# [re](https://docs.python.org/2/library/re.html#module-re)**— Regular expression**

### [**Table of Contents**](https://docs.python.org/2/contents.html)

* [7.2. **re** — Regular expression operations](https://docs.python.org/2/library/re.html)
  + [7.2.1. Regular Expression Syntax](https://docs.python.org/2/library/re.html#regular-expression-syntax)
  + [7.2.2. Module Contents](https://docs.python.org/2/library/re.html#module-contents)
  + [7.2.3. Regular Expression Objects](https://docs.python.org/2/library/re.html#regular-expression-objects)
  + [7.2.4. Match Objects](https://docs.python.org/2/library/re.html#match-objects)
  + [7.2.5. Examples](https://docs.python.org/2/library/re.html#examples)
    - [7.2.5.1. Checking For a Pair](https://docs.python.org/2/library/re.html#checking-for-a-pair)
    - [7.2.5.2. Simulating scanf()](https://docs.python.org/2/library/re.html#simulating-scanf)
    - [7.2.5.3. search() vs. match()](https://docs.python.org/2/library/re.html#search-vs-match)
    - [7.2.5.4. Making a Phonebook](https://docs.python.org/2/library/re.html#making-a-phonebook)
    - [7.2.5.5. Text Munging](https://docs.python.org/2/library/re.html#text-munging)
    - [7.2.5.6. Finding all Adverbs](https://docs.python.org/2/library/re.html#finding-all-adverbs)
    - [7.2.5.7. Finding all Adverbs and their Positions](https://docs.python.org/2/library/re.html#finding-all-adverbs-and-their-positions)
    - [7.2.5.8. Raw String Notation](https://docs.python.org/2/library/re.html#raw-string-notation)

[7.2. re — Regular expression operations — Python 2.7.18 documentation](https://docs.python.org/2/library/re.html#re.MatchObject.start)

Regular Expression Syntax

'.'

(Dot.) In the default mode, this matches any character except a newline. If the [**DOTALL**](https://docs.python.org/2/library/re.html#re.DOTALL) flag has been specified, this matches any character including a newline.

'^'

(Caret.) Matches the start of the string, and in [**MULTILINE**](https://docs.python.org/2/library/re.html#re.MULTILINE) mode also matches immediately after each newline.

'$'

Matches the end of the string or just before the newline at the end of the string, and in [**MULTILINE**](https://docs.python.org/2/library/re.html#re.MULTILINE) mode also matches before a newline. foo matches both ‘foo’ and ‘foobar’, while the regular expression foo$ matches only ‘foo’. More interestingly, searching for foo.$ in 'foo1\nfoo2\n' matches ‘foo2’ normally, but ‘foo1’ in [**MULTILINE**](https://docs.python.org/2/library/re.html#re.MULTILINE) mode; searching for a single $ in 'foo\n' will find two (empty) matches: one just before the newline, and one at the end of the string.

'\*'

Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab\* will match ‘a’, ‘ab’, or ‘a’ followed by any number of ‘b’s.

'+'

Causes the resulting RE to match 1 or more repetitions of the preceding RE. ab+ will match ‘a’ followed by any non-zero number of ‘b’s; it will not match just ‘a’.

'?'

Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. ab? will match either ‘a’ or ‘ab

\*?, +?, ??

The '\*', '+', and '?' qualifiers are all *greedy*; they match as much text as possible. Sometimes this behaviour isn’t desired; if the RE <.\*> is matched against <a> b <c>, it will match the entire string, and not just <a>. Adding ? after the qualifier makes it perform the match in *non-greedy* or *minimal* fashion; as *few* characters as possible will be matched. Using the RE <.\*?> will match only <a>.

{m}

Specifies that exactly *m* copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, a{6} will match exactly six 'a' characters, but not five.

{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. For example, a{3,5} will match from 3 to 5 'a' characters. Omitting *m* specifies a lower bound of zero, and omitting *n* specifies an infinite upper bound. As an example, a{4,}b will match aaaab or a thousand 'a' characters followed by a b, but not aaab. The comma may not be omitted or the modifier would be confused with the previously described form.

{m,n}?

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as *few* repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string 'aaaaaa', a{3,5} will match 5 'a' characters, while a{3,5}? will only match 3 characters.

'\'

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-]), 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 [**LOCALE**](https://docs.python.org/2/library/re.html#re.LOCALE) or [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) 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.

'|'

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 insv ide 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.

(?iLmsux)

(One or more letters from the set 'i', 'L', 'm', 's', 'u', 'x'.) The group matches the empty string; the letters set the corresponding flags: **[re.I](https://docs.python.org/2/library/re.html" \l "re.I" \o "re.I)** (ignore case), **[re.L](https://docs.python.org/2/library/re.html" \l "re.L" \o "re.L)** (locale dependent), **[re.M](https://docs.python.org/2/library/re.html" \l "re.M" \o "re.M)** (multi-line), **[re.S](https://docs.python.org/2/library/re.html" \l "re.S" \o "re.S)** (dot matches all), **[re.U](https://docs.python.org/2/library/re.html" \l "re.U" \o "re.U)** (Unicode dependent), and **[re.X](https://docs.python.org/2/library/re.html" \l "re.X" \o "re.X)** (verbose), for the entire regular expression. (The flags are described in [Module Contents](https://docs.python.org/2/library/re.html#contents-of-module-re).) 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()](https://docs.python.org/2/library/re.html" \l "re.compile" \o "re.compile)** function.

Note that the (?x) flag changes how the expression is parsed. It should be used first in the expression string, or after one or more whitespace characters. If there are non-whitespace characters before the flag, the results are undefined.

(?:...)

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.

(?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. Group references are not supported even if they match strings of some fixed length. 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()**](https://docs.python.org/2/library/re.html#re.search) function rather than the [**match()**](https://docs.python.org/2/library/re.html#re.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('(?<=-)\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 and shouldn’t contain group references. 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'.

The special sequences consist of '\' and a character from the list below. If the ordinary character is not on the list, 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 alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore character. 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, so the precise set of characters deemed to be alphanumeric depends on the values of the UNICODE and LOCALE flags. For example, r'\bfoo\b' matches 'foo', 'foo.', '(foo)', 'bar foo baz' but not 'foobar' or 'foo3'. 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 is also subject to the settings of LOCALE and UNICODE.

\d

When the [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) flag is not specified, matches any decimal digit; this is equivalent to the set [0-9]. With [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE), it will match whatever is classified as a decimal digit in the Unicode character properties database.

\D

When the [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) flag is not specified, matches any non-digit character; this is equivalent to the set [^0-9]. With [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE), it will match anything other than character marked as digits in the Unicode character properties database.

\s

When the [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) flag is not specified, it matches any whitespace character, this is equivalent to the set [ \t\n\r\f\v]. The [**LOCALE**](https://docs.python.org/2/library/re.html#re.LOCALE) flag has no extra effect on matching of the space. If [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) is set, this will match the characters [ \t\n\r\f\v] plus whatever is classified as space in the Unicode character properties database

\S

When the [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) flag is not specified, matches any non-whitespace character; this is equivalent to the set [^ \t\n\r\f\v] The [**LOCALE**](https://docs.python.org/2/library/re.html#re.LOCALE) flag has no extra effect on non-whitespace match. If [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) is set, then any character not marked as space in the Unicode character properties database is matched.

\w

When the [**LOCALE**](https://docs.python.org/2/library/re.html#re.LOCALE) and [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) flags are not specified, matches any alphanumeric character and the underscore; this is equivalent to the set [a-zA-Z0-9\_]. With [**LOCALE**](https://docs.python.org/2/library/re.html#re.LOCALE), it will match the set [0-9\_] plus whatever characters are defined as alphanumeric for the current locale. If [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) is set, this will match the characters [0-9\_] plus whatever is classified as alphanumeric in the Unicode character properties database.

\W

When the [**LOCALE**](https://docs.python.org/2/library/re.html#re.LOCALE) and [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) flags are not specified, matches any non-alphanumeric character; this is equivalent to the set [^a-zA-Z0-9\_]. With [**LOCALE**](https://docs.python.org/2/library/re.html#re.LOCALE), it will match any character not in the set [0-9\_], and not defined as alphanumeric for the current locale. If [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) is set, this will match anything other than [0-9\_] plus characters classified as not alphanumeric in the Unicode character properties database.

\Z

Matches only at the end of the string.

re.**compile**(*pattern*, *flags=0*)

Compile a regular expression pattern into a regular expression object, which can be used for matching using its [**match()**](https://docs.python.org/2/library/re.html#re.RegexObject.match) and [**search()**](https://docs.python.org/2/library/re.html#re.RegexObject.search) methods, described below. The expression’s behavior can be modified by specifying a *flags* value. Values can be any of the following variables, combined using bitwise OR (the | operator).

The sequence

prog = re.compile(pattern)

result = prog.match(string)

is equivalent to

result = re.match(pattern, string)

but using **[re.compile()](https://docs.python.org/2/library/re.html" \l "re.compile" \o "re.compile)** and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

**Note**

The compiled versions of the most recent patterns passed to **[re.match()](https://docs.python.org/2/library/re.html" \l "re.match" \o "re.match)**, **[re.search()](https://docs.python.org/2/library/re.html" \l "re.search" \o "re.search)** or **[re.compile()](https://docs.python.org/2/library/re.html" \l "re.compile" \o "re.compile)** are cached, so programs that use only a few regular expressions at a time needn’t worry about compiling regular expressions.

re.**DEBUG**

Display debug information about compiled expression.

re.**I**

re.**IGNORECASE**

Perform case-insensitive matching; expressions like [A-Z] will match lowercase letters, too. This is not affected by the current locale. To get this effect on non-ASCII Unicode characters such as ü and Ü, add the [**UNICODE**](https://docs.python.org/2/library/re.html#re.UNICODE) flag.

re.**L**

re.**LOCALE**

Make \w, \W, \b, \B, \s and \S dependent on the current locale.

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.

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.

re.**U**

re.**UNICODE**

Make the \w, \W, \b, \B, \d, \D, \s and \S sequences dependent on the Unicode character properties database. Also enables non-ASCII matching for [**IGNORECASE**](https://docs.python.org/2/library/re.html#re.IGNORECASE).

re.**X**

re.**VERBOSE**

This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like \*?, (?: or (?P<...>. When a line contains a # that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such # through the end of the line are ignored.

This means that the two following regular expression objects that match a decimal number are functionally equal:

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

\. # the decimal point

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

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

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 **[MatchObject](https://docs.python.org/2/library/re.html" \l "re.MatchObject" \o "re.MatchObject)** instance. 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

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

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding **[MatchObject](https://docs.python.org/2/library/re.html" \l "re.MatchObject" \o "re.MatchObject)** instance. Return None if the string does not match the pattern; note that this is different from a zero-length match.

Note that even in [**MULTILINE**](https://docs.python.org/2/library/re.html#re.MULTILINE) mode, **[re.match()](https://docs.python.org/2/library/re.html" \l "re.match" \o "re.match)** will only match at the beginning of the string and not at the beginning of each line.

If you want to locate a match anywhere in string, use [**search()**](https://docs.python.org/2/library/re.html#re.search) instead (see also [search() vs. match()](https://docs.python.org/2/library/re.html#search-vs-match)).

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

Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list. (Incompatibility note: in the original Python 1.5 release, *maxsplit* was ignored. This has been fixed in later releases.)

**>>>** 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.']

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

['0', '3', '9

If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string:

**>>>** re.split('(\W+)', '...words, words...')

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

That way, separator components are always found at the same relative indices within the result list (e.g., if there’s one capturing group in the separator, the 0th, the 2nd and so forth).

Note that split will never split a string on an empty pattern match. For example:

**>>>** re.split('x\*', 'foo')

['foo']

**>>>** re.split("(?m)^$", "foo**\n\n**bar**\n**")

['foo\n\nbar\n']

*Changed in version 2.7:*Added the optional flags argument.

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

Return all non-overlapping matches of *pattern* in *string*, as a list of strings. The *string* is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result.

**Note**

Due to the limitation of the current implementation the character following an empty match is not included in a next match, so findall(r'^|\w+', 'two words') returns ['', 'wo', 'words'] (note missed “t”). This is changed in Python 3.7.

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

Return an [iterator](https://docs.python.org/2/glossary.html#term-iterator) yielding **[MatchObject](https://docs.python.org/2/library/re.html" \l "re.MatchObject" \o "re.MatchObject)** instances 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. See also the note about **[findall()](https://docs.python.org/2/library/re.html" \l "re.findall" \o "re.findall)**.

*New in version 2.2.*

*Changed in version 2.4:*Added the optional flags argument.

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

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn’t found, *string* is returned unchanged. *repl* can be a string or a function; if it 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 group 6 in the pattern. For example:

**>>>** 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 object 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 an RE object.

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. Empty matches for the pattern are replaced only when not adjacent to a previous match, so sub('x\*', '-', 'abc') returns '-a-b-c-'.

In string-type repl arguments, in addition to the character escapes and backreferences described above, \g<name> will use the substring matched by the group named name, as defined by the (?P<name>...) syntax. \g<number> uses the corresponding group number; \g<2> is therefore equivalent to \2, but isn’t ambiguous in a replacement such as \g<2>0. \20 would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character '0'. The backreference \g<0> substitutes in the entire substring matched by the RE.

*Changed in version 2.7:*Added the optional flags argument.

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

Perform the same operation as [**sub()**](https://docs.python.org/2/library/re.html#re.sub), but return a tuple (new\_string, number\_of\_subs\_made).

*Changed in version 2.7:*Added the optional flags argument.

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

Escape all the characters in *pattern* except ASCII letters and numbers. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:

**>>>** print re.escape('python.exe')

python\.exe

**>>>** legal\_chars = string.ascii\_lowercase + string.digits + "!#$%&'\*+-.^\_`|~:"

**>>>** print '[*%s*]+' % re.escape(legal\_chars)

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

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

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

re.**purge**()

Clear the regular expression cache.

*exception*re.**error**

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.

Regular Expression Objects

*class*re.**RegexObject**

The **[RegexObject](https://docs.python.org/2/library/re.html" \l "re.RegexObject" \o "re.RegexObject)** class supports the following methods and attributes:

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

Scan through *string* looking for a location where this regular expression produces a match, and return a corresponding **[MatchObject](https://docs.python.org/2/library/re.html" \l "re.MatchObject" \o "re.MatchObject)** instance. 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 optional second parameter pos gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string; the '^' pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter endpos limits how far the string will be searched; it will be as if the string is endpos characters long, so only the characters from pos to endpos - 1 will be searched for a match. If endpos is less than pos, no match will be found, otherwise, if rx is a compiled regular expression object, rx.search(string, 0, 50) is equivalent to rx.search(string[:50], 0).

**>>>** pattern = re.compile("d")

**>>>** pattern = re.compile("d")

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

<\_sre.SRE\_Match object at ...>

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

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

If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding **[MatchObject](https://docs.python.org/2/library/re.html" \l "re.MatchObject" \o "re.MatchObject)** instance. Return None if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the [**search()**](https://docs.python.org/2/library/re.html#re.RegexObject.search) method.

**>>>** 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".*

<\_sre.SRE\_Match object at ...>

If you want to locate a match anywhere in string, use [**search()**](https://docs.python.org/2/library/re.html#re.RegexObject.search) instead (see also [search() vs. match()](https://docs.python.org/2/library/re.html#search-vs-match)).

**split**(*string*, *maxsplit=0*)

Identical to the [**split()**](https://docs.python.org/2/library/re.html#re.split) function, using the compiled pattern.

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

Similar to the **[findall()](https://docs.python.org/2/library/re.html" \l "re.findall" \o "re.findall)** function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for [**match()**](https://docs.python.org/2/library/re.html#re.match).

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

Similar to the **[finditer()](https://docs.python.org/2/library/re.html" \l "re.finditer" \o "re.finditer)** function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for [**match()**](https://docs.python.org/2/library/re.html#re.match).

**sub**(*repl*, *string*, *count=0*)

Identical to the [**sub()**](https://docs.python.org/2/library/re.html#re.sub) function, using the compiled pattern.

**subn**(*repl*, *string*, *count=0*)

Identical to the **[subn()](https://docs.python.org/2/library/re.html" \l "re.subn" \o "re.subn)** function, using the compiled pattern.

**flags**

The regex matching flags. This is a combination of the flags given to [**compile()**](https://docs.python.org/2/library/re.html#re.compile) and any (?...) inline flags in the pattern.

**groups**

The number of capturing groups in the pattern.

**groupindex**

A dictionary mapping any symbolic group names defined by (?P<id>) to group numbers. The dictionary is empty if no symbolic groups were used in the pattern.

**pattern**

The pattern string from which the RE object was compiled.

## Match Objects

*class*re.**MatchObject**

Match objects always have a boolean value of True. Since **match()** and **search()** return None when there is no match, you can test whether there was a match with a simple if statement:

match = re.search(pattern, string)

**if** match:

process(match)

Match objects support the following methods and attributes:

**expand**(*template*)

Return the string obtained by doing backslash substitution on the template string *template*, as done by the [**sub()**](https://docs.python.org/2/library/re.html#re.RegexObject.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.

**group**([*group1*, *...*])

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, *group1* defaults to zero (the whole match is returned). If a *groupN* argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an **[IndexError](https://docs.python.org/2/library/exceptions.html" \l "exceptions.IndexError" \o "exceptions.IndexError)** exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is None. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

**>>>** 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')

If the regular expression uses the (?P<name>...) syntax, the groupN arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an **[IndexError](https://docs.python.org/2/library/exceptions.html" \l "exceptions.IndexError" \o "exceptions.IndexError)** exception is raised.

A moderately complicated example:

**>>>** m = re.match(r"(?P<first\_name>\w+) (?P<last\_name>\w+)", "Malcolm Reynolds")

**>>>** m.group('first\_name')

'Malcolm'

**>>>** m.group('last\_name')

'Reynolds'

Named groups can also be referred to by their index:

**>>>** m.group(1)

'Malcolm'

**>>>** m.group(2)

'Reynolds'

If a group matches multiple times, only the last match is accessible:

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

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

'c3'

**groups**([*default*])

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The *default* argument is used for groups that did not participate in the match; it defaults to None. (Incompatibility note: in the original Python 1.5 release, if the tuple was one element long, a string would be returned instead. In later versions (from 1.5.1 on), a singleton tuple is returned in such cases.)

For example:

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

**>>>** m.groups()

('24', '1632')

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to None unless the default argument is given:

**>>>** 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'*

**groupdict**([*default*])

Return a dictionary containing all the *named* subgroups of the match, keyed by the subgroup name. The *default* argument is used for groups that did not participate in the match; it defaults to None. For example:

**>>>** m = re.match(r"(?P<first\_name>\w+) (?P<last\_name>\w+)", "Malcolm Reynolds")

**>>>** m.groupdict()

{'first\_name': 'Malcolm', 'last\_name': 'Reynolds'}

**start**([*group*])

**end**([*group*])

Return the indices of the start and end of the substring matched by *group*; *group* defaults to zero (meaning the whole matched substring). Return -1 if *group* exists but did not contribute to the match. For a match object *m*, and a group *g* that did contribute to the match, the substring matched by group *g* (equivalent to m.group(g)) is

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

Note that m.start(group) will equal m.end(group) if *group* matched a null string. For example, after m = re.search('b(c?)', 'cba'), m.start(0) is 1, m.end(0) is 2, m.start(1) and m.end(1) are both 2, and m.start(2) raises an **[IndexError](https://docs.python.org/2/library/exceptions.html" \l "exceptions.IndexError" \o "exceptions.IndexError)** exception.

An example that will remove *remove\_this* from email addresses:

**>>>** email = "tony@tiremove\_thisger.net"

**>>>** m = re.search("remove\_this", email)

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

'tony@tiger.net'

**span**([*group*])

For **[MatchObject](https://docs.python.org/2/library/re.html" \l "re.MatchObject" \o "re.MatchObject)** *m*, return the 2-tuple (m.start(group), m.end(group)). Note that if *group* did not contribute to the match, this is (-1, -1). *group* defaults to zero, the entire match.

**pos**

The value of *pos* which was passed to the [**search()**](https://docs.python.org/2/library/re.html#re.RegexObject.search) or [**match()**](https://docs.python.org/2/library/re.html#re.RegexObject.match) method of the **[RegexObject](https://docs.python.org/2/library/re.html" \l "re.RegexObject" \o "re.RegexObject)**. This is the index into the string at which the RE engine started looking for a match.

**endpos**

The value of *endpos* which was passed to the [**search()**](https://docs.python.org/2/library/re.html#re.RegexObject.search) or [**match()**](https://docs.python.org/2/library/re.html#re.RegexObject.match) method of the **[RegexObject](https://docs.python.org/2/library/re.html" \l "re.RegexObject" \o "re.RegexObject)**. This is the index into the string beyond which the RE engine will not go.

**lastindex**

The integer index of the last matched capturing group, or None if no group was matched at all. For example, the expressions (a)b, ((a)(b)), and ((ab)) will have lastindex == 1 if applied to the string 'ab', while the expression (a)(b) will have lastindex == 2, if applied to the same string.

**lastgroup**

The name of the last matched capturing group, or None if the group didn’t have a name, or if no group was matched at all.

**re**

The regular expression object whose [**match()**](https://docs.python.org/2/library/re.html#re.RegexObject.match) or [**search()**](https://docs.python.org/2/library/re.html#re.RegexObject.search) method produced this **[MatchObject](https://docs.python.org/2/library/re.html" \l "re.MatchObject" \o "re.MatchObject)** instance.

**string**

The string passed to [**match()**](https://docs.python.org/2/library/re.html#re.RegexObject.match) or [**search()**](https://docs.python.org/2/library/re.html#re.RegexObject.search).