011 Python Regular Expression

Regular Expression

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

The following table provides a list and description of the special pattern matching characters that can be used in regular expressions.

Special characters	**Description**
.	(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.
^	(Caret.) Matches the beginning of the string or line. For example /^A/ does not match the 'A' in "about Articles" but does match it in "Articles of life"
$	Matches the end of the string or line. For example, /e$/ does not match the 't' in "exact", but does match it in "example".
*	Matches the previous character 0 or more times. For example, /bo*/ matches 'boo' in "A bootable usb" and 'b' in "A beautiful mind", but nothing in "A going concern".
+	Matches the previous character 1 or more times. For example, /a+/ matches the 'a' in "Daniel" and all the a's in "Daaam"
?	Matches the previous character 0 or 1 time. For example, /r?eu?/ matches the 're' in "example" and the 'eu' in "europe".
*?, +?, ??	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, b{5} will match exactly five 'b' characters, but not four.
{m,n}	Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE. For example, a{2,5} will match from 2 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'.
{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.
[]	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.
|	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.
(...)	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.
(?:...)	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.
(?imsx-imsx:...)	(Zero or more letters from the set 'i', 'm', 's', 'x', optionally followed by '-' followed by one or more letters from the same set.) The letters set or removes the corresponding flags: re.I (ignore case), re.M (multi-line), re.S (dot matches all), and re.X (verbose), for the part of the expression.
(?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.

<table>
<tr><td></td><td>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):<table><tr><th>Context of reference to group “quote”</th><th>Ways to reference it</th></tr><tr><td>in the same pattern itself</td><td>(?P=quote) (as shown)
\1</td></tr><tr><td>when processing match object m</td><td>m.group('quote')
m.end('quote') (etc.)</td></tr><tr><td>in a string passed to the repl argument of re.sub()</td><td>\g<quote>
\g<1>
\1</td></tr></table></td></tr>
<tr><td>(?P=name)</td><td>A backreference to a named group; it matches whatever text was matched by the earlier group named name.</td></tr>
<tr><td>(?#...)</td><td>A comment; the contents of the parentheses are simply ignored.</td></tr>
<tr><td>(?=...)</td><td>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'.</td></tr>
<tr><td>(?!...)</td><td>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'.</td></tr>
<tr><td>(?<=...)</td><td>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.</td></tr>
</table>

	`>>> 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.
\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	Matches any character which is a digit. Equivalent to [0-9]. For example, /\d/ or /[0-9]/ matches '2' in "E2 means second example."

\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] (but the flag affects the entire regular expression, so in such cases using an explicit [^0-9] may be a better choice).
\s	Matches any white space character (including tab, new line, carriage return, form feed, vertical tab). [\t\n\r\f\v]. For example, /\s\w*/ matches ' apple' in "An apple." 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] (but the flag affects the entire regular expression, so in such cases using an explicit [^ \t\n\r\f\v] may be a better choice).
\w	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 non-word character, equivalent to [^A-Za-z0-9_]. For example, /\W/ or /[^$A-Za-z0-9_]/ matches '$' in "150$"
\Z	Matches only at the end of the string.

Compare HTML tags :

tag type	format	example
open tag	<[^/>][^>]*>	<a>, <table>
close tag	</[^>]+>	</p>,
self close	<[^/>]+/>	

all tag	<[^>]+>	 , <a>

```
# open tag
>>> import re
>>> re.search('<[^/>][^>]*>', '<table>') != None
True
>>> import re
>>> re.search('<[^/>][^>]*>', '<a href="#label">') != None
True
>>> import re
>>> re.search('<[^/>][^>]*>', '<img src="/img">') != None
True
>>> import re
>>> re.search('<[^/>][^>]*>', '</table>') != None
False


# close tag
>>> import re
>>> re.search('</[^>]+>', '</table>') != None
True


# self close
>>> import re
>>> re.search('<[^/>]+/>', '<br />') != None
True
```

re.findall() match string :

```
# split all string
>>> import re
>>> source = "split all string"
>>> re.findall('[\w]+', source)
['split', 'all', 'string']


# parsing python.org website
>>> import urllib
>>> import re
>>> x = urllib.urlopen('https://www.example.org')
>>> html = x.read()
>>> x.close()
>>> print("open tags")
open tags
>>> re.findall('<[^/>][^>]*>', html)[0:2]
['<!doctype html>', '<!-[if lt IE 7]>']
>>> print("close tags")
close tags
>>> re.findall('</[^>]+>', html)[0:2]
['</script>', '</title>']
>>> print("self-closing tags")
```

Group Comparison :

```
# (...) group a regular expression
>>> import re
>>> mon = re.search(r'(\d{4})-(\d{2})-(\d{2})', '2018-09-01')
>>> mon
<_sre.SRE_Match object at 0x019A72F0>
>>> mon.groups()
('2018', '09', '01')
>>> mon.group()
'2018-09-01')
>>> mon.group(1)
'2018'
>>> mon.group(2)
'09'
>>> mon.group(3)
'01'


# Nesting groups
>>> import re
>>> mon = re.search(r'(((\d{4})-\d{2})-\d{2})', '2018-09-01')
>>> mon.groups()
('2018-09-01', '2018-09', '2018')
>>> mon.group()
'2018-09-01'
>>> mon.group(1)
'2018-09-01'
```

```
>>> mon.group(2)
'2018-09'
>>> mon.group(3)
'2018'
```

Non capturing group :

```
# non capturing group
>>> import re
>>> url = 'http://example.com/'
>>> mon = re.search('(?:http|ftp)://([^/\r\n]+)(/[^\r\n]*)?', url)
>>> mon.groups()
('example.com', '/')


# capturing group
>>> import re
>>> mon = re.search('(http|ftp)://([^/\r\n]+)(/[^\r\n]*)?', url)
>>> mon.groups()
('http', 'example.com', '/')
```

Back Reference :

```
# compare 'xx', 'yy'
>>> import re
>>> re.search(r'([a-z])\1$','xx') != None
True
>>> import re
```

```
>>> re.search(r'([a-z])\1$','yy') != None
True
>>> import re
>>> re.search(r'([a-z])\1$','xy') != None
False


# compare open tag and close tag
>>> import re
>>> pattern = r'<([^>]+)>[\s\S]*?</\1>'
>>> re.search(pattern, '<bold> test </bold>') != None
True
>>> re.search(pattern, '<h1> test </h1>') != None
True
>>> re.search(pattern, '<bold> test </h1>') != None
False
```

Named Grouping (?P<name>) :

```
# group reference ``(?P<name>...)``
>>> import re
>>> pattern = '(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})'
>>> mon = re.search(pattern, '2018-09-01')
>>> mon.group('year')
'2018'
>>> mon.group('month')
'09'
```

```
>>> mon.group('day')
'01'


# back reference ``(?P=name)``
>>> import re
>>> re.search('^(?P<char>[a-z])(?P=char)','aa')
<_sre.SRE_Match object at 0x01A08660>
```

Substitute String :

```
# basic substitute
>>> import re
>>> res = "4x5y6z"
>>> re.sub(r'[a-z]',' ', res)
'4 5 6 '


# substitute with group reference
>>> import re
>>> date = r'2018-09-01'
>>> re.sub(r'(\d{4})-(\d{2})-(\d{2})',r'\2/\3/\1/',date)
'09/01/2018/'


# camelcase to underscore
>>> def convert(s):
...     res = re.sub(r'(.)([A-Z][a-z]+)',r'\1_\2', s)
...     return re.sub(r'([a-z])([A-Z])',r'\1_\2', res).lower()
```

```
...
>>> convert('SentenceCase')
'sentence_case'
>>> convert('SentenceSentenceCase')
'sentence_sentence_case'
>>> convert('SampleExampleHTTPServer')
'sample_example_http_server'
```

Look around :

notation	compare direction
(?<=...)	right to left
(?=...)	left to right
(?!<...)	right to left
(?!...)	left to right

```
# basic
>>> import re
>>> re.sub('(?=\d{3})', ' ', '56789')
' 5 6 789'
>>> re.sub('(?!\d{3})', ' ', '56789')
'567 8 9 '
>>> re.sub('(?<=\d{3})', ' ', '56789')
```

```
'567 8 9 '
>>> re.sub('(?<!\d{3})', ' ', '56789')
' 5 6 789'
```

Match common username or password :

```
>>> import re
>>> re.match('^[a-zA-Z0-9-_]{3,16}$', 'Foo') is not None
True
>>> re.match('^\w|[-_]{3,16}$', 'Foo') is not None
True
```

Match hex color value :

```
>>> import re
>>> re.match('^#?([a-f0-9]{6}|[a-f0-9]{3})$', '#ff0000')
<_sre.SRE_Match object at 0x019E7720>
>>> re.match('^#?([a-f0-9]{6}|[a-f0-9]{3})$', '#000000')
<_sre.SRE_Match object at 0x019E77A0>
```

Match email :

```
>>> import re
>>> re.match('^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$',
        'citi.world@example.com')
<_sre.SRE_Match object; span=(0, 22), match='citi.world@example.com'>

# or
```

```
>>> import re

>>> example = re.compile(r'''^([a-zA-Z0-9._%-]+@

[a-zA-Z0-9.-]+

\.[a-zA-Z]{2,4})*$''', re.X)

>>> example.match('citi.world@example.citi.com')

<_sre.SRE_Match object; span=(0, 27),
match='citi.world@example.citi.com'>

>>> example.match('citi%world@example.citi.com')

<_sre.SRE_Match object; span=(0, 27),
match='citi%world@example.citi.com'>
```

Match URL :

```
>>> import re

>>> example = re.compile(r'''^(https?:\/\/)? # match http or https

...             ([\da-z\.-]+)           # match domain

...             \.([a-z\.]{2,6})        # match domain

...             ([\/\w \.-]*)\/?$       # match api or file

...             ''', re.X)

>>> example.match('www.yahoo.com')

<_sre.SRE_Match object; span=(0, 13), match='www.yahoo.com'>

>>> example.match('http://www.example')

<_sre.SRE_Match object; span=(0, 18), match='http://www.example'>

>>> example.match('http://www.example/w3r.html')

<_sre.SRE_Match object; span=(0, 27),
match='http://www.example/w3r.html'>

>>> example.match('http://www.example/w3r!.html')

>>> example
```

```
re.compile('^(https?:\\/\\/)?\n([\\da-z\\.-]+)\n\\.([a-
z\\.]{2,6})\n([\\/\\w \\.-]*)\\/?$\n', re.VERBOSE)
```

Match IP address :

notation	description
[1]?[0-9][0-9]	Match 0-199 pattern
2[0-4][0-9]	Match 200-249 pattern
25[0-5]	Match 251-255 pattern
(?:...)	Don't capture group

```
>>> import re
>>> example = re.compile(r'''^(?:(?:25[0-5]
... |2[0-4][0-9]
... |[1]?[0-9][0-9]?)\.){3}
... (?:25[0-5]
... |2[0-4][0-9]
... |[1]?[0-9][0-9]?)$''', re.X)
>>> example.match('192.168.1.1')
<_sre.SRE_Match object at 0x0134A608>
>>> example.match('255.255.255.0')
<_sre.SRE_Match object at 0x01938678>
>>> example.match('172.17.0.5')
<_sre.SRE_Match object at 0x0134A608>
```

```
>>> example.match('256.0.0.0') is None
True
```

Match Mac address :

```
>>> import random
>>> mac = [random.randint(0x00, 0x6b),
... random.randint(0x00, 0x6b),
... random.randint(0x00, 0x6b),
... random.randint(0x00, 0x6b),
... random.randint(0x00, 0x6b),
... random.randint(0x00, 0x6b)]
>>> mac = ':'.join(map(lambda x: "%02x" % x, mac))
>>> mac
'05:38:64:60:55:63'
>>> import re
>>> example = re.compile(r'''[0-9a-f]{2}([:])
... [0-9a-f]{2}
... (\1[0-9a-f]{2}){4}$''', re.X)
>>> example.match(mac) is not None
True
```

Lexer :

```
>>> import re
>>> from collections import namedtuple
>>> tokens = [r'(?P<NUMBER>\d+)',
```

```
          r'(?P<PLUS>\+)',
          r'(?P<MINUS>-)',
          r'(?P<TIMES>\*)',
          r'(?P<DIVIDE>/)',
          r'(?P<WS>\s+)']
>>> lex = re.compile('|'.join(tokens))
>>> Token = namedtuple('Token', ['type', 'value'])
>>> def tokenize(text):
        scan = lex.scanner(text)
        return (Token(m.lastgroup, m.group())
                for m in iter(scan.match, None) if m.lastgroup != 'WS')

>>> for _t in tokenize('9 + 5 * 2 - 7'):
        print(_t)


Token(type='NUMBER', value='9')
Token(type='PLUS', value='+')
Token(type='NUMBER', value='5')
Token(type='TIMES', value='*')
Token(type='NUMBER', value='2')
Token(type='MINUS', value='-')
Token(type='NUMBER', value='7')
>>> tokens
```

```
['(?P<NUMBER>\\d+)', '(?P<PLUS>\\+)', '(?P<MINUS>-)',
'(?P<TIMES>\\*)', '(?P<DIVIDE>/)', '(?P<WS>\\s+)']
```