Expressions régulières Python (re)

re --- Regular expression operations

Code source : Lib/re/

Ce module fournit des opérations sur les expressions rationnelles similaires à celles que l'on trouve dans Perl.

Both patterns and strings to be searched can be Unicode strings (str) as well as 8-bit strings (bytes). However, Unicode strings and 8-bit strings cannot be mixed: that is, you cannot match a Unicode string with a bytes pattern or vice-versa; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

Regular expressions use the backslash character ('\') to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write '\\\\' as the pattern string, because the regular expression must be \\, and each backslash must be expressed as \\ inside a regular Python string literal. Also, please note that any invalid escape sequences in Python's usage of the backslash in string literals now generate a SyntaxWarning and in the future this will become a SyntaxError. This behaviour will happen even if it is a valid escape sequence for a regular expression.

La solution est d'utiliser la notation des chaînes brutes en Python pour les expressions rationnelles ; Les *backslashs* ne provoquent aucun traitement spécifique dans les chaînes littérales préfixées par 'r'. Ainsi, r"\n" est une chaîne de deux caractères contenant '\' et 'n', tandis que "\n" est une chaîne contenant un unique caractère : un saut de ligne. Généralement, les motifs seront exprimés en Python à l'aide de chaînes brutes.

Il est important de noter que la plupart des opérations sur les expressions rationnelles sont disponibles comme fonctions au niveau du module et comme méthodes des expressions rationnelles compilées. Les fonctions sont des raccourcis qui ne vous obligent pas à d'abord compiler un objet *regex*, mais auxquelles manquent certains paramètres de configuration fine.

Voir aussi

The third-party regex module, which has an API compatible with the standard library re module, but offers additional functionality and a more thorough Unicode support.

Syntaxe des expressions rationnelles

Une expression rationnelle (*regular expression* ou *RE*) spécifie un ensemble de chaînes de caractères qui lui correspondent ; les fonctions de ce module vous permettent de vérifier si une chaîne particulière correspond à une expression rationnelle donnée (ou si un expression rationnelle donnée correspond à une chaîne particulière, ce qui revient à la même chose).

Les expressions rationnelles peuvent être concaténées pour former de nouvelles expressions : si *A* et *B* sont deux expressions rationnelles, alors *AB* est aussi une expression rationnelle. En général, si une chaîne *p* valide *A* et qu'une autre chaîne *q* valide *B*, la chaîne *pq* validera AB. Cela

est vrai tant que *A* et *B* ne contiennent pas d'opérations de priorité ; de conditions de frontière entre *A* et *B* ; ou de références vers des groupes numérotés. Ainsi, des expressions complexes peuvent facilement être construites depuis de plus simples expressions primitives comme celles décrites ici. Pour plus de détails sur la théorie et l'implémentation des expressions rationnelles, consultez le livre de Friedl [Frie09], ou à peu près n'importe quel livre dédié à la construction de compilateurs.

Une brève explication sur le format des expressions rationnelles suit. Pour de plus amples informations et une présentation plus simple, référez-vous au Guide des expressions régulières.

Les expressions rationnelles peuvent contenir à la fois des caractères spéciaux et ordinaires. Les plus ordinaires, comme 'A', 'a' ou '0' sont les expressions rationnelles les plus simples : elles correspondent simplement à elles-mêmes. Vous pouvez concaténer des caractères ordinaires, ainsi last correspond à la chaîne 'last'. (Dans la suite de cette section, nous écrirons les expressions rationnelles dans ce style spécifique, généralement sans guillemets, et les chaînes à tester 'entourées de simples guillemets'.)

Certains caractères, comme '|' ou '(', sont spéciaux. Des caractères spéciaux peuvent aussi exister pour les classes de caractères ordinaires, ou affecter comment les expressions rationnelles autour d'eux seront interprétées.

Les caractères de répétition ou quantificateurs (*, +, ?, {m,n}, etc.) ne peuvent être directement imbriqués. Cela empêche l'ambiguïté avec le suffixe modificateur non gourmand ? et avec les autres modificateurs dans d'autres implémentations. Pour appliquer une seconde répétition à une première, des parenthèses peuvent être utilisées. Par exemple, l'expression (?:a{6})* valide toutes les chaînes composées d'un nombre de caractères 'a' multiple de six.

Les caractères spéciaux sont :

.

(Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline. (?s:.) matches any character regardless of flags.

^

(Accent circonflexe.) Valide le début d'une chaîne de caractères, ainsi que ce qui suit chaque saut de ligne en mode MULTILINE.

$

Valide la fin d'une chaîne de caractères, ou juste avant le saut de ligne à la fin de la chaîne, ainsi qu'avant chaque saut de ligne en mode MULTILINE. foo valide à la fois *foo* et *foobar*, tandis que l'expression rationnelle foo$ ne correspond qu'à 'foo'. Plus intéressant,
chercher foo.$ dans 'foo1\nfoo2\n' trouve normalement 'foo2', mais 'foo1' en mode MULTILINE ;
chercher un simple $ dans 'foo\n' trouvera deux correspondances (vides) : une juste avant le saut de ligne, et une à la fin de la chaîne.

*

Fait valider par l'expression rationnelle résultante 0 répétition ou plus de l'expression qui précède, avec autant de répétitions que possible. ab* validera 'a', 'ab' ou 'a' suivi de n'importe quel nombre de 'b'.

+

Fait valider par l'expression rationnelle résultante 1 répétition ou plus de l'expression qui précède. ab+ validera 'a' suivi de n'importe quel nombre non nul de 'b' ; cela ne validera pas la chaîne 'a'.

?

Fait valider par l'expression rationnelle résultante 0 ou 1 répétition de l'expression qui précède. ab? correspondra à 'a' ou 'ab'.

*?, +?, ??

Les quantificateurs '*', '+' et '?' sont tous *greedy* (gourmands) ; ils valident autant de texte que possible. Parfois ce comportement n'est pas désiré ; si l'expression rationnelle <.*> est testée avec la chaîne '<a> b <c>', cela correspondra à la chaîne entière, et non juste à '<a>'. Ajouter ? derrière le quantificateur lui fait réaliser l'opération de façon *non-greedy* (ou *minimal*) ; le *moins* de caractères possibles seront validés. Utiliser l'expression rationnelle <.*?> validera uniquement '<a>'.

*+, ++, ?+

Like the '*', '+', and '?' quantifiers, those where '+' is appended also match as many times as possible. However, unlike the true greedy quantifiers, these do not allow back-tracking when the expression following it fails to match. These are known as *possessive* quantifiers. For example, a*a will match 'aaaa' because the a* will match all 4 'a's, but, when the final 'a' is encountered, the expression is backtracked so that in the end the a* ends up matching 3 'a's total, and the fourth 'a' is matched by the final 'a'. However, when a*+a is used to match 'aaaa', the a*+ will match all 4 'a', but when the final 'a' fails to find any more characters to match, the expression cannot be backtracked and will thus fail to match. x*+, x++ and x?+ are equivalent to (?>x*), (?>x+) and (?>x?) correspondingly.

Ajouté dans la version 3.11.

{m}

Spécifie qu'exactement *m* copies de l'expression rationnelle qui précède devront être validées ; un nombre plus faible de correspondances empêche l'expression entière de correspondre. Par exemple, a{6} correspondra exactement à six caractères 'a', mais pas à cinq.

{m,n}

Fait valider par l'expression rationnelle résultante entre *m* et *n* répétitions de l'expression qui précède, cherchant à en valider le plus possible. Par exemple, a{3,5} validera entre 3 et 5 caractères 'a'. Omettre *m* revient à spécifier 0 comme borne inférieure, et omettre *n* à avoir une borne supérieure infinie. Par exemple, a{4,}b correspondra à 'aaaab' ou à un millier de caractères 'a' suivis d'un 'b', mais pas à 'aaab'. La virgule ne doit pas être omise, auquel cas le modificateur serait confondu avec la forme décrite précédemment.

{m,n}?

Fait valider l'expression rationnelle résultante entre *m* et *n* répétitions de l'expression qui précède, cherchant à en valider le moins possible. Il s'agit de la version non gourmande du

précédent quantificateur. Par exemple, dans la chaîne de 6 caractères 'aaaaaa', a{3,5} trouvera 5 caractères 'a', alors que a{3,5}? n'en trouvera que 3.

{m,n}+

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible *without* establishing any backtracking points. This is the possessive version of the quantifier above. For example, on the 6-character string 'aaaaaa', a{3,5}+aa attempt to match 5 'a' characters, then, requiring 2 more 'a's, will need more characters than available and thus fail, while a{3,5}aa will match with a{3,5} capturing 5, then 4 'a's by backtracking and then the final 2 'a's are matched by the final aa in the pattern. x{m,n}+ is equivalent to (?>x{m,n}).

Ajouté dans la version 3.11.

\

Échappe les caractères spéciaux (permettant d'identifier des caractères comme '*', '?' et autres) ou signale une séquence spéciale ; les séquences spéciales sont décrites ci-dessous.

Si vous n'utilisez pas de chaînes brutes pour exprimer le motif, souvenez-vous que Python utilise aussi le *backslash* comme une séquence d'échappement dans les chaînes littérales ; si la séquence d'échappement n'est pas reconnue par l'interpréteur Python, le *backslash* et les caractères qui le suivent sont inclus dans la chaîne renvoyée. Cependant, si Python reconnait la séquence, le *backslash* doit être doublé (pour ne plus être reconnu). C'est assez compliqué et difficile à comprendre, c'est pourquoi il est hautement recommandé d'utiliser des chaînes brutes pour tout sauf les expressions les plus simples.

[]

Utilisé pour indiquer un ensemble de caractères. Dans un ensemble :

- Les caractères peuvent être listés individuellement, e.g. [amk] correspondra à 'a', 'm' ou 'k'.
- Des intervalles de caractères peuvent être indiqués en donnant deux caractères et les séparant par un '-', par exemple [a-z] correspondra à toute lettre minuscule *ASCII*, [0-5][0-9] à tous nombres de deux chiffres entre 00 et 59, et [0-9A-Fa-f] correspondra à n'importe quel chiffre hexadécimal. Si '-' est échappé ([a\-z]) ou s'il est placé comme premier ou dernier caractère (e.g. [-a] ou [a-]), il correspondra à un '-' littéral.
- Special characters except backslash lose their special meaning inside sets. For example, [(+*)] will match any of the literal characters '(', '+', '*', or ')'.
- Backslash either escapes characters which have special meaning in a set such as '-', ']', '^' and '\\' itself or signals a special sequence which represents a single character such as \xa0 or \n or a character class such as \w or \S (defined below). Note that \b represents a single "backspace" character, not a word boundary as outside a set, and numeric escapes such as \1 are always octal escapes, not group references. Special sequences which do not match a single character such as \A and \Z are not allowed.
- Les caractères qui ne sont pas dans un intervalle peuvent être trouvés avec l'ensemble complémentaire (*complementing*). Si le premier caractère de l'ensemble est '^', tous les caractères qui *ne sont pas* dans l'ensemble seront validés. Par

exemple, [^5] correspondra à tout caractère autre que '5' et [^^] validera n'importe quel caractère excepté '^'. ^ n'a pas de sens particulier s'il n'est pas le premier caractère de l'ensemble.

- To match a literal ']' inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both [()[\]{}] and []()[{}] will match a right bracket, as well as left bracket, braces, and parentheses.
- La gestion des ensembles inclus l'un dans l'autre et les opérations d'ensemble comme dans Unicode Technical Standard #18 pourrait être ajoutée par la suite. Ceci changerait la syntaxe, donc pour faciliter ce changement, une exception FutureWarning sera levée dans les cas ambigus pour le moment. Ceci inclut les ensembles commençant avec le caractère '[' ou contenant les séquences de caractères '--', '&&', '~~' et '||'. Pour éviter un message d'avertissement, échapper les séquences avec le caractère antislash ("\").

Modifié dans la version 3.7: L'exception FutureWarning est levée si un ensemble de caractères contient une construction dont la sémantique changera dans le futur.

|

A|B, où *A* et *B* peuvent être deux expressions rationnelles arbitraires, crée une expression rationnelle qui validera soit *A* soit *B*. Un nombre arbitraire d'expressions peuvent être séparées de cette façon par des '|'. Cela peut aussi être utilisé au sein de groupes (voir ci-dessous). Quand une chaîne cible est analysée, les expressions séparées par '|' sont essayées de la gauche vers la droite. Quand un motif correspond complètement, cette branche est acceptée. Cela signifie qu'une fois que *A* correspond, *B* ne sera pas testée plus loin, même si elle pourrait provoquer une plus ample correspondance. En d'autres termes, l'opérateur '|' n'est jamais gourmand. Pour valider un '|' littéral, utilisez \|, ou enveloppez-le dans une classe de caractères, comme [|].

(...)

Valide n'importe quelle expression rationnelle comprise entre les parenthèses, et indique le début et la fin d'un groupe ; le contenu d'un groupe peut être récupéré après qu'une analyse a été effectuée et peut être réutilisé plus loin dans la chaîne avec une séquence spéciale \number, décrite ci-dessous. Pour écrire des '(' ou ')' littéraux, utilisez \(ou \), ou enveloppez-les dans une classe de caractères : [(], [)].

(?...)

Il s'agit d'une notation pour les extensions (un '?' suivant une '(' n'a pas de sens autrement). Le premier caractère après le '?' détermine quel sens donner à l'expression. Les extensions ne créent généralement pas de nouveaux groupes ; (?P<name>...) est la seule exception à la règle. Retrouvez ci-dessous la liste des extensions actuellement supportées.

(?aiLmsux)

(One or more letters from the set 'a', 'i', 'L', 'm', 's', 'u', 'x'.) The group matches the empty string; the letters set the corresponding flags for the entire regular expression:

- re.A (ASCII-only matching)
- re.I (ignore case)
- re.L (locale dependent)

- re.M (multi-line)
- re.S (dot matches all)
- re.U (Unicode matching)
- re.X (verbose)

(The flags are described in Contenu du module.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a *flag* argument to the re.compile() function. Flags should be used first in the expression string.

Modifié dans la version 3.11: Cette construction ne peut être utilisée qu'au début d'une chaîne de caractères.

(?:...)

Une version sans capture des parenthèses habituelles. Valide n'importe quelle expression rationnelle à l'intérieur des parenthèses, mais la sous-chaîne correspondant au groupe *ne peut pas* être récupérée après l'analyse ou être référencée plus loin dans le motif.

(?aiLmsux-imsx:...)

(Zero or more letters from the set 'a', 'i', 'L', 'm', 's', 'u', 'x', optionally followed by '-' followed by one or more letters from the 'i', 'm', 's', 'x'.) The letters set or remove the corresponding flags for the part of the expression:

- re.A (ASCII-only matching)
- re.I (ignore case)
- re.L (locale dependent)
- re.M (multi-line)
- re.S (dot matches all)
- re.U (Unicode matching)
- re.X (verbose)

(The flags are described in Contenu du module.)

The letters 'a', 'L' and 'u' are mutually exclusive when used as inline flags, so they can't be combined or follow '-'. Instead, when one of them appears in an inline group, it overrides the matching mode in the enclosing group. In Unicode patterns (?a:...) switches to ASCII-only matching, and (?u:...) switches to Unicode matching (default). In bytes patterns (?L:...) switches to locale dependent matching, and (?a:...) switches to ASCII-only matching (default). This override is only in effect for the narrow inline group, and the original matching mode is restored outside of the group.

Ajouté dans la version 3.6.

Modifié dans la version 3.7: Les lettres 'a', 'L' et 'u' peuvent aussi être utilisées dans un groupe.

(?>...)

Attempts to match ... as if it was a separate regular expression, and if successful, continues to match the rest of the pattern following it. If the subsequent pattern fails to match, the stack can only be unwound to a point *before* the (?>...) because once exited, the expression, known as an *atomic group*, has thrown away all stack points within itself. Thus, (?>.*). would never match anything because first the .* would match all characters possible, then, having nothing left to match, the final . would fail to match. Since there are no stack points saved in the Atomic Group, and there is no stack point before it, the entire expression would thus fail to match.

Ajouté dans la version 3.11.

(?P<name>...)

Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name *name*. Group names must be valid Python identifiers, and in bytes patterns they can only contain bytes in the ASCII range. Each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.

Les groupes nommés peuvent être référencés dans trois contextes. Si le motif est (?P<quote>['"]).*?(?P=quote) (c.-à-d. correspondant à une chaîne entourée de guillemets simples ou doubles) :

Contexte de référence au groupe *quote*	**Manières de le référencer**
lui-même dans le même motif	• (?P=quote) (comme vu) • \1
en analysant l'objet résultat *m*	• m.group('quote') • m.end('quote') (etc.)
dans une chaîne passée à l'argument *repl* de re.sub()	• \g<quote> • \g<1> • \1

Modifié dans la version 3.12: In bytes patterns, group *name* can only contain bytes in the ASCII range (b'\x00'-b'\x7f').

(?P=name)

Une référence arrière à un groupe nommé ; elle correspond à n'importe quel texte validé plus tôt par le groupe nommé *name*.

(?#...)

Un commentaire ; le contenu des parenthèses est simplement ignoré.

(?=...)

Valide si ... valide la suite, mais ne consomme rien de la chaîne. On appelle cela une assertion *lookahead*. Par exemple, Isaac (?=Asimov) correspondra à la chaîne 'Isaac ' seulement si elle est suivie par 'Asimov'.

(?!...)

Valide si ... ne valide pas la suite. C'est une assertion *negative lookahead*. Par exemple, Isaac (?!Asimov) correspondra à la chaîne 'Isaac ' seulement si elle *n'est pas* suivie par 'Asimov'.

(?<=...)

Valide si la position courante dans la chaîne est précédée par une correspondance sur ... qui se termine à la position courante. On appelle cela une *positive lookbehind assertion*. (?<=abc)def cherchera une correspondance dans 'abcdef', puisque le *lookbehind** mettra de côté 3 caractères et vérifiera que le motif contenu correspond. Le motif ne devra correspondre qu'à des chaînes de taille fixe, cela veut dire que abc ou a|b sont autorisées, mais pas a* ou a{3,4}. Notez que les motifs qui commencent par des assertions *lookbehind* positives ne peuvent pas correspondre au début de la chaîne analysée ; vous préférerez sûrement utiliser la fonction search() plutôt que la fonction match() :

import re

m = re.search('(?<=abc)def', 'abcdef')

m.group(0)

'def'

Cet exemple recherche un mot suivi d'un trait d'union :

m = re.search(r'(?<=-)\w+', 'spam-egg')

m.group(0)

'egg'

Modifié dans la version 3.5: Ajout du support des références aux groupes de taille fixe.

(?<!...)

Valide si la position courante dans la chaîne n'est pas précédée par une correspondance sur .... On appelle cela une *negative lookbehind assertion*. À la manière des assertions *lookbehind* positives, le motif contenu ne peut que correspondre à des chaînes de taille fixe. Les motifs débutant par une assertion *lookbehind* négative peuvent correspondre au début de la chaîne analysée.

(?(id/name)yes-pattern|no-pattern)

Essaiera de faire la correspondance avec yes-pattern si le groupe indiqué par *id* ou *name* existe, et avec no-pattern s'il n'existe pas. no-pattern est optionnel et peut être omis. Par exemple, (<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$) est un motif simpliste pour identifier une adresse courriel, qui validera '<user@host.com>' ainsi que 'user@host.com' mais pas '<user@host.com' ni 'user@host.com>'.

Modifié dans la version 3.12: Group *id* can only contain ASCII digits. In bytes patterns, group *name* can only contain bytes in the ASCII range (b'\x00'-b'\x7f').

Les séquences spéciales sont composées de '\' et d'un caractère de la liste qui suit. Si le caractère ordinaire n'est pas un chiffre *ASCII* ou une lettre *ASCII*, alors l'expression rationnelle résultante validera le second caractère de la séquence. Par exemple, \$ correspond au caractère '$'.

\number

Correspond au contenu du groupe du même nombre. Les groupes sont numérotés à partir de 1. Par exemple, (.+) \1 correspond à 'the the' ou '55 55', mais pas à 'thethe' (notez l'espace après le groupe). Cette séquence spéciale ne peut être utilisée que pour faire référence aux 99 premiers groupes. Si le premier chiffre de *number* est 0, ou si *number* est un nombre octal de 3 chiffres, il ne sera pas interprété comme une référence à un groupe, mais comme le caractère à la valeur octale *number*. À l'intérieur des '[' et ']' d'une classe de caractères, tous les échappements numériques sont traités comme des caractères.

\A

Correspond uniquement au début d'une chaîne de caractères.

\b

Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of word characters. Note that formally, \b is defined as the boundary between a \w and a \W character (or vice versa), or between \w and the beginning or end of the string. This means that r'\bat\b' matches 'at', 'at.', '(at)', and 'as at ay' but not 'attempt' or 'atlas'.

The default word characters in Unicode (str) patterns are Unicode alphanumerics and the underscore, but this can be changed by using the ASCII flag. Word boundaries are determined by the current locale if the LOCALE flag is used.

Note

Inside a character range, \b represents the backspace character, for compatibility with Python's string literals.

\B

Matches the empty string, but only when it is *not* at the beginning or end of a word. This means that r'at\B' matches 'athens', 'atom', 'attorney', but not 'at', 'at.', or 'at!'. \B is the opposite of \b, so word characters in Unicode (str) patterns are Unicode alphanumerics or the underscore, although this can be changed by using the ASCII flag. Word boundaries are determined by the current locale if the LOCALE flag is used.

Note

Note that \B does not match an empty string, which differs from RE implementations in other programming languages such as Perl. This behavior is kept for compatibility reasons.

\d

Pour les motifs Unicode (*str*) :

Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes [0-9], and also many other digit characters.

Matches [0-9] if the ASCII flag is used.

Pour les motifs 8-bits (*bytes*) :

Matches any decimal digit in the ASCII character set; this is equivalent to [0-9].

\D

Matches any character which is not a decimal digit. This is the opposite of \d.

Matches [^0-9] if the ASCII flag is used.

\s

Pour les motifs Unicode (*str*) :

Matches Unicode whitespace characters (as defined by str.isspace()). This includes [\t\n\r\f\v], and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages.

Matches [\t\n\r\f\v] if the ASCII flag is used.

Pour les motifs 8-bits (*bytes*) :

Valide les caractères considérés comme des espacements dans la table ASCII ; équivalent à [\t\n\r\f\v].

\S

Matches any character which is not a whitespace character. This is the opposite of \s.

Matches [^ \t\n\r\f\v] if the ASCII flag is used.

\w

Pour les motifs Unicode (*str*) :

Matches Unicode word characters; this includes all Unicode alphanumeric characters (as defined by str.isalnum()), as well as the underscore (_).

Matches [a-zA-Z0-9_] if the ASCII flag is used.

Pour les motifs 8-bits (*bytes*) :

Matches characters considered alphanumeric in the ASCII character set; this is equivalent to [a-zA-Z0-9_]. If the LOCALE flag is used, matches characters considered alphanumeric in the current locale and the underscore.

\W

Matches any character which is not a word character. This is the opposite of \w. By default, matches non-underscore (_) characters for which str.isalnum() returns False.

Matches [^a-zA-Z0-9_] if the ASCII flag is used.

If the LOCALE flag is used, matches characters which are neither alphanumeric in the current locale nor the underscore.

\Z

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

Most of the escape sequences supported by Python string literals are also accepted by the regular expression parser:

\a \b \f \n

\N \r \t \u

\U \v \x \\

(Notez que \b est utilisé pour représenter les bornes d'un mot, et signifie « *retour arrière* » uniquement à l'intérieur d'une classe de caractères)

'\u', '\U', and '\N' escape sequences are only recognized in Unicode (str) patterns. In bytes patterns they are errors. Unknown escapes of ASCII letters are reserved for future use and treated as errors.

Les séquences octales d'échappement sont incluses dans une forme limitée. Si le premier chiffre est un 0, ou s'il y a trois chiffres octaux, la séquence est considérée comme octale. Autrement, il s'agit d'une référence vers un groupe. Comme pour les chaînes littérales, les séquences octales ne font jamais plus de 3 caractères de long.

Modifié dans la version 3.3: Les séquences d'échappement '\u' et '\U' ont été ajoutées.

Modifié dans la version 3.6: Les séquences inconnues composées de '\' et d'une lettre ASCII sont maintenant des erreurs.

Modifié dans la version 3.8: The '\N{*name*}' escape sequence has been added. As in string literals, it expands to the named Unicode character (e.g. '\N{EM DASH}').

Contenu du module

Le module définit plusieurs fonctions, constantes, et une exception. Certaines fonctions sont des versions simplifiées des méthodes plus complètes des expressions rationnelles compilées. La plupart des applications non triviales utilisent toujours la version compilée.

Flags

Modifié dans la version 3.6: Les constantes d'options sont maintenant des instances de RegexFlag, sous-classe de enum.IntFlag.

class re.**RegexFlag**

An enum.IntFlag class containing the regex options listed below.

Ajouté dans la version 3.11: - added to __all__

re.**A**

re.**ASCII**

Make \w, \W, \b, \B, \d, \D, \s and \S perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode (str) patterns, and is ignored for bytes patterns.

Corresponds to the inline flag (?a).

Note

The U flag still exists for backward compatibility, but is redundant in Python 3 since matches are Unicode by default for str patterns, and Unicode matching isn't allowed for bytes patterns. UNICODE and the inline flag (?u) are similarly redundant.

re.**DEBUG**

Display debug information about compiled expression.

No corresponding inline flag.

re.**I**

re.**IGNORECASE**

Perform case-insensitive matching; expressions like [A-Z] will also match lowercase letters. Full Unicode matching (such as Ü matching ü) also works unless the ASCII flag is used to disable non-ASCII matches. The current locale does not change the effect of this flag unless the LOCALE flag is also used.

Corresponds to the inline flag (?i).

Note that 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). If the ASCII flag is used, only letters 'a' to 'z' and 'A' to 'Z' are matched.

re.**L**

re.**LOCALE**

Make \w, \W, \b, \B and case-insensitive matching dependent on the current locale. This flag can be used only with bytes patterns.

Corresponds to the inline flag (?L).

Avertissement

This flag is discouraged; consider Unicode matching instead. The locale mechanism is very unreliable as it only handles one "culture" at a time and only works with 8-bit locales. Unicode matching is enabled by default for Unicode (str) patterns and it is able to handle different locales and languages.

Modifié dans la version 3.6: LOCALE can be used only with bytes patterns and is not compatible with ASCII.

Modifié dans la version 3.7: Compiled regular expression objects with the LOCALE flag no longer depend on the locale at compile time. Only the locale at matching time affects the result of matching.

re.**M**

re.**MULTILINE**

When specified, the pattern character '^' matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character '$' matches at the end of the string and at the end of each line (immediately preceding each newline). By default, '^' matches only at the beginning of the string, and '$' only at the end of the string and immediately before the newline (if any) at the end of the string.

Corresponds to the inline flag (?m).

re.**NOFLAG**

Indicates no flag being applied, the value is 0. This flag may be used as a default value for a function keyword argument or as a base value that will be conditionally ORed with other flags. Example of use as a default value:

```
def myfunc(text, flag=re.NOFLAG):
   return re.match(text, flag)
```

Ajouté dans la version 3.11.

re.**S**

re.**DOTALL**

Make the '.' special character match any character at all, including a newline; without this flag, '.' will match anything *except* a newline.

Corresponds to the inline flag (?s).

re.**U**

re.**UNICODE**

In Python 3, Unicode characters are matched by default for str patterns. This flag is therefore redundant with **no effect** and is only kept for backward compatibility.

See ASCII to restrict matching to ASCII characters instead.

re.**X**

re.**VERBOSE**

Cette option vous autorise à écrire des expressions rationnelles qui présentent mieux et sont plus lisibles en vous permettant de séparer visuellement les sections logiques du motif et d'ajouter des commentaires. Les caractères d'espacement à l'intérieur du motif sont ignorés, sauf à l'intérieur des classes de caractères ou quand ils sont précédés d'un *backslash* non échappé, ou dans des séquences comme *?, (?: ou (?P<...>.Par exemple, (? : et * ? ne sont pas autorisés. Quand une ligne contient un # qui n'est ni dans une classe de caractères, ni précédé

d'un *backslash* non échappé, tous les caractères depuis le # le plus à gauche jusqu'à la fin de la ligne sont ignorés.

Cela signifie que les deux expressions rationnelles suivantes qui valident un nombre décimal sont fonctionnellement égales :

a = re.compile(r"""\d + # the integral part

\. # the decimal point

\d * # some fractional digits""", re.X)

b = re.compile(r"\d+\.\d*")

Correspond à l'option de groupe (?x).

Fonctions

re.**compile**(*pattern*, *flags=0*)

Compile un motif vers une expression rationnelle compilée, dont les méthodes match() et search(), décrites ci-dessous, peuvent être utilisées pour analyser des textes.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the | operator).

La séquence

prog = re.compile(pattern)

result = prog.match(string)

est équivalente à

result = re.match(pattern, string)

mais utiliser re.compile() et sauvegarder l'expression rationnelle renvoyée pour la réutiliser est plus efficace quand l'expression est amenée à être utilisée plusieurs fois dans un même programme.

Note

Les versions compilées des motifs les plus récents passés à re.compile() et autres fonctions d'analyse du module sont mises en cache, ainsi les programmes qui n'utilisent que quelques expressions rationnelles en même temps n'ont pas à s'inquiéter de la compilation de ces expressions.

re.**search**(*pattern*, *string*, *flags=0*)

Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding Match. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the | operator).

re.**match**(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding Match. Return None if the string does not match the pattern; note that this is different from a zero-length match.

Notez que même en mode MULTILINE, re.match() ne validera qu'au début de la chaîne et non au début de chaque ligne.

Si vous voulez trouver une correspondance n'importe où dans *string*, utilisez plutôt search() (voir aussi Comparaison de search() et match()).

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the | operator).

re.**fullmatch**(*pattern*, *string*, *flags=0*)

If the whole *string* matches the regular expression *pattern*, return a corresponding Match. Return None if the string does not match the pattern; note that this is different from a zero-length match.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the | operator).

Ajouté dans la version 3.4.

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)

Sépare *string* selon les occurrences de *pattern*. Si des parenthèses de capture sont utilisées dans *pattern*, alors les textes des groupes du motif sont aussi renvoyés comme éléments de la liste résultante. Si *maxsplit* est différent de zéro, il ne pourra y avoir plus de *maxsplit* séparations, et le reste de la chaîne sera renvoyé comme le dernier élément de la liste.

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.', maxsplit=1)

['Words', 'words, words.']

re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)

['0', '3', '9']

S'il y a des groupes de capture dans le séparateur et qu'ils trouvent une correspondance au début de la chaîne, le résultat commencera par une chaîne vide. La même chose se produit pour la fin de la chaîne :

re.split(r'(\W+)', '...words, words...')

['', '...', 'words', ', ', 'words', '...', '']

De cette manière, les séparateurs sont toujours trouvés aux mêmes indices relatifs dans la liste résultante.

Adjacent empty matches are not possible, but an empty match can occur immediately after a non-empty match.

re.split(r'\b', 'Words, words, words.')

['', 'Words', ', ', 'words', ', ', 'words', '.']

re.split(r'\W*', '...words...')

['', '', 'w', 'o', 'r', 'd', 's', '', '']

re.split(r'(\W*)', '...words...')

['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the | operator).

Modifié dans la version 3.1: ajout de l'argument optionnel *flags*.

Modifié dans la version 3.7: Gestion du découpage avec un motif qui pourrait correspondre à une chaine de caractère vide.

Obsolète depuis la version 3.13: Passing *maxsplit* and *flags* as positional arguments is deprecated. In future Python versions they will be keyword-only parameters.

re.**findall**(*pattern*, *string*, *flags=0*)

Renvoie toutes les correspondances, sans chevauchements, entre le motif *pattern* et la chaîne *string*, comme une liste de chaînes ou de *n*-uplets. La chaîne *string* est examinée de gauche à droite, et les correspondances sont données dans cet ordre. Le résultat peut contenir des correspondances vides.

Le type du résultat dépend du nombre de groupes capturants dans le motif. S'il n'y en a pas, le résultat est une liste de sous-chaînes de caractères qui correspondent au motif. S'il y a exactement un groupe, le résultat est une liste constituée des sous-chaînes qui correspondaient à ce groupe pour chaque correspondance entre le motif et la chaîne. S'il y a plusieurs groupes, le résultat est formé de *n*-uplets avec les sous-chaînes correspondant aux différents groupes.

re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')

['foot', 'fell', 'fastest']

re.findall(r'(\w+)=(\d+)', 'set width=20 and height=10')

[('width', '20'), ('height', '10')]

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the | operator).

Modifié dans la version 3.7: Les correspondances non vides peuvent maintenant démarrer juste après une correspondance vide précédente.

re.**finditer**(*pattern*, *string*, *flags=0*)

Return an iterator yielding Match objects over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the | operator).

Modifié dans la version 3.7: Les correspondances non vides peuvent maintenant démarrer juste après une correspondance vide précédente.

re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Renvoie la chaîne obtenue en remplaçant les occurrences (sans chevauchement) les plus à gauche de *pattern* dans *string* par le remplacement *repl*. Si le motif n'est pas trouvé, *string* est renvoyée inchangée. *repl* peut être une chaîne de caractères ou une fonction ; si c'est une chaîne, toutes les séquences d'échappement qu'elle contient sont traduites. Ainsi, \n est convertie en un simple saut de ligne, \r en un retour chariot, et ainsi de suite. Les échappements inconnus de lettres ASCII sont réservés pour une utilisation future et sont considérés comme des erreurs. Les autres échappements tels que \& sont laissés intacts. Les références arrières, telles que \6, sont remplacées par la sous-chaîne correspondant au groupe 6 dans le motif. Par exemple :

re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',

    r'static PyObject*\npy_\1(void)\n{',

    'def myfunc():')

'static PyObject*\npy_myfunc(void)\n{'

If *repl* is a function, it is called for every non-overlapping occurrence of *pattern*. The function takes a single Match argument, and returns the replacement string. For example:

def dashrepl(matchobj):

  if matchobj.group(0) == '-': **return** ' '

  else: **return** '-'

re.sub('-{1,2}', dashrepl, 'pro----gram-files')

'pro--gram files'

re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)

'Baked Beans & Spam'

The pattern may be a string or a Pattern.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. If omitted or zero, all occurrences will be replaced.

Adjacent empty matches are not possible, but an empty match can occur immediately after a non-empty match. As a result, sub('x*', '-', 'abxd') returns '-a-b--d-' instead of '-a-b-d-'.

Dans les arguments *repl* de type *string*, en plus des séquences d'échappement et références arrières décrites au-dessus, \g<name> utilisera la sous-chaîne correspondant au groupe nommé name, comme défini par la syntaxe (?P<name>...). \g<number> utilise le groupe numéroté associé ; \g<2> est ainsi équivalent à \2, mais n'est pas ambigu dans un remplacement tel que \g<2>0, \20 serait interprété comme une référence au groupe 20, et non une référence au groupe 2 suivie par un caractère littéral '0'. La référence arrière \g<0> est remplacée par la sous-chaîne entière validée par l'expression rationnelle.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the | operator).

Modifié dans la version 3.1: ajout de l'argument optionnel *flags*.

Modifié dans la version 3.5: Les groupes sans correspondance sont remplacés par une chaîne vide.

Modifié dans la version 3.6: Les séquences d'échappement inconnues dans *pattern* formées par '\' et une lettre ASCII sont maintenant des erreurs.

Modifié dans la version 3.7: Unknown escapes in *repl* consisting of '\' and an ASCII letter now are errors. An empty match can occur immediately after a non-empty match.

Modifié dans la version 3.12: Group *id* can only contain ASCII digits. In bytes replacement strings, group *name* can only contain bytes in the ASCII range (b'\x00'-b'\x7f').

Obsolète depuis la version 3.13: Passing *count* and *flags* as positional arguments is deprecated. In future Python versions they will be keyword-only parameters.

re.**subn**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Réalise la même opération que sub(), mais renvoie une
paire (nouvelle_chaîne, nombre_de_substitutions_réalisées).

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the | operator).

re.**escape**(*pattern*)

Échappe tous les caractères spéciaux de *pattern*. Cela est utile si vous voulez valider une quelconque chaîne littérale qui pourrait contenir des métacaractères d'expressions rationnelles. Par exemple :

print(re.escape('https://www.python.org'))

https://www\.python\.org

legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"

print('[**%s**]+' % re.escape(legal_chars))

[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&'*\+\-\.\^_`\|\~:]+

operators = ['+', '-', '*', '/', '**']

print('|'.join(map(re.escape, sorted(operators, reverse=**True**))))

/|\-|\+|**|*

Cette fonction ne doit pas être utilisée pour la chaîne de remplacement dans sub() et subn(), seuls les antislashs devraient être échappés. Par exemple :

digits_re = r'\d+'

sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'

print(re.sub(digits_re, digits_re.replace('****', r'****'), sample))

/usr/sbin/sendmail - \d+ errors, \d+ warnings

Modifié dans la version 3.3: Le caractère '_' n'est plus échappé.

Modifié dans la version 3.7: Seuls les caractères qui peuvent avoir une signification spéciale dans une expression rationnelle sont échappés. De ce fait, '!', '"', '%', "'", ',', '/', ':', ';', '<', '=', '>', '@', et "`" ne sont plus échappés.

re.**purge**()

Vide le cache d'expressions rationnelles.

Exceptions

exception re.**PatternError**(*msg*, *pattern=None*, *pos=None*)

Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern. The PatternError instance has the following additional attributes:

msg

Le message d'erreur non formaté.

pattern

Le motif d'expression rationnelle.

pos

L'index dans *pattern* où la compilation a échoué (peut valoir None).

lineno

La ligne correspondant à *pos* (peut valoir None).

colno

La colonne correspondant à *pos* (peut valoir None).

Modifié dans la version 3.5: Ajout des attributs additionnels.

Modifié dans la version 3.13: PatternError was originally named error; the latter is kept as an alias for backward compatibility.

Objets d'expressions rationnelles

class re.**Pattern**

Compiled regular expression object returned by re.compile().

Modifié dans la version 3.9: re.Pattern supports [] to indicate a Unicode (str) or bytes pattern. See Type Alias générique.

Pattern.**search**(*string*[, *pos*[, *endpos*]])

Scan through *string* looking for the first location where this regular expression produces a match, and return a corresponding Match. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

Le second paramètre *pos* (optionnel) donne l'index dans la chaîne où la recherche doit débuter ; il vaut 0 par défaut. Cela n'est pas complètement équivalent à un *slicing* sur la chaîne ; le caractère de motif '^' correspond au début réel de la chaîne et aux positions juste après un saut de ligne, mais pas nécessairement à l'index où la recherche commence.

Le paramètre optionnel *endpos* limite la longueur sur laquelle la chaîne sera analysée ; ce sera comme si la chaîne faisait *endpos* caractères de long, donc uniquement les caractères de *pos* à endpos - 1 seront analysés pour trouver une correspondance. Si *endpos* est inférieur à *pos*, aucune correspondance ne sera trouvée ; dit autrement, avec *rx* une expression rationnelle compilée, rx.search(string, 0, 50) est équivalent à rx.search(string[:50], 0).

```
pattern = re.compile("d")

pattern.search("dog")     # Match at index 0

<re.Match object; span=(0, 1), match='d'>

pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

Pattern.**match**(*string*[, *pos*[, *endpos*]])

If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding Match. Return None if the string does not match the pattern; note that this is different from a zero-length match.

Les paramètres optionnels *pos* et *endpos* ont le même sens que pour la méthode search().

```
pattern = re.compile("o")

pattern.match("dog")     # No match as "o" is not at the start of "dog".

pattern.match("dog", 1)  # Match as "o" is the 2nd character of "dog".

<re.Match object; span=(1, 2), match='o'>
```

Si vous voulez une recherche n'importe où dans *string*, utilisez plutôt search() (voir aussi Comparaison de search() et match()).

Pattern.**fullmatch**(*string*[, *pos*[, *endpos*]])

If the whole *string* matches this regular expression, return a corresponding Match.
Return None if the string does not match the pattern; note that this is different from a zero-length match.

Les paramètres optionnels *pos* et *endpos* ont le même sens que pour la méthode search().

pattern = re.compile("o[gh]")

pattern.fullmatch("dog") *# No match as "o" is not at the start of "dog".*

pattern.fullmatch("ogre") *# No match as not the full string matches.*

pattern.fullmatch("doggie", 1, 3) *# Matches within given limits.*

<re.Match object; span=(1, 3), match='og'>

Ajouté dans la version 3.4.

Pattern.**split**(*string*, *maxsplit=0*)

Identique à la fonction split(), en utilisant le motif compilé.

Pattern.**findall**(*string*[, *pos*[, *endpos*]])

Similaire à la fonction findall(), en utilisant le motif compilé, mais accepte aussi des paramètres *pos* et *endpos* optionnels qui limitent la région de recherche comme pour search().

Pattern.**finditer**(*string*[, *pos*[, *endpos*]])

Similaire à la fonction finditer(), en utilisant le motif compilé, mais accepte aussi des paramètres *pos* et *endpos* optionnels qui limitent la région de recherche comme pour search().

Pattern.**sub**(*repl*, *string*, *count=0*)

Identique à la fonction sub(), en utilisant le motif compilé.

Pattern.**subn**(*repl*, *string*, *count=0*)

Identique à la fonction subn(), en utilisant le motif compilé.

Pattern.**flags**

The regex matching flags. This is a combination of the flags given to compile(), any (?...) inline flags in the pattern, and implicit flags such as UNICODE if the pattern is a Unicode string.

Pattern.**groups**

Le nombre de groupes de capture dans le motif.

Pattern.**groupindex**

Un dictionnaire associant les noms de groupes symboliques définis par (?P<id>) aux groupes numérotés. Le dictionnaire est vide si aucun groupe symbolique n'est utilisé dans le motif.

Pattern.**pattern**

La chaîne de motif depuis laquelle l'objet motif a été compilé.

Modifié dans la version 3.7: Ajout du support des fonctions copy.copy() et copy.deepcopy(). Les expressions régulières compilées sont considérées atomiques.

Objets de correspondance

Les objets de correspondance ont toujours une valeur booléenne True.
Puisque match() et search() renvoient None quand il n'y a pas de correspondance, vous pouvez tester s'il y a eu correspondance avec une simple instruction if :

match = re.search(pattern, string)

if match:

  process(match)

class re.**Match**

Match object returned by successful matches and searches.

Modifié dans la version 3.9: re.Match supports [] to indicate a Unicode (str) or bytes match.
See Type Alias générique.

Match.**expand**(*template*)

Return the string obtained by doing backslash substitution on the template string *template*, as done by the sub() method. Escapes such as \n are converted to the appropriate characters, and numeric backreferences (\1, \2) and named backreferences (\g<1>, \g<name>) are replaced by the contents of the corresponding group. The backreference \g<0> will be replaced by the entire match.

Modifié dans la version 3.5: Les groupes sans correspondance sont remplacés par une chaîne vide.

Match.**group**([*group1*, ...])

Renvoie un ou plus sous-groupes de la correspondance. Si un seul argument est donné, le résultat est une chaîne simple ; s'il y a plusieurs arguments, le résultat est un *n*-uplet comprenant un élément par argument. Sans arguments, *group1* vaut par défaut zéro (la correspondance entière est renvoyée). Si un argument *groupN* vaut zéro, l'élément associé sera la chaîne de correspondance entière ; s'il est dans l'intervalle fermé [1..99], c'est la correspondance avec le groupe de parenthèses associé. Si un numéro de groupe est négatif ou supérieur au nombre de groupes définis dans le motif, une exception indexError est levée. Si un groupe est contenu dans une partie du motif qui n'a aucune correspondance, l'élément associé sera None. Si un groupe est contenu dans une partie du motif qui a plusieurs correspondances, seule la dernière correspondance est renvoyée.

m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")

m.group(0) *# The entire match*

'Isaac Newton'

m.group(1) *# The first parenthesized subgroup.*

'Isaac'

m.group(2) *# The second parenthesized subgroup.*

'Newton'

m.group(1, 2) *# Multiple arguments give us a tuple.*

('Isaac', 'Newton')

Si l'expression rationnelle utilise la syntaxe (?P<name>...), les arguments *groupN* peuvent alors aussi être des chaînes identifiant les groupes par leurs noms. Si une chaîne donnée en argument n'est pas utilisée comme nom de groupe dans le motif, une exception IndexError est levée.

Un exemple modérément compliqué :

m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")

m.group('first_name')

'Malcolm'

m.group('last_name')

'Reynolds'

Les groupes nommés peuvent aussi être référencés par leur index :

m.group(1)

'Malcolm'

m.group(2)

'Reynolds'

Si un groupe a plusieurs correspondances, seule la dernière est accessible :

m = re.match(r"(..)+", "a1b2c3") *# Matches 3 times.*

m.group(1) *# Returns only the last match.*

'c3'

Match.__**getitem**__(*g*)

Cela est identique à m.group(g). Cela permet un accès plus facile à un groupe individuel depuis une correspondance :

m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")

m[0] *# The entire match*

'Isaac Newton'

m[1] *# The first parenthesized subgroup.*

'Isaac'

m[2] *# The second parenthesized subgroup.*

'Newton'

Named groups are supported as well:

m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Isaac Newton")

m['first_name']

'Isaac'

m['last_name']

'Newton'

Ajouté dans la version 3.6.

Match.**groups**(*default=None*)

Renvoie un *n*-uplet contenant tous les sous-groupes de la correspondance, de 1 jusqu'au nombre de groupes dans le motif. L'argument *default* est utilisé pour les groupes sans correspondance ; il vaut None par défaut.

Par exemple :

m = re.match(r"(\d+)\.(\d+)", "24.1632")

m.groups()

('24', '1632')

Si on rend la partie décimale et tout ce qui la suit optionnels, tous les groupes ne figureront pas dans la correspondance. Ces groupes sans correspondance vaudront None sauf si une autre valeur est donnée à l'argument *default* :

m = re.match(r"(\d+)\.?(\d+)?", "24")

m.groups() *# Second group defaults to None.*

('24', None)

m.groups('0') *# Now, the second group defaults to '0'.*

('24', '0')

Match.**groupdict**(*default=None*)

Renvoie un dictionnaire contenant tous les sous-groupes *nommés* de la correspondance, accessibles par leurs noms. L'argument *default* est utilisé pour les groupes qui ne figurent pas dans la correspondance ; il vaut None par défaut. Par exemple :

m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")

m.groupdict()

{'first_name': 'Malcolm', 'last_name': 'Reynolds'}

Match.**start**([*group*])

Match.**end**([*group*])

Renvoie les indices de début et de fin de la sous-chaîne correspondant au groupe *group* ; *group* vaut par défaut zéro (pour récupérer les indices de la correspondance complète). Renvoie -1 si *group* existe mais ne figure pas dans la correspondance. Pour un objet

de correspondance *m*, et un groupe *g* qui y figure, la sous-chaîne correspondant au groupe *g* (équivalente à m.group(g)) est

m.string[m.start(g):m.end(g)]

Notez que m.start(group) sera égal à m.end(group) si *group* correspond à une chaîne vide. Par exemple, après m = re.search('b(c?)', 'cba'), m.start(0) vaut 1, m.end(0) vaut 2, m.start(1) et m.end(1) valent tous deux 2, et m.start(2) lève une exception IndexError.

Un exemple qui supprimera *remove_this* d'une adresse mail :

email = "tony@tiremove_thisger.net"

m = re.search("remove_this", email)

email[:m.start()] + email[m.end():]

'tony@tiger.net'

Match.**span**([*group*])

Pour un objet de correspondance *m*, renvoie la paire (m.start(group), m.end(group)). Notez que si *group* ne figure pas dans la correspondance, (-1, -1) est renvoyé. *group* vaut par défaut zéro, pour la correspondance entière.

Match.**pos**

La valeur de *pos* qui a été passée à la méthode search() ou match() d'un objet expression rationnelle. C'est l'index dans la chaîne à partir duquel le moteur d'expressions rationnelles recherche une correspondance.

Match.**endpos**

La valeur de *endpos* qui a été passée à la méthode search() ou match() d'un objet expression rationnelle. C'est l'index dans la chaîne que le moteur d'expressions rationnelles ne dépassera pas.

Match.**lastindex**

L'index entier du dernier groupe de capture validé, ou None si aucun groupe ne correspondait. Par exemple, les expressions (a)b, ((a)(b)) et ((ab)) auront un lastindex == 1 si appliquées à la chaîne 'ab', alors que l'expression (a)(b) aura un lastindex == 2 si appliquée à la même chaîne.

Match.**lastgroup**

Le nom du dernier groupe capturant validé, ou None si le groupe n'a pas de nom, ou si aucun groupe ne correspondait.

Match.**re**

L'expression rationnelle dont la méthode match() ou search() a produit cet objet de correspondance.

Match.**string**

La chaîne passée à match() ou search().

Modifié dans la version 3.7: Ajout du support des fonctions copy.copy() et copy.deepcopy(). Les objets correspondants sont considérés atomiques.

Exemples d'expressions rationnelles

Rechercher une paire

Dans cet exemple, nous nous aidons de la fonction suivante pour afficher de manière plus jolie les objets qui correspondent :

```
def displaymatch(match):
  if match is None:
    return None
  return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Supposez que vous écriviez un jeu de poker où la main d'un joueur est représentée par une chaîne de 5 caractères avec chaque caractère représentant une carte, « a » pour l'as, « k » pour le roi (*king*), « q » pour la reine (*queen*), « j » pour le valet (*jack*), « t » pour 10 (*ten*), et les caractères de « 2 » à « 9 » représentant les cartes avec ces valeurs.

Pour vérifier qu'une chaîne donnée est une main valide, on pourrait faire comme suit :

```
valid = re.compile(r"^[a2-9tjqk]{5}$")
displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
displaymatch(valid.match("akt5e"))  # Invalid.
displaymatch(valid.match("akt"))    # Invalid.
displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

La dernière main, "727ak", contenait une paire, deux cartes de la même valeur. Pour valider cela avec une expression rationnelle, on pourrait utiliser des références arrière comme :

```
pair = re.compile(r".*(.).*\1")
displaymatch(pair.match("717ak"))    # Pair of 7s.
"<Match: '717', groups=('7',)>"
displaymatch(pair.match("718ak"))    # No pairs.
displaymatch(pair.match("354aa"))    # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

Pour trouver de quelle carte est composée la paire, on pourrait utiliser la méthode group() de l'objet de correspondance de la manière suivante :

```
pair = re.compile(r".*(.).*\1")
pair.match("717ak").group(1)
```

'7'

Error because re.match() returns None, which doesn't have a group() method:

pair.match("718ak").group(1)

Traceback (most recent call last):

  File "<pyshell#23>", line 1, in <module>

    re.match(r".*(.).*\1", "718ak").group(1)

AttributeError: 'NoneType' object has no attribute 'group'

pair.match("354aa").group(1)

'a'

Simuler *scanf()*

Python does not currently have an equivalent to scanf(). Regular expressions are generally more powerful, though also more verbose, than scanf() format strings. The table below offers some more-or-less equivalent mappings between scanf() format tokens and regular expressions.

Pour extraire le nom de fichier et les nombres depuis une chaîne comme

/usr/sbin/sendmail - 0 errors, 4 warnings

you would use a scanf() format like

%s - %d errors, %d warnings

L'expression rationnelle équivalente serait

(\S+) - (\d+) errors, (\d+) warnings

Comparaison de *search()* et *match()*

Python offre différentes opérations primitives basées sur des expressions régulières :

- re.match() cherche une correspondance uniquement au début de la chaîne de caractères
- re.search() cherche une correspondance n'importe où dans la chaîne de caractères (ce que fait Perl par défaut)
- re.fullmatch() cherche une correspondance avec l'intégralité de la chaîne de caractères.

Par exemple :

re.match("c", "abcdef") *# No match*

re.search("c", "abcdef") *# Match*

<re.Match object; span=(2, 3), match='c'>

re.fullmatch("p.*n", "python") *# Match*

<re.Match object; span=(0, 6), match='python'>

re.fullmatch("r.*n", "python") *# No match*

Les expressions rationnelles commençant par '^' peuvent être utilisées avec search() pour restreindre la recherche au début de la chaîne :

re.match("c", "abcdef") *# No match*

re.search("^c", "abcdef") *# No match*

re.search("^a", "abcdef") *# Match*

<re.Match object; span=(0, 1), match='a'>

Notez cependant qu'en mode MULTILINE, match() ne recherche qu'au début de la chaîne, alors que search() avec une expression rationnelle commençant par '^' recherchera au début de chaque ligne.

re.match("X", "A**\n**B**\n**X", re.MULTILINE) *# No match*

re.search("^X", "A**\n**B**\n**X", re.MULTILINE) *# Match*

<re.Match object; span=(4, 5), match='X'>

Construire un répertoire téléphonique

split() découpe une chaîne en une liste délimitée par le motif donné. La méthode est inestimable pour convertir des données textuelles vers des structures de données qui peuvent être lues et modifiées par Python comme démontré dans l'exemple suivant qui crée un répertoire téléphonique.

Tout d'abord, voici l'entrée. Elle provient normalement d'un fichier, nous utilisons ici une chaîne à guillemets triples

text = """Ross McFluff: 834.345.1254 155 Elm Street

Ronald Heathmore: 892.345.3428 436 Finley Avenue

Frank Burger: 925.541.7625 662 South Dogwood Way

Heather Albrecht: 548.326.4584 919 Park Place"""

Les entrées sont séparées par un saut de ligne ou plus. Nous convertissons maintenant la chaîne en une liste où chaque ligne non vide aura sa propre entrée :

entries = re.split("**\n**+", text)

entries

['Ross McFluff: 834.345.1254 155 Elm Street',

'Ronald Heathmore: 892.345.3428 436 Finley Avenue',

'Frank Burger: 925.541.7625 662 South Dogwood Way',

'Heather Albrecht: 548.326.4584 919 Park Place']

Finalement, on sépare chaque entrée en une liste avec prénom, nom, numéro de téléphone et adresse. Nous utilisons le paramètre maxsplit de split() parce que l'adresse contient des espaces, qui sont notre motif de séparation :

[re.split(":? ", entry, maxsplit=3) **for** entry **in** entries]

[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],

['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],

['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],

['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]

Le motif :? trouve les deux points derrière le nom de famille, pour qu'ils n'apparaissent pas dans la liste résultante. Avec un maxsplit de 4, nous pourrions séparer le numéro du nom de la rue :

[re.split(":? ", entry, maxsplit=4) **for** entry **in** entries]

[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],

['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],

['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],

['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]

Mélanger les lettres des mots

sub() remplace toutes les occurrences d'un motif par une chaîne ou le résultat d'une fonction. Cet exemple le montre, en utilisant sub() avec une fonction qui mélange aléatoirement les caractères de chaque mot dans une phrase (à l'exception des premiers et derniers caractères) :

```
def repl(m):
  inner_word = list(m.group(2))
  random.shuffle(inner_word)
  return m.group(1) + "".join(inner_word) + m.group(3)

text = "Professor Abdolmalek, please report your absences promptly."
re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

Trouver tous les adverbes

findall() trouve *toutes* les occurrences d'un motif, pas juste la première comme le fait search(). Par exemple, si un écrivain voulait trouver tous les adverbes dans un texte, il devrait utiliser findall() de la manière suivante :

```
text = "He was carefully disguised but captured quickly by police."
re.findall(r"\w+ly\b", text)
['carefully', 'quickly']
```

Trouver tous les adverbes et leurs positions

If one wants more information about all matches of a pattern than the matched text, finditer() is useful as it provides Match objects instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs *and their positions* in some text, they would use finditer() in the following manner:

```
text = "He was carefully disguised but captured quickly by police."
for m in re.finditer(r"\w+ly\b", text):
  print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

Notation brute de chaînes

La notation brute de chaînes (r"text") garde saines les expressions rationnelles. Sans elle, chaque *backslash* ('\') dans une expression rationnelle devrait être préfixé d'un autre *backslash* pour l'échapper. Par exemple, les deux lignes de code suivantes sont fonctionnellement identiques :

```
re.match(r"\W(.)\1\W", " ff ")
```

```
<re.Match object; span=(0, 4), match=' ff '>
```

```
re.match("\\W(.)\\1\\W", " ff ")
```

```
<re.Match object; span=(0, 4), match=' ff '>
```

Pour rechercher un *backslash* littéral, il faut l'échapper dans l'expression rationnelle. Avec la notation brute, cela signifie r"\\". Sans elle, il faudrait utiliser "\\\\", faisant que les deux lignes de code suivantes sont fonctionnellement identiques :

```
re.match(r"\\", r"\\")
```

```
<re.Match object; span=(0, 1), match='\\'>
```

```
re.match("\\\\", r"\\")
```

```
<re.Match object; span=(0, 1), match='\\'>
```

Écrire un analyseur lexical

Un analyseur lexical ou scanner analyse une chaîne pour catégoriser les groupes de caractères. C'est une première étape utile dans l'écriture d'un compilateur ou d'un interpréteur.

Les catégories de texte sont spécifiées par des expressions rationnelles. La technique est de les combiner dans une unique expression rationnelle maîtresse, et de boucler sur les correspondances successives :

```
from typing import NamedTuple
import re


class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int


def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
```

```
        ('OP',      r'[+\-*/]'),     # Arithmetic operators
        ('NEWLINE', r'\n'),          # Line endings
        ('SKIP',    r'[ \t]+'),      # Skip over spaces and tabs
        ('MISMATCH', r'.'),          # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
```

'''

```
for token in tokenize(statements):
    print(token)
```

L'analyseur produit la sortie suivante :

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```