7장. 정규 표현식

Contents

I. 정규 표현식 살펴보기

❖ 정규 표현식(Regular Expressions) 은 복잡한 문자열을 처리할 때 사용하는 기법으로 문자열을 처리하는 모든 곳에서 사용함

❖ **정규 표현식은 왜 필요한가?**

> 주민등록번호를 포함하고 있는 텍스트가 있다. 이 텍스트에 포함된 모든 주민등록번호의 뒷자리를 * 문자로 변경해 보자.

- 정규식을 모르면, 다음과 같은 순서로 프로그램을 작성해야 할 것임
  1. 전체 텍스트를 공백 문자로 나눈다(split).
  2. 나뉜 단어가 주민등록번호 형식인지 조사한다.
  3. 단어가 주민등록번호 형식이라면 뒷자리를 *로 변환한다.
  4. 나뉜 단어를 다시 조립한다.

- 주민등록번호의 뒷자리를 * 문자로 구현한 코드

```
data = """
park 800905-1049118
kim  700905-1059119
"""

result = [ ]
for line in data.split("₩n"):
    word_result = [ ]
    for word in line.split(" "):
        if len(word) == 14 and word[:6].isdigit() and word[7:].isdigit():
            word = word[:6] + "-" + "*******"
        word_result.append(word)
    result.append(" ".join(word_result))
print("₩n".join(result))
```

전체 텍스트

공백 문자마다 나누기

나눈 단어 조립하기

I. 정규 표현식 살펴보기

- 주민등록번호의 뒷자리를 * 문자로 구현한 정규 표현식

```
import re    ←—— 정규 표현식을 사용하기 위한 re 모듈

data = """
park 800905-1049118
kim  700905-1059119
"""

pat = re.compile("(₩d{6})[-]₩d{7}")
print(pat.sub("₩g<1>-*******", data))
```

```
결과값:
park 800905-*******
kim  700905-*******
```

✓ 정규 표현식을 사용하면 훨씬 간편하고 직관적인 코드를 작성할 수 있음

✓ 찾으려는 문자열 또는 바꾸어야 할 문자열의 규칙이 매우 복잡하다면, 정규식의 효용은 더 커지게 됨

❖ 정규 표현식의 기초, 메타 문자

```
. ^ $ * + ? {} [] ₩ | ()
```

- 메타 문자란 원래 그 문자가 가진 뜻이 아닌 특별한 용도로 사용하는 문자를 말함

❖ 문자 클래스 [ ]

- '[ ] 사이의 문자들과 매치'라는 의미를 갖음

정규식	문자열	매치 여부	설명
[abc]	a	Yes	"a"는 정규식과 일치하는 문자인 "a"가 있으므로 매치
	before	Yes	"before"는 정규식과 일치하는 문자인 "b"가 있으므로 매치
	dude	No	"dude"는 정규식과 일치하는 문자인 a, b, c 중 어느 하나도 포함하고 있지 않으므로 매치되지 않음

- 하이픈(-)는 두 문자 사이의 범위(From - To)를 의미함
- ^는 반대(not)를 의미함

- 자주 사용하는 문자 클래스

정규 표현식	설명
[a-zA-Z]	알파벳 모드
[0-9]	숫자
₩d	숫자와 매치, [0-9]와 동일한 표현식
₩D	숫자가 아닌 것과 매치, [^0-9]와 동일한 표현식
₩s	whitespace 문자(space나 tab처럼 공백을 표현하는 문자)와 매치 [₩t₩n₩r₩f₩v]와 동일한 표현식, 맨 앞의 빈칸은 공백문자를 의미
₩S	whitespace 문자가 아닌 것과 매치 [^ ₩t₩n₩r₩f₩v]와 동일한 표현식
₩w	문자+숫자(alphanumeric)와 매치 [a-zA-Z0-9]와 동일한 표현식
₩W	문자+숫자(alphanumeric)가 아닌 문자와 매치 [^a-zA-Z0-9]와 동일한 표현식

✓ 대문자로 사용된 것은 소문자의 반대임

❖ Dot(.)

- Dot(.) 메타 문자는 줄바꿈 문자인 ₩n을 제외한 모든 문자와 매치됨을 의미함

a.b ← "a + 모든 문자 + b"

정규식	문자열	매치 여부	설명
[a.b]	aab	Yes	"aab"는 가운데 문자 "a"가 모든 문자를 의미하는 `.`과 일치하므로 정규식과 매치
	a0b	Yes	"a0b"는 가운데 문자 "0 " 이 모든 문자를 의미하는 `.`과 일치하므로 정규식과 매치
	abc	No	"abc"는 "a"문자와 "b"문자 사이에 어떤 문자라도 하나는있어야 하는 이 정규식과 일치하지 않으므로 매치되지 않음

✓ a[.]b는 문자 . 그대로를 의미함 ("a + Dot(.)문자 + b")

❖ 반복(*)

- * 메타 문자는 반복을 의미함. 0부터 무한대로 반복될 수 있음

ca*t ← * 문자 바로 앞에 있는 a가 0번 이상 반복되면 매치

정규식	문자열	매치 여부	설명
ca*t	ct	Yes	"a"가 0번 반복되어 매치
	cat	Yes	"a"가 0번 이상 반복되어 매치 (1번 반복)
	caaat	Yes	"a"가 0번 이상 반복되어 매치 (3번 반복)

❖ 반복(+)

- \+ 메타 문자도 반복을 의미함. 최소 1번 이상 반복될 때 사용함

ca+t ⟵ "c + a(1번 이상 반복) + t"

정규식	문자열	매치 여부	설명
ca+t	ct	No	"a"가 0번 반복되어 매치되지 않음
	cat	Yes	"a"가 1번 이상 반복되어 매치 (1번 반복)
	caaat	Yes	"a"가 1번 이상 반복되어 매치 (3번 반복)

❖ 반복({m,n}, ?)

- 반복 횟수를 제한하고 싶을 때 사용
- {m,n}는 반복 횟수가 m부터 n까지 매치할 수 있음. m 또는 n을 생략할 수도 있음
- {m,}는 반복 횟수가 m 이상이고, {,n}는 반복 횟수가 n 이하를 의미함
- {1,}은 +와 동일하고, {0,}은 *와 동일함

1. {m}

ca{2}t ← a가 2번 반복되면 매치, "c + a(반드시 2번 반복) + t"

정규식	문자열	매치 여부	설명
ca{2}t	cat	No	"a"가 1번만 반복되어 매치되지 않음
	caat	Yes	"a"가 2번 반복되어 매치

2. {m,n}

ca{2,5}t ← a가 2~5번 반복되면 매치, "c + a(2~5번 반복) + t"

정규식	문자열	매치 여부	설명
ca{2,5}t	cat	No	"a"가 1번만 반복되어 매치되지 않음
	caat	Yes	"a"가 2번 반복되어 매치
	caaaaat	Yes	"a"가 5번 반복되어 매치

3. ?

- ? 메타 문자는 {0, 1}을 의미함

ab?c ← b가 0~1번 사용되면 매치, "a + b(있어도 되고 없어도 된다) + c"

정규식	문자열	매치 여부	설명
ab?c	abc	Yes	"b"가 1번 사용되어 매치
	ac	Yes	"b"가 0번 사용되어 매치

✓ *, +, ? 메타 문자는 모두 {m, n} 형태로 고쳐 쓰는 것이 가능하지만, 가급적 간결한 *, +, ? 메타 문자를 사용하는 것이 좋음

❖ 파이썬에서 정규 표현식을 지원하는 re 모듈

- re 모듈은 파이썬 설치할 때 자동으로 설치되는 기본 라이브러리임

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

  - ✓ re.compile을 사용하여 정규 표현식을 컴파일함
  - ✓ re.compile의 결과로 돌려주는 객체 p(컴파일된 패턴 객체)를 사용하여 그 이후의 작업을 수행함

❖ 정규식을 사용한 문자열 검색

- 컴파일된 패턴 객체는 4가지 메소드를 제공함

메소드	목적
match()	문자열의 처음부터 정규식과 매치되는지 조사함
search()	문자열 전체를 검색하여 정규식과 매치되는지 조사함
findall()	정규식과 매치되는 모든 문자열(substring)을 리스트로 돌려줌
finditer()	정규식과 매치되는 모든 문자열(substring)을 반복 가능한 객체로 돌려줌

✓ match, search는 정규식과 매치될 때는 match 객체를 돌려주고, 매치되지 않을 때는 None을 돌려줌

- **match**
  - ✓ 문자열의 처음부터 정규식과 매치되는지 조사함

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

>>> m = p.match("python")
>>> print(m)
<re.Match object; span=(0, 6), match='python'>

>>> m = p.match("3 python")
>>> print(m)
None
```

← "python"은 정규식에 부합되므로 match 객체를 돌려줌

← 문자 3이 정규식에 부합되지 않으므로 None을 돌려줌

  - ✓ match의 결과로 객체 또는 None 을 돌려주기 때문에 다음의 흐름으로 작성함

```
p = re.compile(정규 표현식)
m = p.match("조사할 문자열")

if m:
    print('Match found: ', m.group())
else:
    print('No match')
```

- **search**
  - ✓ 문자열 전체를 검색하여 정규식과 매치되는지 조사함

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

>>> m = p.search("python")
>>> print(m)
<re.Match object; span=(0, 6), match='python'>    ← match 메소드와 동일하게 매치됨

>>> m = p.search("3 python")
>>> print(m)
<re.Match object; span=(2, 8), match='python'>
```

  - ✓ 문자열 전체를 검색하기 때문에 "3" 이후의 "python" 문자열과 매치됨

- **findall**
  - ✓ 정규식과 매치되는 모든 문자열(substring)을 리스트로 돌려줌

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

>>> result = p.findall("life is too short")
>>> print(result)
['life', 'is', 'too', 'short']
```

- **finditer**
  - ✓ 정규식과 매치되는 모든 문자열(substring)을 반복 가능한 객체로 돌려줌

```
>>> result = p.finditer("life is too short")
>>> print(result)
<callable_iterator object at 0x01F5E390>
>>> 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, search 객체의 메소드

- 어떤 문자열이 매치되었는가?
- 매치된 문자열의 인덱스는 어디부터 어디까지인가?

메소드	목적
group()	매치된 문자열을 돌려줌
start()	매치된 문자열의 시작 위치를 돌려줌
end()	매치된 문자열의 끝 위치를 돌려줌
span()	매치된 문자열의 (시작, 끝)에 해당하는 튜플을 돌려줌

- match, search 객체의 메소드 사용 예

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

```
>>> m = p.match("python")
>>> m.group()
'python'
>>> m.start()
0
>>> m.end()
6
>>> m.span()
(0, 6)
```

```
>>> m = p.search("3 python")
>>> m.group()
'python'
>>> m.start()
2
>>> m.end()
8
>>> m.span()
(2, 8)
```

✓ 메소드 결과값이 다르게 나오는 것을 확인할 수 있음

[모듈 단위로 수행하기]

- re 모듈은 축약된 형태로 사용할 수 있는 방법을 제공함

```
>>> p = re.compile('[a-z]+')
>>> m = p.match("python")
```

```
>>> m = re.match('[a-z]+', "python"
```

✓ 한 번 만든 패턴 객체를 여러 번 사용해야 할 때는 re.compile을 사용하는 것이 좋음

❖ 컴파일 옵션

옵션 이름	약어	목적
DOTALL	S	dot 문자(.)가 줄바꿈 문자를 포함하여 모든 문자와 매치함
IGNORECASE	I	대 · 소문자에 관계 없이 매치함
MULTILINE	M	여러 줄과 매치함 (^, $ 메타 문자의 사용과 관계가 있는 옵션임
VERBOSE	X	verbose 모드를 사용함 (정규식을 보기 편하게 만들 수도 있고, 주석 등을 사용할 수도 있음)

- **DOTALL, S**
  - ✓ dot 문자(.)가 줄바꿈 문자(₩n)를 포함하여 모든 문자와 매치함
  - ✓ re.DATALL 또는 re.S 옵션을 사용해 정규식을 컴파일하면 됨

```
>>> import re
>>> p = re.compile('a.b')
>>> m = p.match('a₩nb')
>>> print(m)
None
```
← 문자열과 정규식이 매치되지 않음

```
>>> p = re.compile('a.b', re.DOTALL)
>>> m = p.match('a₩nb')
>>> print(m)
<re.Match object; span=(0, 3), match='a₩nb'>
```
← ₩n 문자와도 매치

- **IGNORECASE, I**
  - ✓ re.IGNORECASE 또는 re.I 옵션은 대 · 소문자에 관계 없이 매치함

```
>>> p = re.compile('[a-z]', re.I)
>>> p.match('python')
<re.Match object; span=(0, 1), match='p'>
>>> p.match('Python')
<re.Match object; span=(0, 1), match='P'>
>>> p.match('PYTHON')
<re.Match object; span=(0, 1), match='P'>
```

- **MULTILINE, M**
  - ✓ re.MULTILINE 또는 re.M 옵션은 메타 문자인 ^, $와 연관된 옵션임
  - ✓ ^는 문자열의 처음을 의미하고, $은 문자열의 마지막을 의미함

```
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))
```

```
결과값:
['python one']
```

✓ re.MULTILINE 또는 re.M 옵션은 ^, $ 를 문자열의 각 줄마다 적용해 주는 것임

```
import re
p = re.compile("^python₩s₩w+", re.MULTILINE)

data = """python one
life is too short
python two
you need python
python three"""

print(p.findall(data))
```

결과값:
['python one', 'python two', 'python three']

- **VERBOSE, X**
  - ✓ re.VERBOSE 또는 re.X 옵션은 정규식을 보기 편하게 주석 또는 줄 단위로 구분할 수도 있음

```
charref = re.compile(r'&[#](0[0-7]+|[0-9]+|x[0-9a-fA-F]+);')
```

```
charref = re.compile(r"""
 &[#]                    # Start of a numeric entity reference
 (
     0[0-7]+             # Octal form
   | [0-9]+              # Decimal form
   | x[0-9a-fA-F]+       # Hexadecimal form
 )
 ;                       # Trailing semicolon
""", re.VERBOSE)
```

  - ✓ 문자열에 사용된 whitespace는 컴파일할 때 제거됨([ ]안은 제외)
  - ✓ 줄 단위로 #기호를 사용하여 주석문을 작성할 수 있음

❖ 백슬래시 문제

- "₩section"와 같이 ₩ 문자가 포함된 문자열을 찾을 때 혼란을 줌

```
₩section
```

  - ✓ ₩s 문자가 whitespace로 해석되어 의도한 대로 매치가 이루어지지 않음
  - ✓ [₩t₩n₩r₩f₩v]ection 도 동일하게 whitespace로 해석됨
  - ✓ 정규식에서 사용한 ₩ 문자가 문자열 자체임을 알려 주기 위해 백슬래시 2개를 사용하여 이스케이프 처리를 해야 함

```
>>> p = re.compile('₩₩section')      ← ₩₩이 ₩로 변경되어 ₩section이 전달됨
>>> p = re.compile('₩₩₩₩section')   ← ₩₩₩₩ 이 ₩₩로 변경되어 ₩₩section이 전달됨
```

- ₩를 사용한 표현이 너무 복잡하여 Raw String 규칙이 생겨나게 됨

```
>>> p = re.compile(r'₩₩section')
```

  - ✓ 정규식 문자열 앞에 **r** 문자를 삽입하면, Raw String 규칙에 의하여 백슬래시 2개 대신 1개만 써도 2개를 쓴 것과 동일한 의미를 갖게 됨

❖ 그룹(Group)을 만드는 법, 전방 탐색 등 강력한 정규 표현식을 살펴봄

❖ **메타 문자**

+, *, [], {} 등의 메타 문자는 매치가 진행될 때 매치되고 있는 문자열의 위치가 변경되지만(소비된다고 표현), 문자열을 소비시키지 않는 메타 문자도 있음

- |
  - ✓ | 메타 문자는 or과 동일한 의미로 사용됨

```
>>> 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 의미함

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

  - ✓ re 모듈의 축약된 형태
  - ✓ re.MULTILINE을 사용할 경우에는 여러 줄의 문자열일 때 각 줄의 처음과 일치함

- $
  - ✓ $ 메타 문자는 문자열의 끝과 매치함을 의미함

```
>>> 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
```

- ₩A
  - ✓ ₩A는 문자열의 처음과 매치됨을 의미함
  - ✓ ₩A는 줄과 상관없이 전체 문자열의 처음하고만 매치됨

- ₩Z
  - ✓ ₩Z는 문자열의 끝과 매치됨을 의미함
  - ✓ ₩Z는 줄과 상관없이 전체 문자열의 끝과 매치됨

- ₩b
  - ✓ ₩b는 단어 구분자(Word boundary)이며, 단어는 whitespace에 의해 구분됨

```
>>> p = re.compile(r'₩bclass₩b')

>>> print(p.search('no class at all'))
<re.Match object; span=(3, 8), match='class'>

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

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

  - ✓ ₩b는 파이썬 리터럴 규칙에 의하면 백스페이스(BackSpace)를 의미하므로, 구분자임을 알려 주기 위해 r'₩bclass₩b'처럼 Raw string임을 알려주는 기호r을 반드시 붙여 주어야 함

- ₩B
  - ✓ ₩b의 반대로, ₩B는 whitespace로 구분된 단어가 아닌 경우에만 매치됨

```
>>> 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 안됨

❖ 그루핑

- ABC 문자열이 계속해서 반복되는지 조사하는 정규식을 작성하고 싶을 때 사용됨

```
(ABC)+
```

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

- '₩w+₩s+₩d+[-]₩d+[-]₩d+'은 '이름 + " " + 전화번호' 형태의 문자열을 찾는 정규식

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

- 그룹핑하고 '이름' 부분만 뽑아내기

```
>>> 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) 메소드를 사용하여 그루핑된 부분의 문자열만 뽑아낼 수 있음

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

- '전화번호' 부분만 뽑아내기

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

- '국번' 부분만 뽑아내기

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

✓ 그룹이 중첩되어 있는 경우는 바깥쪽부터 시작하여 안쪽으로 들어갈수록 인덱스가 증가함

- 그루핑된 문자열 재참조하기

```
>>> 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 의미
- ✓ 2개의 동일한 단어를 연속적으로 사용해야만 매치됨
- ✓ **₩1**은 정규식의 그룹 중 첫 번째 그룹을 가리키고, **₩2**는 두 번째 그룹을 참조함

❖ 그루핑된 문자열에 이름 붙이기

- 정규식 안에 그룹이 많고 그룹이 추가 및 삭제되면, 그 그룹을 인덱스로 참조한 프로그램도 모두 변경해 주어야 하는 위험이 있음
- 이러한 이유로 그룹 이름을 지정할 수 있게 했음

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

✓ 그룹에 이름을 지어 주려면, (₩w+) → (**?P<name>**₩w+) 변경하면 됨

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

✓ name이라는 그룹 이름으로 참조할 수 있음

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

✓ 그룹 이름도 정규식 안에서 재참조할 때는 (**?P=그룹이름**) 확장 구분을 사용함

❖ 전방 탐색

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

".+:"과 일치하는 문자열로 http:를 돌려줌

✓ http:라는 검색 결과에서 :을 제외하고 싶을 때, 전방 탐색임

정규식	종류	설명
(?=...)	긍정형 전방 탐색	...에 해당하는 정규식과 매치되어야 하며, 조건이 통과되어도 문자열이 소비되지 않음
(?!...)	부정형 전방 탐색	...에 해당하는 정규식과 매치되지 않아야 하며, 조건이 통과되어도 문자열이 소비되지 않음

- **긍정형 전방 탐색**

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

✓ http:라는 검색 결과에서 :을 제외하고 돌려줌

- **부정형 전방 탐색**

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

✓ 확장자가 bat가 아닌 경우에만 통과된다는 의미임

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

✓ 확장자가 bat 또는 exe가 아닌 경우에는 통과된다는 의미임

❖ 문자열 바꾸기

- sub 메소드를 사용하면 정규식과 매치되는 부분을 다른 문자로 쉽게 바꿀 수 있음

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

  - ✓ sub 메소드의 첫 번째 매개변수는 '바꿀 문자열'이 되고, 두 번째 매개변수는 '대상 문자열'이 됨
  - ✓ 바꾸기 횟수를 제어하려면, 세 번째 매개변수로 count 값을 넘기면 됨

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

- **sub 메소드와 유사한 subn 메소드**

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

  - ✓ 반환 결과를 튜플(변경된 문자열, 발생한 횟수)로 돌려줌

- **sub 메소드를 사용할 때 참조 구문 사용하기**

```
>>> 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
```

✓ '**₩g<그룹 이름>**'을 사용하면 정규식의 그룹 이름을 참조할 수 있게 됨

```
>>> 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
```

✓ 그룹 이름 대신 참조 번호를 사용해도 됨

- **sub 메소드의 매개변수로 함수 넣기**

```
>>> 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.'
```

❖ Greedy vs Non-Greedy

```
>>> 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 * 메타 문자는 탐욕스러워서(Greedy) 최대한의 문자열을 모두 소비해 버림

✓ non- greedy 문자인 ?를 사용하면 *의 탐욕을 제한 할 수 있음

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

✓ non-greedy 문자인 ?는 *?, +?, ??, {m,n}?와 같이 사용할 수 있음