07-3 강력한 정규 표현식의 세계로

이제 07-2에서 배우지 않은 몇몇 메타 문자의 의미를 살펴보고 그룹(Group)을 만드는 법, 전방 탐색 등 더욱 강력한 정규 표현식에 대해서 살펴보자.

메타문자
|

메타문자

아직 살펴보지 않은 메타 문자에 대해서 모두 살펴보자. 여기에서 다룰 메타 문자는 앞에서 살펴본 메타 문자와 성격이 조금 다르다. 앞에서 살펴본 `+`, `*`, `[]`, `{}` 등의 메타문자는 매치가 진행될 때 현재 매치되고 있는 문자열의 위치가 변경된다(보통 소비된다고 표현한다). 하지만 이와 달리 문자열을 소비시키지 않는 메타 문자도 있다. 이번에는 이런 문자열 소비가 없는(zerowidth assertions) 메타 문자에 대해 살펴보자.

|

`|` 메타 문자는 or과 동일한 의미로 사용된다. `A|B` 라는 정규식이 있다면 A 또는 B라는 의미가 된다.

```
>>> p = re.compile('Crow|Servo')
>>> m = p.match('CrowHello')
>>> print(m)
<re.Match object; span=(0, 4), match='Crow'>
```

^

`^` 메타 문자는 문자열의 맨 처음과 일치함을 의미한다. 앞에서 살펴본 컴파일 옵션 `re.MULTILINE` 을 사용할 경우에는 여러 줄의 문자열일 때 각 줄의 처음과 일치하게 된다.

다음 예를 보자.

```
>>> print(re.search('^Life', 'Life is too short'))
<re.Match object; span=(0, 4), match='Life'>
>>> print(re.search('^Life', 'My Life'))
None
```

^Life 정규식은 Life 문자열이 처음에 온 경우에는 매치하지만 처음 위치가 아닌 경우에는 매치되지 않음을 알 수 있다.

$

$ 메타 문자는 ^ 메타 문자와 반대의 경우이다. 즉 $ 는 문자열의 끝과 매치함을 의미한다.

다음 예를 보자.

```
>>> print(re.search('short$', 'Life is too short'))
<re.Match object; span=(12, 17), match='short'>
>>> print(re.search('short$', 'Life is too short, you need python'))
None
```

short$ 정규식은 검색할 문자열이 short로 끝난 경우에는 매치되지만 그 이외의 경우에는 매치되지 않음을 알 수 있다.

> ※ ^ 또는 $ 문자를 메타 문자가 아닌 문자 그 자체로 매치하고 싶은 경우에는 \^, \$ 로 사용하면 된다.

\A

\A 는 문자열의 처음과 매치됨을 의미한다. ^ 메타 문자와 동일한 의미이지만 re.MULTILINE 옵션을 사용할 경우에는 다르게 해석된다. re.MULTILINE 옵션을 사용할 경우 ^ 은 각 줄의 문자열의 처음과 매치되지만 \A 는 줄과 상관없이 전체 문자열의 처음하고만 매치된다.

\Z

\Z 는 문자열의 끝과 매치됨을 의미한다. 이것 역시 \A 와 동일하게 re.MULTILINE 옵션을 사용할 경우 $ 메타 문자와는 달리 전체 문자열의 끝과 매치된다.

\b

\b 는 단어 구분자(Word boundary)이다. 보통 단어는 whitespace에 의해 구분된다.

다음 예를 보자.

```
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<re.Match object; span=(3, 8), match='class'>
```

`\bclass\b` 정규식은 앞뒤가 whitespace로 구분된 class라는 단어와 매치됨을 의미한다. 따라서 no class at all의 class라는 단어와 매치됨을 확인할 수 있다.

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

위 예의 the declassified algorithm 문자열 안에도 class 문자열이 포함되어 있긴 하지만 whitespace로 구분된 단어가 아니므로 매치되지 않는다.

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

subclass 문자열 역시 class 앞에 sub 문자열이 더해져 있으므로 매치되지 않음을 알 수 있다.

`\b` 메타 문자를 사용할 때 주의해야 할 점이 있다. `\b` 는 파이썬 리터럴 규칙에 의하면 백스페이스(BackSpace)를 의미하므로 백스페이스가 아닌 단어 구분자임을 알려 주기 위해 `r'\bclass\b'` 처럼 Raw string임을 알려주는 기호 r을 반드시 붙여 주어야 한다.

\B

`\B` 메타 문자는 `\b` 메타 문자와 반대의 경우이다. 즉 whitespace로 구분된 단어가 아닌 경우에만 매치된다.

```
>>> p = re.compile(r'\Bclass\B')
>>> print(p.search('no class at all'))
None
>>> print(p.search('the declassified algorithm'))
<re.Match object; span=(6, 11), match='class'>
>>> print(p.search('one subclass is'))
None
```

class 단어의 앞뒤에 whitespace가 하나라도 있는 경우에는 매치가 안 되는 것을 확인할 수 있다.

그루핑

ABC 문자열이 계속해서 반복되는지 조사하는 정규식을 작성하고 싶다고 하자. 어떻게 해야할까? 지금까지 공부한 내용으로는 위 정규식을 작성할 수 없다. 이럴 때 필요한 것이 바로 그루핑(Grouping) 이다.

위 경우는 다음처럼 그루핑을 사용하여 작성할 수 있다.

```
(ABC)+
```

그룹을 만들어 주는 메타 문자는 바로 ( ) 이다.

```
>>> p = re.compile('(ABC)+')
>>> m = p.search('ABCABCABC OK?')
>>> print(m)
<re.Match object; span=(0, 9), match='ABCABCABC'>
>>> print(m.group())
ABCABCABC
```

다음 예를 보자.

```
>>> p = re.compile(r"\w+\s+\d+[-]\d+[-]\d+")
>>> m = p.search("park 010-1234-1234")
```

\w+\s+\d+[-]\d+[-]\d+ 은 이름 + " " + 전화번호 형태의 문자열을 찾는 정규식이다. 그런데 이렇게 매치된 문자열 중에서 이름만 뽑아내고 싶다면 어떻게 해야 할까?

보통 반복되는 문자열을 찾을 때 그룹을 사용하는데, 그룹을 사용하는 보다 큰 이유는 위에서 볼 수 있듯이 매치된 문자열 중에서 특정 부분의 문자열만 뽑아내기 위해서인 경우가 더 많다.

위 예에서 만약 '이름' 부분만 뽑아내려 한다면 다음과 같이 할 수 있다.

```
>>> p = re.compile(r"(\w+)\s+\d+[-]\d+[-]\d+")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(1))
park
```

이름에 해당하는 \w+ 부분을 그룹 (\w+) 으로 만들면 match 객체의 group(인덱스) 메서드를 사용하여 그루핑된 부분의 문자열만 뽑아낼 수 있다. group 메서드의 인덱스는 다음과 같은 의미를 갖는다.

group(인덱스)	설명
group(0)	매치된 전체 문자열
group(1)	첫 번째 그룹에 해당되는 문자열
group(2)	두 번째 그룹에 해당되는 문자열
group(n)	n 번째 그룹에 해당되는 문자열

다음 예제를 계속해서 보자.

```
>>> p = re.compile(r"(\w+)\s+(\d+[-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(2))
010-1234-1234
```

이번에는 전화번호 부분을 추가로 그룹 `(\d+[-]\d+[-]\d+)` 로 만들었다. 이렇게 하면 `group(2)` 처럼 사용하여 전화번호만 뽑아낼 수 있다.

만약 전화번호 중에서 국번만 뽑아내고 싶으면 어떻게 해야 할까? 다음과 같이 국번 부분을 또 그루핑하면 된다.

```
>>> p = re.compile(r"(\w+)\s+((\d+)[-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(3))
010
```

위 예에서 볼 수 있듯이 `(\w+)\s+((\d+)[-]\d+[-]\d+)` 처럼 그룹을 중첩되게 사용하는 것도 가능하다. 그룹이 중첩되어 있는 경우는 바깥쪽부터 시작하여 안쪽으로 들어갈수록 인덱스가 증가한다.

그루핑된 문자열 재참조하기

그룹의 또 하나 좋은 점은 한 번 그루핑한 문자열을 재참조(Backreferences)할 수 있다는 점이다. 다음 예를 보자.

```
>>> p = re.compile(r'(\b\w+)\s+\1')
>>> p.search('Paris in the the spring').group()
'the the'
```

정규식 `(\b\w+)\s+\1` 은 `(그룹) + " " + 그룹과 동일한 단어` 와 매치됨을 의미한다. 이렇게 정규식을 만들게 되면 2개의 동일한 단어를 연속적으로 사용해야만 매치된다. 이것을 가능하게 해주는 것이 바로 재참조 메타 문자인 `\1` 이다. `\1` 은 정규식의 그룹 중 첫 번째 그룹을 가리킨다.

> ※ 두 번째 그룹을 참조하려면 `\2` 를 사용하면 된다.

그루핑된 문자열에 이름 붙이기

정규식 안에 그룹이 무척 많아진다고 가정해 보자. 예를 들어 정규식 안에 그룹이 10개 이상만 되어도 매우 혼란스러울 것이다. 거기에 더해 정규식이 수정되면서 그룹이 추가, 삭제되면 그 그룹을 인덱스로 참조한 프로그램도 모두 변경해 주어야 하는 위험도 갖게 된다.

만약 그룹을 인덱스가 아닌 이름(Named Groups)으로 참조할 수 있다면 어떨까? 그렇다면 이런 문제에서 해방되지 않을까?

이러한 이유로 정규식은 그룹을 만들 때 그룹 이름을 지정할 수 있게 했다. 그 방법은 다음과 같다.

```
(?P<name>\w+)\s+((\d+)[-]\d+[-]\d+)
```

위 정규식은 앞에서 본 이름과 전화번호를 추출하는 정규식이다. 기존과 달라진 부분은 다음과 같다.

```
(\w+) --> (?P<name>\w+)
```

대단히 복잡해진 것처럼 보이지만 `(\w+)` 라는 그룹에 name이라는 이름을 붙인 것에 불과하다. 여기에서 사용한 `(?...)` 표현식은 정규 표현식의 확장 구문이다. 이 확장 구문을 사용하기 시작하면 가독성이 상당히 떨어지긴 하지만 반면에 강력함을 갖게 된다.

그룹에 이름을 지어 주려면 다음과 같은 확장 구문을 사용해야 한다.

```
(?P<그룹명>...)
```

그룹에 이름을 지정하고 참조하는 다음 예를 보자.

```
>>> p = re.compile(r"(?P<name>\w+)\s+((\d+)[-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group("name"))
park
```

위 예에서 볼 수 있듯이 name이라는 그룹 이름으로 참조할 수 있다.

그룹 이름을 사용하면 정규식 안에서 재참조하는 것도 가능하다.

```
>>> p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)')
>>> p.search('Paris in the the spring').group()
'the the'
```

위 예에서 볼 수 있듯이 재참조할 때에는 `(?P=그룹이름)` 이라는 확장 구문을 사용해야 한다.

전방 탐색

정규식에 막 입문한 사람들이 가장 어려워하는 것이 바로 전방 탐색(Lookahead Assertions) 확장 구문이다. 정규식 안에 이 확장 구문을 사용하면 순식간에 암호문처럼 알아보기 어렵게 바뀌기 때문이다. 하지만 이 전방 탐색이 꼭 필요한 경우가 있으며 매우 유용한 경우도 많으니 꼭 알아 두자.

다음 예를 보자.

```
>>> p = re.compile(".+:")
>>> m = p.search("http://google.com")
>>> print(m.group())
http:
```

정규식 `.+:` 과 일치하는 문자열로 http:를 돌려주었다. 만약 http:라는 검색 결과에서 :을 제외하고 출력하려면 어떻게 해야 할까? 위 예는 그나마 간단하지만 훨씬 복잡한 정규식이어서 그루핑은 추가로 할 수 없다는 조건까지 더해진다면 어떻게 해야 할까?

이럴 때 사용할 수 있는 것이 바로 전방 탐색이다. 전방 탐색에는 긍정(Positive)과 부정(Negative)의 2종류가 있고 다음과 같이 표현한다.

- 긍정형 전방 탐색(`(?=...)`) - `...` 에 해당되는 정규식과 매치되어야 하며 조건이 통과되어도 문자열이 소비되지 않는다.
- 부정형 전방 탐색(`(?!...)`) - `...` 에 해당되는 정규식과 매치되지 않아야 하며 조건이 통과되어도 문자열이 소비되지 않는다.

긍정형 전방 탐색

긍정형 전방 탐색을 사용하면 http:의 결과를 http로 바꿀 수 있다. 다음 예를 보자.

```
>>> p = re.compile(".+(?=:)")
>>> m = p.search("http://google.com")
>>> print(m.group())
http
```

정규식 중 `:` 에 해당하는 부분에 긍정형 전방 탐색 기법을 적용하여 `(?=:)` 으로 변경하였다. 이렇게 되면 기존 정규식과 검색에서는 동일한 효과를 발휘하지만 `:` 에 해당하는 문자열이 정규식 엔진에 의해 소비되지 않아(검색에는 포함되지만 검색 결과에는 제외됨) 검색 결과에서는 `:` 이 제거된 후 돌려주는 효과가 있다.

자, 이번에는 다음 정규식을 보자.

```
.*[.].*$
```

이 정규식은 `파일 이름 + . + 확장자` 를 나타내는 정규식이다. 이 정규식은 foo.bar, autoexec.bat, sendmail.cf 같은 형식의 파일과 매치될 것이다.

이 정규식에 확장자가 "bat인 파일은 제외해야 한다"는 조건을 추가해 보자. 가장 먼저 생각할 수 있는 정규식은 다음과 같다.

```
.*[.][^b].*$
```

이 정규식은 확장자가 b라는 문자로 시작하면 안 된다는 의미이다. 하지만 이 정규식은 foo.bar라는 파일마저 걸러낸다. 정규식을 다음과 같이 수정해 보자.

```
.*[.]([^b]..|.[^a].|..[^t])$
```

이 정규식은 | 메타 문자를 사용하여 확장자의 첫 번째 문자가 b가 아니거나 두 번째 문자가 a가 아니거나 세 번째 문자가 t가 아닌 경우를 의미한다. 이 정규식에 의하여 foo.bar는 제외되지 않고 autoexec.bat은 제외되어 만족스러운 결과를 돌려준다. 하지만 이 정규식은 아쉽게도 sendmail.cf처럼 확장자의 문자 개수가 2개인 케이스를 포함하지 못하는 오동작을 하기 시작한다.

따라서 다음과 같이 바꾸어야 한다.

```
.*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$
```

확장자의 문자 개수가 2개여도 통과되는 정규식이 만들어졌다. 하지만 정규식은 점점 더 복잡해지고 이해하기 어려워진다.

그런데 여기에서 bat 파일말고 exe 파일도 제외하라는 조건이 추가로 생긴다면 어떻게 될까? 이 모든 조건을 만족하는 정규식을 구현하려면 패턴은 더욱더 복잡해질 것이다.

부정형 전방 탐색

이러한 상황의 구원 투수는 바로 부정형 전방 탐색이다. 위 예는 부정형 전방 탐색을 사용하면 다음과 같이 간단하게 처리된다.

```
.*[.](?!bat$).*$
```

확장자가 bat가 아닌 경우에만 통과된다는 의미이다. bat 문자열이 있는지 조사하는 과정에서 문자열이 소비되지 않으므로 bat가 아니라고 판단되면 그 이후 정규식 매치가 진행된다.

exe 역시 제외하라는 조건이 추가되더라도 다음과 같이 간단히 표현할 수 있다.

```
.*[.](?!bat$|exe$).*$
```

문자열 바꾸기

sub 메서드를 사용하면 정규식과 매치되는 부분을 다른 문자로 쉽게 바꿀 수 있다.

다음 예를 보자.

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

sub 메서드의 첫 번째 매개변수는 "바꿀 문자열(replacement)"이 되고, 두 번째 매개변수는 "대상 문자열"이 된다. 위 예에서 볼 수 있듯이 blue 또는 white 또는 red라는 문자열이 colour라는 문자열로 바뀌는 것을 확인할 수 있다.

그런데 딱 한 번만 바꾸고 싶은 경우도 있다. 이렇게 바꾸기 횟수를 제어하려면 다음과 같이 세 번째 매개변수로 count 값을 넘기면 된다.

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

처음 일치하는 blue만 colour라는 문자열로 한 번만 바꾸기가 실행되는 것을 알 수 있다.

[sub 메서드와 유사한 subn 메서드]

subn 역시 sub와 동일한 기능을 하지만 반환 결과를 튜플로 돌려준다는 차이가 있다. 돌려준 튜플의 첫 번째 요소는 변경된 문자열이고, 두 번째 요소는 바꾸기가 발생한 횟수이다.

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn( 'colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
```

sub 메서드 사용 시 참조 구문 사용하기

sub 메서드를 사용할 때 참조 구문을 사용할 수 있다. 다음 예를 보자.

```
>>> p = re.compile(r"(?P<name>\w+)\s+(?P<phone>(\d+)[-]\d+[-]\d+)")
>>> print(p.sub("\g<phone> \g<name>", "park 010-1234-1234"))
010-1234-1234 park
```

위 예는 `이름 + 전화번호` 의 문자열을 `전화번호 + 이름` 으로 바꾸는 예이다. sub의 바꿀 문자열 부분에 `\g<그룹이름>` 을 사용하면 정규식의 그룹 이름을 참조할 수 있게 된다.

다음과 같이 그룹 이름 대신 참조 번호를 사용해도 마찬가지 결과를 돌려준다.

```
>>> p = re.compile(r"(?P<name>\w+)\s+(?P<phone>(\d+)[-]\d+[-]\d+)")
>>> print(p.sub("\g<2> \g<1>", "park 010-1234-1234"))
010-1234-1234 park
```

sub 메서드의 매개변수로 함수 넣기

sub 메서드의 첫 번째 매개변수로 함수를 넣을 수도 있다. 다음 예를 보자.

```
>>> def hexrepl(match):
...     value = int(match.group())
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
'Call 0xffd2 for printing, 0xc000 for user code.'
```

hexrepl 함수는 match 객체(위에서 숫자에 매치되는)를 입력으로 받아 16진수로 변환하여 돌려주는 함수이다. sub의 첫 번째 매개변수로 함수를 사용할 경우 해당 함수의 첫 번째 매개변수에는 정규식과 매치된 match 객체가 입력된다. 그리고 매치되는 문자열은 함수의 반환 값으로 바뀌게 된다.

Greedy vs Non-Greedy

정규식에서 Greedy(탐욕스러운)란 어떤 의미일까? 다음 예제를 보자.

```
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print(re.match('<.*>', s).span())
(0, 32)
>>> print(re.match('<.*>', s).group())
<html><head><title>Title</title>
```

`<.*>` 정규식의 매치 결과로 `<html>` 문자열을 돌려주기를 기대했을 것이다. 하지만 * 메타 문자는 매우 탐욕스러워서 매치할 수 있는 최대한의 문자열인 `<html><head><title>Title</title>` 문자열을 모두 소비해 버렸다. 어떻게 하면 이 탐욕스러움을 제한하고 `<html>` 문자열까지만 소비하도록 막을 수 있을까?

다음과 같이 non-greedy 문자인 `?` 를 사용하면 `*` 의 탐욕을 제한할 수 있다.

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

non-greedy 문자인 `?` 는 `*?` , `+?` , `??` , `{m,n}?` 와 같이 사용할 수 있다. 가능한 한 가장 최소한의 반복을 수행하도록 도와주는 역할을 한다.

마지막 편집일시 : 2019년 7월 25일 7:59 오후

(https://wikidocs.net/105844)

댓글 16 피드백

- **이전글** : 07-2 정규 표현식 시작하기
- **다음글** : 08장 종합문제

TOP