

Python - Regular Expressions

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The Python module **re** provides full support for Perl-like regular expressions in Python. The **re** module raises the exception **re.error** if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.

The *match* Function

This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function –

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

Here is the description of the parameters –

Sr.No.	Parameter & Description
1	pattern This is the regular expression to be matched.
2	string This is the string, which would be searched to match the pattern at the beginning of string.
3	flags You can specify different flags using bitwise OR (). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Method & Description
1	group(num=0) This method returns entire match (or specific subgroup num)
2	groups() This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

Live Demo

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
    print "matchObj.group() : ", matchObj.group()
    print "matchObj.group(1) : ", matchObj.group(1)
    print "matchObj.group(2) : ", matchObj.group(2)
else:
    print "No match!!"
```

When the above code is executed, it produces following result –

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

The *search* Function

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function –

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

Here is the description of the parameters –

Sr.No.	Parameter & Description
1	pattern This is the regular expression to be matched.
2	string This is the string, which would be searched to match the pattern anywhere in the string.
3	flags You can specify different flags using bitwise OR (). These are modifiers, which are listed in the table below.

The `re.search` function returns a **match** object on success, **none** on failure. We use `group(num)` or `groups()` function of **match** object to get matched expression.

Sr.No.	Match Object Methods & Description
1	group(num=0) This method returns entire match (or specific subgroup num)
2	groups() This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

searchObj = re.search( r'(.*) are (.*) .*', line, re.M|re.I)

if searchObj:
    print "searchObj.group() : ", searchObj.group()
    print "searchObj.group(1) : ", searchObj.group(1)
    print "searchObj.group(2) : ", searchObj.group(2)
else:
    print "Nothing found!!"
```

Live Demo

When the above code is executed, it produces following result –

```
searchObj.group() : Cats are smarter than dogs
searchObj.group(1) : Cats
searchObj.group(2) : smarter
```

Matching Versus Searching

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search** checks for a match anywhere in the string (this is what Perl does by default).

Example

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
    print "match --> matchObj.group() : ", matchObj.group()
else:
    print "No match!!"

searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
    print "search --> searchObj.group() : ", searchObj.group()
else:
    print "Nothing found!!"
```

Live Demo

When the above code is executed, it produces the following result –

```
No match!!
search --> searchObj.group() : dogs
```

Search and Replace

One of the most important **re** methods that use regular expressions is **sub**.

Syntax

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method returns modified string.

Example

[Live Demo](#)

```
#!/usr/bin/python
import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

When the above code is executed, it produces the following result –

```
Phone Num : 2004-959-559
Phone Num : 2004959559
```

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these –

Sr.No.	Modifier & Description
1	re.I Performs case-insensitive matching.
2	re.L Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).
3	re.M Makes \$ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
4	re.S Makes a period (dot) match any character, including a newline.
5	re.U Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
6	re.X Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular Expression Patterns

Except for control characters, (+ ? . * ^ \$ () [] { } | \), all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python –

Sr.No.	Pattern & Description
1	^ Matches beginning of line.
2	\$ Matches end of line.
3	. Matches any single character except newline. Using m option allows it to match newline as well.
4	[...] Matches any single character in brackets.
5	[^...] Matches any single character not in brackets
6	re* Matches 0 or more occurrences of preceding expression.
7	re+ Matches 1 or more occurrence of preceding expression.
8	re? Matches 0 or 1 occurrence of preceding expression.
9	re{ n} Matches exactly n number of occurrences of preceding expression.
10	re{ n,} Matches n or more occurrences of preceding expression.
11	re{ n, m} Matches at least n and at most m occurrences of preceding expression.

12	a b Matches either a or b.
13	(re) Groups regular expressions and remembers matched text.
14	(?imx) Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
15	(?-imx) Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
16	(?: re) Groups regular expressions without remembering matched text.
17	(?imx: re) Temporarily toggles on i, m, or x options within parentheses.
18	(?-imx: re) Temporarily toggles off i, m, or x options within parentheses.
19	(?#...) Comment.
20	(?= re) Specifies position using a pattern. Doesn't have a range.
21	(?! re) Specifies position using pattern negation. Doesn't have a range.
22	(?> re) Matches independent pattern without backtracking.

23	\w Matches word characters.
24	\W Matches nonword characters.
25	\s Matches whitespace. Equivalent to <code>[\t\n\r\f]</code> .
26	\S Matches nonwhitespace.
27	\d Matches digits. Equivalent to <code>[0-9]</code> .
28	\D Matches nondigits.
29	\A Matches beginning of string.
30	\Z Matches end of string. If a newline exists, it matches just before newline.
31	\z Matches end of string.
32	\G Matches point where last match finished.
33	\b Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
34	

	\B Matches nonword boundaries.
35	\n, \t, etc. Matches newlines, carriage returns, tabs, etc.
36	\1...\9 Matches nth grouped subexpression.
37	\10 Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Regular Expression Examples

Literal characters

Sr.No.	Example & Description
1	python Match "python".

Character classes

Sr.No.	Example & Description
1	[Pp]ython Match "Python" or "python"
2	rub[ye] Match "ruby" or "rube"
3	[aeiou] Match any one lowercase vowel
4	[0-9] Match any digit; same as [0123456789]
5	[a-z] Match any lowercase ASCII letter
6	[A-Z] Match any uppercase ASCII letter
7	[a-zA-Z0-9] Match any of the above
8	[^aeiou] Match anything other than a lowercase vowel
9	[^0-9] Match anything other than a digit

Special Character Classes

Sr.No.	Example & Description
1	. Match any character except newline
2	\d Match a digit: [0-9]
3	\D Match a nondigit: [^0-9]
4	\s Match a whitespace character: [\t\r\n\f]
5	\S Match nonwhitespace: [^ \t\r\n\f]
6	\w Match a single word character: [A-Za-z0-9_]
7	\W Match a nonword character: [^A-Za-z0-9_]

Repetition Cases

Sr.No.	Example & Description
1	ruby? Match "rub" or "ruby": the y is optional
2	ruby* Match "rub" plus 0 or more ys
3	ruby+ Match "rub" plus 1 or more ys
4	\d{3} Match exactly 3 digits
5	\d{3,} Match 3 or more digits
6	\d{3,5} Match 3, 4, or 5 digits

Nongreedy repetition

This matches the smallest number of repetitions –

Sr.No.	Example & Description
1	<.*> Greedy repetition: matches "<python>perl>"
2	<.*?> Nongreedy: matches "<python>" in "<python>perl>"

Grouping with Parentheses

Sr.No.	Example & Description
1	\D\d+ No group: + repeats \d
2	(\D\d)+ Grouped: + repeats \D\d pair
3	([Pp]ython(,)?)+ Match "Python", "Python, python, python", etc.

Backreferences

This matches a previously matched group again –

Sr.No.	Example & Description
1	([Pp])ython&\1ails Match python&pails or Python&Pails
2	(["'])[^1]*\1 Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.

Alternatives

Sr.No.	Example & Description
1	python perl Match "python" or "perl"
2	rub(y le) Match "ruby" or "ruble"
3	Python(!+ \?) "Python" followed by one or more ! or one ?

Anchors

This needs to specify match position.

Sr.No.	Example & Description
1	^Python Match "Python" at the start of a string or internal line
2	Python\$ Match "Python" at the end of a string or line
3	\APython Match "Python" at the start of a string
4	Python\Z Match "Python" at the end of a string
5	\bPython\b Match "Python" at a word boundary
6	\brub\b \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
7	Python(?!) Match "Python", if followed by an exclamation point.
8	Python(?!) Match "Python", if not followed by an exclamation point.

Special Syntax with Parentheses

Sr.No.	Example & Description
1	R(?#comment) Matches "R". All the rest is a comment
2	R(?i)uby Case-insensitive while matching "uby"
3	R(?i:uby) Same as above
4	rub(?:y le)) Group only without creating \1 backreference