Doğal Dil İşlemeye Giriş

Metin İşleme-Düzenli İfadeler
(Text Processing - Regular Expressions)

Starting with Python

Python is a simple yet powerful programming language with excellent functionality for processing linguistic data. Python can be downloaded for free from http://www.python.org/.

Starting with Python

- Here is a five-line Python program that processes `file.txt` and prints all the words ending in `ing`:

```
>>> for line in open("file.txt"):
...     for word in line.split():
...         if word.endswith('ing'):
...             print (word)
```

Starting with Python

- Reading Local Files:

```
>>> f = open('document.txt')
>>> raw = f.read()
Or
>>> raw = open('document.txt‘).read()
```

Starting with Python

- **To see if the file is in the right directory**

```
>>> import os
>>> os.listdir('.')
['ashishg.txt', 'deneme.txt', 'DLLs', 'Doc', 'HW3.txt',
'include', 'Lib', 'libs', 'python.exe', 'python2.7.exe',
'python2.exe', 'pythonw.exe', 'pythonw2.7.exe',
'pythonw2.exe', 'Scripts', 'stop_words.txt', 'tcl', 'Tools',
'w9xpopen.exe']
```

Starting with Python

- **To see if the file is in the right directory**

```
>>> import os
>>> os.getcwd()                    #Get working directory
'C:\\Python27‘
>>> os.chdir("/temp")              #change directory
>>> os.getcwd()
'C:\\Temp‘
```

Starting with Python

- **To print the file**

```
>>> f = open('document.txt')
>>> raw = f.read()
>>> print raw
```

OR

```
>>> f = open('document.txt')
>>> print f.read()
```

Regular Expressions

- A formal language for specifying text strings
- How can we search for any of these?
  - woodchuck
  - woodchucks
  - Woodchuck
  - Woodchucks …

Regular Expressions: Disjunctions

- Letters inside square brackets []

Pattern	Matches
`[wW]oodchuck`	Woodchuck, woodchuck
`[1234567890]`	Any digit

- Ranges [A-Z]

Pattern	Matches	
`[A-Z]`	An upper case letter	`Drenched Blossoms`
`[a-z]`	A lower case letter	`my beans were impatient`
`[0-9]`	A single digit	`Chapter 1: Down the Rabbit Hole`

Regular Expressions: Negation in Disjunction

- Negations [^Ss]
  - Caret means negation only when first in []

Pattern	Matches	
[^A-Z]	Not an upper case letter	Oyfn pripetchik
[^Ss]	Neither 'S' nor 's'	I have no exquisite reason"
[^e^]	Neither e nor ^	Look here
a^b	The pattern a caret b	Look up a^b now

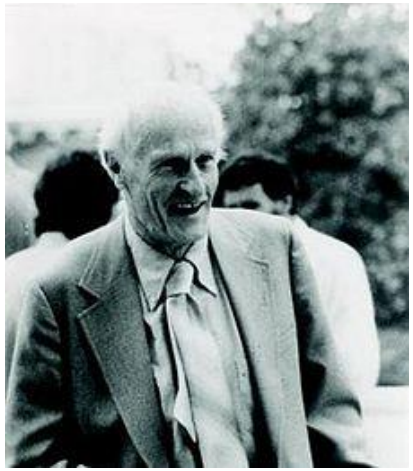
Regular Expressions: More Disjunction

- Woodchucks is another name for groundhog!
- The pipe | for disjunction

Pattern	Matches
`groundhog|woodchuck`	
`yours|mine`	yours mine
`a|b|c`	= [abc]
`[gG]roundhog|[Ww]oodchuck`	

Regular Expressions: ? * + .

Pattern	Matches	
`colou?r`	Optional previous char	color colour
`oo*h!`	0 or more of previous char	oh! ooh! oooh! ooooh!
`o+h!`	1 or more of previous char	oh! ooh! oooh! ooooh!
`baa+`		baa baaa baaaa baaaaa
`beg.n`	Any character	begin begun begun beg3n



Stephen C Kleene

Kleene *, Kleene +

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

- Using re
- The **re** module provides an interface to the regular expression engine, allowing you to compile rEs into objects and then perform matches with them.

- Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```
>>> import re
>>> p = re.compile('ab*')
>>> p
<_sre.SRE_Pattern object at 0x...>
```

Regular Expressions: Anchors ^ $

Pattern	Matches
^[A-Z]	Palo Alto
^[^A-Za-z]	1 "Hello"
\.$	The end.
.$	The end? The end!

```
patt = re.compile('^[^A-Za-z]')

patt.findall('This is a Arg')
Out[76]: []

patt.findall('1 This is a Arg')
Out[77]: ['1']

patt.findall("Hello")
Out[78]: []
```

- Hands on

```
>>> import re
>>> p = re.compile('\.$')     #$ matches end of line
>>> q = re.compile('.$')

>>> print p.search('Is there anybody? Hellooo!! By.')
<_sre.SRE_Match object at 0x02A9B4F0>

>>> print p.findall('Is there anybody? Hellooo!! By.')
['.']
>>> print q.findall('Is there anybody? Hellooo!! By?')
[‘?']
>>> print (p.findall('Is there anybody? Hellooo!! By?'))
[]
```

https://docs.python.org/3/library/re.html?highlight=regex

Example

- Find me all instances of the word “the” in a text.

```
the
```

Misses capitalized examples

```
[tT]he
```

Incorrectly returns `other` or `theology`

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

```
>>> q = re.compile('[^a-zA-Z][tT]he[^a-zA-Z]')
>>> print q.findall('There thee the The this and that the')
[' the ‘]
```

OR

```
>>> p = re.compile('\s[Tt]he\s')          # \s is for space
>>> print p.findall('There thee the The this and that the')
[' the ']
```

Errors

- The process we just went through was based on fixing two kinds of errors
  - Matching strings that we should not have matched (there, then, other)
    - False positives (Type I)
  - Not matching things that we should have matched (The)
    - False negatives (Type II)

Errors cont.

- In NLP we are always dealing with these kinds of errors.
- Reducing the error rate for an application often involves two antagonistic efforts:
  - Increasing accuracy or precision (minimizing false positives)
  - Increasing coverage or recall (minimizing false negatives).

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

- Regular expressions are used to operate on strings, we'll begin with the most common task: matching characters.
- Some characters are special *metacharacters*, and don't match themselves.
- . ^ $ * + ? { } [ ] \ | ( )
- Metacharacters are not active inside classes. For example, [akm$] will match any of the characters 'a', 'k', 'm', or '$'; '$' is usually a metacharacter, but inside a character class it's stripped of its special nature.

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

[ ] : specify a set of characters you are trying to find match
[a-z], [abc], [a-c]

^ : complement: [^5] will match any character but 5

\ : escape - if you need to match a [or \, you can precede them with a backslash to remove their special meaning: \[or \\.

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

‘.’ : will match any character except new line.

| : Alternation, or the “or” operator. If A and B are regular expressions, A|B will match any string that matches either A or B. | has very low precedence in order to make it work reasonably when you’re alternating multi-character strings.Crow|Servo will match either Crow or Servo, not Cro, a 'w' or an 'S', and ervo.

^ : Matches at the beginning of lines. Unless the **MULTILINE** flag has been set, this will only match at the beginning of the string. In **MULTILINE** mode, this also matches immediately after each newline within the string.

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

For example, if you wish to match the word 'From' only at the beginning of a line,

the RE to use is ^From.

```
>>> print (re.search('^From', 'From Here to Eternity') )
<_sre.SRE_Match object at 0x...>
>>> print (re.search('^From', 'Reciting From Memory'))
None
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

$: Matches at the end of a line, which is defined as either the end of the string, or any location followed by a newline character.

```
>>> print re.search('}$', '{block}')
<_sre.SRE_Match object at 0x...

>>>> print re.search('}$', '{block} ')                #space after }
None

>>> print re.search('}$', '{block}\n')
<_sre.SRE_Match object at 0x...>
```

To match a literal '$', use \$ or enclose it inside a character class, as in [$].

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

\A: Matches only at the start of the string. When not in **MULTILINE** mode, \A and ^ are effectively the same. In **MULTILINE** mode, they're different: \A still matches only at the beginning of the string, but ^ may match at any location inside the string that follows a newline character.

\Z: Matches only at the end of the string.

\b: Word boundary. This is a zero-width assertion that matches only at the beginning or end of a word. A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

The following example matches class only when it's a complete word; it won't match when it's contained inside another word.

```
>>> p = re.compile(r'\bclass\b')          # r:raw text; \b: word boundry
>>> print (p.search('no class at all'))
<_sre.SRE_Match object at 0x...>

>>> print (p.search('the declassified algorithm'))
None

>>> print (p.search('one subclass is'))
None
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

In Python's string literals, \b is the backspace character, ASCII value 8. If you're not using raw strings, then Python will convert the \b to a backspace, and your RE won't match as you expect it to. The following example looks the same as our previous re, but omits the 'r' in front of the re string.

```
>>> p = re.compile('\bclass\b')
>>> print p.search('no class at all')
None
>>> print p.search('\b' + 'class' + '\b')
<_sre.SRE_Match object at 0x...>
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

\B : Another zero-width assertion, this is the opposite of \b, only matching when the current position is not at a word boundary.

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

- Repeating
- * specifies that the previous character can be matched zero or more times, instead of exactly once. For example,
  - ca*t will match ct (0 a characters), cat (1 a), caaat (3 a characters)

Repetitions such as * are *greedy*. A step-by-step example will make this more obvious. Let's consider the expression **a[bcd]*b**. This matches the letter 'a', zero or more letters from the class [bcd], and finally ends with a 'b'. Now imagine matching this RE against the string **abcbd**.

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

Matching re a[bcd]*b against the string 'abcbd'

Step	Matched	Explanation
1	`a`	The `a` in the RE matches.
2	`abcbd`	The engine matches `[bcd]*`, going as far as it can, which is to the end of the string.
3	*Failure*	The engine tries to match `b`, but the current position is at the end of the string, so it fails.
4	`abcb`	Back up, so that `[bcd]*` matches one less character.
5	*Failure*	Try `b` again, but the current position is at the last character, which is a `'d'`.
6	`abc`	Back up again, so that `[bcd]*` is only matching `bc`.
6	`abcb`	Try `b` again. This time the character at the current position is `'b'`, so it succeeds.

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

+ : which matches one or more times. ca+t will match cat (1 a), caaat (3 a's), but won't match ct.

?, matches either once or zero times; you can think of it as marking something as being optional. For example, home-?brew matches either homebrew or home-brew.

{m,n}, where *m* and *n* are decimal integers. This qualifier means there must be at least *m* repetitions, and at most *n*. For example, a/{1,3}b will match a/b, a//b, and a///b. It won't match ab.

{0,} is the same as *,

{1,} is equivalent to +, and

{0,1} is the same as ?.

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

- To write a re that matches the string \section, which might be found in a LaTeX file
  - you must escape any backslashes and other metacharacters by preceding them with a backslash, resulting in the string \\section.
  - The resulting string that must be passed to **re.compile()** must be \\section. However, to express this as a Python string literal, both backslashes must be escaped *again*.

Regular Expre

(http://docs.python.org/2/howt

Characters	Sta
\section	Text string to be match
\\section	Escaped backslash for
"\\\\section"	Escaped backslashes for a string literal

Regular String	Raw string
"ab*"	r"ab*"
"\\\\section"	r"\\section"
"\\w+\\s+\\1"	r"\w+\s+\1"

- use Python's raw string notation for regular expressions; backslashes are not handled in any special way in a string literal prefixed with 'r', so
- r"\n" is a two-character string containing '\' and 'n', while
- "\n" is a one-character string containing a newline.

re "\\\\section" equivalent to raw r"\\section"

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

- Performing matches

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*.

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

- **match()** and **search()** return **None** if no match can be found. If they're successful,
- a *match object* instance is returned, containing information about the match: where it starts and ends, the substring it matched, and more.
- RE for representing e-mail : \b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b

```
>>> text = open('ashishg.txt').read()
>>> import re
>>> p = re.compile('\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b')
>>> m = p.match(text)
>>> print m
None
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

Extracting e-mails

```
>>> p = re.compile('[A-Za-z0-9._%+-]+@[A-Za-z0-9]+\.[A-Za-z0-9.]+')
>>> text = “Here are some contact information: Bahar Kara e-mail:
    bkara@gmail.com, tel no: (232) 311 3204. Cemre Candemir: e-mail:
    cemre.demir@erbakan.edu.tr. Her phone number is 0(505) 3328999. Here
    505-332-4540, another +90(231)3747576."
>>> liste = p.findall(text)
>>> liste
['bkara@gmail.com', 'cemre.demir@erbakan.edu.tr.']
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

Extracting phone numbers

```
>>> print re.findall ('[- + ( ) \d]*\d+', text)
[' (232) 311 3204', ' 0(505) 3328999', ' 505-332-4540', ' +90(231)3747576']
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

- We can query *match object*.

Method/Attribute	Purpose
`group()`	Return the string matched by the RE
`start()`	Return the starting position of the match
`end()`	Return the ending position of the match
`span()`	Return a tuple containing the (start, end) positions of the match

```
>>> p = re.compile('\w+')      #Matches any alphanumeric character; this is equivalent to the
                               # class [a-zA-Z0-9_]
>>> m = p.match('tempo')
>>> m
<_sre.SRE_Match object at 0x...> #match object is created at address
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

```
>>> m.group()
'tempo'
>>> m.span()
(0, 5)
>>> m.start(), m.end()
(0, 5)
>>>
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

- Since the **match()** method only checks if the RE matches at the start of a string, **start()** will always be zero.
- However, the **search()** method of patterns scans through the string, so the match may not start at zero in that case.

```
>>> print p.match('::: message')
None
>>> m = p.search('::: message'); print m
 <_sre.SRE_Match object at 0x...>
>>> m.group()
'message‘
>>> m.span()
(4, 11)
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

```
>>> p = re.compile('[ab]*')
>>> print (p.match('abababababab').span())
(0, 10)
```

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
        print ('Match found: ', m.group())
else:
        print ('No match‘)
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

```
>>> p = re.compile('\d+') #Matches any decimal digit; this is equivalent to the class [0-9].
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
```

findall() has to create the entire list before it can be returned as the result.
The **finditer()** method returns a sequence of *match object* instances as an *iterator*.

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable-iterator object at 0x...>
>>> for match in iterator:
        print match.span()


(0, 2)
(22, 24)
(29, 31)
```

Regular Expressions

(http://docs.python.org/2/howto/regex.html)

```
>>> for match in p.finditer(text):
        print (text[match.start():match.end()])
```

bkara@gmail.com

cemre.demir@erbakan.edu.tr.