정규 표현식

정규 표현식 기초 메타 문자

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

문자 클래스 [ ]

정규 표현식이 [abc]라면 이 표현식의 의미는 "a, b, c 중 한 개의 문자와 매치"를 뜻한다. 이해를 돕기 위해 문자열 "a", "before", "dude"가 정규식 [abc]와 어떻게 매치되는지 살펴보자.

- "a"는 정규식과 일치하는 문자인 "a"가 있으므로 매치
- "before"는 정규식과 일치하는 문자인 "b"가 있으므로 매치
- "dude"는 정규식과 일치하는 문자인 a, b, c 중 어느 하나도 포함하고 있지 않으므로 매치되지 않음

[ ] 안의 두 문자 사이에 하이픈(-)을 사용하면 두 문자 사이의 범위(From - To)를 의미한다. 예를 들어 [a-c]라는 정규 표현식은 [abc]와 동일하고 [0-5]는 [012345]와 동일하다.

다음은 하이픈(-)을 사용한 문자 클래스의 사용 예이다.

- [a-zA-Z] : 알파벳 모두
- [0-9] : 숫자

문자 클래스 안에 `^` 메타 문자를 사용할 경우에는 반대(not)라는 의미를 갖는다. 예를 들어 `[^0-9]`라는 정규 표현식은 숫자가 아닌 문자만 매치된다.

```
[0-9] 또는 [a-zA-Z] 등은 무척 자주 사용하는 정규 표현식이다. 이렇게 자주 사용하는 정규식은 별도의 표기법으로 표현할 수 있다. 다음을 기억해 두자.

\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 공백문자(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+숫자(alphanumeric)가 아닌 문자와 매치, [^a-zA-Z0-9_]와 동일한 표현식이다.
```

> 1. 특수 문자가 있는 문자를 찾아보자.
> 2. 숫자를 포함하지 않는 문자를 찾아보자.
> 3. 공백이 포함되지 않은 문자를 찾아보자.
> 4. babo 문자가 포함된 문자를 찾아보자.

5. ap로 시작하고 e로 끝나는 문자를 찾아보자.
6. !로 끝나는 문자를 찾아보자.
7. my로 시작하는 문자를 찾아보자.

```
import re

list = ["hello^^ everyone", "my name is choi jooho 2 time!!",
        "are you my babo?", "apple", "babo"]

# 1번
print("1번~~~~~~~~~~~~~~~~~~~~~")
# p = re.compile("\!|\?|\@|\#|\%|\^|\&|\*|\(|\)|\[|\]")
p = re.compile("[^ a-zA-Z0-9]")  # 숫자, 문자, 공백이 아닌것 (특수문자)
for s in list:
    m = p.search(s)
    try:
        print(m)
    except:
        pass
print("="*30)

# 2번
print("2번~~~~~~~~~~~~~~~~~~~~~")
p = re.compile("\d")
for s in list:
    m = p.search(s)
    if m == None:
        print(s)
print("="*30)

# 3번
print("3번~~~~~~~~~~~~~~~~~~~~~")
p = re.compile("\s")
for s in list:
    m = p.search(s)
    if m == None:
        print(s)
print("="*30)

# 4번
print("4번~~~~~~~~~~~~~~~~~~~~~")
p = re.compile("babo")
for s in list:
    m = p.search(s)
    print(m)
print("="*30)

# 5번
```

```
print("5번~~~~~~~~~~~~~~~~~~~~")
p = re.compile("ap.+e")
for s in list:
    m = p.match(s)  # findall
    print(m)

print("="*30)

# 6번
print("6번~~~~~~~~~~~~~~~~~~~~")
p = re.compile("!$")
for s in list:
    m = p.search(s)
    try:
        print(m)
    except:
        pass
print("="*30)

# 7번
print("7번 첫번째 방법~~~~~~~~~~~~~~~~~~~~")
p = re.compile("^my")
for s in list:
    m = p.match(s)
    try:
        print(m)
    except:
        pass
print("7번 두번째 방법~~~~~~~~~~~~~~~~~~~~")
p = re.compile("my.+")
for s in list:
    m = p.match(s)
    try:
        print(m)
    except:
        pass
print("7번 세번째 방법 search로 하면 틀림~~~~~~~~~~~~~~~~~~~~")
p = re.compile("my.+")
for s in list:
    m = p.search(s)  # search는 일부분만 검색해서  my가 중간에 있어도 찾음
    try:
        print(m)
    except:
        pass
print("="*30)
```

re 메서드

- match는 문자열의 처음부터 매칭되는 것을 찾는다
- search는 문자열의 일부부만 매칭되는 것을 찾는다.
- findall은 정규식과 매치되는 모든 문자열을 리스트로 돌려준다.

match 메서드

 image-20220714153811515

문자열 중 특정문자 하나라도 포함되는 것 찾기 (|) or문법

```
hello|world
```

> 문자열에 hello나 world가 포함되는 것을 찾는다.
>
> 이때 match가 아닌 search를 사용해야 한다.

특정 문자로 시작되거나, 끝나는 것 (^, $)

```
^hello  -> hello everyone
hello$  -> everyone hello
```

Dot(.)

```
a.b
```

위 정규식의 의미는 다음과 같다.

> "a + 모든문자 + b"

- abc
- acd
- akd

```
import re

s = "akb"

p = re.compile("a.b")
m = p.match(s)
print(m)
```

```
a[.]b
```

위 정규식의 의미는 다음과 같다.

> "a.b"

문자열 매치가 들어가면 무조건 가운데 . 이 들어와야 한다.

```
import re

s = "a.b"

p = re.compile("a[.]b")
m = p.match(s)
print(m)
```

반복(*)

```
ca*t
```

위 정규식의 의미는 다음과 같다.

> *의 의미는 바로 직전에 있는 a의 문자가 무한대로 반복될 수 있다는 의미이다.
>
> a가 포함되지 않아도 된다.

- ct
- caaaaat
- caat

- cat
- caaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaat

반복(+)

```
ca+t
```

위 정규식의 의미는 다음과 같다.

> +의 의미는 a가 최소 1번 이상 반복될 때 사용한다.
>
> a가 꼭 포함되어야 한다.

- ct 안됨
- cat
- caaaat

반복 ({m,n}, ?)

반복 횟수를 제한할 때 사용한다.

1. {m}

```
ca{3}t
```

> c로 시작하고 a를 반드시 3번 반복 t로 끝난다.

- caaat

```
import re

s = "caaat"

p = re.compile("ca{3}t")
m = p.match(s)
print(m)
```

2.{m,}

```
ca{3,}t
```

> c로 시작하고 a를 반드시 3이상 반복하고 t로 끝난다.

- caaat
- caaaaaaaaaat

```
import re

s = "caaaaaaaaat"

p = re.compile("ca{3,}t")
m = p.match(s)
print(m)
```

3. {,n}

```
ca{,3}t
```

> c로 시작하고 a가 3이하로 반복되고 t로 끝난다.

- ct
- cat
- caaat

4. {m,n}

```
ca{1,3}t
```

> c로 시작하고 a가 1개에서 3개 까지만 가능하다.

- cat
- caat
- caaat

```
import re

s = "caaat"

p = re.compile("ca{1,3}t")
m = p.match(s)
print(m)
```

있어도 되고 없어도 되는 ?

```
ab?c
```

> b가 있어도 되고 없어도 된다

```
s = "ac"

p = re.compile("ab?c")
m = p.match(s)
print(m)
```