Python tutorial – Regular expressions

Introduction

Regular expressions are used to identify if a certain pattern exists in a given sequence of characters usually strings. They help in manipulating textual data, which is often a pre-requisite for all Natural Language Processing (NLP) tasks. Regular expressions are extremely powerful as they help in extensive data manipulation just using simple commands.

Regular expressions are used in multiple ways in various platforms, like, command line tools, plain text editors, programming languages. See the table below:

Command line tools	Plain Text editors	Programming Languages
grep	vi	Perl
egrep	ed	Python
sed	emacs	Java, Javascript

Regular expressions in Python

In Python, regular expressions are supported by the **re** module. If one wants to start using it in the Python scripts, you have to import this module with the help of **import**

```
#!usr/bin/python3

import re
```

Basic patterns in Regular expressions

Ordinary characters are the simplest regular expressions which can be used for finding a sequence of strings. They match themselves exactly and do not have a special meaning in their regular expression syntax.

Examples: 'A', 'a', 'X', '5'.

r - This is called a raw string literal. It changes how the string literal is interpreted. Such literals are stored as they appear.

```
#!usr/bin/python3
import re
adjective = r"happy"
comparative = "happier"
if re.match(adjective, comparative):
   print("Match!")
else:
   print("Not a match!")
```

Output- Not a match

Special characters

Special characters do not match themselves as they are but carry a special meaning when used in regular expressions.

. A period - Matches any single character except newline character.

```
#!usr/bin/python3

import re

adjective = r"h.p.y"

comparative = "happy"

if re.match(adjective, comparative):

   print("Match!")

else:

  print("Not a match!")
```

Output- Match

Sl.No	**Special character**	**description**
1.	\w	Matches any single letter, digit or underscore.

2.	\W	Matches any character not part of \w (lowercase w)
3.	\s	Matches a single whitespace character like: space, newline, tab, return.
4.	\S	Matches any character not part of \s (lowercase s).
5.	\t	Matches tab.
6.	\n	Matches newline.
7.	\r	Matches return.
8.	\d	Matches decimal digit 0-9.
9.	^	Matches a pattern at the start of the string.
10	$	Matches a pattern at the end of string.
11	[abc]	Matches a or b or c.
12	[a-zA-Z0-9]	Matches any letter from (a to z) or (A to Z) or (0 to 9). Characters that are not within a range can be matched by complementing the set. If the first character of the set is ^, all the characters that are not in the set will be matched.
13	\A	Matches only at the start of the string. Works across multiple lines as well.
14	\b	Matches only the beginning or end of the word.
15	\	If the character following the backslash is a recognized escape character, then the special meaning of the term is taken. For example, \n is considered as newline. However, if the character following the \ is not a recognized escape character, then the \ is treated like any other character and passed through.
16	+	Checks for one or more characters to its left.
17	*	Checks for zero or more characters to its left.

18	?	Checks for exactly zero or one character to its left.
19	.*	Any pattern of characters
20	{x}	Repeat exactly x number of times.
21	{x,}	Repeat at least x times or more.
22	{x,y}	Repeat at least x times but no more than y times.

Note – The +, .* and * are called the greedy characters.

Grouping functions:

Parts of a regular expression pattern bounded by parenthesis () are called groups. The parenthesis does not change what the expression matches, but rather forms groups within the matched sequence. The plain match.group () without any argument is still the whole matched text as usual.

re Python library functions:

The re library in Python provides several functions that makes natural language processing easy. Let us look at some of such functions:

1. search()

With this function, you scan through the given string/sequence looking for the first location where the regular expression produces a match. It returns a corresponding match object if found, else returns None if no position in the string matches the pattern. Note that None is different from finding a zero-length match at some point in the string.

Syntax- **re.search (pattern, string, flag=0)**

```
#!usr/bin/python3

import re

adjective = r"happy"

comparative = "happy and sad"

if re.search("[0-9]", “adjective"):
   print("match")
else:
   print("Not a match")
```

Output- Not a match

2. Match()

Returns a corresponding match object if zero or more characters at the beginning of string match the pattern. Else it returns None, if the string does not match the given pattern.

Syntax- **re.match (pattern, string, flag=0)**

```
#!usr/bin/python3
import re
adjective = r"happy"
comparative = "happy and sad"
if re.match("a", adjective):
   print("match")
else:
   print("Not a match")
```

Output- match

3. findall()

Finds all the possible matches in the entire sequence and returns them as a list of strings. Each returned string represents one match.

Syntax- **re.findall(pattern,string,flag=0)**

```
#!usr/bin/python3
import re
nouns = "cat, mat, camp, march, train, tramp, couch"
```

```
findall = re.findall('^[ct].*', nouns)
for i in findall:
    print(findall)
```

Output- cat,camp,train,tramp,couch

4. sub()

This is the substitute function. It returns the string obtained by replacing or substituting the leftmost non-overlapping occurrences of pattern in string by the replacement repl. If the pattern is not found then the string is returned unchanged.

Syntax- re.sub(pattern, repl, string, count=0, flags=0)

```
#!usr/bin/python3
import re
nouns = "cat rat tap mat cap van"
substitute = re.sub(r'[crm]at','bat', nouns)
print(substitute)
```

Output – bat bat tap bat cap van

5. compile()

Compiles a regular expression pattern into a regular expression object. When you need to use an expression several times in a single program, using the compile() function to save the resulting regular expression object for reuse is more efficient. This is because the compiled versions of the most recent patterns passed to compile() and the module-level matching functions are cached.

Syntax- compile(pattern, flags=0)

```
#!usr/bin/python3
import re
comp = re.compile("happy")
```

```
sequence = "sad and happy"

print(comp.search(sequence).group())
```

Output- happy

Note-An expression's behavior can be modified by specifying a flags value. One can add flag as an extra argument to the various functions that have been mentioned earlier. Some of the flags used are: IGNORECASE, DOTALL, MULTILINE, VERBOSE, etc.

References

https://www.datacamp.com/community/tutorials/python-regular-expression-tutorial

https://en.wikibooks.org/wiki/Regular_Expressions/Introduction

https://en.wikibooks.org/wiki/Regular_Expressions/Basic_Regular_Expressions