Regular Expression (정규식)

이상엽

1. Regular expression

What is regular expression?

정규식은 특정한 문자들의 조합으로 구성된 패턴(pattern)이라고 보면 됩니다. 특정한 패턴을 만들어서 그 패턴을 만족하는 문자열이 문서 (혹은 텍스트)에 존재하는지를 파악하고, 매치되는 패턴의 텍스트 정보를 추출하는 등의 작업을 하는데 사용하게 됩니다. 즉, 문서에서 특정한 정보를 찾는 경우, 그리고 특정한 정보를 포함하고 있는지를 파악할 때 유용하게 사용할 수 있습니다.

Comment [S11]: a sequence of characters that define a search pattern 라고 정의됩니다.

패턴 (pattern) 만들기

먼저 특정한 문자열의 패턴 (정규식)을 어떻게 만들 수 있는지를 살펴보도록 하겠습니다. 패턴을 만들 때 사용되는 문자는 크게 일반문자와 특수문자로 구분됩니다. 일반문자는 문자(character) 자체가 그 문자 원래의 의미를 갖는 것입니다. 예를 들어, a 는 문자 a, b 는 문자 b 그대로를 의미하며 정규식에서 이러한 character 를 literal 이라고 표현합니다.

특수문자는 metacharacter 라고 표현되는데, 이는 원래 문자의 의미와는 다른 의미를 갖고 있는 character 들을 일컫습니다. 주요 metacharacters 는 다음과 같습니다.[1] metacharacter 가 아닌 character 들은 일반문자 즉, literal 이라고 보면 됩니다.

주요 Metacharacters

- \w -- Matches a word character (e.g., a, b, c 등)
- \W -- Matches a nonword character (e.g., ^, &, $ 등)
- \s -- Matches a whitespace (e.g., a space, tab, new-line 등)
- \S -- Matches a nonwhitespace (숫자, 문자, 그외 기호들)
- \d -- Matches digits. Equivalent to [0-9].
- \D -- Matches nondigits (문자, 공백문자 등)

[1] 보다 자세한 내용은 https://www.tutorialspoint.com/python/python_reg_expressions.htm 를 참조하시기 바랍니다.

■ . -- \n 을 제외한 모든 character

예제

① 패턴: 'abc'

'abc'라는 세개의 character 로 구성이 된 패턴이 있다면 이는 문자그대로 abc 와 매치하게 됩니다. 왜냐하면 a, b, c 모두 일반문자이기 때문이 그렇습니다. 예를 들어서 '123abc456'이라는 문자열에서 'abc'라는 패턴을 찾는다면 123 이후에 abc 가 존재하기 때문에 해당 abc 와 매치하게 됩니다. 하지만, 'abcd'라는 패턴을 찾는다고 한다면 이를 만족하는 문자열이 없기 때문에 아무런 문자열도 매치되지 않습니다.

② 패턴: '\w\w'

'\w\w' 라는 패턴은 두개의 word characters 와 매치되게 됩니다. 그래서 '\w\w'를 'abcd'를 찾는다면 만족하는 것은 'ab', 'cd' 두 개가 매치되게 됩니다. 그리고 만약 '\w\w'를 '우리나라'에서 찾는다면 '우리'와 '나라'가 매치되게 됩니다. 이와 같은 결과가 나오는 이유는 정규식이 작동하는 방식때문에 그렇습니다. '우리나라'의 경우를 살펴보겠습니다. 정규식은 '우리나라' 텍스트의 앞부분에서 부터 '\w\w'라는 패턴을 만족 문자열을 찾습니다. 그러면 먼저 매치가 되는 문자열이 '우리'입니다. 그리고 그 과정을 중단하는 것이 아니라, 남아있는 문자열에 대해서 같은 과정을 반복합니다. 즉, '우리나라'에서 이미 '우리'가 만족되었기 때문에 새로운 탐색과정에서는 '우리'가 제외되고, 나머지 부분인 '나라'만 사용됩니다. '나라' 역시'\w\w' 패턴을 만족하므로 '나라'도 패턴과 매치되는 문자열의 결과로 반환이 됩니다. 그래서 최종 결과가 '우리', '나라'가 됩니다.

[]의 기능

[와]도 정규식에서 사용되는 metacharacter 입니다. 특, 특정한 역할을 하게 됩니다.

[]는 []안에 존재하는 character 들 중에서 어느 하나라도 만족하게 되면 패턴이 만족하게 됩니다. 예를 들어서, '[abc]'라는 패턴이 있다면, 이는 a 또는 b 또는 c 라는 것을 의미합니다. a, b, c 중 하나의 character 만 존재하더라도 해당 패턴을 만족하게 되는 것입니다. 또 다른 예를 들어보겠습니다. 'appl[ea]'라는 패턴을 만들었다면, 이는 apple 또는 appla 둘 모두와 매치되게 됩니다. 즉 'appl[ea]'이라는 패턴은 결국 'apple' or 'appla' 를 의미하는 것입니다.

Hypen 을 사용해서 []안에서 characters 의 range 를 표현할 수 있습니다. 예를 들어, [0-9]라고 표현하면 이는 0, 1, 2, …, 9 를 의미하게 됩니다. 마찬가지로 [a-zA-Z] 라고 표현하면 모든 영문 알파벳이 포함되게 됩니다.

Negation

Comment [S12]: '123abc456' 은 우리가 정보를 찾고자 하는 문서라고 생각할 수 있습니다. 그리고 'abc' 는 정보를 찾는데 사용하고자 하는 정규식 패턴입니다.

Comment [S13]: 즉, 패턴에서 사용된 일반문자 그대로와 매치가 됩니다.

Comment [S14]: \w 라고 하는 두개의 특수문자를 사용했습니다.

[]안에서 사용되는 ^ 기호는 negation 기능을 수행할 수 있습니다. 예를 들어, [^0-9]와 같이 표현하게 되면 숫자가 아닌 모든 character 와 매치되게 됩니다. 또는 appl[^ea]라고 표현한다면 apple 과 appla 가 아닌 appl 로 시작하는 모든 5 개의 연속된 characters 를 만족하게 됩니다.

Comment [S15]: Caret 이라고 표현

Alternation 기능

Alteration 은 여러 개의 표현들 중에서 어느 하나라도 만족하는 것을 찾고자 하는 경우에 사용할 수 있습니다. 이를 위해서는 | (pipe symbol)을 사용하게 됩니다. | 는 or (또는)의 의미를 갖습니다. 예를 들어서 'yes|no' 라는 패턴이 있다면 'yes' 또는 'no' 둘중 하나가 만족하는 경우에 매치가 되게 됩니다.

'License: yes|no'과 같은 패턴이 있는 경우에는 'License: yes' 또는 'no'라는 문자열과 매치가 되게 됩니다. 만약, 우리가 찾고자 하는 것이 'License: yes' 또는'License: no'라고 한다면 이를 위해서는 다음과 같이 소괄호를 사용해서 정규식을 만들어야 합니다.
'License: (yes|no)'
이렇게 하면, 괄호 안의 내용만을 가지고 alternation 을 하게 됩니다.

Quantifier

Quantifier 는 하나의 character 또는 여러개의 characters 가 얼마나 반복되는지를 정의할 때 사용하게 됩니다.

주요한 quantifier 로는 다음과 같은 것들이 있습니다.

? -- 0 or 1

* -- 0 or more

+ -- 1 or more

{n} -- exactly n repetitions

{n, m} -- n <= # repetitions <=#

{ ,n} -- #repetitions <= n

{n, } -- #repetitions >= n

? 는 앞의 character 또는 character set 이 0 번 나오거나 1 번 나오는 것을 의미합니다. 예를 들어서 \d?라는 정규식은 숫자가 한번도 나오지 않거나 1 번 나오는 문자열과 매치되게 됩니다.

예를 들어, 사용자가 전화번호를 다음 3 개의 형태로 입력할 수 있다라고 가정합시다.

555-555-555

555 555 555

555555555

이러한 경우에 이 3 개 형태 모두의 입력 값과 매치하는 정규식은 다음과 같이 만들 수 있습니다.

'\d+[-\s]?\d+[-\s]?\d+'

* 는 앞의 character 또는 character set 이 0 번 또는 그 이상 나오는 것을 의미합니다. 예를 들어서 \d*라는정규식은 숫자가 한번도 나오지 않거나 한번 이상 나오는 문자열과 매치되게 됩니다.

+ 는 앞의 character 또는 character set 이 한번 이상 나오는 것을 의미합니다. 예를 들어서 \d*라는정규식은 숫자가 한번 이상 나오는 문자열과 매치되게 됩니다.

{}는 앞의 character 또는 character set 이 정확한 숫자만큼 반복되는 정규식을 만들고자 하는 경우에 사용합니다. 예를 들어 \d{2}라는 정규식은 숫자가 정확하게 두번 반복된 문자열과 매치되게 됩니다. 중괄호 안에 숫자를 두개 입력할 수도 있습니다. 예를 들어, \d{2,4}라는 정규식은 숫자가 적어도 2 개이상 4 개 이하인 문자열과 매치되게 됩니다.

2. Python 에서의 사용

Python 에서는 re 라는 모듈을 사용해서 regular expression 을 사용할 수 있습니다. 이를 위해서는 다음과 같이 import 하면 됩니다.

```
import re
```

re 의 주요 함수

패턴을 만들어 re 의 함수와 함께 사용할 때는 주로 아래의 2 가지 방법을 사용합니다.

① compile() 사용해서 패턴 만들기

```
pattern = re.compile(r'test')
pattern.function(text)
```

② 패턴을 별도로 만들지 않고, 문자열에서 패턴을 찾는데 사용되는 함수에 직접 argument 로 입력하기

re.function(pattern, text)

방법①이 메모리 관리에 있어서 더 효율적인데 두번째 방법에 비해서 좀 번거로운 면이 있습니다. 만약, 여러분들이 다루는 데이터가 크지 않다면 두번째 방법을 사용해도 메모리 관리에 있어서는 큰 차이가 없습니다.

Python 에서 정규식 pattern 을 만드는 경우에 r 을 패턴 앞에 관용적으로 붙여 줍니다.
예) r'우리나라'

여기서 r 은 raw string 을 의미합니다. 원래의 목적은 문자열에 사용되는 특수 문자나 기호 등의 의미를 제거하기 위한 것입니다. re 의 버전이 지속적으로 업데이트 되면서 원래의 목적이 어느 정도 없어진 상태입니다.

Searching

텍스트에서 정규식 패턴을 만족하는 문자열을 찾는데 사용되는 함수에는 match, search, findall 등이 있습니다.

match 와 search 는 패턴을 만족하는 문자열을 단 하나만 찾습니다. 만족하는 문자열이 여러개가 있다고 할지라도 가장 먼저 만족하는 하나의 문자열을 찾은 다음에 찾는 과정을 멈추게 됩니다.

match: 문자열의 맨 앞에서만 찾는다.

```
re.match(pattern, text)
```

문자열의 시작하는 위치를 임의로 지정할 수 있다.

```
re.match(pattern, text, 2)
```

search: 위치가 상관없다.

```
re.search(pattern, text)
```

예)

```
text = '대한민국의 수도는 서울입니다'
re.match(r'서울', text)
re.search(r'서울', text)
```

라는 코드가 있는 경우에는, re.match(r'서울', text)는 매치되는 결과가 없는 반면에, re.search(r'서울', text)은 text 라는 변수에 저장되어 있는 문장 중간의 '서울'이라는 단어를 찾게 됩니다. match 함수와 search 함수가 반환하는 결과는 re object 입니다. 즉, re.match(r'서울', text)는 None 을 리턴하게 되고, < _sre.SRE_Match object; span=(10, 12), match='서울'>라는 결과를 리턴하게 됩니다. 만약에 매치된 문자열을 추출하고 싶다면 소괄호()를 사용해서 grouping 을 해야 합니다.

results = re.search(r'(서울)', text)

이라고 코딩하게 되면, results 변수에는 위에서 언급한 re object 가 저장되게 됩니다. 여기에 아래와 같이 group 이라는 함수를 사용하게 되면 매치되는 문자열을 추출할 수 있습니다.

results.group()

=> '서울'

이라는 결과를 리턴합니다. 그룹이 여러개인 경우에는 숫자로 그룹을 지정할 수 있습니다.

다음과 같은 문자열이 있다고 가정을 합시다.

text = '2002 년 12 월 19 일 기사입니다'

여기서 년도, 월, 일 정보를 추출하고 싶다면 다음과 같이 코딩할 수 있습니다.

result1 = re.search(r'(\d{4}년 \d{1,2}월 \d{1,2}일)', text) 또는

result2 = re.search(r'(\d{4})년 (\d{1,2})월 (\d{1,2})일', text)

result1 에는 매치된 그룹이 하나 밖에 없습니다. 그래서 그 결과를 추출하기 위해서는 다음과 같이 타이핑하면 됩니다.

result1.group()

=> '2002 년 12 월 19 일'

그에 비해서 result2 에는 매치되는 그룹이 3 개나 있습니다. 첫번째 그룹은 년도 정보가, 두번째 그룹에는 월 정보, 세번째 그룹에는 일 정보가 포함되어 있습니다.

result2.group() 는 매치되는 전체 결과('2002 년 12 월 19 일')를 반환합니다. 반면에 result2.group(1)은 첫번째 그룹의 정보만을 반환하게 되며 ('2002'), result2.group(2)는 두번째 그룹 정보 (즉, '12'),

Comment [S16]: 매치되는 그룹이 하나 밖에 없기 때문에 result1.group(), result1.group(0), result1.group(1) 모두 같은 결과 (즉, '2002 년 12 월 19 일')를 리턴하게 됩니다.

Comment [S17]: result2.group(0)도 같은 결과를 반환합니다.

result2.group(3)은'19'를 반환하게 됩니다. result2.groups()는 만족하는 3 개의 그룹의 결과를 모두(즉, ('2002', '12', '19')) 반환합니다.

findall()

findall() 함수는 인자로 입력되는 정규식 패턴을 만족하는 모든 결과를 list 의 형태로 반환합니다.

앞에서 살펴본 text = '대한민국의 수도는 서울입니다' 라는 텍스트가 있다라고 하는 경우에,

re.findall(r'대한|수도', text)

는 ['대한', '수도']의 결과를 반환합니다. 즉, '대한', '수도' 의 패턴과 만족하는 문자열이 있기 때문에 그 결과가 모두 반환되는 것입니다. 하지만, match 나 search 는 위에서 언급한 것 처럼 패턴을 만족하는 문자열이 많이 있더라도, 그 중에서 가장 먼저 나오는 문자열만 반환을 합니다. 위의 text 에 대해서 아래와 같이 search 를 사용하면,

re.search(r'대한|수도', text)

아래와 같은 결과가 나옵니다.

<_sre.SRE_Match object; span=(0, 2), match='대한'>

즉 '대한'만 반환이 된 것입니다.

또 다른 예를 살펴보겠습니다.

text1 = '2002 년 12 월 19 일 기사입니다'

위의 text1 에 대해서 아래의 코드를 실행하면,

re.findall(r'(\d{4})년 (\d{1,2})월 (\d{1,2})일', text1)

반환되는 결과는 [('2002', '12', '19')] 입니다. 물런 '년', '월', '일' 도 매치가 되지만, 최종적으로 findall()에 의해서 반환되는 결과는 소괄호로 묶어진 내용뿐입니다.

이러한 결과는 아래의 예에서도 확인될 수 있습니다.

re.findall(r'자살 (교육|방지|범죄|폭탄)', '자살 폭탄으로 인해, 자살 교육 실시가 중요하고')

위의 결과로는 ['폭탄', '교육']이 반환됩니다.

만약 '자살' 이라는 문자열까지 반환하고 싶다면, 아래와 같이 할 수 있습니다.

re.findall(r'(자살) (교육|방지|범죄|폭탄)', '자살 폭탄으로 인해, 자살 교육 실시가 중요하고')

=> [('자살', '폭탄'), ('자살', '교육')]

findall() 함수는 아래와 같이 동일한 내용을 의미하는 다른 표현들을 고려해서 문자열의 정보를 추출하고자 하는 경우에 사용할 수 있습니다.

Example 1)

text = '우울증 걸린 대한민국'

re.findall(r'(낳는|앓는|걸린) (사회|대한민국)', text)

=> [('걸린', '대한민국')]

text1 = '우울증 앓는 사회'

re.findall(r'(낳는|앓는|걸린) (사회|대한민국)', text1)

=> [('앓는', '사회')]

Modifying a string

Python 의 re 모듈에는 패턴을 만족하는 문자열을 찾는 역할을 하는 함수 뿐만 아니라, 기존의 문자열을 특정한 방식으로 변경하는데 사용되는 함수들도 존재합니다.

1) split()

split()함수는 패턴을 이용해서 텍스트를 쪼개고자 할 때 사용합니다. 예를 들어, word character 가 아닌 character 를 이용해서 주어진 텍스트 또는 문자열을 쪼개고자 하는 경우에는 아래의 코드를 사용할 수 있습니다.

re.split(r'\W','자살-폭탄으로 인해, 자살 교육 실시가 중요하고')

=> ['자살', '폭탄으로', '인해', '', '자살', '교육', '실시가', '중요하고']

'자살-폭탄으로 인해, 자살 교육 실시가 중요하고'의 텍스트에 존재하는 word character 가 아닌 character 에는 '-', ',' 와 띄어쓰기가 존재합니다. 따라서 그러한 character 를 기준으로 해당 문자열이 쪼개진 것을 확인할 수 있습니다.

문자열 함수인 split()과 비교

문자열 함수 중에 하나인 split()도 re 에서 제공되는 split()과 비슷한 역할을 합니다. 하지만, 문자열 함수 split()은 주어진 텍스트를 쪼개는데 사용되는 character 가 여러개인 경우에는 해당 함수를 여러번 적용해야 한다는 단점이 존재합니다.

2) sub():

기존 문자열을 새로운 문자열을 가지고 대체 (substitute) 하고자 할 때 사용할 수 있는 re 함수가 sub()입니다. sub()는 주어진 정규식 패턴을 만족하는 부분을 특정한 문자열로 대체를 합니다.

예를 들어, '010-123-1234'라는 문자열에 존재하는 하이픈 (-)기호를 띄어쓰기로 대체하고자 하는 경우에는 아래 코드를 사용할 수 있습니다.

```
re.sub(r'\-', ' ', '010-123-1234')
```

=> '010 123 1234'

Comment [S18]: 정규식에서 하이픈 기호는 특수한 역할을 하는 특수문자입니다. 즉, []안에서 사용되는 경우에는 range 를 의미합니다. 이러한 특수문자를 문자그대로 사용하고자 하는 경우에는 앞에 \를 붙여주면 됩니다. 즉, \-은 실제 하이픈과 매치됩니다.

문자열 함수인 replace()와 비교

문자열 함수인 replace()도 re 모듈의 sub()와 비슷한 역할을 합니다. 하지만, 위의 split()의 경우와 마찬가지로 대체하고자 하는 문자열이나 character 가 여러개인 경우에는 replace() 함수를 여러번 사용해야 되는 불편함이 있습니다.

문장 간 띄어쓰기가 안 되어 있는 경우

sub() 함수는 텍스트 전처리를 하는데도 유용하게 사용될 수 있습니다. 예를 들어서, 수집한 raw 데이터가 문장간 띄어쓰기가 잘 안되어 있는 텍스트 데이터라고 하는 경우에 우리는 sub()함수를 사용해서 문제를 해결할 수 있습니다. 아래의 예를 보도록 하겠습니다. old_text 에 저장되어 있는 문자열을 보면 두개의 문장이 서로 띄어쓰기 없이 연결되어 있습니다.

```
old_text = 'Today is Monday.Tomorrow is Tuesday.'
```

이 두개의 문장을 분리하기 위해서는 아래의 코드를 사용할 수 있습니다.

```
new_text = re.sub(r'([\.\?\!])([A-Z])',r'\1 \2', old_text)
```

Comment [S19]: \1: 매치되는 결과의 첫번째 그룹에 속한 내용
\2: 매치되는 결과의 두번째 그룹에 속한 내용
\1 과 \2 사이에 띄어쓰기가 포함되어 있습니다.

sub() 함수에 첫번째 인자로 사용되는 패턴에 소괄호를 사용하게 되면, old_text 의 문자열 중에서 소괄호 안 패턴을 만족하는 문자열의 내용이 컴퓨터 메모리 상에 저장이 됩니다. 그리고 그렇게 저장된 문자열의 정보를 \1 또는 \2 같은 방식으로 사용할 수가 있습니다. \1 은 첫번째 소괄호의 패턴을 만족하는 문자열이고 \2 는 두번째 소괄호의 패턴을 만족하는 문자열입니다.

new_text

=> 'Today is Monday. Tomorrow is Tuesday.'

또 다른 예)

영어의 경우에는 마침표가 문장을 구분하는 용도로 사용되는 것 뿐만아니라, 약자를 표현할 때도 사용됩니다. 예를 들어서, U.S.A., Mr. Dr. 등에 사용됩니다. 그런데 문장 단위로 텍스트를 분리한 후에 텍스트를 분석해야 하는 경우에는 이러한 마침표가 들어간 약어들을 먼저 마침표가 없는 표현으로 변환하는 것이 필요합니다. 이는 아래 코드와 같이 수행할 수 있습니다.

```
old_text = 'Seattle is a city in the U.S.A.'
new_text = re.sub(r'([A-Z])(\.)',r'\1',old_text)
new_text
=> 'Seattle is a city in the USA'
```

기호 없애기

sub() 함수는 불필요한 기호를 없애는 데도 유용하게 사용됩니다.

```
text = 'The * sign means several different things: multiplication, footnote, etc.'
cleaned_text = re.sub(r'[^\.\w\d\s]','',text)
cleaned_text
=> 'The  sign means several different things multiplication footnote etc.'
```

Comment [S110]: 이 패턴은\.\w\d\s 이 아닌 하나의 character 를 의미합니다. 즉, 마침표(.)가 아니거나, word character (\w)가 아니거나, 숫자(\d)가 아니거나, 공백문자 (\s)아닌 character 를 의미합니다. 이는 결국 기호들 중에서 마침표가 아닌 모든 기호들을 의미하게 됩니다.

특정한 표현이 뒤따르지 않는 패턴(혹은 표현) 찾기

이럴 때는 ?!를 동시에 사용합니다.

```
text3='초콜릿렛'
```

text4='초콜릿맛'

text5 = '초콜릿맛 초콜릿렛 초콜릿사과 초콜릿딸기'

re.findall(r'초콜릿(?!렛)', text3)

Out[14]: []

re.findall(r'초콜릿(?!렛)', text4)

Out[14]: ['초콜릿']

re.findall(r'초콜릿(?!렛)', text5)

Out[14]: ['초콜릿', '초콜릿', '초콜릿']

re.findall(r'초콜\w{1}(?!렛)', text3)

Out[15]: []

re.findall(r'초콜\w{0,3}(?!렛)', text3)

Out[13]: ['초콜릿렛']

text4 = '초콜릿바'

re.findall(r'초콜릿(?!렛)', text4)

Out[6]: ['초콜릿']

또 다른 예

아래 결과를 이해하려면 "정규식은 패턴의 앞부분부터 차례로 맞춰간다"라는 것을 기억하는 것이 필요합니다.

text3='초콜릿렛'

re.findall(r'초콜\w{0}(?!렛)', text3) (사실 이는 re.findall(r'초콜(?!렛)', text3) 과 같습니다.)

['초콜']

Comment [S111]: 초콜릿이라는 표현이 들어간 문서 중에서 초콜릿렛이 들어간 문서는 제외하고자 하는 경우에 사용할 수 있습니다.

Comment [S112]: 왜 이런 결과가 나오는지 바로 위의 코드와 비교해 보세요.

일단 먼저 '초콜'까지는 매치가 됩니다. 그 다음으로 나오는 것이 (?!렛)이라는 패턴입니다. 그런데 세번째로 나오는 글자가 '릿'이므로 ('렛'이 아니고), '초콜'이 매치가 되고 그 값이 리턴이 됩니다. 만약 text3 ='초콜렛' 이 었다면, '초콜'은 매치가 되지만, 그 다음 글자가 '렛' 이기 때문에 아무것도 리턴되지 않을 것입니다.

만약 아래와 같이 입력하면 어떻게 될까요?

re.findall(r'초콜\w{1}(?!렛)', text3)

이 경우는 패턴의 ' 초콜\w{1}'과 text3 의 '초콜릿'이 매치가 됩니다. 그런데 그 다음 글자가 '렛' 이기 때문에 (?!렛)에 해당해서 아무런 결과도 리턴되지 않습니다. 즉,

[]

re.findall(r'초콜\w{2}(?!렛)', text3)의 경우는

일단 text3 ='초콜릿렛'의 네글자가 패턴의 '초콜\w{2}'와 매치가 됩니다. 그리고 그 다음에는 아무런 글자도 나오지 않기 때문에 (?!렛)에 해당 부분이 없는 것이지요. 따라서 '초콜릿렛' 이라는 4 개의 글자가 리턴이 됩니다.

['초콜릿렛']

re.findall(r'초콜\w{3}(?!렛)', text3)의 경우는

text3 = '초콜릿렛' 이 패턴의 ' 초콜\w{3}'와 매치가 되지 않으므로 아무런 결과를 리턴하지 않습니다.
여기서는 (?!렛)는 아무런 역할도 하지 않습니다.

제목에 한단어만 사용된 경우

문서의 제목을 가지고 문서를 선택해야하는 경우가 있습니다. 그러한 경우에, 제목에 특정한 단어 하나만 사용된 경우를 구분해야 하는 경우가 있을 것입니다. 예를 들어서, 문서 (예, 신문기사)의 제목이 '우울증' 이라는 하나의 단어만을 사용되서 표현된 문서를 찾고자 하는 경우에는 아래와 같은 코드를 사용할 수 있습니다.

text = '우울증'

re.findall(r'^우울증$',text)

=> ['우울증']

Comment [S113]: ^ 기호가 [] 아닌 곳에서 사용되면, 해당 패턴을 주어진 문자열의 처음에서 찾아라는 것을 의미합니다. $는 문자열의 마지막에서 찾으라는 것을 의미합니다. 따라서 ^우울증$ 이라는 패턴은 우울증이라는 표현이 문자열의 처음과 끝에 존재를 하는 경우에만 찾습니다. 즉, 문자열에 우울증이라는 표현만 저장된 경우에 매치됩니다.

```
text1 = '우울증은 치매 유발'

re.findall(r'^우울증$',text1)

Out[4]: []

text2 = '우울증 환자'

re.findall(r'^우울증$',text2)

Out[5]: []
```

그리고 많은 경우 같은 의미를 갖는 표현을 다른 단어들을 가지고 사용합니다. 예를 들어서, 대전을 ‘대전시’라고 표현할수도 있고, ‘대전광역시’라고 표현할 수도 있습니다. 두 단어는 표현은 다르지만, 결국 ‘대전’이라는 동일한 도시를 의미합니다. 텍스트 분석에서는 이러한 서로 다른 표현의 단어를 같은 단어로 처리하는 것이 필요합니다. 그럴 때는 아래와 같은 방법을 사용할 수 있습니다.

```
text2 = '대전시'

re.findall(r'(대전)(광역)?시',text2)

Out[7]: [('대전', '')]
```