

STRING

Recap Python String

파이썬 String의 특징

- 원시 자료형이자, 불변 타입이다.
- 큰 따옴표 " 혹은 작은 따옴표 ' 표기된다.
- 따옴표를 3개 연달아 쓰면 여러 줄을 넣을 수 있다.
 - `"""text"""`, `"""여러줄 적기"""`
- Indexing 및 Slicing이 가능하다.
- 덧셈 및 곱셈이 가능하다
- in & not in 연산이 가능하다
 - tuple과 다소 다르게 작동
- Unicode로 처리된다.

```
>>> text = """
... 여러줄 적기
... 가 가능합니다
... """
```

```
>>> text
'\n여러줄 적기\n가 가능합니다\n'
```

Special Characters

Escape 문자 \ 를 사용하여 특수 문자 활용

문자	설명
\ [Enter]	다음 줄과 연속임을 표현
\\	\ 문자
\'	' 문자
\"	" 문자
\b	백스페이스
\n	줄 바꾸기
\t	TAB 키
\e	ESC 키

```
>>> text = '\
... 이렇게 적으면 \
... 엔터 없이 \
... 여러 줄을 적어요\
... '
```

```
>>> text
'이렇게 적으면 엔터 없이 여러 줄을 적어요'
```

```
>>> text = 'print할 때랑 \n평가할 때랑 달라요'
```

```
>>> text
'print할 때랑 \n평가할 때랑 달라요'
```

```
>>> print(text)
print할 때랑
평가할 때랑 달라요
```

Raw String

r“<TEXT>” 형태로 \ 를 무시하고 문자 그대로 취급 가능

```
>>> string = "여기서는 역 슬래시는 \n 특별한 의미를 가져요"
>>> string
'여기서는 역 슬래시는 \n 특별한 의미를 가져요'
>>> print(string)
여기서는 역 슬래시는
특별한 의미를 가져요

>>> raw_string = r"여기서는 역 슬래시가 \n 특별한 의미를 안 가져요"
>>> raw_string
'여기서는 역 슬래시가 \\n 특별한 의미를 안 가져요'
>>> print(raw_string)
여기서는 역 슬래시가 \n 특별한 의미를 안 가져요
```

역 슬래시가 일반 문자로 변환

```
>>> locate = "C:\\Users\\hojuncho"
>>> print(locate)
C:\Users\hojuncho

>>> locate = r"C:\Users\hojuncho"
>>> print(locate)
C:\Users\hojuncho
```

주로 정규 표현식에서 사용

String Functions

함수 형태	기능
<code>len(string)</code>	문자 개수 반환
<code>string.upper()</code>	대문자로 변환
<code>string.lower()</code>	소문자로 변환
<code>string.capitalize()</code>	문자열 시작 문자를 대문자로 변환
<code>string.title()</code>	단어 시작을 대문자로 변환

```
>>> text = 'this is the Python course'
>>> len(text)
25

>>> text.upper()
'THIS IS THE PYTHON COURSE'
>>> text.lower()
'this is the python course'
>>> text.capitalize()
'This is the python course'
>>> text.title()
'This Is The Python Course'
```

String Functions

함수 형태	기능
string.strip()	좌우 공백 제거
string.lstrip()	왼쪽 공백 제거
string.rstrip()	오른쪽 공백 제거
string.isdigit()	숫자 형태인지 확인
string.isupper()	대문자로만 이루어져 있는지 확인
string.islower()	소문자로만 이루어져 있는지 확인

```
>>> test = '    공백이 \t 있어요 \t\n '
>>> test.strip()
'공백이 \t 있어요'
>>> test.lstrip()
'공백이 \t 있어요 \t\n '
>>> test.rstrip()
'    공백이 \t 있어요'
```

```
>>> '12345'.isdigit()
True
>>> '1.24e-3'.isdigit()
False
>>> 'Capitalize'.isupper()
False
>>> 'lower_case'.islower()
True
```

String Pattern Matching

함수 형태	기능
string.count(pattern)	문자열 string 내에 pattern 등장 횟수 반환
string.find(pattern)	문자열 string 내에 pattern 첫 등장 위치 반환 (앞에서 부터)
string.rfind(pattern)	문자열 string 내에 pattern 첫 등장 위치 반환 (뒤에서 부터)
string.startswith(pattern)	문자열 string이 pattern으로 시작하는지 확인
string.endswith(pattern)	문자열 string이 pattern으로 끝나는지 확인

```
>>> text = 'abc_text_abc_ee'
>>> pattern = 'abc'

>>> text.count(pattern)
2
>>> text.find(pattern)
0
>>> text.rfind(pattern)
9
>>> text.startswith(pattern)
True
>>> text.endswith(pattern)
False
```

String Split & Join

함수 형태	기능
string.split()	공백을 기준으로 문자열 나누기
string.split(pattern)	Pattern을 기준으로 문자열 나누기
string.join(iterable)	String을 중간에 두고 iterable 원소들 합치기

```
>>> text = '한국어 abc 테스트 \n abc 중 \t 입니다'

>>> text.split()                                # 공백으로 나누기
['한국어', 'abc', '테스트', 'abc', '중', '입니다']

>>> text.split('abc')                          # 'abc' 패턴으로 나누기
['한국어 ', ' 테스트 \n ', ' 중 \t 입니다']

>>> ' '.join(text.split())                     # ' '를 중간에 두고 문자열들 합치기
'한국어 abc 테스트 abc 중 입니다'

>>> ', '.join(str(i) for i in range(10))       # ', '를 중간에 두고 문자열들 합치기
'0, 1, 2, 3, 4, 5, 6, 7, 8, 9'
```


String Formatting

Print등을 할 때 보기 좋게 값들을 확인하고 싶음

- 일정한 형태로 변수들을 문자열로 출력
- + 를 써서 구현은 할 수 있지만....

```
>>> a, b, c = 10, 1.725, 'sample'
>>> str(a) + ": " + str(b) + " - " + c # 하기도 보기도 불편
'10: 1.725 - sample'

>>> "%d: %f - %s" % (a, b, c)          # %-formatting
'10: 1.725000 - sample'

>>> "{}: {} - {}".format(a, b, c)      # .format 함수
'10: 1.725 - sample'


>>> f"{a}: {b} - {c}"                  # f-string
'10: 1.725 - sample'
```

%-formatting

“%datatype” % variables 형태로 표현

- C나 Java에서 주로 쓰이는 방식

"Art: %5d, Price per Unit: %8.2f" % (453, 59.058)



```
>>> "Art: %5d, Price per Unit: %8.2f" % (453, 59.058)
'Art: 453, Price per Unit: 59.06'
```

%datatype	설명
%s	문자열 (str)
%d	정수 (int)
%f	부동소수점 (float)
%o	8진수
%x	16진수

Padding

“%[padding]datatype” 형태로 패딩 가능

```
>>> "%d+%d+%d" % (1, 10, 100)
'1+10+100'
>>> "%4d+%4d+%4d" % (1, 10, 100)
'   1+   10+  100'
>>> "%-4d+%-4d+%-4d" % (1, 10, 100)
'1   +10   +100 '
>>> "%04d+%04d+%04d" % (1, 10, 100)
'0001+0010+0100'
```

“%[padding].[precision]datatype” 형태로 정확도 지정

```
>>> "%f+%f+%f" % (123.4, 12.34, 1.234)
'123.400000+12.340000+1.234000'
>>> "%.3f+%.3f+%.3f" % (123.4, 12.34, 1.234)
'123.400+12.340+1.234'
>>> "%8.3f+%8.3f+%8.3f" % (123.4, 12.34, 1.234)
' 123.400+   12.340+    1.234'
>>> "%08.3f+%08.3f+%08.3f" % (123.4, 12.34, 1.234)
'0123.400+0012.340+0001.234'
```

String.format Method

“{0}”.format(variables) 형태로 표현

`"Art: {0:5d}, Price per Unit: {1:8.2f}".format(453, 59.058)`



순서 설정

```
>>> a, b, c = 10, 1.725, 'sample'
>>> "{}: {} - {}".format(a, b, c)
'10: 1.725 - sample'
>>> "{0}: {1} - {2}".format(a, b, c)
'10: 1.725 - sample'
>>> "{0}: {2} - {1}".format(a, b, c)
'10: sample - 1.725'
```

패딩 설정

```
>>> "{0}+{1}+{2}".format(123.4, 12.34, 1.234)
'123.4+12.34+1.234'
>>> "{0:.3f}+{1:.3f}+{2:.3f}".format(123.4, 12.34, 1.234)
'123.400+12.340+1.234'
>>> "{:8.3f}+{:8.3f}+{:8.3f}".format(123.4, 12.34, 1.234)
' 123.400+  12.340+   1.234'
```

Naming

순서가 헛갈리지 않게 각 위치에 이름 붙이기

“%([name])format” 형태

```
>>> "%(first)5.2f - %(second)5.2f" % {"first": 10.2, "second": 5.62}
'10.20 - 5.62'
```

“{[name]:format}” 형태

```
>>> "{first:5.2f} - {second:5.2f}".format(first=10.2, second=5.62)
'10.20 - 5.62'

>>> "{first:5.2f} - {second:5.2f}".format(**{"first": 10.2, "second": 5.62})
'10.20 - 5.62'
```

F-string

f“{variable}” 형태로 표현

- Python 3.6 이상 부터 지원
- 변수 위치에 원하는 변수를 위치하면 됨

변수 삽입

```
>>> a, b, c = 10, 1.725, 'sample'
>>> f"{a}: {b} - {c}"
'10: 1.725 - sample'
>>> f"{a}: {c} - {b}"
'10: sample - 1.725'
```

패딩 설정

```
>>> value = 12.34
>>> f"{value*10}+{value}+{value/10}"
'123.4+12.34+1.234'
>>> f"{value*10:.3f}+{value:.3f}+{value/10:.3f}"
'123.400+12.340+1.234'
>>> f"{value*10:8.3f}+{value:8.3f}+{value/10:8.3f}"
' 123.400+  12.340+   1.234'
```

Finding Patterns

야 이거 **#%이름#**거 아니냐?
#%이름#에게 물어봐봐
#%이모티콘#

문자열에서 **#%SOMETHING#**를 찾거나 치환하는 방법은?

- Find 메소드는 정확히 일치하는 문자열만 찾을 수 있음
- 문자열에서 특정한 패턴을 정의하고 찾는 방법이 필요

Regular Expression

정규 표현식

- 특정한 규칙을 가진 문자열의 집합을 표현하는데 사용하는 형식 언어
- 많은 텍스트 편집기와 프로그래밍 언어에서 문자열 검색과 치환에 활용

패턴

`\d{3}-\d{4}-\d{4}`

`\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`

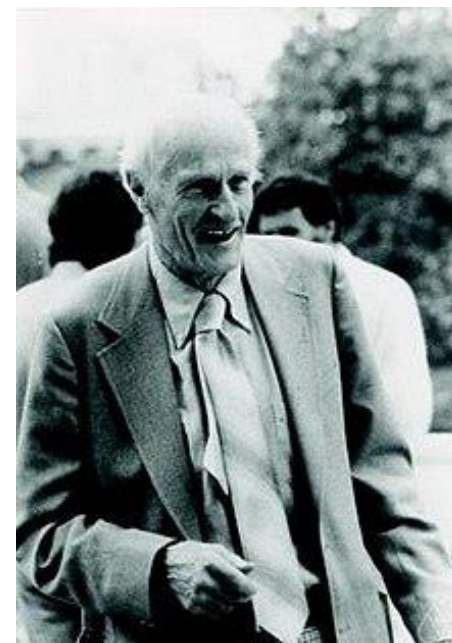
`#%[^#]+#`

예시

010-1234-5678

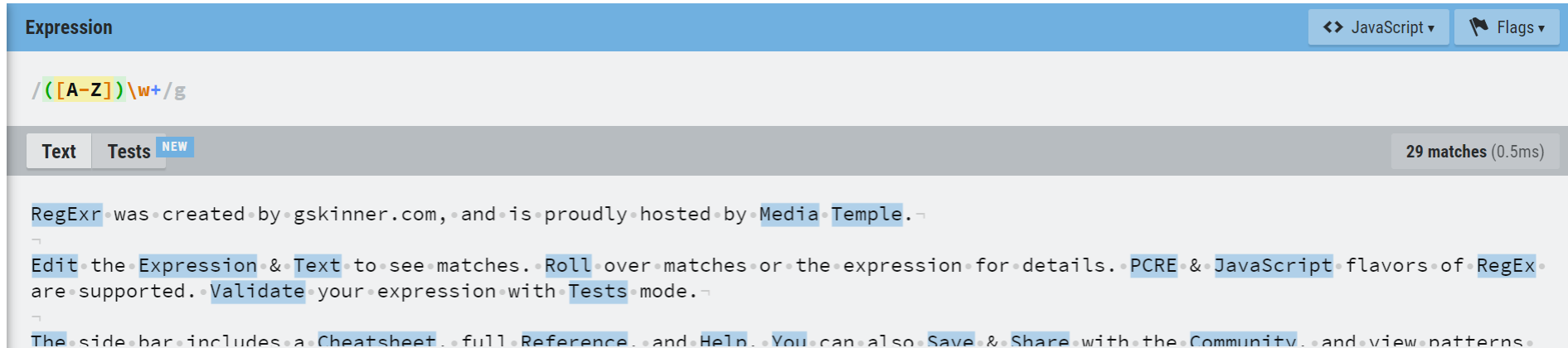
192.168.0.20

#%이모티콘#



Stephen Cole Kleene

Regular Expression



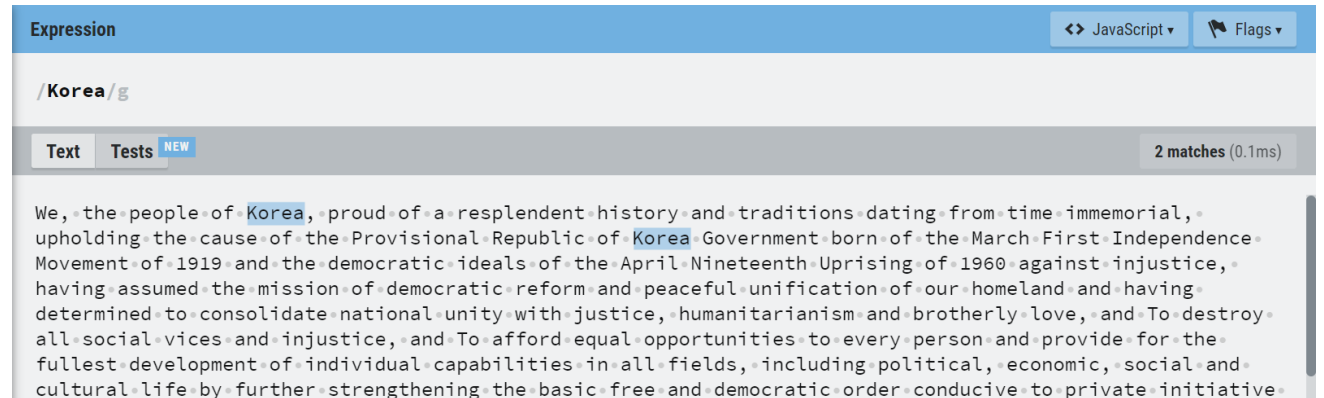
정규 표현식 공부하기 → 언어 하나 더 배우기

- 정규식 연습장을 활용해보자 (<http://www.regexr.com/>)
- 문법이 매우 방대
 - ppt 슬라이드 30장 정도 더 필요
 - 그래도 기본적인 몇 가지 요소를 설명하자면...

Regular Expression

- 일반 텍스트 패턴

- 정확히 일치되는 문자열 찾기
- .find 함수와 동일



- 특수 문자

- 정규식만에서 쓰이는 문자
- 공백, 영단어 등
- 일반 문자와 섞어 씀

특수 글자	설명	Regex
\w	영숫자 + “_”	[A-Za-z0-9_]
\W	(영숫자 + “_”)를 제외한 문자	[^A-Za-z0-9_]
\d	숫자	[0-9]
\D	숫자가 아닌 문자	[^0-9]
\s	공백 문자	[\t\r\n\v\f]
\S	공백이 아닌 문자	[^\t\r\n\v\f]
\b	단어 경계	(?<=\W)(?=\w) (?<=\w)(?=\W)

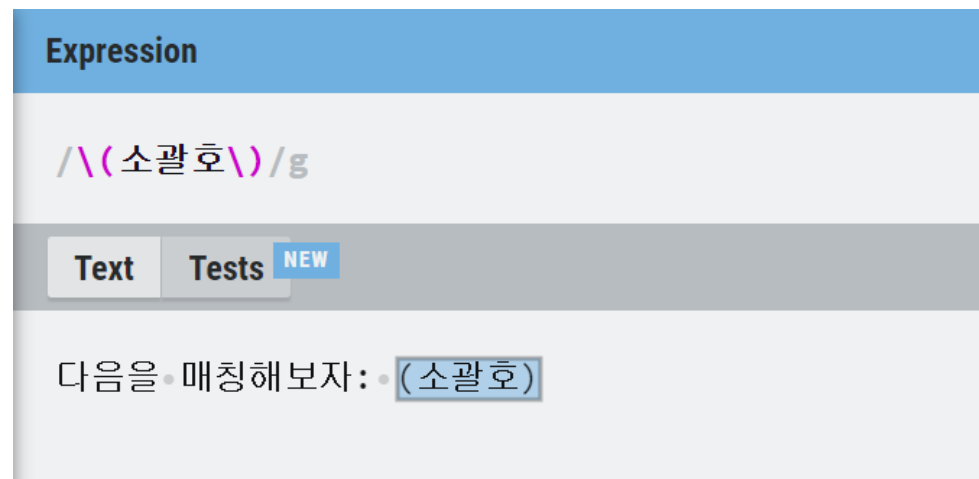
Meta Character

- 메타 문자: 정규식의 문법적인 요소를 담당하는 문자

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

- 이 문자들은 특수한 의미의 문자이므로 사용 불가능

- 만약 문자 그대로 매칭하고 싶다면 Escape 문자 \ 붙여 사용



Regular Expression: Character Class

문자 클래스 []

- [와] 사이의 문자들 중 하나와 매칭
- “-”를 사용하여 범위를 지정 가능
 - [a-z] [A-Z0-9] [\d\s]

Expression
/[abcde]/g
Text Tests NEW
My·number·is·010-1234-5678

Expression
/[0-9]/g
Text Tests NEW
My·number·is·010-1234-5678

부정 [^]

- [^와] 사이에 없는 문자를 매칭
- “-”를 사용하여 범위를 지정 가능
 - [^a-z] [^A-Z0-9] [^s]

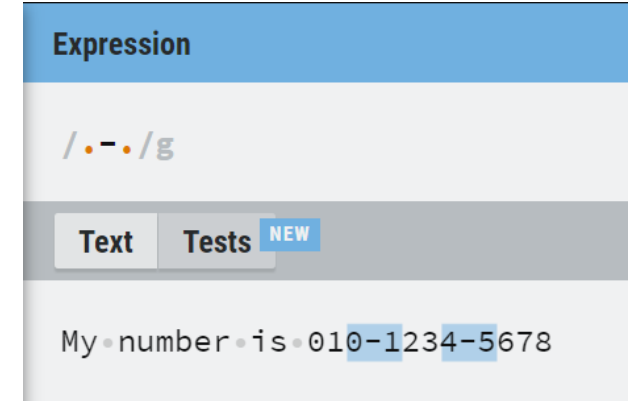
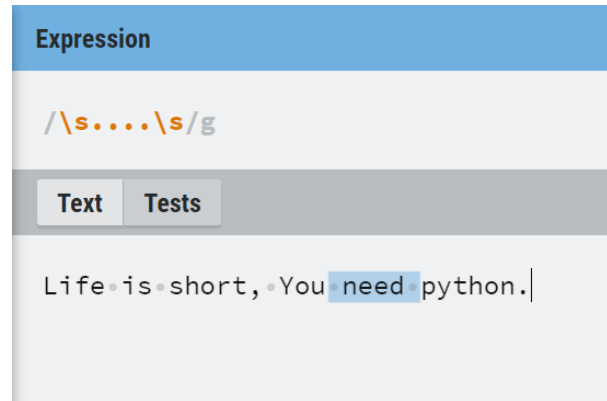
Expression
/[^cho]/g
Text Tests NEW
abcde·fghij·klm·n

Expression
/[^s\d]/g
Text Tests NEW
My·number·is·010-1234-5678

Regular Expression: Repetition

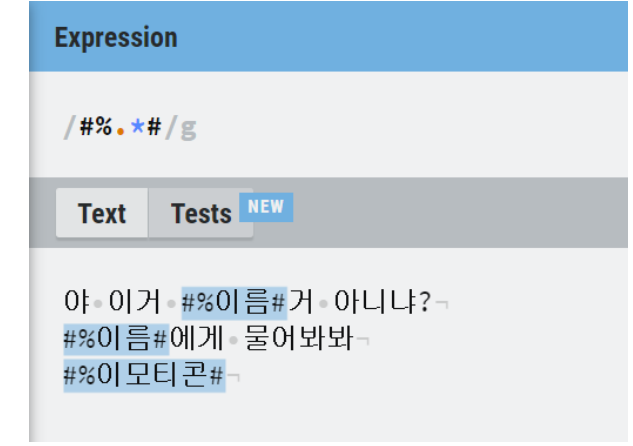
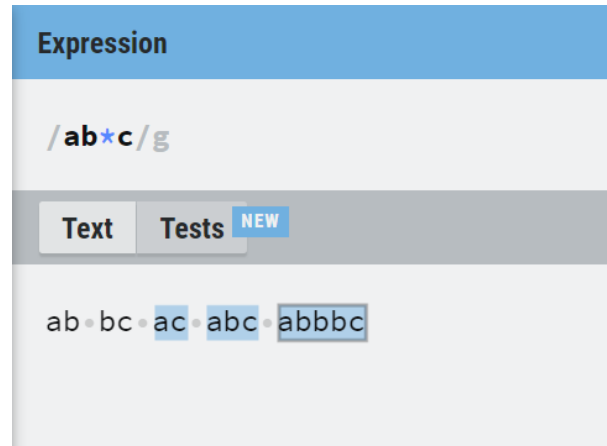
문자 .

- 아무 문자나 하나를 매칭
- 줄 바꿈 문자 \n는 제외



0회 이상 *

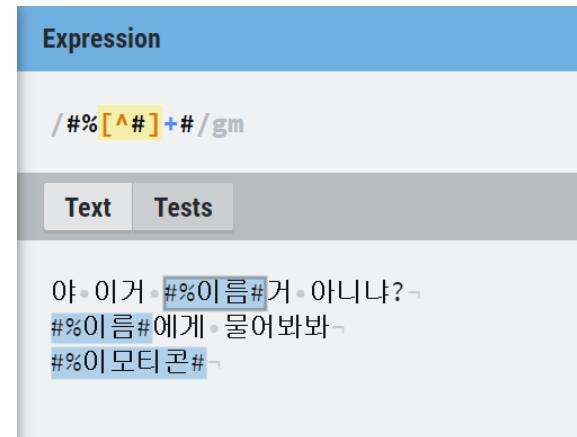
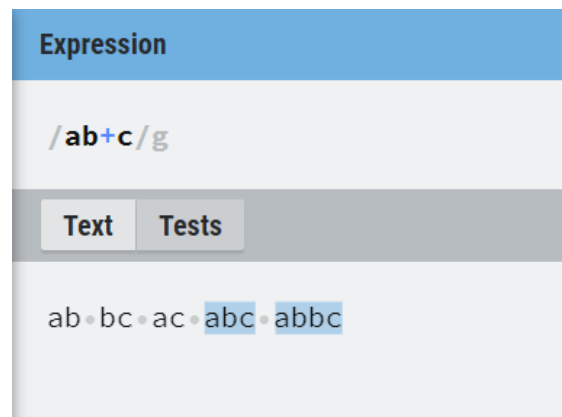
- 앞의 문자를 0개 이상 포함



Regular Expression: Repetition

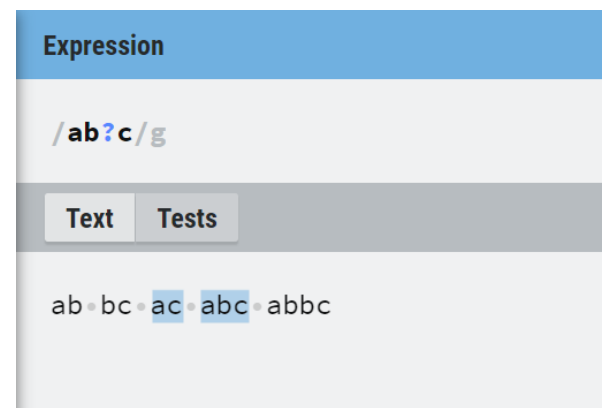
1회 이상 +

- 앞의 문자를 1개 이상 포함



0 또는 1회 ?

- 앞의 문자가 없을 수도 있음



Regular Expression: Repetition

반복 횟수 지정 {m,n}

- m번 이상 n번 이하 반복
- 숫자가 하나만 주어졌을 경우
 - {m}: m번 반복
 - {m,}: m번 이상 반복
 - {,n}: n번 이하 반복

Expression	
<code>/0\d{1,2}-\d{3,4}-\d{4}/g</code>	
Text	Tests
<pre>010-1234-5678 051-123-4567 042-1234-526 02-444-1235 024-23-4343</pre>	

Lazy Matching ?

- * + {} 와 같은 반복 연산은 욕심쟁이
 - 최대한 길게 반복
- 최대한 짧게 매칭하기 위해선 뒤에 ? 삽입
 - *?, +?, {n,m}?

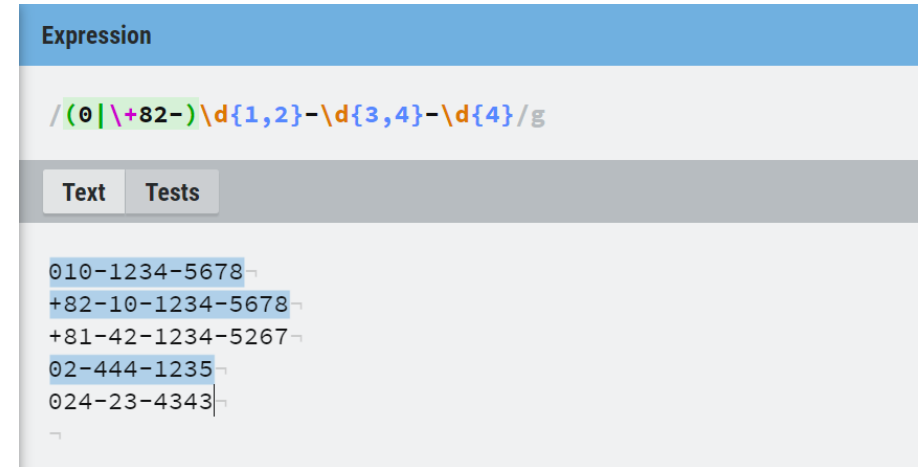
Expression	
<code>/<.+>/gm</code>	
Text	Tests
<pre>This is a <div> simple div </div> test</pre>	

Expression	
<code>/<.+?>/gm</code>	
Text	Tests
<pre>This is a <div> simple div </div> test</pre>	

Regular Expression: Boundary

선택 |

- 여러 식 중 하나를 선택



Expression

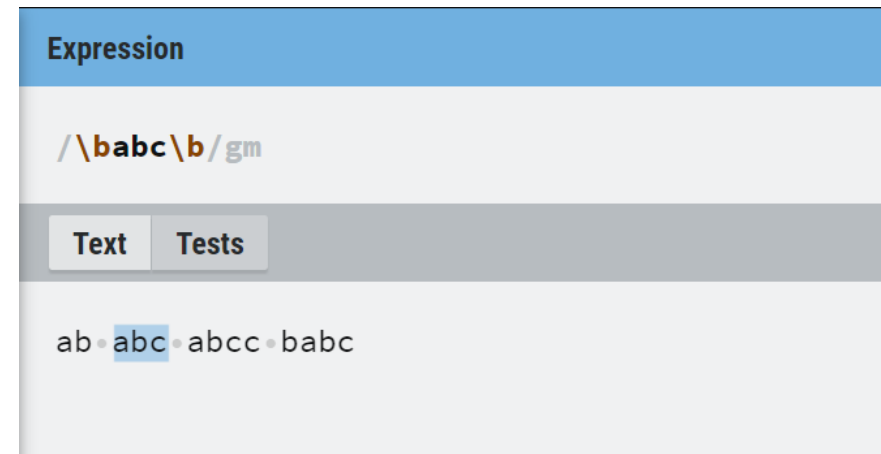
```
/(\d|\+82-)\d{1,2}-\d{3,4}-\d{4}/g
```

Text Tests

```
010-1234-5678
+82-10-1234-5678
+81-42-1234-5267
02-444-1235
024-23-4343
```

단어 경계 \b

- 단어의 경계 지점인지 확인



Expression

```
/\babc\b/gm
```

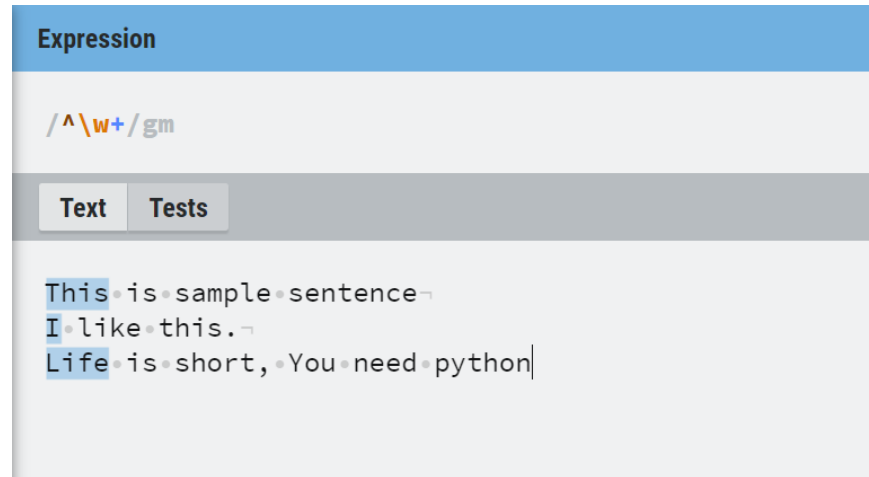
Text Tests

```
ab•abc•abcc•babc
```


Regular Expression: Boundary

줄의 시작 ^

- 줄이나 문자열의 시작점
- Multiline flag 필요



Expression

```
^\w+/gm
```

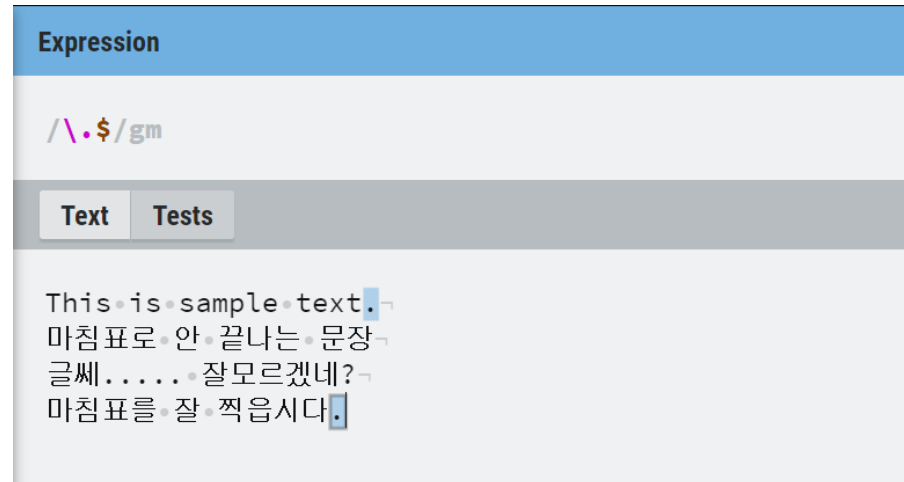
Text Tests

This is sample sentence.
I like this.
Life is short, You need python

The screenshot shows a regex testing interface. The expression `^\w+/gm` is entered. The 'Text' tab is selected, showing the sample text. The 'Tests' tab is also visible. The matches for the expression are highlighted in blue: 'This', 'I', and 'Life'.

줄의 끝 \$

- 줄이나 문자열의 끝
- Multiline flag 필요



Expression

```
/\.$/gm
```

Text Tests

This is sample text.
마침표로 안 끝나는 문장
글쎄.....잘 모르겠네?
마침표를 잘 찍읍시다.

The screenshot shows a regex testing interface. The expression `/\.$/gm` is entered. The 'Text' tab is selected, showing the sample text. The 'Tests' tab is also visible. The matches for the expression are highlighted in blue: 'This is sample text.', '마침표로 안 끝나는 문장', '글쎄.....잘 모르겠네?', and '마침표를 잘 찍읍시다.'.

Regular Expression: Capture

그룹 캡처 ()

- 괄호이므로 우선 순위가 있음
- 해당 문자열을 캡처한다.
- 캡처된 텍스트를 불러올 수 있다
 - \1, \2, \3, ..., \[NUMBER]

```
Expression
/(\w{2,}).+\1/gm

Text Tests

tomato
abcde
one-to-one
btomatu
abcdebch
```

비 그룹 캡처 (?:)

- 괄호이므로 우선 순위가 있음
- 캡처를 하지는 않는다
- 그냥 괄호다

```
Expression
/(?:0|\+82-)\d{1,2}-(\d{4})-\1/gm

Text Tests

010-1234-1234
010-1234-5678
+82-10-5678-5678
+82-4123-1234
```

Regular Expression: Condition

뒷 패턴 확인 $D(?=R)$

- R이 바로 뒤에 있는 D를 매칭
- R 부분은 포함되지 않음

Expression

```
/\w+(?=ism)/gm
```

Text Tests

tourism
I don't like idealism
He is socialism

앞 패턴 확인 $(?<=R)D$

- R이 바로 앞에 있는 D를 매칭
- R 부분은 포함되지 않음

Expression

```
/(?!pre)\w+/gm
```

Text Tests

This is predefined function
Word preprocessing is important

Regular Expression in Python

그래서 파이썬에서는 어떻게 쓰나요?

```
string = \
"""
010-1234-1234
010-1234-5678
+82-10-5678-5678
+82-4123-1234
"""

pattern = r'^(?:0|\+82-)\d{1,2}-(\d{4})-\d{1}$' # 패턴이 raw string이어야 함
```

→ 파이썬 표준 **re** 패키지를 사용

```
import re

for match in re.finditer(pattern, string, re.MULTILINE):
    print("전체 문자열", match.group(0))
    print(r"\1 문자열", match.group(1))
```

Regular Expression in Python

탐색

```
# 처음 매칭되는 문자열 탐색  
match = re.search(pattern, string, re.MULTILINE)  
print(match.group(0))
```

모두 탐색

```
# 모든 매칭되는 문자열 탐색  
for match in re.finditer(pattern, string, re.MULTILINE):  
    print(match.group(0))
```

치환

```
# 매칭되는 패턴 치환  
repl = r"치환됨-\1"  
print(re.sub(pattern, repl, string, flags=re.MULTILINE))
```

나누기

```
# 매칭되는 패턴을 기준으로 나누기  
splited = re.split(pattern, string, flags=re.MULTILINE)  
print(splited)
```

Regular Expression Compile

정규 표현식을 평가하는 것은 오래 걸림

```
for string in dataset:  
    match = re.search(pattern, string, re.MULTILINE)  
    print(match.group(0))
```



정규 표현식을 미리 컴파일 → 훨씬 빠름

```
compiled = re.compile(pattern, flags=re.MULTILINE)  
  
for string in dataset:  
    match = compiled.search(string)  
    print(match.group(0))
```