

AGENDA

1. Introduction to **Regular Expressions**
2. Metacharacters in **Regular Expressions**
3. Functions and Constants **To work with RegEx**

I

Introduction to Regular Expressions



Data Science
Academy

RegEx

- A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern with a formal syntax. Regular expressions are typically used in applications that involve a lot of text processing.
- As a data scientist/engineer, having a solid understanding of Regex can help you perform various data preprocessing very easily.
- There are multiple open-source implementations of regular expressions, each sharing a common core syntax but with different extensions or modifications to their advanced features. Python has a built-in package called re, which can be used to work with Regular Expressions.

reg[ular]
expr[essio]n

RegEx

- A **Regular Expression** (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 letters 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

Module "Re"

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

```
Import re
```

```
import re

pattern = "^a...s$"
test_string = "abyss"
result = re.match(pattern, test_string)

if result :
    print("Search successful.")
else:
    print("Search unsuccessful.")
```

Module "Re"

- Regex functionality in Python resides in a module named re. The re module contains many useful functions and methods, most of which you'll learn about in the next tutorial in this series.
- 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.
- There are other several functions defined in the re module to work with RegEx. Before we explore that, let's learn about regular expressions themselves.
- **Key Idea:** Regex works at the character-level, not word-level.

Expectation



word 'cool'

Reality



'c' then 'o' then 'o' then
'l'

Module "Re"

- The implication of this is that the regex r'cool' would match the following sentences as well.

Batman is coolest

He bought a watercooler

Batman is supercool

White space character

- We can detect special characters such as whitespace and newlines using special escape sequences.

Name	Regex	Example
Whitespace	\s	Batman is cool
Tab	\t	Batman is cool
Newline	\n	Batman is cool He is a human

II

Metacharacters in Regular Expressions



Data Science
Academy

Specify Pattern using RegEx

- To specify regular expressions, metacharacters are used. in 3rd page , ^ and \$ are metacharacters.

MetaCharacters:

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

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

MetaCharacters

- **[] - Square brackets**
- Square brackets specifies a set of characters you wish to match.
- For example, the following pattern matches any of the characters 'a', 'e', 'i', 'o', and 'u'.

Regex: [aeiou]

apple

MetaCharacters

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

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

MetaCharacters

- 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.

MetaCharacters

- **. . – 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 (contain 4 characters)

MetaCharacters

- ^ - **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
	abc	1 match
^ab	acb	No match(starts with a but not followed by b)

regex = ^hey

hey

regex = ^hey

He said "hey"

regex = ^hey

hey man!

MetaCharacters

- **\$ - 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

regex = !\$

love it!

regex = !\$

Cool! See
ya

MetaCharacters

- * - **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
	maan	1 match
	main	No match(a is not followed by n)
	woman	1 match

MetaCharacters

- + - **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
	maan	1 match
	main	No match (a is not followed by n)
	woman	1 match

MetaCharacters

- ? - **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
	maan	No match (more than one a)
	main	No match (a is not followed by n)
	woman	1 match

MetaCharacters

- **{}** - **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)
	abc daaaat	1 match (at daaaat)

MetaCharacters

- 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?
<code>[0-9]{2,4}</code>	ab123csde	1 match (at 123)
	12 and 345673	3 matches (12, 3456, 73)
	1 and 2	No match

MetaCharacters

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

Expression	String	Matched?
a b	cde	No match
	ade	1 match (match at <u>a</u> de)
	acdbea	3 matches (at <u>a</u> cd <u>b</u> <u>e</u> <u>a</u>)

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

MetaCharacters

- **() - 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 <u>abxz</u>)
	axzbc cabxz	2 matches (at <u>axzbc</u> <u>cabxz</u>)

MetaCharacters

- \ - **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

Special Sequences

- \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 foo	Match
foo\b	the afoo test	Match
	the afootest	No match

Special Sequences

- \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
	the foo	No match
foo\b	the afoo test	No match
	the afootest	Match

Special Sequences

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

Expression	String	Matched?
\d	12abc3	3 matches (at <u>1</u> 2abc <u>3</u>)
	Python	No match

Special Sequences

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

Expression	String	Matched?
\D	1ab34"50	3 matches (at 1 <u>a</u> b34"50)
	1345	No match

Special Sequences

- `\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

Special Sequences

- `\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

Special Sequences

- \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 <u>1</u> <u>2</u> &" : ; <u>c</u>)
	%"> !	No match

Special Sequences

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

Expression	String	Matched?
\W	1a2%c	1 match (at 1a2 <u>%</u> c)
	Python	No match

Special Sequences

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

Expression	String	Matched?
Python <code>\Z</code>	I like Python	1 match
	I like Python Programming	No match
	Python is fun	No match



Functions and Constants

To work with RegEx

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: ', '. ']
```

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

maxsplit

- 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. ']
```

- 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()

```
import re
# multiline string
string = "abc 12\nde 23 \n f45 6"
# matches all whitespace characters
pattern = "\s+"
# empty string
replace = " "
new_string = re.sub(pattern, replace, string)
print(new_string)
# Output : abc 12de 23 f45 6
```


count

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

```
import re

# multiline string
string = "abc 12\nde 23 \n f45 6"

# matches all whitespace characters
pattern = "\s+"
replace = " "

new_string = re.sub(r"\s+", replace, string, 1)
print(new_string)

# Output :
abc 12de 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()

```
# Program to remove all whitespaces
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 :
abc 12de 23
f45 6
```

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
```

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()
0
>>> match.end()
6
```

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

```
>>> match.span()
(0,6)
```

match.re and match.string

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

```
match.re
#Output
re.compile(r'\APython', re.UNICODE)
match.string
#Output
'Python is fun'
```

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

```
import re
string = "\n and \r are escape sequences."

result = re.findall(r"[\n\r]", string)
print(result)
# Output :[ '\n', '\r' ]
```

Thank **You!**



Data Science
Academy