정규 표현식

- 텍스트에 포함된 특정 문자열을 검색하는 것
- ex) 주민번호 뒷자리를 '*' 문자로 변경 하시오
    - 문자가 주민번호 형식(123456-1234567) 인지 검사
    - '-(하이픈)' 뒷자리를 '*'로 변경

re 모듈

- 파이썬에서 정규 표현식 지원 (cf. Java에서의 Pattern 객체와 유사)
  사용법 1) Compile → Matching

```python
import re # 정규표현식 모듈

p = re.compile('[a-z]+') # re 내장모듈 내(.) compile 메서드를 사용.
                         # compile 메서드는 "패턴 객체"를 반환한다.

m = p.match("python")    # 패턴 객체(p)에는 또다시 검색 메서드가 있다.
```

  사용법 2) Compile + Matching

```python
import re # 정규표현식 모듈

m = re.match('[a-z]+', 'Python')
```

- Compile을 사용하면 패턴 객체(p)를 재사용 가능 → 시간 단축

패턴 객체(p)

- re.compile() – 정규식 패턴을 파이썬이 사용할 수 있는 정규식 객체로 컴파일
  - match(), search()와 같은 메소드를 통해 사용됨

- 검색, 반환, 위치, 변경, 분리 관련 된 작업을 지원

```
1   import re # 정규표현식 모듈
2
3   p = re.compile('a')
4   print(dir(p))
```

```
['__class__', '__copy__', '__deepcopy__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '
__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'findall',
 'finditer', 'flags', 'fullmatch', 'groupindex', 'groups', 'match', 'pattern', 'scanner', 'search', 'split', 'sub'
, 'subn']
```

메타 문자

- 메타문자란 원래 그 문자가 가진 뜻이 아닌 특별한 용도로 사용되는 문자
- `. ^ $ + ? { } [ ] \ | ( )`

메타 문자	설명
[ ]	문자 클래스
.	¥n을 제외한 모든 문자와 매치 (점 하나는 글자 하나를 의미)
*	0회 이상 반복 (업어도 상관 없음)
+	1회 이상 반복 (무조건 한번 이상 등장해야 함)
{m, n}	m회 이상 n회 이하
|	or 조건식을 의미
^	문자열의 시작 의미
$	문자열의 끝을 의미
?	0회 이상 1회 이하
¥	이스케이프, 또는 메타 문자를 일반 문자로 인식하게 한다
( )	그룹핑, 추출할 패턴을 지정한다.

[,]

- Character class
- 어떤 내용이든지 들어 갈 수 있다.
- [,]에 포함 된 내용들은 or로 연결된다.
  - 하이픈(-) 사용 가능 ex) [a-z]는 a부터 z까지, [0-9]는 0부터 9까지
  - 두 문자 사이의 범위 (From ~ To)
- [abc]의 의미는 '**a** or **b** or **c**'의 의미.
  - 즉, 'a, b, c' 중에서 한 개의문자와 매칭

[,]

1) 정규표현식 [abc]가 있다. 다음중 매칭되는 것은?

   (1) a (2) before (3) dude

2) 정규표현식 [a-c]가 있다. 다음중 같은 의미는?

   (1) a 또는 c (2) a 또는 b 또는 c (3) [abc] (4) a 그리고 b 그리고 c

3) 정규 표현식 a[a-z0-9]z가 있다. 다음중 매칭(Y) 되는 것은? (응용)

   (1) a!012z (2) aBz (3) a999z (4) azX09z (5) a9z

[,]

원래 표현식	축약 표현	부연 설명	사용처
[0-9]	¥d	숫자를 찾는다	숫자
[^0-9]	¥D	숫자가 아닌 것을 찾는다	텍스트 + 특수문자 + 화이트스페이스
[¥t¥n¥r¥f¥v]	¥s	whitespace 문자인 것을 찾는다	스페이스, TAB, 개행(new line)
[^ ¥t¥n¥r¥f¥v]	¥S	whitespace 문자가 아닌 것을 찾는다	텍스트 + 특수문자 + 숫자
[a-zA-Z0-9]	¥w	문자+숫자인 것을 찾는다. (특수문자는 제외. 단, 언더스코어 포함)	텍스트 + 숫자
[^a-zA-Z0-9]	¥W	문자+숫자가 아닌 것을 찾는다.	특수문자 + 공백

Dot(.)

- 도트는 하나의 문자 하나를 의미
  - 도트 두개는 문자 두개를 의미
  - 문자는 숫자(0-9)나 특수문자(!@#$%^& 등)을 포함

1) 정규 표현식 a.z가 있다. 다음중 매칭(Y) 되는 것은?
   (1) akdz (2) axz (3) abdeZ (4) aBDEz (5) axcz

2) 정규표현식 a.z가 있다. 다음중 매칭(Y) 되는 것은?
   (1) a&z (2) a!z (3) a0z (4)akz

*, +

- * ➔ 바로 앞 문자가 0번 이상 반복
- + ➔ 바로 앞 문자가 1번 이상 반복

표현식	설명	매칭 예시
.*	선행문자가 .이므로 하나 이상의 문자를 포함하는 문자열(공백 문자열 제외)	모든 문자가 출력될 거라고 생각하기 쉽지만, .이 공백 문자열은 제외하기 때문에 첫줄만 출력된다. 모두 선택을 하고자 한다면 .+로 출력하는 것이 적절해보인다.`
ab*c	b를 0번 또는 여러번 반복되도 상관없음	ac, az, a123c, abbbb
like.*	선행문자가 .이므로 like에 0 또는 하나 이상의 문자가 추가된 문자열	like, likely ,likelihood

*, +

- * ➔ 바로 앞 문자가 0번 이상 반복
- + ➔ 바로 앞 문자가 1번 이상 반복

형식	설명	매칭 예시
ca+t	a가 1번 이상 반복되야 함	cat, caaaat, caaaaaaaat
car+ot	r이 1번 이상 반복되어야 함	carrot
like.+	선행문자가 .이므로 like에 하나 이상 문자열이 추가되어야 함	liekly, liker (단, like는 안된다.)
[A-Z]+	대문자로만 이루어진 문자열	ABC, DEF, ZAX

*, +

```python
import re # 정규표현식 모듈

source = "Luke Skywarker 02-123-4567 luke@daum.net"  # \w와 \w+의 차이

m1 = re.findall('\w', source) # 단어가 아니라 문자 단위로 출력
m2 = re.findall('\w+', source) # 단어 단위로 출력
print("m1 : ", m1)
print("m2 : ", m2)

```

```
m1 :  ['L', 'u', 'k', 'e', 'S', 'k', 'y', 'w', 'a', 'r', 'k', 'e', 'r', '0', '2', '1', '2', '3', '4', '5', '6', '7'
, 'l', 'u', 'k', 'e', 'd', 'a', 'u', 'm', 'n', 'e', 't']
m2 :  ['Luke', 'Skywarker', '02', '123', '4567', 'luke', 'daum', 'net']
```

반복 { }

- 앞 문자의 반복 횟수 지정

표현식	설명	"ct cat caat caaat caaaat"
ca{2}t	a가 2회 반복되어야 함	caat
ca{2,5}t	a가 2회 이상 5회 이하 반복되어야 함	caat, caaat, caaaat
ca{0, }t	반복횟수 0회 이상 (*와 동일)	ct, cat, caat, caaat, caaaat
cat{0, 1}t	반복횟수 0회 ~ 1회 이하 (?와 동일)	ct, cat
cat{ , 3}	반복횟수 0회 이상 ~ 3회 이하	ct, cat, caat, caat

반복 {}

```python
import re

source = "python pytthon pyttttthon pyttttttttthon pythhhhhon pyhon"
m1 = re.findall("pyt{2}hon", source)
m2 = re.findall("pyt{2,5}hon", source)
m3 = re.findall("pyt{0,}hon", source)
m4 = re.findall("pyt{0,1}hon", source)
m5 = re.findall("pyth{,5}on", source)
print("m1 : ", m1)
print("m2 : ", m2)
print("m3 : ", m3)
print("m4 : ", m4)
print("m5 : ", m5)
```

```
m1 :  ['pytthon']
m2 :  ['pytthon', 'pyttttthon']
m3 :  ['python', 'pytthon', 'pyttttthon', 'pyttttttttthon', 'pyhon']
m4 :  ['python', 'pyhon']
m5 :  ['python', 'pythhhhhon']
```

반복 ?

- 선행하는 문자가 있어도 되고, 없어도 됨

표현식	설명	예시
ab?c	b가 있어도 되고 없어도 된다.	ac, ab

```
import re

source = "py pyt pyth pytho python pycham"
m1 = re.findall("py?", source)
print("m1 : ", m1)
```

```
m1 :  ['py', 'py', 'py', 'py', 'py', 'py']
```

Greedy vs. Non-greedy

- *, + → Greedy Operator (최대 일치)
- ? → Non-greedy Operator (최소 일치)

```
import re
source = '<li>나이키</li><li>아디다스</li><li>퓨마</li>'
m = re.match('<li>.*</li>', source)
if m:
    print(m.group())
```

```
<li>나이키</li><li>아디다스</li><li>퓨마</li>
```

Greedy vs. Non-greedy

- *, + → Greedy Operator (최대 일치)
- ? → Non-greedy Operator (최소 일치)

```
import re
source = '<li>나이키</li><li>아디다스</li><li>퓨마</li>'
m = re.match('<li>.*?</li>', source)
if m:
    print(m.group())
```

```
<li>나이키</li>
```

- `.`은 문자 1개를 의미
- `*`는 해당 패턴이 0회 이상 올 수 있다.
- `.*`은 문자가 있거나 없을수도 있다.
- `.*Lady` : 앞에 아무 문자열(또는 빈) 이후 Lady로 끝나는 패턴을 의미한다.

OR |

```
import re
p = re.compile('a|b') # a 또는 b를 찾는다.
m = p.findall("abcdefg")
print(m)
```

```
['a', 'b']
```

^

```
import re
m1 = re.findall("^Life", "Life is too short")
m2 = re.findall("^is", "Life is too short")
print("m1 결과 : ", m1)
print("m2 결과 : ", m2)
```

```
m1 결과 :  ['Life']
m2 결과 :  []
```

$

```
import re
m1 = re.findall("short$", "Life is too short")
m2 = re.findall("short$", "Life is too short. So what?")

print("m1 결과 : ", m1)
print("m2 결과 : ", m2)
```

```
m1 결과 :  ['short']
m2 결과 :  []
```

( ) : 그룹핑

```
import re
p = re.compile(r"(\w+)\s+(\d+[-]\d+[-]\d+)")
m = p.search("park 010-1234-1234")
print(m.group())
print(m.group(1))
print(m.group(2))
```

```
park
010-1234-1234
```

match 객체의 메소드

메서드	설명
match	문자열의 처음부터 정규식과 매치되는지 조사
search	문자열 전체를 검색하여 정규식과 매치되는지 조사
findall	정규식과 매치되는 모든 문자열(substring)을 리스트로 리턴
split	패턴으로 나누기
sub	패턴 대체하기

match vs. search

- match – 처음이 일치하지 않으면 None 반환
- search – 처음이 일치하지 않더라도 전체를 검색

- 원하는 것을 검색하게 되면 작동 ***중지***

```
import re

# match와 search
source13 = '''All That Is Gold Does Not Glitter"
'''
match = re.match("Not", source13)
search = re.search("Not", source13)

if match:
    print("match : ", match.group())

if search:
    print("search : ", search.group())
```

```
search :  Not
```

```
import re

# match와 search
source13 = '''All That Is Gold Does Not Glitter"
'''
match = re.match("All", source13)
search = re.search("All", source13)

if match:
    print("match : ", match.group())

if search:
    print("search : ", search.group())
```

```
match :  All
search :  All
```

findall

- findall() – 정규식과 매치되는 모든 문자열을 [리스트] 형식으로 반환

```
p = re.compile('[a-z]+') # 소문자(a-z)가 1회 이상 반복되는 것을 검색
m = p.findall("Life is to short")
print(m)
print(type(m))
```

```
['ife', 'is', 'to', 'short']
<class 'list'>
```

finditer

- finditer() – 정규식과 매치되는 모든 문자열을 iterator 객체로 반환

```
text = "Life is to short"
p = re.compile('[a-z]+') # 소문자(a-z)가 1회 이상 반복되는 것을 검색
m = p.finditer(text)
print(m)
print(type(m))
for r in m:
  if r:
    print(r, text[r.start():r.end()], r.group())
```

```
<callable_iterator object at 0x00000216222EB550>
<class 'callable_iterator'>
<_sre.SRE_Match object; span=(1, 4), match='ife'> ife ife
<_sre.SRE_Match object; span=(5, 7), match='is'> is is
<_sre.SRE_Match object; span=(8, 10), match='to'> to to
<_sre.SRE_Match object; span=(11, 16), match='short'> short short
```

match 객체의 메소드

method	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.

```
p = re.compile('[a-z]+')
m = p.match("python")
print(m.group())
print(m.start())
print(m.end())
print(m.span())
```

```
python
0
6
(0, 6)
```