Regular Expression in Python

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

Python has a built-in package called re, which can be used to work with Regular Expressions.

```
import re
```

When you have imported the re module, you can start using regular expressions:

```
txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
```

The re module offers a set of functions that allows us to search a string for a match:

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

re.findAll

```
txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

The list contains the matches in the order they are found.

If no matches are found, an empty list is returned:

re.search()

The search() function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

```
txt = "The rain in Spain"
x = re.search("\s", txt)
```

If no matches are found, the value None is returned:

re.split()

The split() function returns a list where the string has been split at each match:

```
txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

re.sub()

The sub() function replaces the matches with the text of your choice:

```
txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

WildCard in Python

A wildcard is a symbol that can be used in place of or in addition to one or more characters. In computer programmes, languages, search engines, including operating systems, wildcards are used to condense search criteria. The question mark (?) and the asterisk () are the most popular wildcards.

Types of wild cards

The asterisk (*) or the character can be used to specify any number of characters. The asterisk * is typically used at the conclusion of the root word and when it is necessary to look for root words with a variety of possible ends.

For instance, if we use the word "game" as an example, the phrases "gamer" and "games" would appear in all search results. Depending on the search parameters and other words, there might be additional words in addition to these two

The question mark or the character? denotes one. Any of the letters in the root word may be used with it. When a word contains several other spellings, the use of the question mark operator speeds up the process.

Instead of the question mark wildcard, the dot or. character is utilised for the single character representation.

Like how the asterisk (*) symbol is employed, the.+ characters are used to match one or more characters. So, in Python, our regex code might look something like this to seek for all words beginning with the root "work":

```
1.words = ["car", "apple", "work", "working",
2.          "goat", "worker"]
3.for word in words:
4.      # Instead of using the * symbol, use the.+ symbol.
5.      if re.search('work.+', word) :
6.          print (word)
```

Raw string

When a string in Python is prefixed with the letter r or R, as in r'...' and R'...', it becomes a raw string. In contrast to a conventional string, a raw string considers backslashes () as literal characters. When working with strings that include a lot of backslashes, such as regular expressions or directory paths on Windows, raw strings are helpful.

Character classes

In Python, regex character classes are sets of characters or ranges of characters enclosed by square brackets [].

For example, [a-z] it means match any lowercase letter from a to z.

[abc]	Match the letter a or b or c
[abc][pq]	Match letter a or b or c followed by either p or q.
[^abc]	Match any letter except a, b, or c (negation)
[0-9]	Match any digit from 0 to 9. inclusive (range)
[a-z]	Match any lowercase letters from a to z. inclusive (range)
[A-Z]	Match any UPPERCASE letters from A to Z. inclusive (range)
[a-zA-z]	Match any lowercase or UPPERCASE letter. inclusive (range)
[m-p2-8]	Ranges: matches a letter between m and p and digits from 2 to 8, but not p2
[a-zA-Z0-9_]	Match any alphanumeric character

```
import re

s = 'mother of all battles'

result = re.findall(r'[aeiou]', s)

print result
```

```
1.import re
2.str = re.compile('hel.o')
3.a = ['hello', 'welcome', 'to', 'java', 'point']
4.match_is = [string for string in a if re.match_is(str, string)]
5.print(match_is)
```

Special Sequences

Special Sequence	Description	Examples	
\A	Matches if the string begins with the given character	\Afor	for geeks
			for the world
\b	Matches if the word begins or ends with the given character. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word.	\bge	geeks
			get
\B	It is the opposite of the \b i.e. the string should not start or end with the given regex.	\Bge	together
			forge

\d	Matches any decimal digit, this is equivalent to the set class [0-9]	\d	123
			gee1
\D	Matches any non-digit character, this is equivalent to the set class [^0-9]	\D	geeks
			geek1
\s	Matches any whitespace character.	\s	gee ks
			a bc a
\S	Matches any non-whitespace character	\S	a bd
			abcd

\w	Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9_].	\w	123
			geeKs4
\W	Matches any non-alphanumeric character.	\W	>$
			gee<>
\Z	Matches if the string ends with the given regex	ab\Z	abcdab
			abababab

Quantifiers

In regular expressions, quantifiers match the preceding characters or character sets a number of times

Quantifier	Name	Meaning
*	Asterisk	Match its preceding element zero or more times.
+	Plus	Match its preceding element one or more times.

?	Question Mark	Match its preceding element zero or one time.
{ *n* }	Curly Braces	Match its preceding element exactly n times.
{ *n* ,}	Curly Braces	Match its preceding element at least n times.
{ *n* , *m* }	Curly Braces	Match its preceding element from n to m times

```
import re

s = """CPython, IronPython, and JPython
       are major Python's implementation"""

matches = re.finditer('\w*Python', s)

for match in matches:
    print(match)
```

```
import re

s = "Python 3.10 was released in 2021"

matches = re.finditer('\d+', s)

for match in matches:
    print(match)
```

```
import re

s = "What color / colour do you like?"

matches = re.finditer('colou?r', s)

for match in matches:
    print(match)
```

```
import re

s = "It was 11:05 AM"

matches = re.finditer('\d{2}:\d{2}', s)

for match in matches:
    print(match)
```

```
import re

s = "5-5-2021 or 05-05-2021 or 5/5/2021"

matches = re.finditer('\d{1,}-\d{1,}-\d{4}', s)

for match in matches:
    print(match)
```

```
import re

s = "5-5-2021 or 05-05-2021 or 5/5/2021"

matches = re.finditer('\d{1,2}-\d{1,2}-\d{4}', s)

for match in matches:
    print(match)
```

Greedy matches in regex

By default, all quantifiers work in a greedy mode. It means that the quantifiers will try to match their preceding elements as much as possible.

Suppose you have the following HTML fragment that represents a button element:

```
s = '<button type="submit" class="btn">Send</button>'
```

```
import re

s = '<button type="submit" class="btn">Send</button>'

pattern = '".+"'
matches = re.finditer(pattern, s)

for match in matches:
    print(match.group())
```

To fix this issue, you need to instruct the quantifier (+) to use the non-greedy (or lazy) mode instead of the greedy mode

```
import re

s = '<button type="submit" class="btn">Send</button>'

pattern = '".+?"'
matches = re.finditer(pattern, s)

for match in matches:
    print(match.group())
```

Grouping in regex

A group is a part of a regex pattern enclosed in parentheses () metacharacter. We create a group by placing the regex pattern inside the set of parentheses (and) . For example, the regular expression (cat) creates a single group containing the letters ‘c’, ‘a’, and ‘t’.

```
import re


target_string = "The price of PINEAPPLE ice cream is 20"

---

# two groups enclosed in separate ( and ) bracket

result = re.search(r"(\b[A-Z]+\b).+(\b\d+)", target_string)


# Extract matching values of all groups
```

```
print(result.groups())

# Output ('PINEAPPLE', '20')
```

```
# Extract match value of group 1

print(result.group(1))

# Output 'PINEAPPLE'


# Extract match value of group 2

print(result.group(2))

# Output 20
```