11장 파이썬 정규 표현식

- 정규 표현식은 꽤 오랜 기간 코드를 작성해 온 프로그래머라도 잘 모를 수 있는 고급 주제여서 초보자에게 어려울 수 있다. 하지만 정규 표현식을 배워 익히기만 하면 아주 달콤한 열매를 맛볼 수 있다.
- 프로그래밍 입문자가 이해하기에는 어려운 내용이니 부담 갖지 말고 편하게 학습한다.

11.1 정규 표현식 살펴보기

- 정규 표현식(Regular Expression)은 복잡한 문자열을 처리할 때 사용하는 기법으로, 파이썬만의 고유 문법이 아니라 문자열을 처리하는 모든 곳에서 사용된다.
- 정규 표현식을 사용하면 아래 예제와 같이 코드가 상당히 간결해진다.

[ch11_regExpressions/ex01_re.py]

```
#-*- coding: utf-8 -*-

# 과제: 주민등록번호를 포함하고 있는 텍스트가 있다. 이 텍스트에 포함된 모든 주민등록번호의
# 뒷자리를 * 문자로 변경하시오.
#
# 정규 표현식을 전혀 모르면 ...
# 1. 전체 텍스트를 공백 문자로 나눈다.(split)
# 2. 나누어진 단어들이 주민등록번호 형식인지 조사한다.
# 3. 단어가 주민등록번호 형식이라면 뒷자리를 '*'로 변환한다.
# 4. 나누어진 단어들을 다시 조립한다.
data = """
park 800905-1049118
kim 700905-1059119
"""

result = []  # ['park 800905-*******', 'kim 700905-*******']
for line in data.split("\n"):
    word_result = [] # ['park', '800905-*******']
    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))

'''
park 800905-*******
kim 700905-*******

'''


# 반면에 정규식을 사용하면 다음처럼 훨씬 간편하고 직관적인 코드를 작성할 수 있다.
import re # 정규 표현식을 사용하기 위한 re모듈
data = """
park 800905-1049118
kim 700905-1059119
"""
pat = re.compile("(\d{6})[-]\d{7}")
print(pat.sub("\g<1>-*******",data))  # \g<그룹이름>을 참조한다. 즉 (\d{6})를 지칭한다.
'''
park 800905-*******
kim 700905-*******
'''
```

11.2 정규 표현식 시작하기

11.2.1 메타 문자

- 메타 문자란 원래 그 문자가 가진 뜻이 아닌 특별한 용도로 사용하는 문자를 말한다.
- 정규 표현식에 아래 메타 문자를 사용하면 특별한 의미를 갖게 된다.

```
. ^ $ * + ? { } [ ] \ | ( )
```

(1) 문자 클래스 []

- 문자 클래스로 만들어진 정규식은 '[와] 사이의 문자들과 매치'라는 의미를 갖는다. 예를 들어 ...
  - [a-c] or [abc]: a, b, c중 한 개의 문자와 매치, 예) a, before, dude(x)
  - [a-zA-Z]: 알파벳 모두
  - [0-9]: 숫자
- 자주 사용하는 문자 클래스의 별도 표기법
  - \d: 숫자와 매치, [0-9]와 동일한 표현식이다.
  - \D: 숫자가 아닌 것과 매치, [^0-9]와 동일한 표현식이다.
  - \s: whitespace 문자와 매치, [\t\n\r\f\v]와 동일한 표현식이다. 맨 앞의 빈 칸은 공백 문자(space)를 의미한다.
  - \S: whitespace 문자가 아닌 것과 매치, [^\t\n\r\f\v]와 동일한 표현식이다.
  - \w: 문자+숫자(alphanumeric)와 매치, [a-zA-Z0-9]와 동일한 표현식이다.
  - \W: 문자+숫자가 아닌 문자와 매치, [^a-zA-Z0-9]와 동일한 표현식이다.

(2) Dot(.)

- 줄바꿈 문자인 \n를 제외한 모든 문자와 매치됨을 의미한다.
  - a.b: a와 b 사이에 줄바꿈 문자를 제외한 어떤 문자가 들어가도 모두 매치, 예) aab, a0b, abc(x)
  - a[.]b: a와 b 사이에 Dot(.) 문자가 있으면 매치, 예) a.b, a0b(x)

(3) 반복(*)

- * 바로 앞에 있는 문자가 0부터 무한대로 반복될 수 있다는 의미이다.
  - ca*t: 문자 바로 앞에 있는 a가 0번 이상 반복되면 매치, 예) ct, cat, caaaat

(4) 반복(+)

- 최소 1번 이상 반복될 때 사용한다. 즉, *가 반복 횟수 0부터라면 +는 반복 횟수 1부터인 것이다.
  - ca+t: +문자 바로 앞에 있는 a가 1번 이상 반복되면 배치, 예) ct(x), cat, caaat

(5) 반복({m,n},?)

- {} 메타 문자를 이용하면 반복 횟수를 고정시킬 수 있다.
  - ca{2}t: a가 2번 반복되면 매치, 예) cat(x), caat
  - ca{2,5}t: a가 2~5번 반복되면 매치, 예) cat(x), caat, caaaaat, caaaaaat(x)
  - ab?c: b가 0~1번 사용되면 매치, 예) abc, ac

11.2.2 파이썬에서 정규 표현식을 지원하는 re 모듈

- 파이썬은 정규 표현식을 지원하기 위해 re(regular expression) 모듈을 제공한다.
- re 모듈은 파이썬이 설치될 때 자동으로 설치되는 기본 라이브러리로, 사용 방법은 다음과 같다.

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

11.2.3 정규식을 이용한 문자열 검색

- 컴파일된 패턴 객체는 다음과 같은 4가지 메소드를 제공한다.

메서드	목적
match()	문자열의 처음부터 정규식과 매치되는지 조사한다.
search()	문자열 전체를 검색하여 정규식과 매치되는지 조사한다.
findall()	정규식과 매치되는 모든 문자열(substring)을 리스트로 리턴한다.
finditer()	정규식과 매치되는 모든 문자열을 반복 가능한 객체로 리턴한다.

[ch11_regExpressions/ex02_search.py]

```
#-*- coding: utf-8 -*-

# 파이썬에서 정규 표현식을 지원하는 re 모듈
import re

# 정규식을 이용한 문자열 검색
p = re.compile('[a-z]+') # 컴파일된 패턴 객체, a~z 문자가 최소 1번 이상 반복되어야 한다.

# match: 문자열의 처음부터 정규식과 매치되는지 조사한다.
m = p.match("python")
print(m)
print(m.group())
# <re.Match object; span=(0, 6), match='python'>
# python
m = p.match("3 python")
print(m)
# None

# search: 문자열 전체를 검색하여 정규식과 매치되는지 조사한다.
m = p.search("python")
print(m)
print(m.group())
# <re.Match object; span=(0, 6), match='python'>
# python
m = p.search("3 python")
print(m)
print(m.group())
# <re.Match object; span=(2, 8), match='python'>
# python
```

```
# findall: 정규식과 매치되는 모든 문자열을 리스트로 리턴한다.
result = p.findall("life is too short short3")
print(result)
# ['life', 'is', 'too', 'short', 'short']

# finditer: 정규식과 매치되는 모든 문자열을 반복 가능한 객체로 리턴한다. 반복 가능한 객체가 포함하는 각각의 요소는 match 객체이다.
result = p.finditer("life is too short")
print(result)
# <callable_iterator object at 0x00000000025AABE0>
```

11.2.4 match 객체의 메서드

■ match 메서드와 search 메서드를 수행한 결과로 돌려준 match 객체의 메서드를 사용하면 "어떤 문자열이 매치되었는가?", "매치된 문자열의 인덱스는 어디부터 어디까지인가?"와 같은 궁금증을 해결할 수 있다.

메서드	목적
group()	매치된 문자열을 리턴한다.
start()	매치된 문자열의 시작 위치를 리턴한다.
end()	매치된 문자열의 끝 위치를 리턴한다.
span()	매치된 문자열의 (시작,끝)에 해당되는 튜플을 리턴한다.

[ch11_regExpressions/ex02_match.py]

```
#-*- coding:utf-8 -*-

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

# match 객체의 메서드
m = p.match("python")
print(m.group())  # 매치된 문자열을 리턴한다.
# 'python'
print(m.start())  # 매치된 문자열의 시작 위치를 리턴한다.
# 0
print(m.end())    # 매치된 문자열의 끝 위치를 리턴한다.
# 6
print(m.span())   # 매치된 문자열의 (시작,끝)에 해당되는 튜플을 리턴한다.
# (0, 6)

# search 객체의 메서드
m = p.search("3 python")
print(m.group())
# 'python'
print(m.start())
# 2
print(m.end())
# 8
print(m.span())
# (2, 8)
```

~~11.2.5 컴파일 옵션~~

■ 정규식을 컴파일할 때 다음 옵션을 사용할 수 있다.

옵션명	약자	설명
DOTALL	S	줄바꿈 문자를 포함하여 모든 문자와 매치할 수 있도록 한다.
IGNORECASE	I	대.소문자에 관계 없이 매치할 수 있도록 한다.
MULTILINE	M	여러 줄과 매치할 수 있도록 한다. (^, $ 메타 문자의 사용과 관계가 있는 옵션이다.)
VERBOSE	X	verbose 모드를 사용할 수 있도록 한다. (정규식을 보기 편하게 만들 수도 있고 주석 등을 사용할 수도 있다.)

[ch11_regExpressions/ex02_compile.py]

```
#-*- coding:utf-8 -*-

# DOTALL, S
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)


# IGNORECASE, I
p = re.compile('[a-z]', re.I)
p.match('python')
p.match('Python')
p.match('PYTHON')


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

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


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

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

```
""", re.VERBOSE)
```

11.2.6 백슬래시 문제

- 정규 표현식을 파이썬에서 사용하려 할 때 혼란을 주는 요소가 한 가지 있는데, 바로 백슬래시(\)이다.

```
p = re.compile('\section')   # \s문자가 whitespace로 해석되어 의도한 대로 매치가 이루어지지 않는다.
p = re.compile('\\section')  # \문자가 문자열 그 자체임을 알려주기 위해 백슬래시 2개를 사용한다.
p = re.compile('\\\\section')  # 정규식 엔진에 \\문자를 전달하려면 파이썬은 \\\\처럼 4개나 사용해야 한다.
p = re.compile(r'\\section')   # Raw String 규칙은 백슬래시 1개만 써도 2개를 쓴 것과 동일한 의미를 갖게 된다.
```

11.3 강력한 정규 표현식의 세계로

11.3.1 메타 문자

- 이전에 살펴본 메타 문자들은 모두 매치되는 문자열을 소모시킨다. 문자열이 일단 소모되어 버리면 그 부분은 검색 대상에서 제외되지만 소모되지 않은 경우에는 다음에 또 다시 검색 대상이 된다.
- +,*,[],{} 등의 메타 문자는 매치가 진행될 때 현재 매치되고 있는 문자열의 위치가 변경(소모)된다. 하지만 이와 달리 문자열을 소모시키지 않는(zero-width assertions) 메타 문자들도 있다.

[ch11_regExpressions/ex03_meta.py]

```
#-*- coding: utf-8 -*-
import re

# | 메타 문자는 or과 동일한 의미로 사용된다.
p = re.compile('Crow|Servo')
m = p.match('CrowHello')
print(m)
print(m.group())
# <re.Match object; span=(0, 4), match='Crow'>
# Crow


# ^ 메타 문자는 문자열의 맨 처음과 일치함을 의미한다.
print(re.search('^Life', 'Life is too short'))
print(re.search('^Life', 'My Life'))
# <re.Match object; span=(0, 4), match='Life'>
# None


# $ 메타 문자는 ^ 메타 문자와 반대의 경우이다. 즉, $는 문자열의 끝과 매치함을 의미한다.
print(re.search('short$', 'Life is too short'))
print(re.search('short$', 'Life is too short, you need python'))
# <re.Match object; span=(12, 17), match='short'>
# None


# \b는 단어 구분자(word boundary)이다. 보통 단어는 whitespace에 의해 구분이 된다.
p = re.compile(r'\bclass\b')
print(p.search('no class at all'))
```

```
print(p.search('the declassified algorithm'))
# <re.Match object; span=(3, 8), match='class'>
# None

# \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
```

11.3.2 그룹핑

[ch11_regExpressions/ex03_grouping.py]

```
#-*- coding: utf-8 -*-

import re

# ABC라는 문자열이 계속에서 반복되는지 조사하는 정규식
p = re.compile('(ABC)+')
m = p.search('ABCABCABC OK?')
print(m)
# <re.Match object; span=(0, 9), match='ABCABCABC'>
print(m.group(0)) # group(0)는 매치된 전체 문자열
#ABCABCABC

p = re.compile(r"(\w+)\s+((\d+)[-]\d+[-]\d+)")
m = p.search("park 010-1234-1234")
print(m.group(1)) # group(1)는 첫 번째 그룹에 해당되는 문자열
# park
print(m.group(2)) # group(2)는 두 번째 그룹에 해당되는 문자열
# 010-1234-1234
print(m.group(3)) # group(3)는 국번 부분을 그룹핑한 것이다.
# 010
```

11.3.3 그룹핑된 문자열에 이름 붙이기

■ 정규식은 그룹을 만들 때 그룹 이름을 지정할 수 있다. 예) (?P<groupName>\w+)\s+

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

11.3.4 전방 탐색

■ 전방 탐색에는 긍정(Positive)과 부정(Negative)의 두 종류가 있다.

정규식	종류	설명
(?=...)	긍정형 전방 방식	...에 해당하는 정규식과 매치되어야 하며 조건이 통과되어도 문자열이 소비

		되지 않는다.
(?!...)	부정형 전방 방식	...에 해당하는 정규식과 매치되지 않아야 하며 조건이 통과되어도 문자열이 소비도지 않는다.

```
# 정규식 ".+:"과 일치하는 문자열로 http:를 돌려준다.
p = re.compile(".+:")
m = p.search("http://google.com")
print(m.group())
# http:


# 기존 정규식과 검색에서는 동일한 효과를 발휘하지만 :에 해당하는 문자열이 정규식 엔진에 의해 소비되지 않아
# 검색 결과에서는 :이 제거된 후 돌려준다.
p = re.compile(".+(?=:)")
m = p.search("http://goople.com")
print(m.group())
# http
```

11.3.5 문자열 바꾸기

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

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

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

~~11.3.6 Greedy vs Non-Greedy~~

```
# * 메타 문자는 greedy(탐욕스러워서) 매치할 수 있는 최대한의 문자열을 모두 소모시킨다.
s = '<html><head><title>Title</title>'
print(len(s))
# 32
print(re.match('<.*>', s).span())
# (0, 32)
print(re.match('<.*>', s).group())
# <html><head><title>Title</title>

# non-greedy 문자인 ?을 사용하면 *의 탐욕을 제안할 수 있다.
print(re.match('<.*?>', s).group())
# <html>
```

[과제] 연습문제

Q1. 정규 표현식 - 문자열

다음 중 정규식 "a[.]{3,}b"과 매치되는 문자열은 무엇일가?

[정답]
a....b # "a" 문자 1개 + "." 문자 3개 이상 + "b" 문자 1개

Q2. 정규 표현식 - 인덱스 번호
다음 코드의 결과값은 무엇일까?

```
import re
p = re.compile('[a-z]+')
m = p.search("5 python")
m.start() + m.end()
```

[정답]
10 # 시작 인덱스(m.start())=2, 마지막 인덱스(m.end())=8

Q3. 정규 표현식 - 핸드폰 번호
다음과 같은 문자열에서 핸드폰 번호 뒷자리인 숫자 4개를 ####로 바꾸는 프로그램을 정규식을 이용하여 작성해 보자.

```
"""
park 010-9999-9988
kim 010-9909-7789
lee 010-8789-7766
"""
```

[정답]

```
import re

s = """
park 010-9999-9988
kim 010-9909-7789
lee 010-8789-7766
"""

pat = re.compile("(\d{3}[-]\d{4})[-]\d{4}")
result = pat.sub("\g<1>-####", s)

print(result)
```

Q4. 정규 표현식 - 이메일 주소
다음은 이메일 주소를 나타내는 정규식이다. 이 정규식은 park@naver.com, kim@daum.net, lee@myhome.co.kr 등과 매치된다. 긍정형 전방 탐색 기법을 이용하여 .com, .net이 아닌 이 메일 주소는 제외시키는 정규식을 작성해 보자.

[정답]

```
import re

pat = re.compile(".*[@].*[.](?:com$|net$).*$")
```


```
print(pat.match("park@naver.com"))
print(pat.match("kim@daum.net"))
print(pat.match("lee@myhome.co.kr"))
```