Tutorial – Python Regex – Shiksha Online

Shiksha Online

Updated on Nov 22, 2022 19:35 IST

Introduction to Regular Expressions

Sometimes in your code, you might need to recognize patterns in text, like e-mail address, phone numbers, web page address, ZIP codes, etc. You probably would have used some wildcard patterns on the command line such as **ls * .py**, which by the way lists all filenames that end with .py.In this blog, you will learn about Python Regex

Similarly, in Python, you can use regular expressions to check if a string contains the specified search pattern that's relevant to you. ***Regular Expressions are combinations of characters that describe a search pattern for matching characters in other strings.*** With python regular expressions(Python Regex), you can:

- Validate the data. For example, you can check that a string last name contains only letters, spaces, apostrophes and hyphens

- You can extract data from the text, commonly known as scraping. For example, locating all the URLs in a given web page
- Cleaning up your data. For example, fixing typos, ensuring consistent data formats, handling incomplete data, etc.
- Converting your data to another format. For example, formatting space separated data into CSV (comma separated values)

Python makes Regular Expressions(Python Regex) available through the **re** module.

Table of contents

re Module

To use regular expressions in Python, first, you need to import the Python Standard Libraries re module.

```
#Importing the re module
import re
```

Once you import the re module, you can start using the regular expressions(Python Regex). For a simple match, all you need to do is define a string pattern that you want to match and the source string that you want to match the pattern against. Something like this:

```
#Simple example using match()
import re
test_string1 = "123@abcd456"

#backslash is escaped, so we just have '' & 't'
test_string2 = "\t123@abcd456"

pattern1 = r"123"              #raw string literal
```

```
pattern2 = "\t123"

#match() checks whether the source begins with the pattern
x = re.match(pattern1, test_string1)
y = re.match(pattern2, test_string2)
print(x.group())
print(y)
```

In the above example, '123' is the pattern and '123@abcd456' is the source string that you want to search. The function match() checks whether the source string 'test_string1' begins with the string pattern1. It does! The group() function returns one or more subgroups of the match, which is "123" in our case.

You might have noticed that the pattern variable begins with r, which shows that the string is a raw string literal. A raw string literal is slightly different from a string literal. In string literal, you have to use a backslash to escape "escape sequences", such as tabs ( ), newlines ( ), etc. However, with raw string literals, " " is just a string of 2 characters and n. So, in the above example, when you try to find the pattern2 in test_string2, the match () function returns **None**.

Moving on, match() is not the only way to search a string for a pattern match. Here are several others that you can use:

Function	Description
match()	Matching is done from the start of the string only
search()	Returns the first match (anywhere in the string unlike match())
findall()	Returns a list of non-overlapping matches
split()	Splits the source wherever the pattern matches and returns a list of string pieces
sub()	Replaces all the occurrences of a pattern in source to another string

Find First Match With search()

As mentioned in the above table, you can use the search() function to find the pattern anywhere in the source string, not just the beginning.

```
#Using search() to find the first match
import re
test_string = "0123@abcd456123"
pattern = r"123"
x = re.search(pattern, test_string)if x:
  print(x.group())
```

Find All Matches With findall()

The above example looked only for one match or rather the 1st match. To print all the matches and to know how many matches occurred, you can use findall() function.

#Find All Matches With findall()

```
#Find All Matches With findall()
import re
test_string = "0123@abcd123efgh123"
pattern = r"123"
x = re.findall(pattern, test_string)
print(x)
print("Found", len(x), "matches")
```

Split the Matches Using split()

You can use the split() function to split a string into a list by a pattern. The split() function splits the source wherever the pattern matches and returns a list of string pieces as shown below.

```
#Split the string by pattern using split()
import re
test_string = "0123@abcd123efgh123"
pattern = r"123"
x = re.split(pattern, test_string)
print("Found", len(x), "matches")
print(x)
```

Replace the Matches Using sub()

To replace all the occurrences of the pattern in the given string you can use sub() function as shown below.

#Replace all the matches with sub()

```
#Replace all the matches with sub()
import re
test_string = "0123@abcd123efgh123"
pattern = r"123"
x = re.sub("123", "456", test_string)   #replace 123 by 456
print("The original string is :", test_string)
print("The new string is :", x)
```

Metacharacters and Special Sequences

Regular Expressions(Python Regex) contain various special symbols called metacharacters, as shown in the table below:

Symbol	Description
[ ]	A set of characters
^	Start of source string
$	End of source string
*	Zero or more occurrences
+	One or more occurrences
?	Zero or one occurrence
{ }	Exactly the specified number of occurrences
.	Any character except newline
	Indicates a special sequence
|	Either or
()	Capture or group

Then we have some common predefined character classes and the groups of characters they match. To escape the special meaning and just use the metacharacter as its literal value, you should precede it with an additional backslash (Eg \n).

Character Class	Matches
d	Matches any digit (0-9)
D	Matches any character that is not a digit
A	Matches if the pattern is the beginning of the string
s	Any whitespace character (such as spaces, tabs, and newlines)
S	Matches when the string does not contain any whitespace character
w	Matches when the string has any word characters (also referred to as alphanumeric characters – that is, an uppercase or lowercase letter, any digit or an underscore
W	Matches if the string does not contain any word characters
Z	Matches if the pattern is at the end of the string
	Matches when a pattern is at the beginning or at the end of the word
B	Matches when a pattern is anywhere but at the beginning and at the end of the string

Also read:

Python Dictionary Practice Programs For Beginners

Working with Python dictionary types can seem slightly difficult if you are not familiar with its features. The best way to master it is by solving problem statements. It is...**read more**

Python Lists Practice Programs For Beginners

Test your Python Lists knowledge with practice programs! In this blog, we will solve 12 python lists exercises for beginners.

Introduction to Python Dictionary (With Examples)

In this tutorial, you will learn all about Dictionary in Python: creating dictionaries, accessing dictionary items, looping through a dictionary, and other dictionary operations with the help of examples.

So guys, we now have enough information. Let us take a sample string and try to use these symbols and special character classes to understand Regular Expressions(Python Regex) much better!

```
#Example
import re

print("Examples using str1")
str1 = "I was born on 26-12-1947 and I"
print("str1 is:", str1, "
")

m = re.search(r'd{2}-d{2}-d{4}', str1)
print(m.group())

n = re.findall(r'^I | I$', str1)
print(n)

n1 = re.search(r'd{3,}', '12345')
print(n1.group())

n2 = re.search(r'd{6,}', '12345')
print(n2)

n3 = re.search(r'd{3,6}', '12')
print(n3)

print("
Examples using str2")
str2 = "Hey! I wish I might have tasty dish of fish tonight."
```

```
print("str2 is:", str2, "
")

x = re.findall(r'[wfd]ish', str2)
print(x)

y = re.findall(r'ghtW', str2)
print(y)

z = re.findall(r'[fsh]+', str2)
print(z)

p = re.findall('□dish', str2)
print(p)

p1 = re.findall(r'□dish', str2)
print(p1)

str3 = 'Dean Winchester, e-mail: demon1@supern.com'
pattern1 = r'([A-Z][a-z]+ [A-Z][a-z]+), e-mail: (w+@w+.w{3})'
result1 = re.search(pattern1, str3)
print(result1.groups())
```

The points that we can infer from the example are:

- r'd{2}-d{2}-d{4}' matches the string consists of two digits, one dash, two digits, one dash, and finally four digits
- r'^I | I$' matches character 'I' at the beginning or end of string
- r'd{3,}'matches strings containing at least three digits. To generalize {n, } qualifier matches a string with at least n occurrences of a pattern or subexpression
- r'd{3,6}' matches strings containing 3 to 6 digits. {n, m} qualifier matches strings with between n and m (inclusive) occurrences of subexpression
- r'[wfd]ish' matches all the occurrences of words 'wish', or 'fish', or 'dish' in given string
- r'ghtW' matches 'ght' followed by a non-alphanumeric character in a string
- r'[fsh]+' matches one or more runs of f, d, or h
- '□dish' should match any word that begins with 'dish' but it doesn't. That is because □ is

interpreted as an escape character backspace. To escape the meaning of the □ and interpret it as a raw string use ‘r’ at the beginning

- R’□dish’ matches any word that begins with ‘dish’

Let’s now analyze the regular expression *pattern1*, shown at the end of the example.

- We are using parentheses metacharacter to capture the substrings in a match
- ([A-Z][a-z]+ [A-Z][a-z]+)matches two words separated by a whitespace character. Each of these words should begin with a capital letter
- e-mail: has literal characters that match themselves
- (w+@w+.w{3}) matches a simple email address that contains one or more alphanumeric characters, followed by ‘@’ character, again followed by one or more alphanumeric character, a dot, and exactly three alphanumeric characters
- In the next step, the match object’s **groups()** method returns a tuple of captured substrings
- The match object’s group() returns entire match as single string, whereas groups() returns the tuple of matched substrings separated by regular expression
- You can also provide an argument to group() methods such as group(1) or group(2), or group(n) to display the nth substring

This way, you can use the powerful pattern-matching abilities of regular expressions to clean, transform, validate your data with functions from the **re module**. Python regular expression(Python Regex) also allows you to compile your pattern so that you can reuse it later in the program. In some cases, you can even alter the behavior of regular expressions with the help of flags. I highly recommend you practice more examples to master the concept.

If you have any queries, please let us know in the comment section. Happy coding!