Module 7: Regular Expressions

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expression are popularly known as regex or regexp.Usually, such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation.Large scale text processing in data science projects requires manipulation of textual data. The regular expressions processing is supported by many programming languages including Python. Python's standard library has re module for this purpose.Since most of the functions defined in re module work with raw strings, let us first understand what the raw strings are.

Raw Strings

Regular expressions use the backslash character ('\') to indicate special forms or to allow special characters to be used without invoking their special meaning. Python on the other hand uses the same character as escape character. Hence Python uses the raw string notation.

A string become a raw string if it is prefixed with r or R before the quotation symbols. Hence 'Hello' is a normal string were are r'Hello' is a raw string.

```
>>> normal="Hello"
>>> print (normal)
Hello
>>> raw=r"Hello"
>>> print (raw)
Hello
```

In normal circumstances, there is no difference between the two. However, when the escape character is embedded in the string, the normal string actually interprets the escape sequence, where as the raw string doesn't process the escape character.

```
>>> normal="Hello\nWorld"
>>> print (normal)
Hello
World
>>> raw=r"Hello\nWorld"
>>> print (raw)
Hello\nWorld
```

In the above example, when a normal string is printed the escape character '\n' is processed to introduce a newline. However because of the raw string operator 'r' the effect of escape character is not translated as per its meaning.

Metacharacters

Most letters and characters will simply match themselves. However, some characters are special metacharacters, and don't match themselves. Meta characters are characters having a special meaning, similar to * in wild card.

Here's a complete list of the metacharacters –

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

Sr.No.	Metacharacters & Description
1	[abc] match any of the characters a, b, or c
2	[a-c] which uses a range to express the same set of characters.
3	[a-z] match only lowercase letters.
4	[0-9] match only digits.
5	'^' complements the character set in [].[^5] will match any character except'5'.

'\'is an escaping metacharacter. When followed by various characters it forms various special sequences. If you need to match a [or \, you can precede them with a backslash to remove their special meaning: \[or \\.

Predefined sets of characters represented by such special sequences beginning with '\' are listed below –

Sr.No.	Metacharacters & Description
1	\d Matches any decimal digit; this is equivalent to the class [0-9].
2	\D Matches any non-digit character; this is equivalent to the class [^0-9].
3	**\s**Matches any whitespace character; this is equivalent to the class [\t\n\r\f\v].
4	\S Matches any non-whitespace character; this is equivalent to the class [^\t\n\r\f\v].
5	\w Matches any alphanumeric character; this is equivalent to the class [a-zAZ0-9_].

6	\W Matches any non-alphanumeric character. equivalent to the class [^a-zAZ0-9_].
7	. Matches with any single character except newline '\n'.
8	? match 0 or 1 occurrence of the pattern to its left
9	+ 1 or more occurrences of the pattern to its left
10	* 0 or more occurrences of the pattern to its left
11	\b boundary between word and non-word and /B is opposite of /b
12	[..] Matches any single character in a square bracket and [^..] matches any single character not in square bracket.
13	\ It is used for special meaning characters like \. to match a period or \+ for plus sign.
14	{n,m} Matches at least n and at most m occurrences of preceding
15	a| b Matches either a or b

Python's re module provides useful functions for finding a match, searching for a pattern, and substitute a matched string with other string etc.

The re.match() Function

This function attempts to match RE pattern at the start of string with optional flags. Following is the syntax for this function –

```
re.match(pattern, string, flags=0)
```

Here is the description of the parameters –

Sr.No.	Parameter & Description
1	**pattern** This is the regular expression to be matched.
2	**String** This is the string, which would be searched to match the pattern at the beginning of string.
3	**Flags** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The re.match() function returns a match object on success, None on failure. A match object instance contains information about the match: where it starts and ends, the substring it matched, etc.The match object's start() method returns the starting position of pattern in the string, and end() returns the endpoint.If the pattern is not found, the match object is None.We use group(num) or groups() function of match object to get matched expression.

Sr.No.	Match Object Methods & Description
1	**group(num=0)**This method returns entire match (or specific subgroup num)
2	**groups()**This method returns all matching subgroups in a tuple (empty if there weren't any)

EXAMPLE:

```
import re
line = "Cats are smarter than
dogs"
matchObj = re.match( r'Cats',
line)
print (matchObj.start(),
matchObj.end())
print ("matchObj.group() : ",
matchObj.group())
```

It will produce the following output –

```
0 4
matchObj.group() : Cats
```

Here is the description of the parameters –

Sr.No.	Parameter & Description
1	**Pattern** This is the regular expression to be matched.
2	**String** This is the string, which would be searched to match the pattern anywhere in the string.
3	**Flags** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The re.search function returns a match object on success, none on failure. We use group(num) or groups() function of match object to get the matched expression.

```
import re
line = "Cats are smarter than
dogs"
matchObj = re.search( r'than',
line)
print (matchObj.start(),
matchObj.end())
print ("matchObj.group() : ",
matchObj.group())
```

It will produce the following output –

```
17 21
matchObj.group() : than
```

Matching Vs Searching

Python offers two different primitive operations based on regular expressions, match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string (this is what Perl does by default).

EXAMPLE:

```
import re
line = "Cats are smarter than dogs";
matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print ("match --> matchObj.group() : ",
matchObj.group())
else:
   print ("No match!!")
searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
   print ("search --> searchObj.group() : ",
searchObj.group())
else:
   print ("Nothing found!!")
```

When the above code is executed, it produces the following output −

```
No match!!
search --> matchObj.group() : dogs
```

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

1	**re.I** Performs case-insensitive matching.
2	**re.L** Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).
3	**re.M** Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
4	**re.S** Makes a period (dot) match any character, including a newline.
5	**re.U** Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
6	**re.X** Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular Expression Patterns

Except for control characters, (+ ? . * ^ $ ( ) [ ] { } | \), all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python –

Sr.No.	Pattern & Description
1	**^** Matches beginning of line.
2	**$** Matches end of line.
3	**.** Matches any single character except newline. Using m option allows it to match newline as well.
4	**[...]** Matches any single character in brackets.
5	**[^...]** Matches any single character not in brackets
6	**re*** Matches 0 or more occurrences of preceding expression.
7	**re+** Matches 1 or more occurrence of preceding expression.
8	**re?** Matches 0 or 1 occurrence of preceding expression.
9	**re{ n}** Matches exactly n number of occurrences of preceding expression.
10	**re{ n,}** Matches n or more occurrences of preceding expression.
11	**re{ n, m}** Matches at least n and at most m occurrences of preceding expression.
12	**a| b** Matches either a or b.

13	**(re)** Groups regular expressions and remembers matched text.
14	**(?imx)** Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
15	**(?-imx)** Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
16	**(?: re)** Groups regular expressions without remembering matched text.
17	**(?imx: re)** Temporarily toggles on i, m, or x options within parentheses.
18	**(?-imx: re)** Temporarily toggles off i, m, or x options within parentheses.
19	**(?#...)** Comment.
20	**(?= re)** Specifies position using a pattern. Doesn't have a range.
21	**(?! re)** Specifies position using pattern negation. Doesn't have a range.
22	**(?> re)** Matches independent pattern without backtracking.
23	**\w** Matches word characters.
24	**\W** Matches nonword characters.

25	**\s** Matches whitespace. Equivalent to [\t\n\r\f].
26	**\S** Matches nonwhitespace.
27	**\d** Matches digits. Equivalent to [0-9].
28	**\D** Matches nondigits.
29	**\A** Matches beginning of string.
30	**\Z** Matches end of string. If a newline exists, it matches just before newline.
31	**\z** Matches end of string.

32	**\G** Matches point where last match finished.
33	**\b** Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
34	**\B** Matches nonword boundaries.
35	**\n, \t, etc.** Matches newlines, carriage returns, tabs, etc.
36	**\1...\9** Matches nth grouped subexpression.
37	**\10** Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Special Character Classes

Sr.No.	Example & Description
1	**.** Match any character except newline
2	**\d** Match a digit: [0-9]
3	**\D** Match a nondigit: [^0-9]
4	**\s** Match a whitespace character: [\t\r\n\f]
5	**\S** Match nonwhitespace: [^ \t\r\n\f]
6	**\w** Match a single word character: [A-Za-z0-9_]
7	**\W** Match a nonword character: [^A-Za-z0-9_]

Anchors

This needs to specify match position.

Sr.No.	Example & Description
1	**^Python** Match "Python" at the start of a string or internal line
2	**Python$** Match "Python" at the end of a string or line
3	**\APython** Match "Python" at the start of a string
4	**Python\Z** Match "Python" at the end of a string
5	**\bPython\b** Match "Python" at a word boundary

6	**\brub\B** \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
7	**Python(?=!)** Match "Python", if followed by an exclamation point.
8	**Python(?!!)** Match "Python", if not followed by an exclamation point.

SOME PYTHON EXAMPLE WITH REGEX

Example:This code uses regular expressions to check if a list of strings starts with “The”. If a string begins with “The,” it’s marked as “Matched” otherwise, it’s labeled as “Not matched”.

```
import re
regex = r'^The'
strings = ['The quick brown fox', 'The lazy dog', 'A quick brown
fox']
for string in strings:
    if re.match(regex, string):
        print(f'Matched: {string}')
    else:
        print(f'Not matched: {string}')
```

OUTPUT:

```
Matched: The quick brown fox
Matched: The lazy dog
Not matched: A quick brown fox
```

EXAMPLE: This code uses a regular expression to check if the string ends with “World!”. If a match is found, it prints “Match found!” otherwise, it prints “Match not found”.

```
import re

string = "Hello World!"
pattern = r"World!$"

match = re.search(pattern, string)
if match:
    print("Match found!")
else:
    print("Match not found.")
```

OUTPUT: Match found!

EXAMPLE: This code uses a regular expression (\d+) to find all the sequences of one or more digits in the given string. It searches for numeric values and stores them in a list. In this example, it finds and prints the numbers “123456789” and “987654321” from the input string.

```
import re
string = """Hello my Number is 123456789 and
        my friend's number is 987654321"""
regex = '\d+'

match = re.findall(regex, string)
print(match)
```

OUTPUT:

```
['123456789', '987654321']
```

EXAMPLE: The code uses a regular expression pattern [a-e] to find and list all lowercase letters from ‘a’ to ‘e’ in the input string “Aye, said Mr. Gibenson Stark”. The output will be ['e', 'a', 'd', 'b', 'e'], which are the matching characters.

```
import re
p = re.compile('[a-e]')

print(p.findall("Aye, said Mr. Gibenson
Stark"))
```

OUTPUT:

```
['e', 'a', 'd', 'b', 'e', 'a']
```