07-2 정규 표현식 시작2

re모듈

파이썬에서 정규 표현식을 지원하는 모듈(re: regular expression)

>>> import re

>>> p = re.compile('ab*')

정규식을 사용한 문자열 검색

컴파일된 패턴객체를 사용하여 문자열 검색을 수행하기. 컴파일된 패턴객체는 다음과 같은 4가지 메서드 제공함

메서드	목적
match()	문자열의 처음부터 정규식과 매치되는지 조사
search()	문자열 전체를 검색하여 정규식과 매치되는지 조사
findall()	정규식과 매치되는 모든 문자열을 리스트로 돌려줌
finditer()	정규식과 매치되는 모든 문자열을 반복 가능한(iterable)객체로 돌려줌

▼ match메서드와 search메서드

- match와 search는 매치될 땐 match객체를, 매치되지 않을 때는 None을 돌려줌

▼ 사용 예시

>>> import re

>>> p = re.compile('[a-z]+')

>>> m = p.match("python")

>>> print(m)

'<re.Match object; span=(0,6), match='python'>' #매치되었으므로 match객체 돌려줌

>>> m = p.match("3 python")

>>> print(m)

'None' #처음에 나오는 3이 정규식에 부합하지 않으므로 None을 돌려줌

>>> m = p.search("python")

>>> print(m)

'<re.Match object; span=(0,6), match='python'> #매치되었으므로 match객체 돌려줌

>>> m = p.search("3 python")

>>> print(m)

'<re.Match object; span=(2,8), match='python'> #match와 달리 search는 처음부터 일치하는지 찾는 게 아니라 문자열 전체를 검색하기 때문에 "3" 이후의 "python"문자열과 매치되므로 match객체 돌려줌

→ 따라서 match메서드와 search메서드는 문자열의 처음부터 검색할지의 여부에 따라 사용

▼ findall메서드

문자열의 각 단어를 각각 컴파일한 정규식과 매치하여 돌려줌

>>> p = re.compile('[a-z]+')

>>> result = p.findall("life is too short")

'['life', 'is', 'too', 'short']

>>> p = re.compile('[0-9]+')

>>> result = p.findall("life is too short")

[ ] (매치되는 단어가 없기 때문에 빈 리스트 돌려줌)

▼ finditer

findall과 동일하지만 다른 점은 결과로 반복가능한(iterable) 객체를 돌려줌(반복 가능한 객체가 포함하는 각각의 요소는 match객체)

>>> p = re.compil('[a-z]+')

>>> result = p.finditer("life is too short")

>>> print(result)

‘<callable_iterator object at 0x10c3e2020>’

>>> for r in result: print(r)

‘<re.Match object; span=(0,4), match=’life’>

‘<re.Match object; span=(5,7), match=’is’>

‘<re.Match object; span=(8,11), match=’too’>

‘<re.Match object; span=(12,17), match=’short’>

Match 객체의 메서드

match메서드와 search메서드를 수행한 결과로 돌려준 match객체란 무엇일까?

메서드	목적
group()	매치된 문자열 돌려줌
start()	매치된 문자열의 시작 위치를 돌려줌
end()	매치된 문자열의 끝위치 돌려줌
span()	매치된 문자열의 (시작, 끝)에 해당하는 튜플 돌려줌

- match객체에서 start()의 결괏값은 항상 0(match메서드는 항상 문자열의 시작부터 조사하기 때문)

+J2P)

▼ 모듈 단위로 수행하기

re 모듈에서는 re.compile로 컴파일하고 그 객체로 이후의 작업을 수행하는 것을 축약한 형태로 사용할 수 있도록 지원해줌

```
#축약 전
p = re.compile(’[a-z]+’)
m = p.match(”python”)

#축약 후
m = re.match('[a-z]+', "python")
```

컴파일 옵션

옵션이름	약어	설명
DOTALL	S	dot문자가 줄바꿈 문자를 포함하여 모든 문자와 매치
IGNORECASE	I	대,소문자 상관없이 매치
MULTILINE	M	여러 줄과 매치
VERBOSE	X	verbose모드를 사용한다

▼ DOTALL, S

\n(줄바꿈문자)도 포함하여 매치하고 싶다면 re.DOTALL or re.S 옵션으로 정규식 컴파일하면 됨

```
import re
p = re.compile('a.b',re.DOTALL)
m = p.match('a\nb')
print(m)
'<re.Match object; span = (0,3), match='a\nb'>
```

▼ IGNORECASE, I

대소문자 구별 없이 매치를 수행할 때

```
import re
p = re.compile('[a-z]',re.I)
p.match('python')
'<re.Match object; span=(0,6), match='p'>
p.match('Python')
'<re.Match object; span=(0,6), match='P'>
p.match('PYTHON')
'<re.Match object; span=(0,6), match='P'>

#원래는 [a-z]니까 대문자랑 매치되면 안 되는데 re.I 옵션으로 대소문자 구별 없이 매치됨
```

▼ MULTILINE, M

^,$와 연관된 옵션

메타문자 ^: 문자열의 처음 ex) ^python은 항상 python으로 시작하는 문자열이랑만 매치

메타문자 $: 문자열의 마지막 ex) $python은 마지막이 항상 python으로 끝나야 매치됨을 의미

```
import re
p = re.compile("^python\s\w+")
data = """python one
life is too short
python two
you need python
python three"""
print(p.findall(data))
'[python one']
```

^python\s\w+ 의 의미: python이라는 문자열로 시작하고(^python) 그 뒤에 whitespace(\s), 그 뒤에 단어(\w)가 와야한다는 의미

⚠️그러나 ^메타문자를 각 라인의 처음으로 인식시키고 싶은 경우도 발생함

→ 이럴 때 사용하는 옵션이 MULTILINE, M

```
import re
p = re.compile("^python\s\w+", re.M)
data = """python one
life is too short
you need python
python three"""
print(p.findall(data))
'[python one', 'python three']

# re.M 옵션으로 각 라인의 ^메타문자를 인식하여 첫번째 줄과 세번째 줄이랑 매치함
```

▼ VERBOSE, X (아니 이게 몬 소리지)

이해하기 어려운 정규식을 주석 또는 줄 단위로 구분할 수 있게 해줌

▼ 백슬래시 문제

\문자가 백슬래스 문자열 자체임을 알려주기 위해서는 백슬래시를 2번 써야함(\\)

\section → \s + ection으로 받아들임

\\section → \section으로 받아들임

또한 \\section을 전달하려면 \\\\section처럼 백슬래시를 4번이나 써야한다...

이렇게 백슬래시가 반복되는 정규식이라면 정규식이 너무 복잡해지기 때문에 이를 해결하기 위해

Raw String 규칙 생겨남

>>> p = re.compile(r'\\section\)