portail

Syntaxe des expressions régulières en Python

La barre oblique inversée \ précède un caractère spécial à utiliser tel quel, sans son utilisation particulière.

Syntaxe des expressions régulières

Une expression régulière (ou RE) spécifie un ensemble de chaînes qui lui correspond.

Les expressions régulières peuvent être concaténées pour former de nouvelles expressions régulières ; si A et B sont deux expressions régulières, AB est également une expression régulière.

La plupart des caractères ordinaires, comme **A**, **a** ou **0** correspondent simplement à leur caractère. Vous pouvez concaténer des caractères ordinaires, donc last correspond à la chaîne 'last'.

Des caractères spéciaux, comme **|** ou **(**, modifient l'interprétation des expressions régulières

Les caractères de répétition (*, +,?, {M, n}, etc.) ne peuvent pas être imbriqués. Pour appliquer une seconde répétition à une répétition, vous pouvez utiliser des parenthèses. Par exemple, l'expression **(?:a{6})*** correspond à un multiple de six caractères “a”.

Les caractères spéciaux sont :

. (Point)
: Par défaut, cela correspond à n'importe quel caractère sauf une nouvelle ligne.

^ (Flèche haut)
: Correspond au début de la chaîne
en mode MULTILINE, correspond également immédiatement après chaque nouvelle ligne.

$
: Correspond à la fin de la chaîne ou juste avant la nouvelle ligne à la fin de la chaîne, et en mode MULTILINE, correspond également avant une nouvelle ligne.
foo correspond à “foo” et à “foobar”, tandis que l’expression régulière **foo$** ne correspond qu’à “foo”.
Plus intéressant : rechercher **foo.$** dans 'foo1\nfoo2\n' correspond à ‘foo2’ normalement, mais à ‘foo1’ en mode en mode MULTILINE; rechercher un seul **$** dans 'foo\n' trouvera deux correspondances (vides): une juste avant la nouvelle ligne et une à la fin de la chaîne.

*
: correspond à 0 répétition ou plus de l'expression régulière précédente.
ab* correspond à «a», «ab» ou «a» suivi d’un nombre quelconque de «b».

+
: correspond à une ou plusieurs répétitions de l'expression régulière précédente.
ab+ correspondr à «a» suivi de tout nombre non nul de «b» ; il ne correspondra pas simplement à «a».

?
: correspond à 0 ou 1 répétition de l'expression régulière précédente.
ab? correspond à “a” ou “ab”.

*?, +?, ??
: '*', '+', and '?' correspondent à autant de texte que possible.
Ce n'est pas toujours ce que l'on souhaite ;
si l'expression régulière <.*> est comparée à '<a> b <c>', elle correspondra à la chaîne entière et pas seulement à <a>.
En ajoutant un ? après le qualificatif, autant de caractères que possible seront appariés.
L'expression régulière <.*?> ne correspond qu'à '<a>'.

{m}
: correspond à exactement m copies de l'expression précédente ; s'il y a moins de correspondances, font que l'expression régulière ne correspond pas.
Par exemple, a{6} correspond à exactement six caractères “a”, mais pas à cinq.

{m,n}
: correspond à de m à n répétitions de l'expression précédente, en essayant de faire correspondre autant de répétitions que possible.
Par exemple, a{3,5} correspondra à de 3 à 5 caractères “a”.
Omettre m spécifie une limite inférieure égale à zéro et omettre n spécifie une limite supérieure infinie.
Par exemple, a{4,}b correspond à “aaaab” ou à un millier de caractères “a” suivis d'un “b”, mais pas “aaab”.
La virgule ne peut pas être omise pour éviter la confusion avec la forme décrite précédemment.

{m,n}?
: correspond à de m à n répétitions de l'expression, en essayant de faire correspondre le moins possible de répétitions.
Par exemple, sur la chaîne de 6 caractères 'aaaaaa', a{3,5} correspond à 5 caractères 'a', alors que a{3,5}? correspond à 3 caractères.

\
: Either escapes special characters (permitting you to match characters like '*', '?', and so forth), or signals a special sequence; special sequences are discussed below.
If you’re not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn’t recognized by Python’s parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it’s highly recommended that you use raw strings for all but the simplest expressions.

[]
: Used to indicate a set of characters. In a set:

  - Characters can be listed individually, e.g. [amk] will match 'a', 'm', or 'k'.
  - Ranges of characters can be indicated by giving two characters and separating them by a '-', for example [a-z] will match any lowercase ASCII letter, [0-5][0-9] will match all the two-digits numbers from 00 to 59, and [0-9A-Fa-f] will match any hexadecimal digit. If - is escaped (e.g. [a\-z]) or if it’s placed as the first or last character (e.g. [-a] or [a-]), it will match a literal '-'.
  - Special characters lose their special meaning inside sets. For example, [(+*)] will match any of the literal characters '(', '+', '*', or ')'.
  - Character classes such as \w or \S (defined below) are also accepted inside a set,

although the characters they match depends on whether ASCII or LOCALE mode is in force.

- Characters that are not within a range can be matched by complementing the set. If the first character of the set is '^', all the characters that are not in the set will be matched. For example, [^5] will match any character except '5', and [^^] will match any character except '^'. ^ has no special meaning if it’s not the first character in the set.
- 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 both match a parenthesis.
- Support of nested sets and set operations as in Unicode Technical Standard #18 might be added in the future. This would change the syntax, so to facilitate this change a FutureWarning will be raised in ambiguous cases for the time being. That includes sets starting with a literal '[' or containing literal character sequences '–', '&&', '~~', and '||'. To avoid a warning escape them with a backslash.

|

A|B where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the '|' in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by '|' are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. In other words, the '|' operator is never greedy. To match a literal '|', use \|, or enclose it inside a character class, as in [|].

(…)

Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the \number special sequence, described below. To match the literals '(' or ')', use \(or \), or enclose them inside a character class: [(], [)].

(?…)

This is an extension notation (a '?' following a '(' is not meaningful otherwise). The first character after the '?' determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; (?P<name>…) is the only exception to this rule. Following are the currently supported extensions.

(?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: 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), and re.X (verbose), for the entire regular expression. (The flags are described in Module Contents.) 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.

(?:…)

A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group cannot be retrieved after performing a match or referenced later in the pattern.

(?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: 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), and re.X (verbose), for the part of the expression. (The flags are described in Module Contents.) 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 byte pattern (?L:…) switches to locale depending 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.

(?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 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. Named groups can be referenced in three contexts. If the pattern is (?P<quote>['"]).*?(?P=quote) (i.e. matching a string quoted with either single or double quotes):

Context of reference to group "quote"	**Ways to reference it**
in the same pattern itself	(?P=quote) (as shown)\ \1
when processing match object m	m.group('quote')\ m.end('quote') (etc.)
in a string passed to the repl\ argument of re.sub()	\g<quote>\ \g<1>\ \1

(?P=name)

A backreference to a named group; it matches whatever text was matched by the earlier group named name.

(?#…)

A comment; the contents of the parentheses are simply ignored.

(?=…)

Matches if … matches next, but doesn't consume any of the string. This is called a lookahead assertion. For example, Isaac (?=Asimov) will match 'Isaac ' only if it's followed by 'Asimov'.

(?!…)

Matches if … doesn't match next. This is a negative lookahead assertion. For example, Isaac (?!Asimov) will match 'Isaac ' only if it's not followed by 'Asimov'.

(?⇐…)

Matches if the current position in the string is preceded by a match for … that ends at the current position. This is called a positive lookbehind assertion. (?⇐abc)def will find a match in 'abcdef', since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that abc or a|b are allowed, but a* and a{3,4} are not. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the search() function rather than the match() function:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

(?<!...)

Matches if the current position in the string is not preceded by a match for …. This is called a negative lookbehind assertion. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

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

Will try to match with yes-pattern if the group with given id or name exists, and with no-pattern if it doesn't. no-pattern is optional and can be omitted. For example, (<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$) is a poor email matching pattern, which will match with 'user@host.com' as well as 'user@host.com', but not with '<user@host.com' nor 'user@host.com>'.

The special sequences consist of '\' and a character from the list below. If the ordinary character is not an ASCII digit or an ASCII letter, then the resulting RE will match the second character. For example, \$ matches the character '$'.

\number

Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, (.+) \1 matches 'the the' or '55 55', but not 'thethe' (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of number is 0, or number is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value number. Inside the '[' and ']' of a character class, all numeric escapes are treated as characters.

\A

Matches only at the start of the string.

\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/end of the string. This means that r'\bfoo\b' matches 'foo', 'foo.', '(foo)', 'bar foo baz' but not 'foobar' or 'foo3'. By default Unicode alphanumerics are the ones used in Unicode patterns, but this can be changed by using the ASCII flag. Word boundaries are determined by the current locale if the LOCALE flag is used. 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'py\B' matches 'python', 'py3', 'py2', but not 'py', 'py.', or 'py!'. \B is just the opposite of \b, so word characters in Unicode 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.

\d

For Unicode (str) patterns:Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes [0-9], and also many other digit characters. If the ASCII flag is used only [0-9] is matched. For 8-bit (bytes) patterns:Matches any decimal digit; this is equivalent to [0-9].

\D
: Matches any character which is not a decimal digit. This is the opposite of \d. If the ASCII flag is used this becomes the equivalent of [^0-9].

\s
: For Unicode (str) patterns:Matches Unicode whitespace characters (which includes [\t\n\r\f\v], and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages). If the ASCII flag is used, only [\t\n\r\f\v] is matched.
For 8-bit (bytes) patterns:Matches characters considered whitespace in the ASCII character set; this is equivalent to [\t\n\r\f\v].

\S
: Matches any character which is not a whitespace character. This is the opposite of \s. If the ASCII flag is used this becomes the equivalent of [^ \t\n\r\f\v].

\w
: For Unicode (str) patterns:Matches Unicode word characters; this includes most characters that can be part of a word in any language, as well as numbers and the underscore. If the ASCII flag is used, only [a-zA-Z0-9_] is matched.
For 8-bit (bytes) patterns: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. If the ASCII flag is used this becomes the equivalent of [^a-zA-Z0-9_]. If the LOCALE flag is used, matches characters considered alphanumeric in the current locale and the underscore.

\Z
: Matches only at the end of the string.

Most of the standard escapes supported by Python string literals are also accepted by the regular expression parser:

```
\a      \b      \f      \n
\r      \t      \u      \U
\v      \x      \\
```

(Note that \b is used to represent word boundaries, and means “backspace” only inside character classes.)

'\u' and '\U' escape sequences are only recognized in Unicode patterns. In bytes patterns they are errors. Unknown escapes of ASCII letters are reserved for future use and treated as errors. Octal escapes are included in a limited form. If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length.

Voir aussi

- **(en)** https://docs.python.org/3/library/re.html

Basé sur « Article » par Auteur.