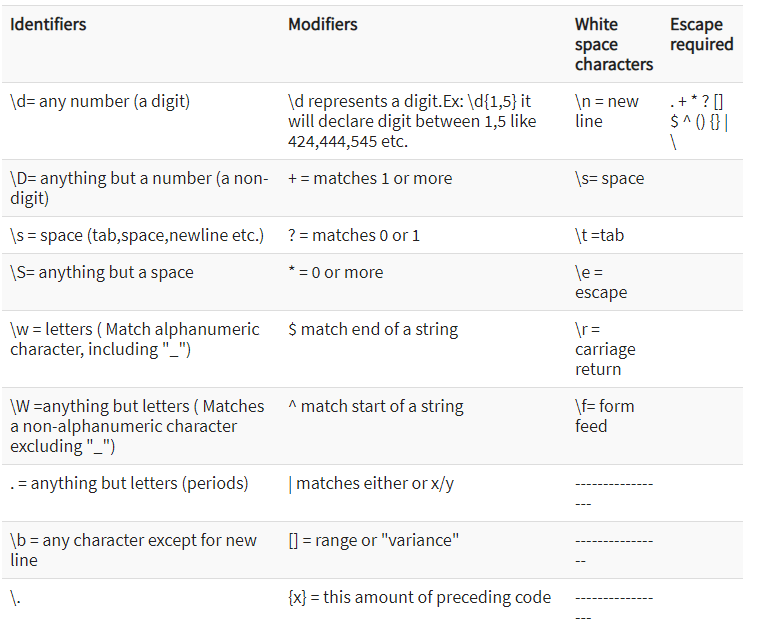
**What is Regular Expression**

A regular expression in a programming language is a special text string used for describing a search pattern. It is extremely useful for extracting information from text such as code, files, log, spreadsheets or even documents.

For instance, a regular expression could tell a program to search for specific text from the string and then to print out the result accordingly. Expression can include

* Text matching
* Repetition
* Branching
* Pattern-composition etc.

In Python, a regular expression is denoted as RE (REs, regexes or regex pattern) are imported through **re module**. Python supports regular expression through libraries. In Python regular expression supports various things like **Modifiers, Identifiers, and White space characters**.



## w+ and ^ Expression

* **"^":**This expression matches the start of a string
* **"w+**": This expression matches the alphanumeric character in the string

import re

xx = "guru99,education is fun"

r1 = re.findall(r"^\w+",xx)

print(r1)

## \s expression in re.split function

import re

xx = "guru99,education is fun"

r1 = re.findall(r"^\w+", xx)

print((re.split(r'\s','we are splitting the words')))

print((re.split(r's','split the words')))

Similarly, there are series of other regular expressions in Python that you can use in various ways in Python like \d,\D,$,\.,\b, etc.

## Using regular expression methods

* re.match()
* re.search()
* re.findall()

## Using re.match()

In this method, the expression "w+" and "\W" will match the words starting with letter 'g' and thereafter, anything which is not started with 'g' is not identified.

import re

list = ["guru99 get", "guru99 give", "guru gel@enium"]

for element in list:

    z = re.match("(g\w+)\W(g\w+)", element)

    if z:

        print((z.groups()))

## Finding Pattern in Text (re.search())

The search() function takes the "pattern" and "text" to scan from our main string and returns a match object when the pattern is found or else not match if the pattern is not found.

import re

patterns = ['software testing', 'guru99']

text = 'software testing is fun?'

for pattern in patterns:

    print('Looking for "%s" in "%s" ->' % (pattern, text), end=' ')

    if re.search(pattern, text):

        print('found a match!')

else:

    print('no match')

## Using re.findall for text

## Re.findall() module is used when you want to iterate over the lines of the file, it will return a list of all the matches in a single step. For example, here we have a list of e-mail addresses, and we want all the e-mail addresses to be fetched out from the list, we use the re.findall method. It will find all the e-mail addresses from the list.

import re

abc = 'guru99@google.com, careerguru99@hotmail.com, users@yahoomail.com'

emails = re.findall(r'[\w\.-]+@[\w\.-]+', abc)

for email in emails:

    print(email)

The special characters are:

.

(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.

import re

xx = "uru99,education is fun"

r1 = re.search(r".", xx)

print(r1.group())

^

(Caret.) Matches the start of the string, and in 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 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 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] 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.

Changed in version 3.7: FutureWarning is raised if a character set contains constructs that will change semantically in the future.

|

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.

New in version 3.6.

Changed in version 3.7: The letters 'a', 'L' and 'u' also can be used in a 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.

(?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',

>>>

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'

Changed in version 3.5: Added support for group references of fixed length.

(?<!...)

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 which are neither alphanumeric in the current locale nor 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

\N      \r      \t      \u

\U      \v      \x      \\

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

'\u', '\U', and '\N' 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.

Changed in version 3.3: The '\u' and '\U' escape sequences have been added.

Changed in version 3.6: Unknown escapes consisting of '\' and an ASCII letter now are errors.

Changed in 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}').

## **Module Contents**

re.**compile**(pattern, flags=0)

Compile a regular expression pattern into a [regular expression object](https://docs.python.org/3/library/re.html#re-objects), which can be used for matching using its [match()](https://docs.python.org/3/library/re.html#re.Pattern.match), [search()](https://docs.python.org/3/library/re.html#re.Pattern.search) and other methods, described below.

prog = re.compile(pattern)

result = prog.match(string)

re.**DOTALL**

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

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 object](https://docs.python.org/3/library/re.html#match-objects).

import re

abc = "abtrolkkd,astabcrmmr,astronomer"

emails = re.search('(ab)+', abc)

print(emails.groups())

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 object](https://docs.python.org/3/library/re.html#match-objects)

import re

abc = "abtrolkkd,astabcrmmr,astronomer"

emails = re.match('(ab)+', abc)

print(emails.groups())

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

if the whole string matches the regular expression pattern, return a corresponding [match object](https://docs.python.org/3/library/re.html#match-objects).

import re

abc = "ab"

emails = re.fullmatch('(ab)+', abc)

print(emails.groups())

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.

import re

abc = "cabcdab"

emails = re.split('(ab)+', abc)

print(emails)

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.

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

Return an [iterator](https://docs.python.org/3/glossary.html#term-iterator) yielding [match objects](https://docs.python.org/3/library/re.html#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.

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.

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

Perform the same operation as [sub()](https://docs.python.org/3/library/re.html#re.sub), but return a tuple (new\_string, number\_of\_subs\_made).

re.**escape**(pattern)

Escape special characters in pattern. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it

import re

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