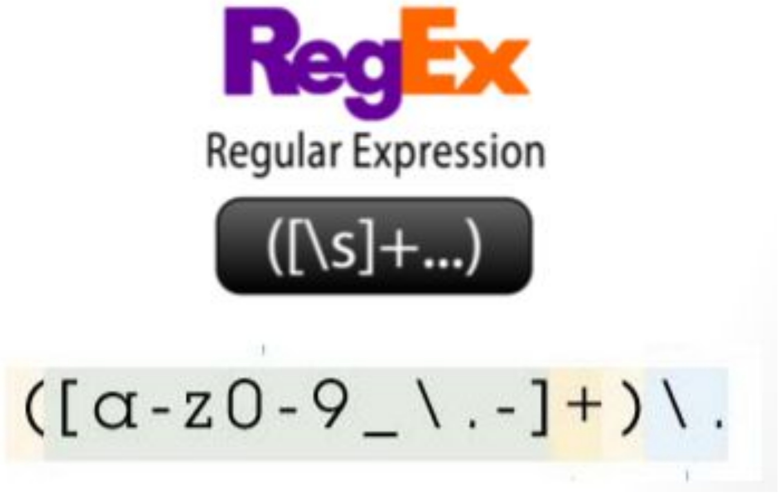
Topic 2 (Pt 2): Python Regular Expressions for NLP



Python RE

- A *regular expression* is a special sequence of characters that match or find other strings or sets of strings, using a specialized syntax held in a pattern.
- The Python **re** (called REs, or regexes, or regex patterns) is a tiny but highly specialized programming language embedded inside Python that provides full support for Perl-like regular expressions, made available through the re module.

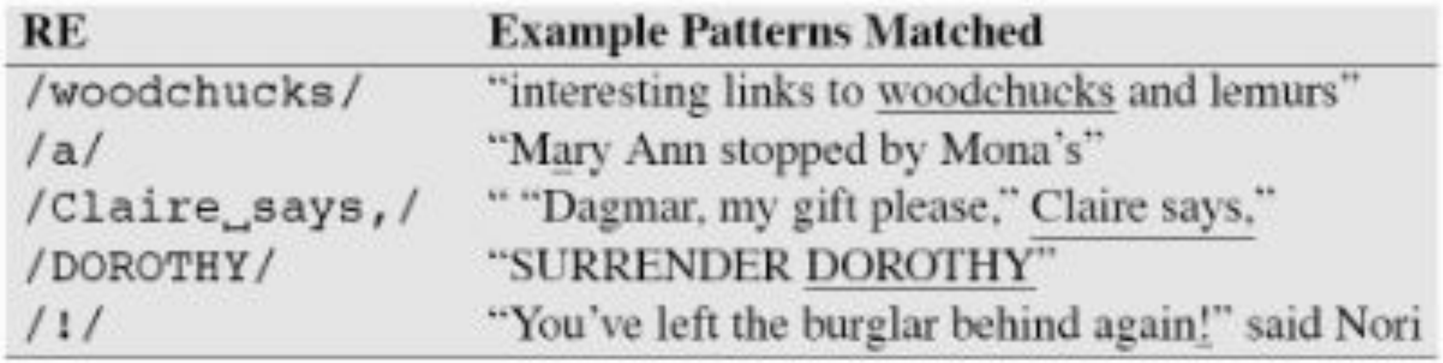
Python RE (cont)

- Can be used with specified rules for the set of possible strings that you want to match; (ie., English sentences, e-mail addresses, html addresses, etc..).
- Ask questions such as:
  - "Does this string match the pattern?", **OR**
  - "Is there a match for the pattern anywhere in this string?"
- Can also be used to modify/replace/split strings
  - https://docs.python.org/3/howto/regex.html
  - https://docs.python.org/3/library/re.html

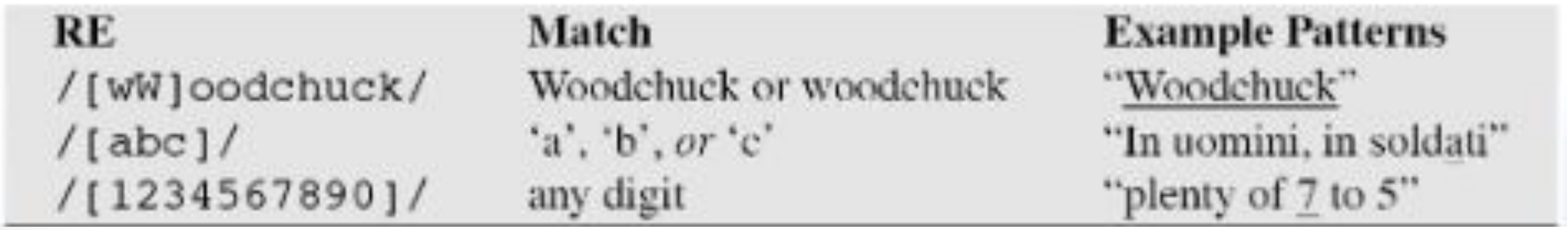
Python RE (cont)

- There are various characters, which would have special meaning when they are used in regular expression.
- To avoid any confusion while dealing with regular expressions, we would use Raw Strings as r'expression‘ (the letter ‘r’, followed by the intended expression).

Common RE in Perl/Python, Unix Grep



RE	Example Patterns Matched
/woodchucks/	"interesting links to woodchucks and lemurs"
/a/	"Mary Ann stopped by Mona's"
/Claire␣says,/	" "Dagmar, my gift please," Claire says,"
/DOROTHY/	"SURRENDER DOROTHY"
/!/	"You've left the burglar behind again!" said Nori

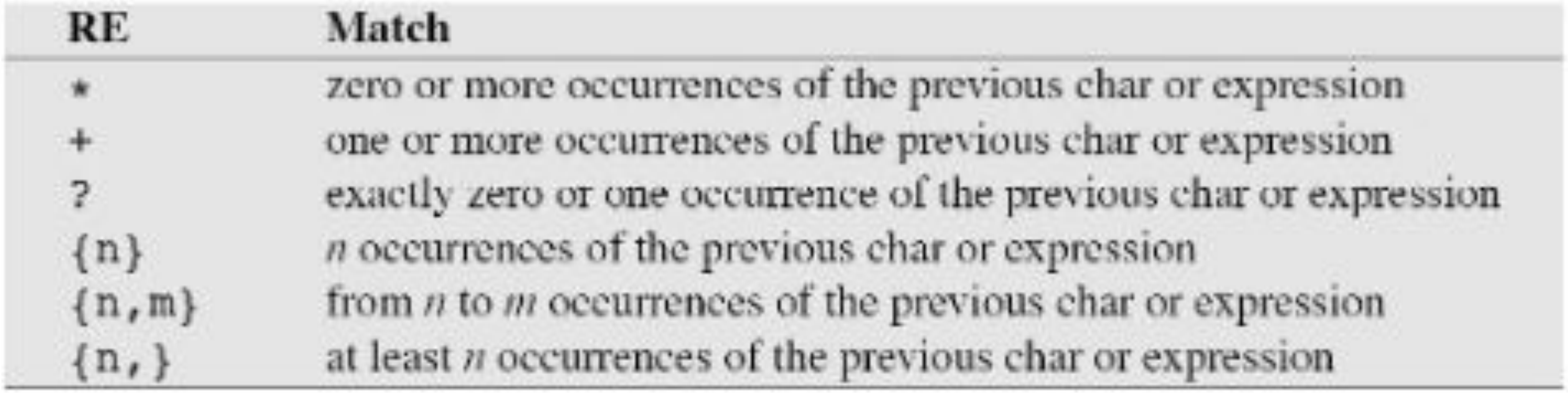


RE	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	"Woodchuck"
/[abc]/	'a', 'b', *or* 'c'	"In uomini, in soldati"
/[1234567890]/	any digit	"plenty of 7 to 5"

A set of digits but match one character at a time, not the whole sequence

`/[bc]oil/`
`matches boil OR coil and NOT bcoil`

Common RE in Perl/Python, Unix Grep



RE	Match
`*`	zero or more occurrences of the previous char or expression
`+`	one or more occurrences of the previous char or expression
`?`	exactly zero or one occurrence of the previous char or expression
`{n}`	*n* occurrences of the previous char or expression
`{n,m}`	from *n* to *m* occurrences of the previous char or expression
`{n,}`	at least *n* occurrences of the previous char or expression

RE in Python

- Matching characters in Python

metachar	Meaning	Example
.	Matches any char except for newline	beg.n matches any character between 'beg' and 'n'. E.g., begun, begin, beg'n, etc
^	Complement or matches start of string	^z will match beginning of string with 'z' [^z] will match any char except z
$	Matches end of string or before newline	'ther$' will match any word ending with '…ther'
*	Matches 0 or more repetition	xy* matches x, xy or x followed by any no. of y's
+	Matches 1 or more repetition	ab+ matches ab and a followed by at least one b (e.g., abb, abbb, etc…)
?	Matches 0 or 1 repetitions of the preceding RE	ab? will match either 'a' or 'ab'.

RE in Python

metachar	Meaning	Example
{ }	Match exactly the no. of copies stated in { }	m{6} will match exactly 6 'm' characters (i.e, 'mmmmmm')
[ ]	Matches a set of character class	[abc] or [a-c] will match either a,b, or c [a-z] will match either a thru z (case sensitive)
\	Escape metacharacters	\[will match '[' in a pattern. E.g: [aeiou**\[**],will match a,e,i,o,u or [
|	Match either expression on each side	A|B will match either A or B
( )	Matches whatever regular expression inside parentheses	(abcdef) will match 'abcdef' (the whole sequence in parentheses)

Advanced Operators

metachar	Expansion	Match	Example
\d	[0-9]	Any digit	There are 5 cats
\D	[^0-9]	Any non-digit	red ball
\w	[a-zA-Z0-9]	Any alphanumeric or space	RM9
\W	[^\w]	A non-alphanumeric	####
\s	[\r\t\n\f]	Whitespace, space or tab	
\S	[^\s]	Non whitespace	in school

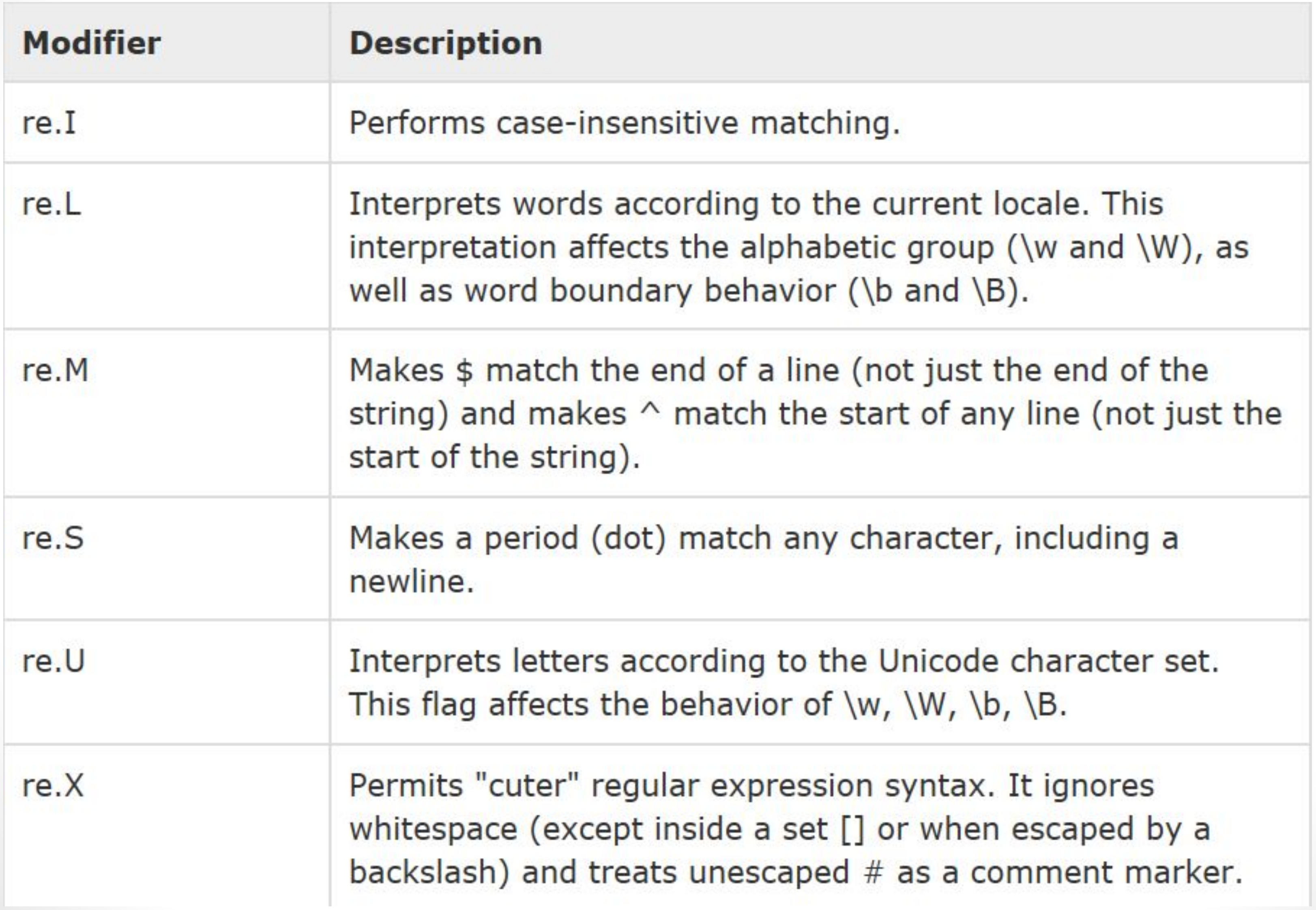
Python re.match( )

- This function attempts to match RE *pattern* to a *string* with optional *flags*.
- Syntax for match() with 2 required and 1 optional arguments/parameters:

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

Param	**Description**	**Example**
pattern	Regular expression or pattern to be matched	pattern re.match(‘[bcs]oil’, "Boiling some water", re.I) string flag
string	String to be searched for matching with pattern	
flags	Indicate different flags using bitwise OR (|). These are modifiers (explained in Table 3)	

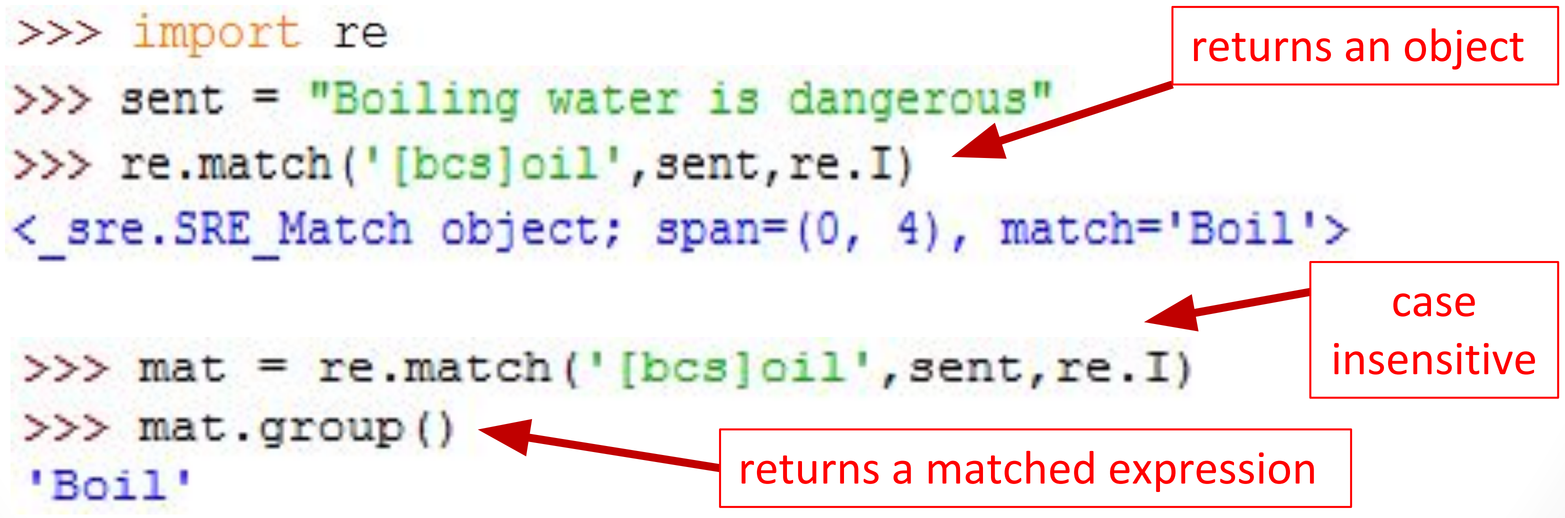
RE modifiers: Option flags (Table 3)



Modifier	Description
re.I	Performs case-insensitive matching.
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).
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).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
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.

Python re.match( )

- The *re.match* function returns a **match** object(single match) on success, **None** on failure.
- Use *group(num)* or *groups()* function of **match** object to get matched expression



```
>>> import re
>>> sent = "Boiling water is dangerous"
>>> re.match('[bcs]oil',sent,re.I)
<_sre.SRE_Match object; span=(0, 4), match='Boil'>
```

```
>>> mat = re.match('[bcs]oil',sent,re.I)
>>> mat.group()
'Boil'
```

RE : Matched subgroups

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any)

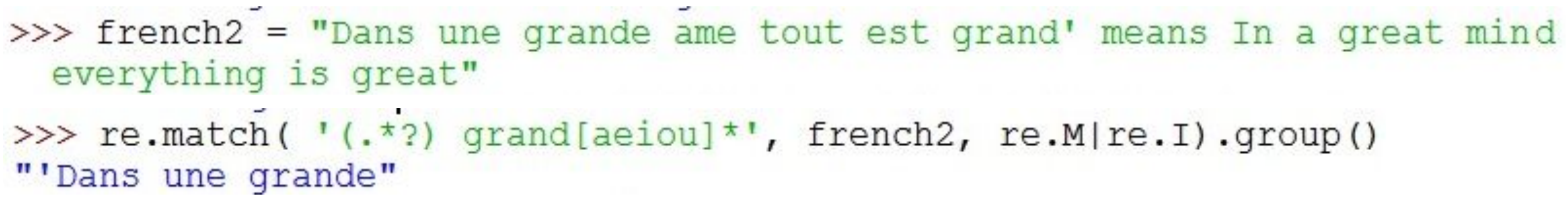
```
>>> import re
>>> text = "A scripting language is more suitable for NLP than a compiled language"
>>> searchObj = re.search( r'(.*) is (.*?) .*', text, re.M|re.I)

>>> if searchObj:
        print("searchObj.group() : ", searchObj.group())
        print("searchObj.group(1) : ", searchObj.group(1))
        print("searchObj.group(2) : ", searchObj.group(2))
    else:
        print("Found nothing!!")
```

```
searchObj.group() :  A scripting language is more suitable for NLP than a compiled language
searchObj.group(1) :  A scripting language
searchObj.group(2) :  more
```

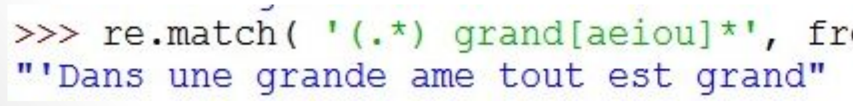
Python re.match( )

- More examples of matched expressions :
  - match a string that begins with with 0 or exactly 1 string of any combination, followed by the word “grand”, “granda”, “grande”, “grandi”, “grando” or “grandu”:



```
>>> french2 = "Dans une grande ame tout est grand' means In a great mind
  everything is great"
>>> re.match( '(.*?) grand[aeiou]*', french2, re.M|re.I).group()
"'Dans une grande"
```

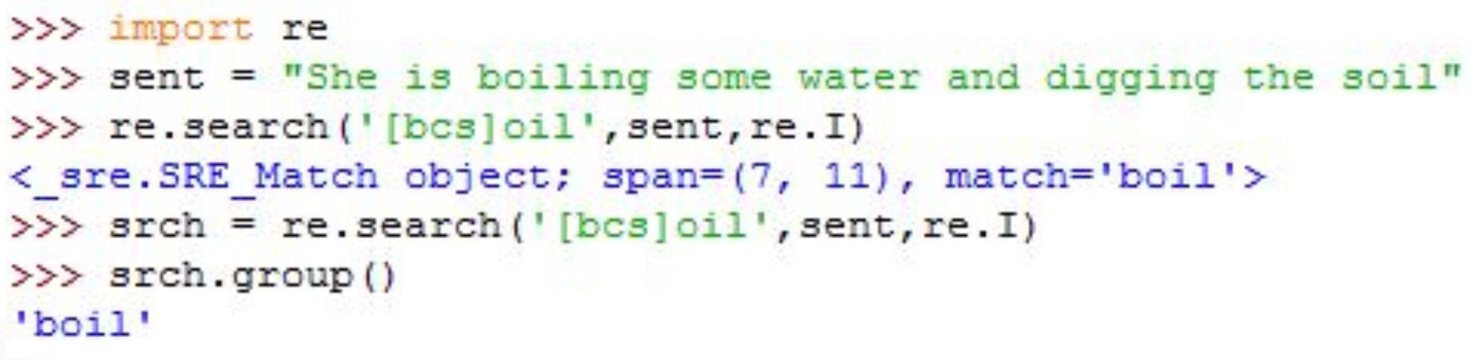
  - match the string that starts with 0 or more strings of any combination, followed by the word “grand”, “granda”, “grande”, “grandi”, “grando” or “grandu”:



```
>>> re.match( '(.*) grand[aeiou]*', french2, re.M|re.I).group()
"'Dans une grande ame tout est grand"
```

Python search( )

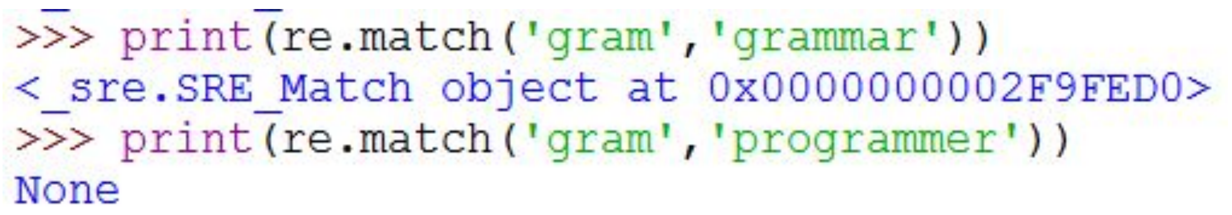
- re.search() searches for first occurrence of RE *pattern* within *string* with optional *flags*.
- Returns a **match** object on success, **None** on failure and uses *group(num)* or *groups()* to get matched expression
- Syntax : re.search(pattern, string, flags = 0)



```
>>> import re
>>> sent = "She is boiling some water and digging the soil"
>>> re.search('[bcs]oil',sent,re.I)
<_sre.SRE_Match object; span=(7, 11), match='boil'>
>>> srch = re.search('[bcs]oil',sent,re.I)
>>> srch.group()
'boil'
```

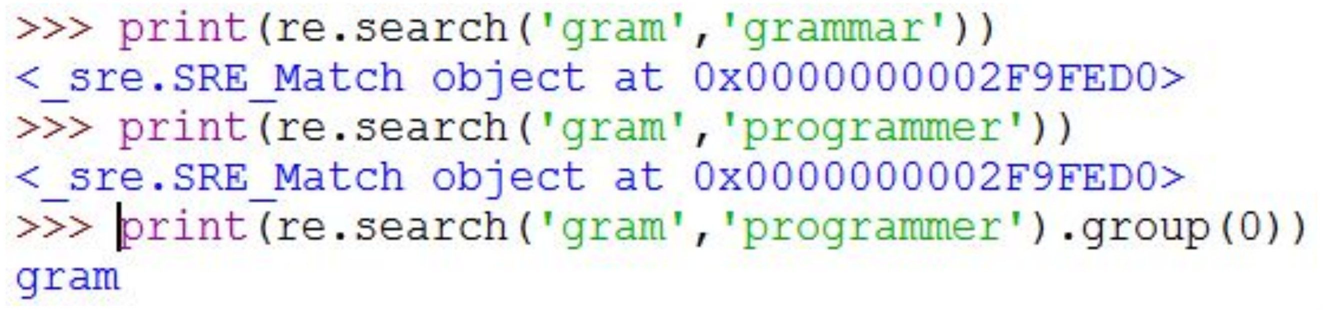
RE match( ) vs search( )

- re.match() function only checks if the RE matches at the beginning of the string
- match() only reports a successful match which starts at position 0;



```
>>> print(re.match('gram','grammar'))
<_sre.SRE_Match object at 0x0000000002F9FED0>
>>> print(re.match('gram','programmer'))
None
```

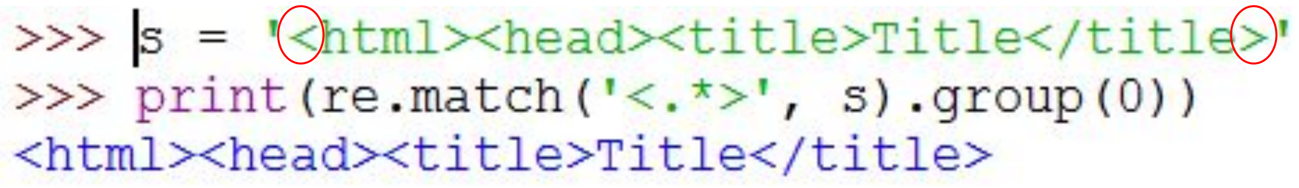
- re.search() scans forward through the string for a match.



```
>>> print(re.search('gram','grammar'))
<_sre.SRE_Match object at 0x0000000002F9FED0>
>>> print(re.search('gram','programmer'))
<_sre.SRE_Match object at 0x0000000002F9FED0>
>>> print(re.search('gram','programmer').group(0))
gram
```

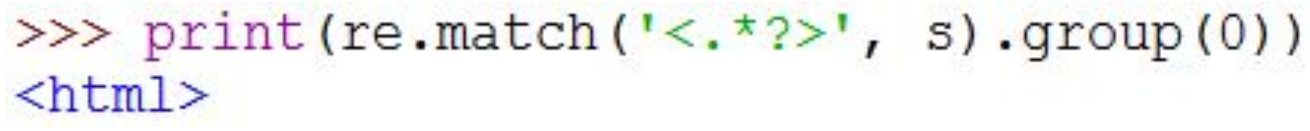
Greedy vs Ungreedy

- Repeating a regular expression as in a* attempts to consume as much of the pattern as possible because of its greedy nature .*.



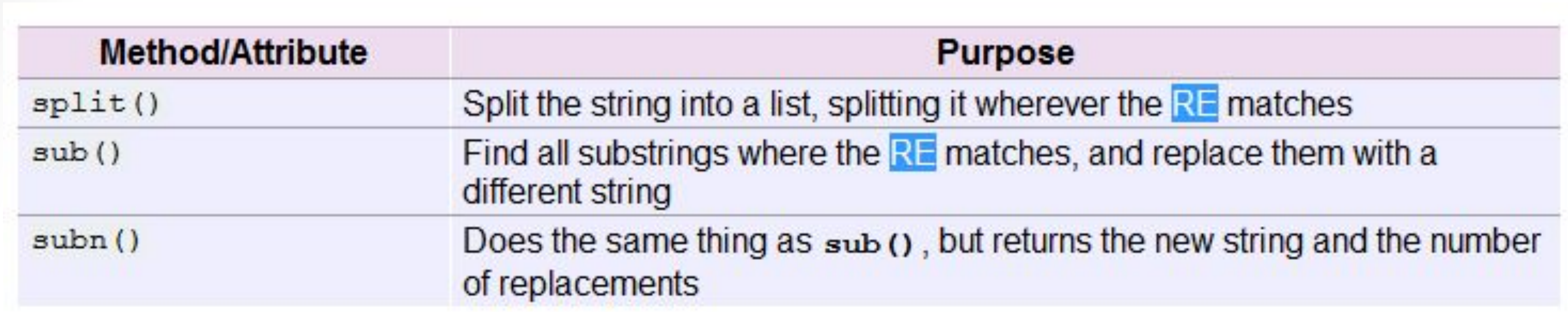
```
>>> s = '<html><head><title>Title</title>'
>>> print(re.match('<.*>', s).group(0))
<html><head><title>Title</title>
```

- Use the non-greedy qualifiers *?, +?, ??, or {m,n}?, which match as little text as possible.



```
>>> print(re.match('<.*?>', s).group(0))
<html>
```

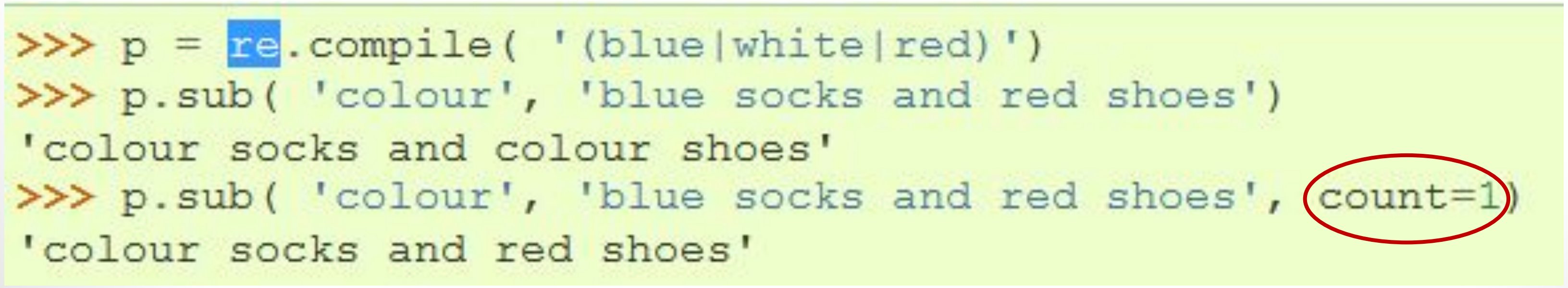
Modifying Strings



Method/Attribute	Purpose
`split()`	Split the string into a list, splitting it wherever the RE matches
`sub()`	Find all substrings where the RE matches, and replace them with a different string
`subn()`	Does the same thing as `sub()`, but returns the new string and the number of replacements

- *.sub* Returns the string obtained by replacing the leftmost non-overlapping occurrences of the RE in *string* by the replacement *replacement*

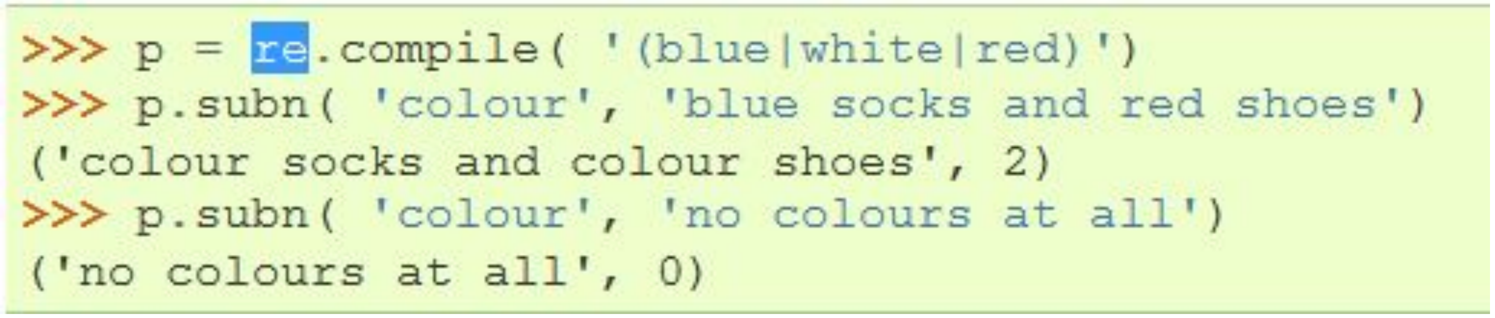
Another way of defining a pattern is to compile it first.



```
>>> p = re.compile( '(blue|white|red)')
>>> p.sub( 'colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub( 'colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

Modifying Strings

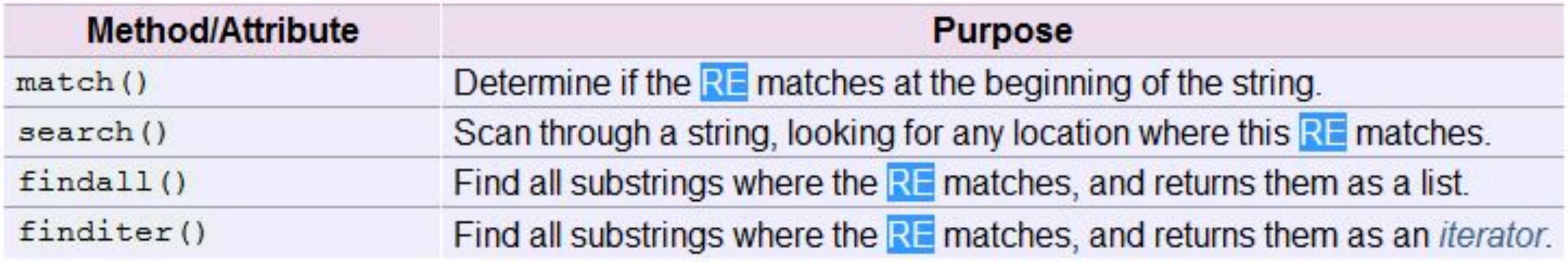
- The *.subn()* method does the same work, but returns a 2-tuple containing the new string value and the number of replacements performed



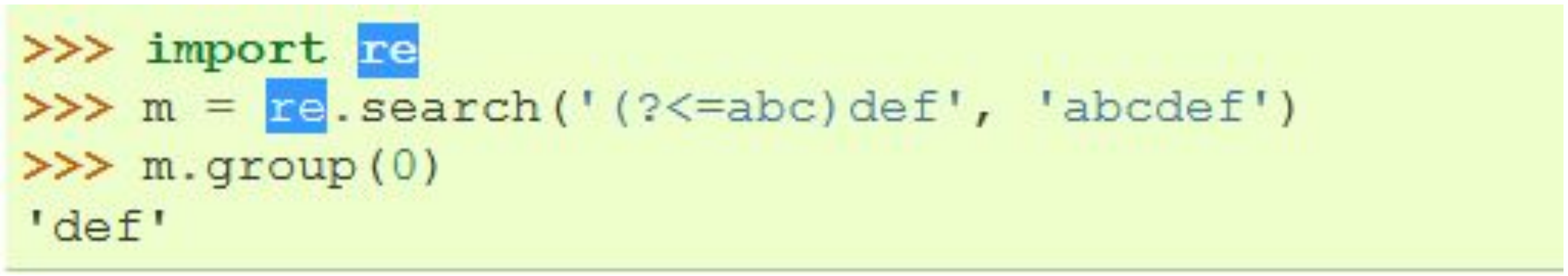
```
>>> p = re.compile( '(blue|white|red)')
>>> p.subn( 'colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn( 'colour', 'no colours at all')
('no colours at all', 0)
```

RE in Python

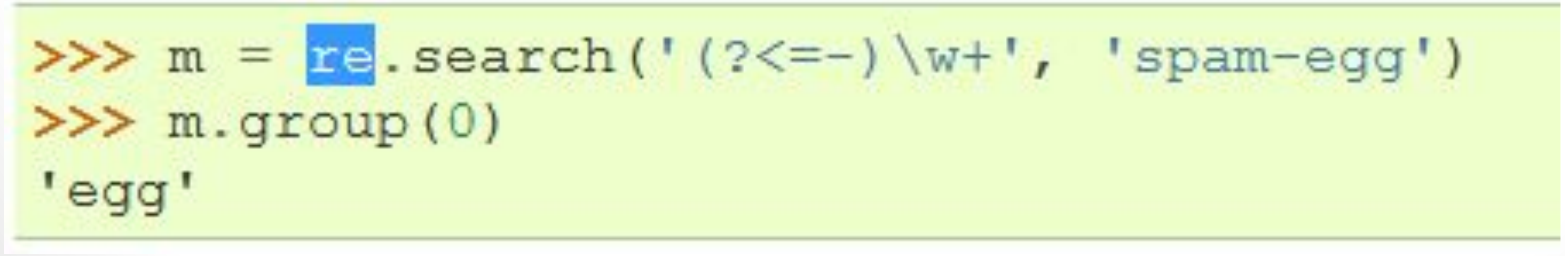
- More RE methods



Method/Attribute	Purpose
`match()`	Determine if the RE matches at the beginning of the string.
`search()`	Scan through a string, looking for any location where this RE matches.
`findall()`	Find all substrings where the RE matches, and returns them as a list.
`finditer()`	Find all substrings where the RE matches, and returns them as an *iterator*.



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



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

Example problem for RE in NLP

- Write a regular expression that finds all the instances of the word ‘the’ in the text:

“The boy eats another cake in the kitchen”

/the/ : This pattern only finds lower case ‘the’

/[tT]he/ : This pattern finds all words with ‘the’ or ‘The’ in it

/\b[tT]he\b/ : This pattern finds all ‘the’ or ‘The’ (\b matches boundaries)

Example solution for RE in NLP

- Matching 'the' that precedes an underscore or numbers (e.g., "the_" or "the25"):

  /[^a-zA-Z][tT]he[^a-zA-Z]/

- Problem: This pattern will not match *the* when it begins a line or ends a line (e.g., *"The book…"* or *"… the."*)
- Solution:

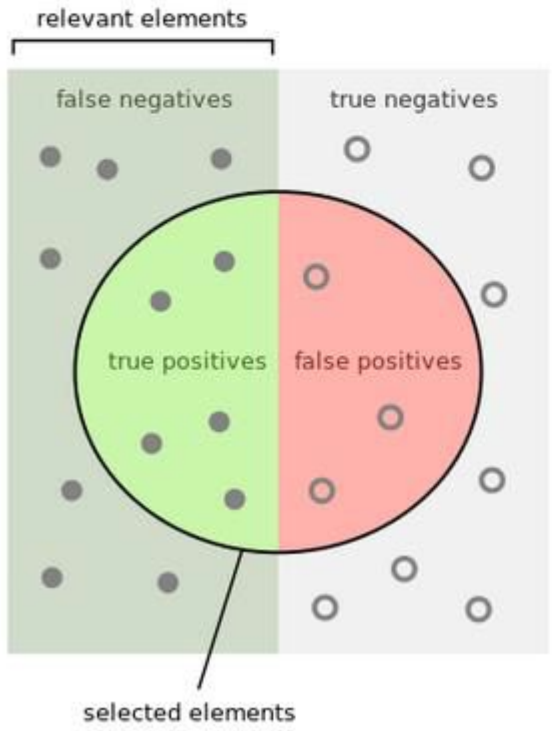
  /(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)/

RE Errors

- In RE, the process involves fixing two kinds of errors:
  - Matching strings that we should not have matched (*there*, *then*, *other*)
    - False positives (Type I error)
  - Not matching strings that we should have matched (*The*):
    - False negatives (Type II error)

Errors

- Reducing error rates for an application involves 2 efforts:
  - Increase accuracy or precision (i.e., minimize false positives)
  - Increase coverage or recall (i.e., minimize false negatives)


relevant elements
false negatives
true negatives
true positives
false positives
selected elements