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 **Re**gular **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`	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.")
```

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`.

If you already know the basics of RegEx, jump to Python RegEx.

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

`[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.

`'\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

`^ab`	`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

`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

`?` - 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

	`abc dat`	No match
	`abc daat`	1 match (at `daat`)
`a{2,3}`	`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?
	`ab123csde`	1 match (match at `ab123csde`)
`[0-9]{2,4}`	`12 and 345673`	3 matches (`12`, `3456`, `73`)
	`1 and 2`	No match

`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`)

`\` - Backslash

Backslash `\` 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

	`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`	`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

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_]`

`\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

Tip: To build and test regular expressions, you can use RegEx tester tools such as regex101 (https://regex101.com/). This tool not only helps you in creating regular expressions, but it also helps you learn it.

Now you understand the basics of RegEx, let's discuss how to use RegEx in your Python code.

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()

```
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()

```
result = re.split(pattern, string)
print(result)

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

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.']
```

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()

```
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
```

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.

```
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()` expect it returns a tuple of 2 items containing the new string and the number of substitutions made.

Example 4: re.subn()

```
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)
```

---

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)
```

```
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
```

Here, `match` contains a match object.

Match object

You can get methods and attributes of a match object using dir() (/python-programming/methods/built-in/dir) function.

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

there is a match.

Example 6: Match object

```
import re

string = '39801 356, 2102 1111'

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

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, 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)
```

expression object. Similarly, `string` attribute returns the passed string.

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

>>> match.string
'39801 356, 2102 1111'
```

We have covered all commonly used methods defined in the `re` module. If you want to learn more, visit Python 3 re module (https://docs.python.org/3/library/re.html).

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`.

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']
```

Next Tutorial:
Python Examples (/python-programming/examples)

Previous Tutorial:
Python Property (/python-programming/property)

Get App (https://www.programiz.com/learn-python?utm_campaign=programiz-homepage&utm_source=programiz-website-python-app-popup)

Related Tutorials

Python Tutorial

Python strptime()

(/python-programming/datetime/strptime)

Python Library

Python String split()

(/python-programming/methods/string/split)

Python Library

Python String title()

Get App (https://www.programiz.com/learn-python?utm_campaign=programiz-homepage&utm_source=programiz-website-python-app-popup)

(/python-programming/methods/string/strip)