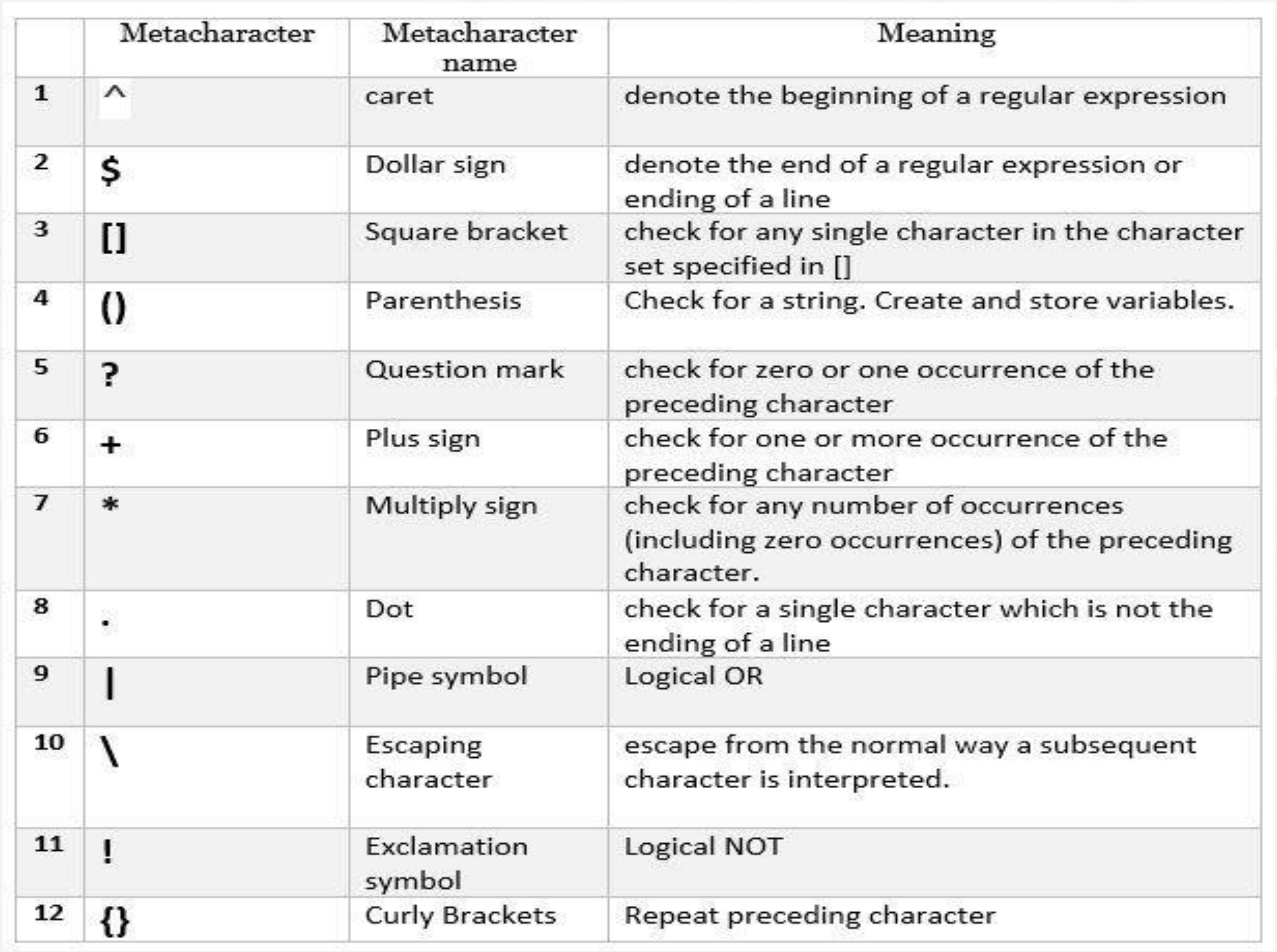
In Python

Ayyappan Murugesan
Phone: 7010774329
Call: 8870483557

Introduction

- Does this string match the pattern?
- Is there a match for the pattern anywhere in this string?
- You can also use REs to modify a string or to split it apart in various ways
- Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C

Matching Characters | Meta Characters



	Metacharacter	Metacharacter name	Meaning
1	^	caret	denote the beginning of a regular expression
2	$	Dollar sign	denote the end of a regular expression or ending of a line
3	[]	Square bracket	check for any single character in the character set specified in []
4	()	Parenthesis	Check for a string. Create and store variables.
5	?	Question mark	check for zero or one occurrence of the preceding character
6	+	Plus sign	check for one or more occurrence of the preceding character
7	*	Multiply sign	check for any number of occurrences (including zero occurrences) of the preceding character.
8	.	Dot	check for a single character which is not the ending of a line
9	|	Pipe symbol	Logical OR
10	\	Escaping character	escape from the normal way a subsequent character is interpreted.
11	!	Exclamation symbol	Logical NOT
12	{}	Curly Brackets	Repeat preceding character

Frequently used short forms

Short Form	Actual Pattern	Description
\d	[0-9]	Matches any decimal digit
\D	[^0-9]	Matches any non-digit character
\s	[\t\n\r\f\v]	Matches any whitespace character
\S	[^ \t\n\r\f\v]	Matches any non-whitespace character
\w	[a-zA-Z0-9_]	Matches any alphanumeric character
\W	[^a-zA-Z0-9_]	Matches any non-alphanumeric character

Repeater {} Usage

Pattern	Meaning
{ 2 }	Pattern occur two times continuously
{2,5}	Pattern repeat 2 or 3 or 4 or 5 times
{2,}	Pattern should repeat at least 2 times (2 times or +2 times)
{,2}	Pattern should repeat at most 2 times only (0 time or 1 time or 2 times)

Single Character Matching

Description	Pattern	Example
Directly Specify	1.@ 2. A 3. \.	1. @ 2. A 3. .
Digit	[0-9]	9
Alphabetic	[a-zA-Z] or [aA-zZ]	Z
Alpha-Numeric	[0-9aA-zZ]	0
Space	[ ] or \s	
Any Character	.	-
This or That	a | b	1.a 2.b

Multiple Character Matching

Description	Pattern	Example
Directly Specify	1.@gmail 2. Ayyappan 3. \.com	1. @gmail 2. Ayyappan 3. .com
Digit	[0-9]+	90
Alphabetic	[a-zA-Z]+ or [aA-zZ]+	Abc
Alpha-Numeric	[0-9aA-zZ]+	A00
Space	[ ]+ or \s	
Any Character	.*	A#2
This or That	[a-z]+ | [0-9]+	1.ayyappan 2.265054

How to write pattern?

- List out all the possible mandatory criteria
- List out all the possible optional criteria
- Write simple pattern for all those criteria
- Join one by one based on the matching positions

Example: Email Validation

S.No	List	Pattern
1	Start with alpha lower case character	^[a-z]
2	Next characters should be alpha numeric characters with dot & underscore characters allowed. Mail Id should have atleast more then 3 characters	[aA-zZ._]{2,}
3	@ character should present	@
4	Domain name should more than 2 alpha numeric characters with - character	[aA-zZ0-9\-]{2,}
5	. Character should present next	\.
6	2 or three alpha character represent domain type	[a-z]{2,3}
7	. Character may be present next (It is optional)	\.
8	2 alpha characters may be present next (it is also optional)	[a-z]{2}

```
^[a-z] [aA-zZ._]{2,}@[aA-zZ0-9\-]{2,}\. [a-z]{2,3}(\.[a-z]{2})?$
```

re Module in Python

compile(pattern, flags=0)	Compile a regular expression pattern into a regular expression object
search(pattern, string, flags=0)	Scan through string looking for the first location where the regular expression pattern produces a match, and return a corresponding match object
match(pattern, string, flags=0)	If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding match object
fullmatch(pattern, string, flags=0)	If the whole string matches the regular expression pattern, return a corresponding match object
split(pattern, string, maxsplit=0, flags=0)	Split string by the occurrences of pattern
findall(pattern, string, flags=0)	Return all non-overlapping matches of pattern in string, as a list of strings
finditer(pattern, string, flags=0)	Return an iterator yielding match objects over all non-overlapping matches for the RE pattern in string
sub(pattern, repl, string, count=0, flags =0)	Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl
subn(pattern, repl, string, count=0, flags=0)	Perform the same operation as sub(), but return a tuple
escape(pattern)	Escape all the characters in pattern except ASCII letters, numbers and '_'
purge()	Clear the regular expression cache.
error(msg, pattern=None, pos=None)	Exception raised when a string passed to one of the functions here is not a valid regular expression

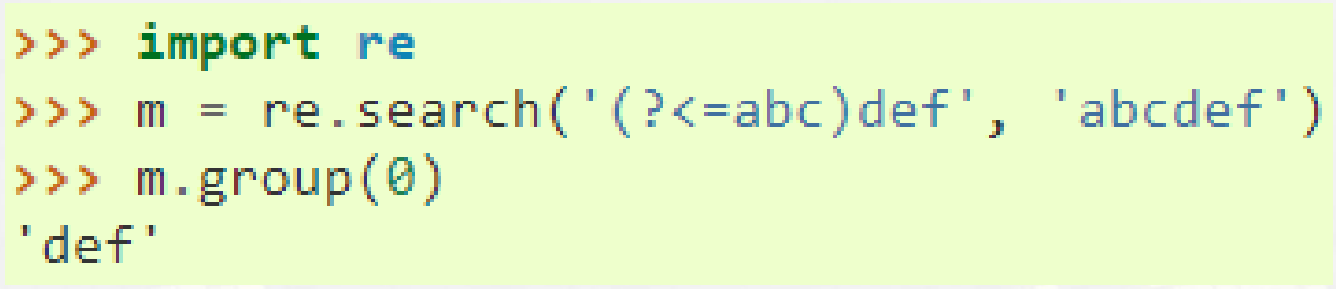
Regular Expression Object

search(*string*[, *pos*[, *endpos*]])	Scan through string looking for the first location where this regular expression produces a match, and return a corresponding match object
match(string[, pos[, endpos]])	If zero or more characters at the beginning of string match this regular expression, return a corresponding match object
fullmatch(string[, pos[, endpos]])	If the whole string matches this regular expression, return a corresponding match object
split(string, maxsplit=0)	Identical to the split() function, using the compiled pattern
findall(string[, pos[, endpos]])	Similar to the findall() function, using the compiled pattern, but also accepts optional pos and endpos parameters
finditer(string[, pos[, endpos]])	Similar to the finditer() function, using the compiled pattern, but also accepts optional pos and endpos parameters
sub(repl, string, count=0)	Identical to the sub() function, using the compiled pattern.
subn(repl, string, count=0)	Identical to the subn() function, using the compiled pattern.

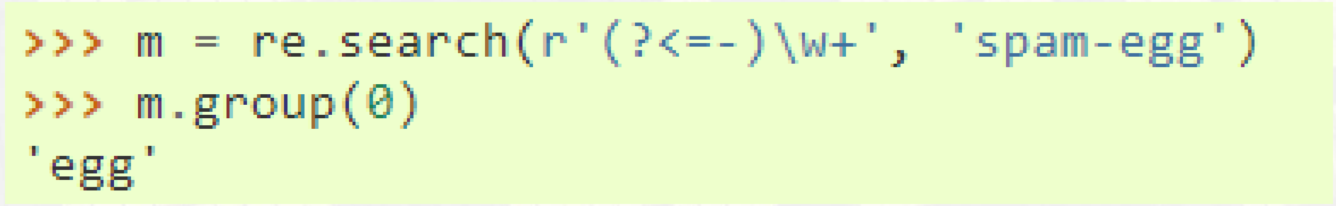
Match Object

expand(template)	Return the string obtained by doing backslash substitution on the template string template
group([group1, …])	Returns one or more subgroups of the match. If there is a single argument, the result is a single string
.__getitem__(g)	This is identical to m.group(g). This allows easier access to an individual group from a match
groups(default=None)	Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern.
groupdict(default=None)¶	Return a dictionary containing all the named subgroups of the match, keyed by the subgroup name.
start([group])	Return the indices of the start and end of the substring matched by group; group defaults to zero (meaning the whole matched substring)
end([group])	
span([group])	For a match m, return the 2-tuple (m.start(group), m.end(group))

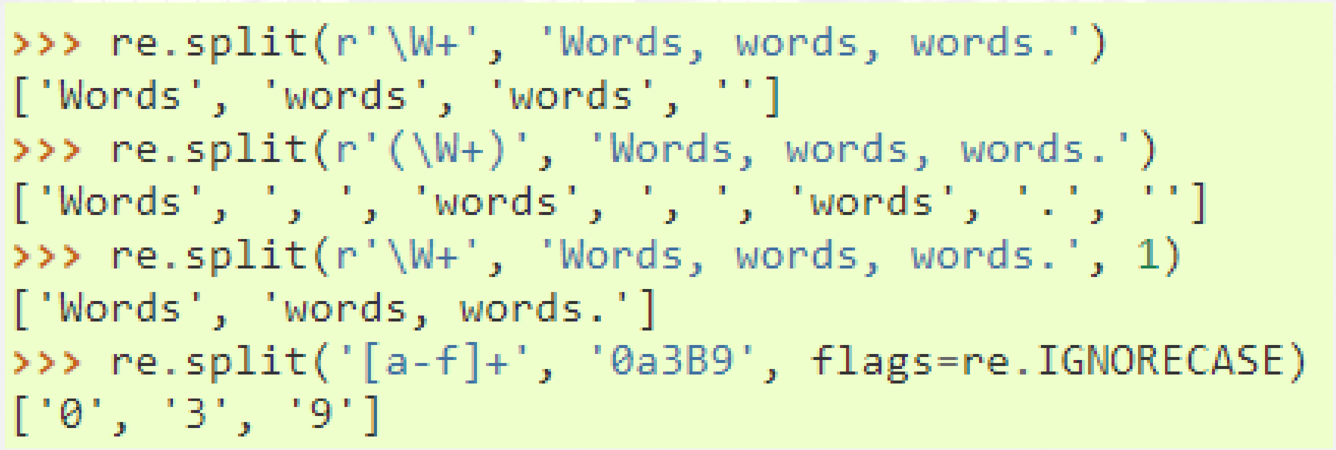
Examples



```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```



```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```



```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<_sre.SRE_Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<_sre.SRE_Match object; span=(1, 2), match='o'>
```

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Query ??

Ayyappan Murugesan
Phone: 7010774329
Call: 8870483557