Regular Expression (RegEx) in Python

Regular Expression (RegEx), also known as Regular Expression, is a segment of special characters that follow certain patterns.

The next chapter of the Advanced Python, TipsMake.com lesson will work with you to learn about **Regular Expression (RegEx)** with the module `re` with specific examples to make it easier for you to understand and grasp the knowledge of RegEx. Let's follow it!

Regular Expression (RegEx) , also known as a **regular expression,** is a segment of special characters that follow certain patterns, representing strings or a set of strings. For example:

```
^a.s$
```

The above code specifies the RegEx rule: any string with five letters, starting with `a` and ending with `s` .

Expression
Example string
Describe
`^a.s$` `abs` Not suitable for only 3 characters `alias` Match `abyss` Match `Alias` Inappropriate because the initial capital letters `An` `abacus` Not suitable because the first letter capitalizes `A` and more than 5 characters

Regular Expression in Python is expressed through the module, so the first thing when you want to use Regular Expression is to import the module into the program. Try with the example above:

```
import re pattern = '^a.s$' test_string = 'abyss' result = re.match(pattern, tes
```

Here we have just used the `re.match()` function to search for the *`test_string`* corresponding to the *`pattern`* . The method returns the corresponding object if the search is successful, returns `None` if not found.

Most languages ??support Regular Expression, including JavaScript, C #, Java, PHP, Ruby, SQL, Oracle, Perl . but most commonly used in Unix / Linux.

There are some other functions in the module to operate with RegEx. Before diving into these functions, learn more about RegEx's regular expression.

Syntax pattern used in RegEx Python

The pattern we understand is a sample object, a compiled version of a regular expression. To specify a regular expression, we use special characters, including:

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

In the example above is `^` and `$` .

Square brackets `[]`

Square brackets are used to represent the set of characters you want to match.

Expression
Example string
Describe
`[abc]` `a` Match the character `a` `ac` Match the character `a` or `c` `Hey Jude` Does not match

Here, `[abc]` will match if the string you passed contains any characters `a` , `b` or `c` .

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

1. `[ae]` similar to `[abcde]` .
2. `[1-4]` similar to `[1234]` .
3. `[0-39]` similar to `[01239]` .

If the first character of the set is `^` then all characters not defined in the set will be matched.

1. `[^abc]` means matching strings without characters `a` , `b` or `c` .
2. `[^0-9]` means matching strings without any alphanumeric characters.

Special characters in `[]` will be treated as regular characters.

1. `[(+)]` matches any string with characters `(` , `+` or `)` .

Dots `.`

The dot matches any regular single character except the new line character `'n'` .

Expression
Example string
Describe
`.` `a` Doesn't match because there is only one `ac` character Match because there are two `acd` characters Match because there are two or more characters

Hats `^`

The caret symbol `^` is used to match the character topping a string.

Expression
Example string
Describe
`^a` `a` Match because starting with `a` `abc` Match because starting with `a` `bac` Does not match because `a` not the first `^ab` `abc` Match because starting with `ab` `acb` Does not match, starting with `a` but not the next character `b`

Symbol of Dollar $

The Dollar `$` symbol is used to match the character that ends a string.

Expression
Example string
Describe
`a$` `a` Match because ending in `a` `formula` Matches because of ending with `a` `cab` Does not match because `a` not in the last position

Asterisk *

The asterisk symbol `*` may match a string with or without a predefined character. This character can be repeated many times without any limit.

Expression
Example string
Describe
`ma*n` `mn` Match because of the previous character `*` may not appear `man` Match because of the full occurrence of `maaaan` characters Match because of the previous character `*` may appear more often `main` Does not match because of a pattern, `n` does not lie `woman` matching because there is a full appearance of characters

Plus sign +

The plus symbol `+` can match the string with one or more characters defined before it. This character can be repeated many times without any limit.

Expression
Example string
Describe
`ma+n` `mn` Does not match because the previous character `+` does not appear `man` joints because of the full occurrence of `maaaan` characters Match because the previous character `+` may appear many times `main` not match because unlike the pattern, `n` does not lie `woman` matching because there is a full appearance of characters

Question mark ?

The question mark icon may match a string with or without a predefined character. This character can **not** be repeated many times, only the amount is limited with **one occurrence** .

Expression
Example string
Describe
`ma?n` `mn` Match the previous letter `?` Can not appear `man` Match because of the full occurrence of `maaaan` characters Do not match because of the previous character `?` can only appear 1 time `main` not match because does not resemble pattern, does not lie `a` `woman` Matches because of the full appearance of characters

Brace {}

The curly brackets use the general formula: `{n,m}` , representing the character before it can appear at least n times up to m times. `n` and `m` are positive integers and n = m.

1. If `n` left blank, this value defaults to 0.
2. If left blank `m` , this value is infinite by default.

Expression
Example string
Describe
`a{2,3}` `abc dat` Does not match because it does not meet the condition of the `abc daat` because there are 2 characters `a` (`a`) of the `aabc daaat` because there are 2 and 3 characters `a` (`aa bc` and `d aaa t`) `aabc daaaat` Match because there appear 2 and 3 characters `a` (`aa bc` and `d aaa at`)

Try another example: RegEx `[0-9] {2, 4}` This matches the string with at least 2 digits and no more than 4 digits.

Expression
Example string
Describe
`[0-9]{2,4}` `ab123csde` Matches because of the conditions: `ab 123 csde` `12 and 345673` Matches because the conditions: `12` and `3456 73` `1 and 2` Not matched because the string has only 1 digit

Vertical markers |

Vertical window icon `|` This may match the string that exists 1 in 2 characters defined before and after it.

Expression
Example string
Describe
`a|b` `cde` Does not match because `a` , `b` does not appear `ade` Match because the condition is satisfied, `a` appears: `a de` `acdbea` Matches because the condition is satisfied, `a` and `b` appear: `a cd b e a`

Here, `a|b` matches any string containing `a` or `b` .

Parentheses ( )

Parentheses `()` are used to group patterns together, the string will match the regular expression inside these brackets.

For example: `(a|b|c)xz` matches any string with `a` or `b` or `c` preceding `xz` .

Expression
Example string
Describe
`(a|b|c)xz` `ab xz` Does not match because `a` or `b` is preceded, but not in `abxz` with `xz` `abxz` Matches because of the condition, `b` appears before `xz` : `a bxz` `axz cabxz` Matches because the condition is satisfied , both `a` and `b` appear before `xz` : `axz ca bxz`

Backslash

A backslash is used to exit special characters, ie when standing before a special character, will turn this character into a lowercase character, you can search for this special character in the string like other common characters.

For example: `$a` will match the string containing the preceding character `$ a` . Here, the Dollar `$` symbol is not used to match a string that ends with the character that comes with it as in the RegEx tool, `$` is just a normal character.

However, a backslash will also turn a character often adjacent to the back into a special character.

For example, the case of the letter `b` without a backslash will match the lowercase `b` characters, but when it adds a backslash, it becomes a special character, does not match any any more.

Some patterns go with

1. `A` - Matches the characters that follow it at the beginning of the string.

Expression
Example string
Describe
The match is because it lies in the `In the sun` sequence. The match is not at the beginning of the chain

2. `b` - Match the characters specified at the beginning or end of the word.

Expression
Example string
Describe
`bfoo football` Match for qualified condition, `foo` is at the beginning of `a football` because of the condition, `foo` located at the second word in the string `afootball` Does not match because `foo` is in the middle of the word in the string. foob `the foo` Match because of the condition, `foo` is at the end of the string `the afoo test` Match because of the condition, `foo` is at the end of the 2nd word in `the afootest` string `the afootest` not match because `foo` is in the middle of the word in the string.

3. `B` - Contrary to `b` , matches the designated characters not at the beginning or end of the word.

Expression
Example string
Describe
`bfoo football` Does not match because `foo` is at the beginning of `a football` string `a football` not match because `foo` is at the beginning of the 2nd word in the string `afootball` Match because `foo` is in the middle of the word in the string. foob `the foo` Doesn't match because `foo` is at the end of `the afoo test` string `the afoo test` not match because `foo` is at the end of the 2nd word in `the afootest` string because `foo` is in the middle of the word in the string.

4. `d` - Matches with alphanumeric characters, equivalent to `[0-9]` .

Expression
Example string

Describe
`d` `12abc3` Matches because of the condition: `12 abc 3` `Python` Does not match because no integers appear

5. `D` - Matches non-numeric characters, equivalent to `[^0-9]`.

Expression
Example string
Describe
`D` `1ab34"50` Matches because of the conditions: `1 ab 34 " 50` `1345` Does not match because the whole sequence of integers appears

6. `s` - Matches any space character, equivalent to `[ tnrfv]`.

Expression
Example string
Describe
`Python RegEx` Matches because the string has `PythonRegEx` whitespace `PythonRegEx` not match because the string has no spaces

7. `S` - Matches any non-whitespace character, equivalent to `[^ tnrfv]`.

Expression
Example string
Describe
`S` `ab` Matches because the string has `a b` character `a b` Does not match because the whole string is whitespace

8. `w` - Match with any alphabetic and alphanumeric character, equivalent to `[a-zA-Z0-9_]`.

Note: The underscore _ also considered an alphanumeric character.

Expression
Example string
Describe
`w` `12&": ;c` Matches because the string has alphanumeric characters `12&": ;c` `%"> !` Does not match because the string has no alphanumeric characters

9. `W` - Match any character other than letters and numbers, equivalent to `[^a-zA-Z0-9_]`.

Expression
Example string
Describe
`w` `1a2%c` Matches because the string has non-alphanumeric characters `1a2 % c` `Python` Does not match because the string has only alphanumeric characters

Note: The underscore _ also considered an alphanumeric character.

Tips : To build RegEx regular expressions, you can use the RegEx test tool like `regex101` . This tool not only creates regular expressions but also helps you learn it more thoroughly.

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

Regular Expression in Python

Regular Expression in Python is expressed via the `re` module, so when you first want to use the regular expression, you need to import the module into the program.

```
import re
```

This module has a lot of methods, functions and constants to work with RegEx. TipsMake.com will list some of the best used examples along with an example so you can easily visualize and capture.

re.findall ()

The re.findall () method returns a list of strings containing all the matching pattern results.

Syntax:

```
findall(partern, string)
```

Inside:

1. `pattern` is RegEx.
2. `string` is the string to match.

Example : Extract the numbers from the given string: *"hello 12 hi 89. Howdy 34"*

```
import re string = 'hello 12 hi 89. Howdy 34' pattern = 'd+' result = re.findall
```

Results returned:

```
['12', '89', '34']
```

re.split ()

The method re.split () uses regular expressions to break strings into substring and return the list of these substring.

Syntax:

```
re.split(pattern, string, maxsplit)
```

Inside:

1. `pattern` is RegEx.
2. `string` is the string to match.
3. `maxsplit` (integer) is the maximum number of strings to be interrupted. If left blank, Python will match and cut all the strings that meet the criteria.

Example: Interrupting at a location with a space character:

```
import re string = 'The rain in Vietnam.' pattern = 's' result = re.split(patter
```

Results returned:

```
['The', 'rain', 'in', 'Vietnam.']
```

Example: Breaking the string in the first space character:

```
import re string = 'The rain in Vietnam.' pattern = 's' result = re.split(patter
```

Result:

```
['The', 'rain in Vietnam.']
```

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

re.sub ()

This is one of the most important methods used with Regular Expression

Re.sub () will replace all the results that match the pattern in the string with another content passed and return the modified string.

Syntax:

```
re.sub(pattern, replace, string, count)
```

Inside:

1. `pattern` is RegEx.
2. `replace` is the content instead of the result string matching the `pattern` .
3. `string` is the string to match.
4. `count` (integer) is the number of substitutions. If left blank, Python will treat this value to 0, match and replace all qualified strings.

For example : The program code deletes all spaces

```
import re # chu?i nhi?u dòng string = 'abc 12 de 23 n f45 6' # so kh?
p các ký t? kho?ng tr?ng pattern = 's+' # chu?i r?
ng replace = '' new_string = re.sub(pattern, replace, string) print(new_string)
```

Results returned:

```
abc12de23f456
```

If no matching result is found, `re.sub()` will return an empty string.

For example: The program code deletes the first 2 spaces

```
 import re # chu?i nhi?
u dòng string = 'abc 12 de 23 n f45 6 n quantrimang website' # so kh?
p các ký t? kho?ng tr?
ng pattern = 's+' replace = '' new_string = re.sub(r's+', replace, string, 2) pri
```

Output returns:

```
 abc12de23 f45 6 quantrimang website
```

re.subn ()

The re.subn () method uses the same as `re.sub()` above, but the returned result includes a tuple containing two values: the new string after being replaced and the number of replacements performed.

```
 import re # chu?i nhi?
u dòng string = 'abc 12 de 23 n f45 6 n quantrimang website' # so kh?
p các ký t? kho?ng tr?ng pattern = 's+' # chu?i r?
ng replace = '' new_string = re.subn(pattern, replace, string) print(new_string)
```

Results returned:

```
 ('abc12de23f456quantrimangwebsite', 6)
```

re.search ()

The re.search () method is used to search for strings matching the RegEx pattern. If the search succeeds, *re.search ()* returns the matching object, otherwise, it returns `None` .

Syntax:

```
 search(pattern, string)
```

Inside:

1. `pattern` is RegEx.
2. `string` is the string to match.

```
import re string = "TipsMake.com la website ban co the hoc Python" # Kiem tra xem
?u t?n t?i chu?i kh?
p print("Tim thay 'Quantrimang' nam o dau chuoi") # in ra thong bao nay else: pri
```

Results returned:

```
 Tim thay 'Quantrimang' nam o dau chuoi
```

In this example, `match` contains the appropriate object that matches the pattern.

Object match

Some methods and properties are often used with match objects.

match.group ()

The group () method returns the parts of the string that match the pattern.

```
import re string = '39801 356, 2102 1111' pattern = '(d{3}) (d{2})' match = re.s
?u t?n t?i chu?i kh?p print(match.group()) # in ra k?t qu?
 else: print("Không kh?p") # Không thì hi?n thông báo # Output: 801 35
```

Here, the variable `match` contains the match object.

We have a pattern `(d{3}) (d{2})` divided into two small groups `(d{3})` and `(d{2})` . You can get part of the string corresponding to subgroups in this parenthesis as follows:

```
>>> match.group(1) '801' >>> match.group(2) '35' >>> match.group(1, 2) ('801', '
```

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

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

```
>>> match.start() 2 >>> match.end() 8
```

The span () function returns tuple containing the start and end index of the appropriate string section.

```
>>> match.span() (2, 8)
```

match.re and match.string

The `re` attribute of the match object will return a regular expression. Similarly, the string attribute returns the string passed in the code.

```
>>> match.re re.compile('(d{3}) (d{2})') >>> match.string '39801 356, 2102 1111'
```

Above are all the most commonly used methods in module re.

Use the prefix r before RegEx

When the prefix `r` or `R` is used before a regular expression that represents the next string it is only normal characters.

For example: `'n'` is a newline, and `r'n'` means that the string consists of two characters: backslash and `n` .

Backslash used to exit the characters as mentioned above. However, use prefix `r` first then it is just a normal character.

```
import re string = 'n and r are escape sequences.' result = re.findall(r'[nr]', s
```

Previous article: Declare @property in Python

You finished reading the article "**Regular Expression (RegEx) in Python**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.

© 2019 TipsMake.com