정규 표현식
(Regular Expression)

정규 표현식(Regular Expression)

파이썬에서는 정규 표현식 모듈 re을 지원하므로, 이를 이용하면 특정 규칙이 있는 텍스트 데이터를 빠르게 정제할 수 있다.

특수 문자	설명
.	한 개의 임의의 문자를 나타냅니다. (줄바꿈 문자인 ₩n는 제외)
?	앞의 문자가 존재할 수도 있고, 존재하지 않을 수도 있습니다. (문자가 0개 또는 1개)
*	앞의 문자가 무한개로 존재할 수도 있고, 존재하지 않을 수도 있습니다. (문자가 0개 이상)
+	앞의 문자가 최소 한 개 이상 존재합니다. (문자가 1개 이상)
^	뒤의 문자로 문자열이 시작됩니다.
$	앞의 문자로 문자열이 끝납니다.
{숫자}	숫자만큼 반복합니다.
{숫자1, 숫자2}	숫자1 이상 숫자2 이하만큼 반복합니다. ?, *, +를 이것으로 대체할 수 있습니다.
{숫자,}	숫자 이상만큼 반복합니다.
[ ]	대괄호 안의 문자들 중 한 개의 문자와 매치합니다. [amk]라고 한다면 a 또는 m 또는 k 중 하나라도 존재하면 매치를 의미합니다. [a-z]와 같이 범위를 지정할 수도 있습니다. [a-zA-Z]는 알파벳 전체를 의미하는 범위이며, 문자열에 알파벳이 존재하면 매치를 의미합니다.
[^문자]	해당 문자를 제외한 문자를 매치합니다.
|	A|B와 같이 쓰이며 A 또는 B의 의미를 가집니다.

정규 표현식

문자 규칙	설명
₩	역 슬래쉬 문자 자체를 의미합니다
₩d	모든 숫자를 의미합니다. [0-9]와 의미가 동일합니다.
₩D	숫자를 제외한 모든 문자를 의미합니다. [^0-9]와 의미가 동일합니다.
₩s	공백을 의미합니다. [₩t₩n₩r₩f₩v]와 의미가 동일합니다.
₩S	공백을 제외한 문자를 의미합니다. [^ ₩t₩n₩r₩f₩v]와 의미가 동일합니다.
₩w	문자 또는 숫자를 의미합니다. [a-zA-Z0-9]와 의미가 동일합니다.
₩W	문자 또는 숫자가 아닌 문자를 의미합니다. [^a-zA-Z0-9]와 의미가 동일합니다.

모듈 함수	설명
re.compile()	정규표현식을 컴파일하는 함수입니다. 다시 말해, 파이썬에게 전해주는 역할을 합니다. 찾고자 하는 패턴이 빈번한 경우에는 미리 컴파일해놓고 사용하면 속도와 편의성면에서 유리합니다.
re.search()	문자열 전체에 대해서 정규표현식과 매치되는지를 검색합니다.
re.match()	문자열의 처음이 정규표현식과 매치되는지를 검색합니다.
re.split()	정규 표현식을 기준으로 문자열을 분리하여 리스트로 리턴합니다.
re.findall()	문자열에서 정규 표현식과 매치되는 모든 경우의 문자열을 찾아서 리스트로 리턴합니다. 만약, 매치되는 문자열이 없다면 빈 리스트가 리턴됩니다.
re.finditer()	문자열에서 정규 표현식과 매치되는 모든 경우의 문자열에 대한 이터레이터 객체를 리턴합니다.
re.sub()	문자열에서 정규 표현식과 일치하는 부분에 대해서 다른 문자열로 대체합니다.

정규 표현식 실습

1) .기호: .은 한 개의 임의의 문자를 나타냅니다. 예를 들어서 정규 표현식이 a.c라고 하면 a와 c 사이에는 어떤 1개의 문자라도 올 수 있다. 즉, akc, azc, avc, a5c, a!c와 같은 형태는 모두 a.c의 정규 표현식과 매치

```
import re
r=re.compile("a.c")
r.search("kkk") # 아무런 결과도 출력되지 않는다.
r.search("abc")
<_sre.SRE_Match object; span=(0, 3), match='abc'>
```

위의 코드는 search의 입력으로 들어오는 문자열에 정규표현식 패턴 a.c이 존재하는지를 확인하는 코드다.

(.)은 어떤 문자로도 인식될 수 있기 때문에 abc라는 문자열은 a.c라는 정규 표현식 패턴으로 매치되는 것을 볼 수 있다.

2) ?기호: ?는 ? 앞의 문자가 존재할 수도 있고, 존재하지 않을 수도 있는 경우

예를 들어서 정규 표현식이 ab?c 인 경우 이 정규 표현식에서의 b는 있다고 취급할 수도 있고, 없다고 취급할 수도 있다.

즉, abc와 ac 모두 매치할 수 있다.

```
R=re.compile( " ab?c " )
r.search("abbc") # 아무런 결과도 출력되지 않는다.
r.search("abc")
<_sre.SRE_Match object; span=(0, 3), match='abc'>
```

```
r.search("ac")
<_sre.SRE_Match object; span=(0, 2), match='ac'>
```

b가 없는 것으로 판단하여 ac를 매치

정규 표현식 실습

1) .기호: .은 한 개의 임의의 문자를 나타냅니다. 예를 들어서 정규 표현식이 a.c라고 하면 a와 c 사이에는 어떤 1개의 문자라도 올 수 있다. 즉, akc, azc, avc, a5c, a!c와 같은 형태는 모두 a.c의 정규 표현식과 매치

```
import re
r=re.compile("a.c")
r.search("kkk") # 아무런 결과도 출력되지 않는다.
r.search("abc")
<_sre.SRE_Match object; span=(0, 3), match='abc'>
```

위의 코드는 search의 입력으로 들어오는 문자열에 정규표현식 패턴 a.c이 존재하는지를 확인하는 코드다.

(.)은 어떤 문자로도 인식될 수 있기 때문에 abc라는 문자열은 a.c라는 정규 표현식 패턴으로 매치되는 것을 볼 수 있다.

2) ?기호: ?는 ? 앞의 문자가 존재할 수도 있고, 존재하지 않을 수도 있는 경우

예를 들어서 정규 표현식이 ab?c 인 경우 이 정규 표현식에서의 b는 있다고 취급할 수도 있고, 없다고 취급할 수도 있다.

즉, abc와 ac 모두 매치할 수 있다.

```
R=re.compile( " ab?c " )
r.search("abbc") # 아무런 결과도 출력되지 않는다.
r.search("abc")
<_sre.SRE_Match object; span=(0, 3), match='abc'>
```

```
r.search("ac")
<_sre.SRE_Match object; span=(0, 2), match='ac'>
```

b가 없는 것으로 판단하여 ac를 매치

정규 표현식 실습

3) *기호

*은 바로 앞의 문자가 0개 이상일 경우를 나타낸다.

앞의 문자는 존재하지 않을 수도 있으며, 또는 여러 개일 수도 있다.

예를 들어서 정규 표현식이 ab*c라면

ac, abc, abbc, abbbc 등과 매치할 수 있으며

b의 갯수는 무수히 많아도 상관없다.

```
import re
r=re.compile("ab*c")
r.search("a") # 아무런 결과도 출력되지 않는다.
r.search("ac")
<_sre.SRE_Match object; span=(0, 2), match='ac'>
r.search("abc")
<_sre.SRE_Match object; span=(0, 3), match='abc'>
r.search("abbbc")
<_sre.SRE_Match object; span=(0, 6), match='abbbbc'>
```

4) +기호:+는 *와 유사 하지만 다른 점은 앞의 문자가 최소 1개 이상이어야 한다는 점

정규 표현식이 ab+c라고 한다면, ac는 매치되지 않는다. 하지만 abc, abbc, abbbc 등과 매치할 수 있으며 b의 갯수는 무수히 많을 수 있다.

```
import re
r=re.compile("ab+c")
r.search("ac") # 아무런 결과도 출력되지 않는다.
r.search("abc")
<_sre.SRE_Match object; span=(0, 3), match='abc'>
r.search("abbbbc")
<_sre.SRE_Match object; span=(0, 6), match='abbbbc'>
```

5) ^기호:^는 시작되는 글자를 지정

정규표현식이 ^a라면 a로 시작되는 문자열만을 찾아낸다.

```
import re
r=re.compile("^a")
r.search("bbc") # 아무런 결과도 출력되지 않는다.
r.search("ab")
<_sre.SRE_Match object; span=(0, 1), match='a'>
```

bbc는 a로 시작되지 않지만, ab는 a로 시작되기 때문에 매치

정규 표현식 실습

6) {숫자} 기호

문자에 해당 기호를 붙이면, 해당 문자를 숫자만큼 반복한 것을 나타냄

ab{2}c라면 a와 c 사이에 b가 존재하면서 b가 2개인 문자열에 대해서 매치한다.

```
import re
r=re.compile("ab{2}c")
r.search("ac") # 아무런 결과도 출력되지 않는다.
r.search("abc") # 아무런 결과도 출력되지 않는다.
r.search("abbc")
<_sre.SRE_Match object; span=(0, 4), match='abbc'>
r.search("abbbbbc") # 아무런 결과도 출력되지 않는다.
```

8) {숫자,} 기호:숫자 이상 만큼 반복

```
import re
r=re.compile("a{2,}bc")
r.search("bc") # 아무런 결과도 출력되지 않는다.
r.search("aa") # 아무런 결과도 출력되지 않는다.
r.search("aabc")
<_sre.SRE_Match object; span=(0, 4), match='aabc'>
r.search("aaaaaaaabc")
<_sre.SRE_Match object; span=(0, 10), match='aaaaaaaabc'>
```

7) {숫자1, 숫자2} 기호

문자에 해당 기호를 붙이면, 해당 문자를 숫자1 이상 숫자2 이하만큼 반복

정규 표현식이 ab{2,8}c라면 a와 c 사이에 b가 존재하면서 b는 2개 이상 8개 이하인 문자열에 대해서 매치

```
import re
r=re.compile("ab{2,8}c")
r.search("ac") # 아무런 결과도 출력되지 않는다.
r.search("abc") # 아무런 결과도 출력되지 않는다.
r.search("abbc")
<_sre.SRE_Match object; span=(0, 4), match='abbc'>
r.search("abbbbbbbbc")
<_sre.SRE_Match object; span=(0, 10), match='abbbbbbbbc'>
r.search("abbbbbbbbbc") # 아무런 결과도 출력되지 않는다.
```

정규 표현식 실습

9) [ ] 기호: [ ]안에 문자들을 넣으면 그 문자들 중 한 개의 문자와 매치라는 의미

[abc]: a 또는 b또는 c가 들어가있는 문자열과 매치

[a-zA-Z]:는 알파벳 전부를 의미 [0-9]:는 숫자 전부를 의미

```
import re
r=re.compile("[abc]") # [abc]는 [a-c]와 같다.
r.search("zzz") # 아무런 결과도 출력되지 않는다.
r.search("a")
<_sre.SRE_Match object; span=(0, 1), match='a'>
r.search("aaaaaaa")
<_sre.SRE_Match object; span=(0, 1), match='a'>
r.search("baac")
<_sre.SRE_Match object; span=(0, 1), match='b'>
```

```
import re
r=re.compile("[a-z]")
r.search("AAA") # 아무런 결과도 출력되지 않는다.
r.search("aBC")
<_sre.SRE_Match object; span=(0, 1), match='a'>
r.search("111") # 아무런 결과도 출력되지 않는다.
```

10) [^문자] 기호

[^문자]:^ 기호 뒤에 붙은 문자들을 제외한 모든 문자를 매치하는 역할

[^abc]: a 또는 b 또는 c가 들어간 문자열을 제외한 모든 문자열을 매치

```
import re
r=re.compile("[^abc]")
r.search("a") # 아무런 결과도 출력되지 않는다.
r.search("ab") # 아무런 결과도 출력되지 않는다.
r.search("b") # 아무런 결과도 출력되지 않는다.
r.search("d")
<_sre.SRE_Match object; span=(0, 1), match='d'>
r.search("1")
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

정규 표현식 모듈 함수 예제

re.match() 와 re.search()의 차이

search()가 정규 표현식 전체에 대해서 문자열이 매치하는지를 본다면, match()는 문자열의 첫 부분부터 정규 표현식과 매치하는지를 확인한다.

문자열 중간에 찾을 패턴이 있다고 하더라도, match 함수는 문자열의 시작에서 패턴이 일치하지 않으면 찾지 않는다.

```
import re
r=re.compile("ab.")
r.search("kkkabc")
<_sre.SRE_Match object; span=(3, 6), match='abc'>
r.match("kkkabc")  #아무런 결과도 출력되지 않는다.
r.match("abckkk")
<_sre.SRE_Match object; span=(0, 3), match='abc'>
```

위의 경우 정규 표현식이 ab. 이기때문에, ab 다음에는 어떤 한 글자가 존재할 수 있다는 패턴을 의미

search 모듈 함수에 kkkabc라는 문자열을 넣어 매치되는지 확인한다면 abc라는 문자열에서 매치되어 Match object를 리턴한다.

하지만 match 모듈 함수의 경우 앞 부분이 ab.와 매치되지 않기때문에, 아무런 결과도 출력되지 않는다.

하지만 반대로 abckkk로 매치를 시도해보면, 시작 부분에서 패턴과 매치되었기 때문에 정상적으로 Match object를 리턴한다.

정규 표현식 모듈 함수 예제

re.split()

split() 함수는 입력된 정규 표현식을 기준으로 문자열들을 분리하여 리스트로 리턴

자연어 처리에 있어서 가장 많이 사용되는 정규 표현식 함수 중 하나인데, 토큰화에 유용하게 쓰일 수 있기 때문

```
import re
text="사과 딸기 수박 메론 바나나"
re.split(" ",text)
```

['사과', '딸기', '수박', '메론', '바나나']

위의 예제의 경우 입력 텍스트로부터 공백을 기준으로 문자열 분리를 수행하였고, 결과로서 리스트를 리턴하는 모습을 볼 수 있다.

```
import re
text="""사과
딸기
수박
메론
바나나"""
re.split("₩n",text)
```

['사과', '딸기', '수박', '메론', '바나나']

```
import re
text="사과+딸기+수박+메론+바나나"
re.split("₩+",text)
```

['사과', '딸기', '수박', '메론', '바나나']

정규 표현식 모듈 함수 예제

re.findall(): 정규 표현식과 매치되는 모든 문자열들을 리스트로 리턴

단, 매치되는 문자열이 없다면 빈 리스트를 리턴

```
import re
text="""이름 : 김철수
전화번호 : 010 - 1234 - 1234
나이 : 30
성별 : 남"""
re.findall("₩d+",text)
```

['010', '1234', '1234', '30']

정규 표현식으로 숫자를 입력하자, 전체 텍스트로부터 숫자만 찾아내서 리스트로 리턴하는 것을 볼 수 있다. 하지만 만약 입력 텍스트에 숫자가 없다면 빈 리스트를 리턴

```
re.findall("₩d+", "문자열입니다.")
```

[] # 빈 리스트를 리턴한다.

re.sub(): 정규 표현식 패턴과 일치하는 문자열을 찾아 다른 문자열로 대체할 수 있다.

```
import re
text="Regular expression : A regular expression, regex or regexp[1] (sometimes called a rational expression)[2][3] is, in theoretical computer science and formal language theory, a sequence of characters that define a search pattern."

re.sub('[^a-zA-Z]',' ',text)
```

'Regular expression A regular expression regex or regexp sometimes called a rational expression is in theoretical computer science and formal language theory a sequence of characters that define a search pattern '

위와 같은 경우, 영어 문장에 각주 등과 같은 이유로 특수 문자가 섞여있다. 자연어 처리를 위해 특수 문자를 제거하고 싶다면 알파벳 외의 문자는 공백으로 처리하는 등의 사용 용도로 쓸 수 있다.

정규 표현식 텍스트 전처리 예제

```
import re
text = """100 John    PROF
101 James   STUD
102 Mac   STUD"""
re.split('₩s+', text)
```

['100', 'John', 'PROF', '101', 'James', 'STUD', '102', 'Mac', 'STUD']

-'₩s+'는 공백을 찾아내는 정규표현식

-뒤에 붙는 +는 최소 1개 이상의 패턴을 찾아낸다.

-s는 공백을 의미하기 때문에 최소 1개 이상의 공백인 패턴을 찾아낸다.

-입력으로 테이블 형식의 데이터를 텍스트에 저장하였다.

-각 데이터가 공백으로 구분되었다.

```
re.findall('₩d+',text)
```

['100', '101', '102]

여기서 ₩d는 숫자에 해당되는 정규표현식

+를 붙였으므로 최소 1개 이상의 숫자에 해당하는 값을 찾는다.

정규 표현식 텍스트 전처리 예제

이번에는 텍스트로부터 대문자인 행의 값만 가져오자.

이 경우에는 정규 표현식에 대문자를 기준으로 매치시키면 되지만 정규 표현식에 대문자라는 기준만을 넣을 경우에는 문자열을 가져오는 것이 아니라 모든 대문자 각각을 갖고 오게 된다.

re.findall('[A-Z]',text)

['J', 'P', 'R', 'O', 'F', 'J', 'S', 'T', 'U', 'D', 'M', 'S', 'T', 'U', 'D']

이 경우, 여러가지 방법이 있겠지만 대문자가 연속적으로 4번 등장하는 경우로 조건을 추가하자.

re.findall('[A-Z]{4}',text)

['PROF', 'STUD', 'STUD']

대문자로 구성된 문자열들을 제대로 가져오는 것을 볼 수 있다. 이름의 경우에는 대문자와 소문자가 섞여있는 상황

이름에 대한 행의 값을 갖고 오고 싶다면 처음에 대문자가 등장하고, 그 후에 소문자가 여러 번 등장하는 경우에 매치하게 한다.

re.findall('[A-Z][a-z]+',text)

['John', 'James', 'Mac']