**Python RegEx**

In this tutorial, you will learn about regular expressions (RegEx), and use Python's re module to work with RegEx (with the help of examples).

A **Reg**ular **Ex**pression (RegEx) is a sequence of characters that defines a search pattern. For example,

^a...s$

The above code defines a RegEx pattern. The pattern is: **any five letter string starting with a and ending with s**.

A pattern defined using RegEx can be used to match against a string.

| Expression | String | Matched? |
| --- | --- | --- |
| ^a...s$ | abs | No match |
| alias | Match |
| abyss | Match |
| Alias | No match |
| An abacus | No match |

Python has a module named re to work with RegEx. Here's an example:

import re

pattern = '^a...s$'

test\_string = 'abyss'

result = re.match(pattern, test\_string)

if result:

print("Search successful.")

else:

print("Search unsuccessful.")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we used re.match() function to search pattern within the test\_string. The method returns a match object if the search is successful. If not, it returns None.

**Specify Pattern Using RegEx**

To specify regular expressions, metacharacters are used. In the above example, ^ and $ are metacharacters.

**MetaCharacters**

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:

**[]** **.** **^** **$** **\*** **+** **?** **{}** **()** **\** **|**

**[] - Square brackets**

Square brackets specifies a set of characters you wish to match.

| Expression | String | Matched? |
| --- | --- | --- |
| [abc] | a | 1 match |
| ac | 2 matches |
| Hey Jude | No match |
| abc de ca | 5 matches |

Here, [abc] will match if the string you are trying to match contains any of the a, b or c.

You can also specify a range of characters using - inside square brackets.

* [a-e] is the same as [abcde].
* [1-4] is the same as [1234].
* [0-39] is the same as [01239].

You can complement (invert) the character set by using caret ^ symbol at the start of a square-bracket.

* [^abc] means any character except a or b or c.
* [^0-9] means any non-digit character.

. - **Period**

A period matches any single character (except newline '\n').

| Expression | String | Matched? |
| --- | --- | --- |
| .. | a | No match |
| ac | 1 match |
| acd | 1 match |
| acde | 2 matches (contains 4 characters) |

^ - **Caret**

The caret symbol ^ is used to check if a string **starts with** a certain character.

| Expression | String | Matched? |
| --- | --- | --- |
| ^a | a | 1 match |
| abc | 1 match |
| bac | No match |
| ^ab | abc | 1 match |
| acb | No match (starts with a but not followed by b) |

$ - **Dollar**

The dollar symbol $ is used to check if a string **ends with** a certain character.

| Expression | String | Matched? |
| --- | --- | --- |
| a$ | a | 1 match |
| formula | 1 match |
| cab | No match |

\* - **Star**

The star symbol \* matches **zero or more occurrences** of the pattern left to it.

| Expression | String | Matched? |
| --- | --- | --- |
| ma\*n | mn | 1 match |
| man | 1 match |
| maaan | 1 match |
| main | No match (a is not followed by n) |
| woman | 1 match |

+ - **Plus**

The plus symbol + matches **one or more occurrences** of the pattern left to it.

| Expression | String | Matched? |
| --- | --- | --- |
| ma+n | mn | No match (no a character) |
| man | 1 match |
| maaan | 1 match |
| main | No match (a is not followed by n) |
| woman | 1 match |

? - **Question Mark**

The question mark symbol ? matches **zero or one occurrence** of the pattern left to it.

| Expression | String | Matched? |
| --- | --- | --- |
| ma?n | mn | 1 match |
| man | 1 match |
| maaan | No match (more than one a character) |
| main | No match (a is not followed by n) |
| woman | 1 match |

{} - **Braces**

Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.

| Expression | String | Matched? |
| --- | --- | --- |
| a{2,3} | abc dat | No match |
| abc daat | 1 match (at daat) |
| aabc daaat | 2 matches (at aabc and daaat) |
| aabc daaaat | 2 matches (at aabc and daaaat) |

Let's try one more example. This RegEx [0-9]{2, 4} matches at least 2 digits but not more than 4 digits

| Expression | String | Matched? |
| --- | --- | --- |
| [0-9]{2,4} | ab123csde | 1 match (match at ab123csde) |
| 12 and 345673 | 3 matches (12, 3456, 73) |
| 1 and 2 | No match |

| - **Alternation**

Vertical bar | is used for alternation (or operator).

| Expression | String | Matched? |
| --- | --- | --- |
| a|b | cde | No match |
| ade | 1 match (match at ade) |
| acdbea | 3 matches (at acdbea) |

Here, a|b match any string that contains either a or b

() - **Group**

Parentheses () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz

| Expression | String | Matched? |
| --- | --- | --- |
| (a|b|c)xz | ab xz | No match |
| abxz | 1 match (match at abxz) |
| axz cabxz | 2 matches (at axzbc cabxz) |

\ - **Backslash**

Backlash \ is used to escape various characters including all metacharacters. For example,

\$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way.

If you are unsure if a character has special meaning or not, you can put \ in front of it. This makes sure the character is not treated in a special way.

**Special Sequences**

Special sequences make commonly used patterns easier to write. Here's a list of special sequences:

\A - Matches if the specified characters are at the start of a string.

| Expression | String | Matched? |
| --- | --- | --- |
| \Athe | the sun | Match |
| In the sun | No match |

\b - Matches if the specified characters are at the beginning or end of a word.

| Expression | String | Matched? |
| --- | --- | --- |
| \bfoo | football | Match |
| a football | Match |
| afootball | No match |
| foo\b | the foo | Match |
| the afoo test | Match |
| the afootest | No match |

\B - Opposite of \b. Matches if the specified characters are **not** at the beginning or end of a word.

| Expression | String | Matched? |
| --- | --- | --- |
| \Bfoo | football | No match |
| a football | No match |
| afootball | Match |
| foo\B | the foo | No match |
| the afoo test | No match |
| the afootest | Match |

\d - Matches any decimal digit. Equivalent to [0-9]

| Expression | String | Matched? |
| --- | --- | --- |
| \d | 12abc3 | 3 matches (at 12abc3) |
| Python | No match |

\D - Matches any non-decimal digit. Equivalent to [^0-9]

| Expression | String | Matched? |
| --- | --- | --- |
| \D | 1ab34"50 | 3 matches (at 1ab34"50) |
| 1345 | No match |

\s - Matches where a string contains any whitespace character. Equivalent to [ \t\n\r\f\v].

| Expression | String | Matched? |
| --- | --- | --- |
| \s | Python RegEx | 1 match |
| PythonRegEx | No match |

\S - Matches where a string contains any non-whitespace character. Equivalent to [^ \t\n\r\f\v].

| Expression | String | Matched? |
| --- | --- | --- |
| \S | a b | 2 matches (at a b) |
|  | No match |

\w - Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9\_]. By the way, underscore \_ is also considered an alphanumeric character.

| Expression | String | Matched? |
| --- | --- | --- |
| \w | 12&": ;c | 3 matches (at 12&": ;c) |
| %"> ! | No match |

\W - Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9\_]

| Expression | String | Matched? |
| --- | --- | --- |
| \W | 1a2%c | 1 match (at 1a2%c) |
| Python | No match |

\Z - Matches if the specified characters are at the end of a string.

| Expression | String | Matched? |
| --- | --- | --- |
| Python\Z | I like Python | 1 match |
| I like Python Programming | No match |
| Python is fun. | No match |

**Python RegEx**

Python has a module named re to work with regular expressions. To use it, we need to import the module.

import re

The module defines several functions and constants to work with RegEx.

**re.findall()**

The re.findall() method returns a list of strings containing all matches.

**Example 1: re.findall()**

# Program to extract numbers from a string

import re

string = 'hello 12 hi 89. Howdy 34'

pattern = '\d+'

result = re.findall(pattern, string)

print(result)

# Output: ['12', '89', '34']

If the pattern is not found, re.findall() returns an empty list.

**re.split()**

The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

**Example 2: re.split()**

import re

string = 'Twelve:12 Eighty nine:89.'

pattern = '\d+'

result = re.split(pattern, string)

print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']

[Run Code](https://www.programiz.com/python-programming/online-compiler)

If the pattern is not found, re.split() returns a list containing the original string.

You can pass maxsplit argument to the re.split() method. It's the maximum number of splits that will occur.

import re

string = 'Twelve:12 Eighty nine:89 Nine:9.'

pattern = '\d+'

# maxsplit = 1

# split only at the first occurrence

result = re.split(pattern, string, 1)

print(result)

# Output: ['Twelve:', ' Eighty nine:89 Nine:9.']

[Run Code](https://www.programiz.com/python-programming/online-compiler)

By the way, the default value of maxsplit is 0; meaning all possible splits.

**re.sub()**

The syntax of re.sub() is:

re.sub(pattern, replace, string)

The method returns a string where matched occurrences are replaced with the content of replace variable.

**Example 3: re.sub()**

# Program to remove all whitespaces

import re

# multiline string

string = 'abc 12\

de 23 \n f45 6'

# matches all whitespace characters

pattern = '\s+'

# empty string

replace = ''

new\_string = re.sub(pattern, replace, string)

print(new\_string)

# Output: abc12de23f456

[Run Code](https://www.programiz.com/python-programming/online-compiler)

If the pattern is not found, re.sub() returns the original string.

You can pass count as a fourth parameter to the re.sub() method. If omited, it results to 0. This will replace all occurrences.

import re

# multiline string

string = 'abc 12\

de 23 \n f45 6'

# matches all whitespace characters

pattern = '\s+'

replace = ''

new\_string = re.sub(r'\s+', replace, string, 1)

print(new\_string)

# Output:

# abc12de 23

# f45 6

**re.subn()**

The re.subn() is similar to re.sub() except it returns a tuple of 2 items containing the new string and the number of substitutions made.

**Example 4: re.subn()**

# Program to remove all whitespaces

import re

# multiline string

string = 'abc 12\

de 23 \n f45 6'

# matches all whitespace characters

pattern = '\s+'

# empty string

replace = ''

new\_string = re.subn(pattern, replace, string)

print(new\_string)

# Output: ('abc12de23f456', 4)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**re.search()**

The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

If the search is successful, re.search() returns a match object; if not, it returns None.

match = re.search(pattern, str)

**Example 5: re.search()**

import re

string = "Python is fun"

# check if 'Python' is at the beginning

match = re.search('\APython', string)

if match:

print("pattern found inside the string")

else:

print("pattern not found")

# Output: pattern found inside the string

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, match contains a match object.

**Match object**

You can get methods and attributes of a match object using [dir()](https://www.programiz.com/python-programming/methods/built-in/dir) function.

Some of the commonly used methods and attributes of match objects are:

**match.group()**

The group() method returns the part of the string where there is a match.

**Example 6: Match object**

import re

string = '39801 356, 2102 1111'

# Three digit number followed by space followed by two digit number

pattern = '(\d{3}) (\d{2})'

# match variable contains a Match object.

match = re.search(pattern, string)

if match:

print(match.group())

else:

print("pattern not found")

# Output: 801 35

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, match variable contains a match object.

Our pattern (\d{3}) (\d{2}) has two subgroups (\d{3}) and (\d{2}). You can get the part of the string of these parenthesized subgroups. Here's how:

>>> match.group(1)

'801'

>>> match.group(2)

'35'

>>> match.group(1, 2)

('801', '35')

>>> match.groups()

('801', '35')

**match.start(), match.end() and match.span()**

The start() function returns the index of the start of the matched substring. Similarly, end() returns the end index of the matched substring.

>>> match.start()

2

>>> match.end()

8

The span() function returns a tuple containing start and end index of the matched part.

>>> match.span()

(2, 8)

**match.re and match.string**

The re attribute of a matched object returns a regular expression object. Similarly, string attribute returns the passed string.

>>> match.re

re.compile('(\\d{3}) (\\d{2})')

>>> match.string

'39801 356, 2102 1111'

**Using r prefix before RegEx**

When r or R prefix is used before a regular expression, it means raw string. For example, '\n' is a new line whereas r'\n' means two characters: a backslash \ followed by n.

Backlash \ is used to escape various characters including all metacharacters. However, using r prefix makes \ treat as a normal character.

**Example 7: Raw string using r prefix**

import re

string = '\n and \r are escape sequences.'

result = re.findall(r'[\n\r]', string)

print(result)

# Output: ['\n', '\r']