, we can return to the features that simplify working with groups in complex REs. Since groups are numbered from left to right and a complex expression may use many groups, it can become difficult to keep track of the correct numbering. Modifying such a complex RE is annoying, too : insert a new group near the beginning and you change the numbers of everything that follows it.

Sometimes you'll want to use a group to collect a part of a regular expression, but aren't interested in retrieving the group's contents. You can make this fact explicit by using a non-capturing group : `(?:...)`, where you can replace the `...` with any other regular expression.

```
>>> 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.

The syntax for a named group is one of the Python-specific extensions : `(?P<name>...)`. *name* is, obviously, the name of the group. Named groups also behave exactly like capturing groups, and additionally associate a name with a group. The match object methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group's name. Named groups are still given numbers, so you can retrieve information about a group in two ways :

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

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.

The syntax for backreferences in an expression such as `(...)\1` refers to the number of the group. There's naturally a variant that uses the group name instead of the number. This is another Python extension : `(?P=name)` indicates that the contents of the group called *name* should again be matched at the current point. The regular expression for finding doubled words, `\b(\w+)\s+\1\b` can also be written as `\b(?P<word>\w+)\s+(?P=word)\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 chaine.

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 :

```
.*[.].*$
```

Notice that the `.` needs to be treated specially because it's a metacharacter ; I've put it inside a character class. Also notice the trailing `$` ; this is added to ensure that all the rest of the string must be included in the extension. This regular expression matches `foo.bar` and `autoexec.bat` and `sendmail.cf` and `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()`	Does the same thing as `sub()`, but returns the new string and the number of replacements

5.1 Découpage de chaînes

The `split()` method of a pattern splits a string apart wherever the RE matches, returning a list of the pieces. It's similar to the `split()` method of strings but provides much more generality in the delimiters that you can split by; `split()` only supports splitting by whitespace or by a fixed string. As you'd expect, there's a module-level `re.split()` function, too.

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

5.2 Recherche et substitution

Another common task is to find all the matches for a pattern, and replace them with a different string. The `sub()` method takes a replacement value, which can be either a string or a function, and the string to be processed.

.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.

Here's a simple example of using the `sub()` method. It replaces colour names with the word `colour` :

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

The `subn()` method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed :

```
>>> 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)
```

Empty matches are replaced only when they're not adjacent to a previous match.

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

If *replacement* is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes such as `\j` are left alone. Backreferences, such as

\6, are replaced with the substring matched by the corresponding group in the RE. This lets you incorporate portions of the original text in the resulting replacement string.

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+')
>>> 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*

Sometimes using the re module is a mistake. If you're matching a fixed string, or a single character class, and you're not using any re features such as the IGNORECASE flag, then the full power of regular expressions may not be required.

Strings have several methods for performing operations with fixed strings and they're usually much faster, because the implementation is a single small C loop that's been optimized for the purpose, instead of the large, more generalized regular expression engine.

One example might be replacing a single fixed string with another one ; for example, you might replace `word` with `deed`. `re.sub()` seems like the function to use for this, but consider the `replace()` method. Note that `replace()` will also replace `word` inside words, turning `swordfish` into `sdeedfish`, but the naive RE `word` would have done that, too. (To avoid performing the substitution on parts of words, the pattern would have to be `\bword\b`, in order to require that `word` have a word boundary on either side. This takes the job beyond `replace()`"s abilities.)

Another common task is deleting every occurrence of a single character from a string or replacing it with another single character. You might do this with something like `re.sub('\n', ' ', S)`, but `translate()` is capable of doing both tasks and will be faster than any regular expression operation can be.

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()*

The `match()` function only checks if the RE matches at the beginning of the string while `search()` will scan forward through the string for a match. It's important to keep this distinction in mind. Remember, `match()` will only report a successful match which will start at 0 ; if the match wouldn't start at zero, `match()` will *not* report it.

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

On the other hand, `search()` will scan forward through the string, reporting the first match it finds.

```
>>> 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 REs 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
```

(suite sur la page suivante)

(suite de la page précédente)

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

The RE matches the `'<'` in `<html>`, and the `.*` consumes the rest of the string. There's still more left in the RE, though, and the > can't match at the end of the string, so the regular expression engine has to backtrack character by character until it finds a match for the >. The final match extends from the `'<'` in `<html>` to the `'>'` in `</title>`, which isn't what you want.

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 vavite 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.

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

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.

The most complete book on regular expressions is almost certainly Jeffrey Friedl's Mastering Regular Expressions, published by O'Reilly. Unfortunately, it exclusively concentrates on Perl and Java's flavours of regular expressions, and doesn't contain any Python material at all, so it won't be useful as a reference for programming in Python. (The first edition covered Python's now-removed `regex` module, which won't help you much.) Consider checking it out from your library.

Notes