기초빅데이터프로그래밍

정규표현식

RE 모듈 정규식 사용하기

서강대학교
SOGANG UNIVERSITY

주민등록번호 가리기 실습(남여표시)

```
data='''
park sunje 890901-1074422
kim sunhee 990103-2079912
'''
```

```
park sunje 남 890901-*******
kim sunhee 여 990103-*******
```

- 위 data에서 주민등록번호의 뒤 7자리숫자들을 가리고 성별도 구분하도록 프로그램 하시오.
- 파일에 있는 주민등록번호도 처리할 수 있도록 확장하시오.

```
홍길동 560922-1089123 02-705-8491
홍길동 560922-1089123 042-7052-8491
김바한솔 911212-1089123 042-705-8491
김연찬 920922-1089123 031-7054-8491
```

Testdata.txt

정규표현식

- 정규표현식을 정의: 문자열에 대한 표현을 메타문자로 표시함
- 정규표현식을 실행 : 실제 문자열을 넣고 정규식과 매칭 여부 검증

➢ **Meta characters**: 원래 그 문자가 가진 뜻이 아닌 특별한 용도로 사용되는 문자

➢ **. ^ $ * + ? { } [ ] ₩ | ( )**

정규 표현식에 메타 문자들이 사용되면 특별한 의미를 갖게 된다.

문자클래스(character class, [])

- 문자클래스를 만드는 메타문자인 **[**와 **]** 사이에는 어떤 문자도 사용가능
- 문자클래스로 만들어진 정규식은 "**[과]사이의 문자들과 매치**"라는 의미
  - **[a-zA-Z] : 알파벳 모두**
  - **[0-9] : 숫자**
  - **^ 메타문자는 반대(not)의 의미: [^0-9] 숫자가 아닌 문자만 매치**

Example	Description
[Pp]ython	Match "Python" or "python"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

축약형 문자표현

- 축약형 문자표현
- **대문자로** 사용된 것은 **소문자의 반대임**

  - ✓ ₩**d** - 숫자와 매치, [0-9]와 동일한 표현식
  - ✓ ₩**D** - 숫자가 아닌 것과 매치, [^0-9]와 동일한 표현식
  - ✓ ₩**s** - whitespace 문자와 매치, [₩t₩n₩r₩f₩v]와 동일한 표현식이다. 맨 앞의 빈칸은 공백문자(space)를 의미
  - ✓ ₩**S** - whitespace 문자가 아닌 것과 매치, [^ ₩t₩n₩r₩f₩v]와 동일한 표현식
  - ✓ ₩**w** - 문자+숫자(alphanumeric)와 매치, [a-zA-Z0-9]와 동일한 표현식
  - ✓ ₩**W** - alphanumeric이 아닌 문자와 매치, [^a-zA-Z0-9]와 동일한 표현식

축약형 문자표현-세부

Pattern	Description
₩w	Matches **word characters.**
₩W	Matches nonword characters.
₩s	Matches **whitespace**. Equivalent to [₩t₩n₩r₩f].
₩S	Matches nonwhitespace.
₩d	Matches **digits**. Equivalent to [0-9].
₩D	Matches nondigits.
₩A	Matches beginning of string.
₩Z	Matches end of string. If a newline exists, it matches just before newline.
₩z	Matches end of string.
₩G	Matches point where last match finished.
₩b	Matches **word boundaries** when outside brackets. Matches **backspace** (0x08) when inside brackets.
₩B	Matches **nonword boundaries**.
₩n, ₩t, etc.	Matches newlines, carriage returns, tabs, etc.
₩1...₩9	Matches **nth grouped subexpression**.
₩10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

축약형 문자표현: 예

- 알파벳 소문자 d로 시작하고 하나이상의 문자 스트링 그룹
  + 알파벳문자가 아닌 문자
  + 알파벳 소문자 d로 시작하는 하나이상의 스트링 그룹으로 구성

- "dog dot", "do don't", "dumb-dumb", "no match", "dumb-bumb“

  "(d₩w+)₩W(d₩w+)“

  : d: 알파벳 소문자 d
  ₩w+: 하나이상의 문자
  ₩W: 문자가 아닌 캐릭터

- ('dog', 'dot'), ('do', 'don'), ('dumb', 'dumb')

```
import re
data = """
"dog dot", "do don't", "dumb-dumb", "no match", "dumb-bumb"
"""
p=re.compile('(d\w+)\W(d\w+)')
p.findall(data)
```

```
[('dog', 'dot'), ('do', 'don'), ('dumb', 'dumb')]
```

```
import re
data = """
"dog dot", "do don't", "dumb-dumb", "no match", "dumb-bumb"
"""
p=re.compile('d\w+\Wd\w+')
p.findall(data)
```

```
['dog dot', 'do don', 'dumb-dumb']
```

^ / $

^

- 문자열의 시작
- 컴파일 옵션 re.MULTILINE 을 사용할 경우에는 **여러 줄의 문자열에서는 각 라인의 처음과 일치**
- ^ 문자를 메타문자가 아닌 문자 그 자체로 매치하고 싶은 경우에는 [^] 처럼 사용하거나 ₩^ 로 사용

$

- **문자열의 맨 마지막부터 일치함을** 의미
- $ 문자를 메타문자가 아닌 문자 그 자체로 매치하고 싶은 경우에는 [$] 처럼 사용하거나 ₩$ 로 사용

Anchor 처리 예시

- 특정 위치를 고정하여 처리할 경우 사용

Example	Description
^Python	Match "Python" at the start of a string or internal line
Python$	Match "Python" at the end of a string or line
₩APython	Match "Python" at the start of a string
Python₩Z	Match "Python" at the end of a string
₩bPython₩b	Match "Python" at a word boundary
₩brub₩B	**₩B is nonword boundary**: match "rub" in "rube" and "ruby" but not alone
Python(?=!)	Match "Python", if followed by an exclamation point.
Python(?!!)	Match "Python", if not followed by an exclamation point.

- **₩A**
- ₩A는 문자열의 처음과 매치됨을 의미한다. ^와 동일한 의미이지만 , re.MULTILINE 옵션을 사용할 경우 ^은 라인별 문자열의 처음과 매치되지만 **₩A는 라인과 상관없이 전체 문자열의 처음하고만 매치된다**

- **₩Z**
- ₩Z는 문자열의 끝과 매치됨을 의미한다. 이것 역시 e.MULTILINE 옵션을 사용할 경우 $ 메타문자와는 **달리 전체 문자열의 끝과 매치된다.**

- **? = ! & ? ! !**

```
print(re.search('Python(?=!)', "Wow! Python!"))
```

```
<_sre.SRE_Match object; span=(5, 11), match='Python'>
```

```
print(re.search('Python(?!!)', "Wow! Python!"))
```

```
None
```

```
print(re.search('Python(?=&)', "Wow! Python&"))
```

```
<_sre.SRE_Match object; span=(5, 11), match='Python'>
```

\b

\b는 단어 구분자(Word boundary)이다. 보통 단어는 whitespace에 의해 구분이 된다.

₩b는 파이썬 리터럴 규칙에 의하면 백스페이스(Back Space)를 의미하므로 백스페이스가 아닌 Word Boundary임을 알려주기 위해 **raw string**임을 알려주는 기호 **r**을 반드시 붙여주어야 한다.

```
import re
p=re.compile(r'\bclass\b')
print(p.search('no class at all'))
```

```
<_sre.SRE_Match object; span=(3, 8), match='class'>
```

```
print(p.search('the declassified algorithm'))
```

```
None
```

```
print(p.search('one subclass is'))
```

```
None
```

\B

\B 메타문자는 \b 메타문자의 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'))
```

```
<_sre.SRE_Match object; span=(6, 11), match='class'>
```

```
print(p.search('one subclass is'))
```

```
None
```

```
print(re.search(r'\Bclass\b', 'one subclass is'))
```

```
<_sre.SRE_Match object; span=(7, 12), match='class'>
```

DOT(.)

- **dot(.)** 메타문자는 줄바꿈 문자 **₩n를 제외한 어떤 문자와 매치됨을** 의미함, 즉, 개행 문자를 제외한 문자 1자를 의미한다.
- re.DOTALL 이라는 옵션을 주면 ₩n문자와도 매치된다
  - **a.b** : "a + 문자 + b"
    - "a0b":match, "**abc": not match**
  - **a[.]b** : "a + Dot(.)문자 + b"
    - Dot 문자 .
    - **"a.b" :match**

```
import re
dp =re.compile('a.b')
print(dp.match("asb"))
```

```
<_sre.SRE_Match object; span=(0, 3), match='asb'>
```

```
print(dp.match("ab"))
```

```
None
```

```
print(dp.match("a b"))
```

```
<_sre.SRE_Match object; span=(0, 3), match='a b'>
```

```
print(dp.match("avgb"))
```

```
None
```

반복 (*)

- * 바로 앞에 있는 문자 a가 **0부터 무한개 까지** 반복될 수 있다는 의미
  (사실은 유한함, 2억개 정도...)

정규식	문자열	Match 여부	설명
c**a***t	ct	Yes	"a"가 **0번 반복되어 매치**
c**a***t	cat	Yes	"a"가 0번 이상 반복되어 매치 (1번 반복)
c**a***t	caaat	Yes	"a"가 0번 이상 반복되어 매치 (3번 반복)

반복 (+)

- +는 **최소 1개 이상의 반복을** 필요로 하는 메타문자

정규식	문자열	Match 여부	설명
ca+t	ct	**No**	"a"가 0번 반복되어 매치되지 않음
ca+t	cat	Yes	"a"가 1번 이상 반복되어 매치 (1번 반복)
ca+t	caaat	Yes	"a"가 1번 이상 반복되어 매치 (3번 반복)

반복 (?)

- **?** 메타문자가 의미하는 것은 **{0, 1} # 0 또는 1**

정규식	문자열	Match 여부	설명
a**b?**c	abc	Yes	"b"가 1번 사용되어 매치
a**b?**c	ac	Yes	"b"가 0번 사용되어 매치

반복 ({m,n})

- {} 메타문자를 이용하면 반복횟수를 고정시킬 수 있다.
- {m, n} 정규식을 사용하면 반복횟수가 m부터 n인 것을 매치
- {1,}은 +와 동일하며 {0,}은 *와 동일

정규식	문자열	Match 여부	설명
ca**{2}**t	cat	No	"a"가 1번만 반복되어 매치되지 않음(**반드시 2번 반복되어야 함**)
ca{2}t	caat	**Yes**	"a"가 2번 반복되어 매치
ca**{2,5}**t	cat	No	"a"가 1번만 반복되어 매치되지 않음 **(2번에서 5번까지 반복가능)**
ca{2,5}t	caat	**Yes**	"a"가 2번 반복되어 매치
ca{2,5}t	caaaaat	**Yes**	"a"가 5번 반복되어 매치

백슬래시(₩) 문제

- 예를 들어 LaTex파일 내에 있는 "₩section" 이라는 문자열을 찾기 위한 정규식을 만든다고 할때,
- "₩section" : 이 정규식은 ₩s 문자가 whitespace로 해석되어
  [₩t₩n₩r₩f₩v]ection 동일한 의미
- re.compile("₩₩section") : 파이썬 문자열 리터럴 규칙에 의하여 ₩₩이 ₩로 변경되므로
  ✓ ₩₩ 문자를 전달하려면 파이썬은 ₩₩₩₩ 처럼 백슬래시를 4개나 사용
- **r**"₩section" : Raw String 규칙에 의하여 백슬래시 두 개 대신 한 개만 써도 두 개를 쓴 것과 동일한 의미
- raw string notation: ₩를 escape 문자가 아닌 일반 문자로 취급

Alternatives (|,or)

- | 메타문자는 "or"의 의미와 동일
- A|B 라는 정규식이 있다면 이것은 A 또는 B라는 의미

Example	Description
python|perl	Match "python" or "perl"
rub(y|le))	Match "ruby" or "ruble"
Python(!+|₩?)	"Python" followed by **one or more !** or **one ?**

전방탐색 (?= ...) (?! ...)

- **긍정형 전방 탐색((?=...))** - ... 에 해당되는 정규식과 매치되어야 하며 조건이 통과되어도 문자열이 소모되지 않는다.
- **부정형 전방 탐색((?!...))** - ...에 해당되는 정규식과 매치되지 않아야 하며 조건이 통과되어도 문자열이 소모되지 않는다.
- 소모되지 않는 것은 검색에는 포함되지만 검색 결과에는 제외됨을 의미

```
import re
p = re.compile(".+:")
m = p.search("http://google.com")
print(m.group())
```

```
http:
```

```
p = re.compile(".+(?=:)")
m = p.search("http://google.com")
print(m.group())
```

```
http
```

```
p = re.compile(".+(?=::)")
m = p.search("http:://google.com")
print(m.group())
```

```
http
```

후방 탐색 (?<=...) (?<!...)

- 후방탐색: 텍스트를 반환하기 전에 뒤쪽을 탐색 look backward/ look behind
- **긍정형 후방 탐색((?<=...))**
- **부정형 후방 탐색((?<!...))**

- 전방탐색과 후방탐색의 비교

```
result = re.search('\w+(?=:)', 'problem1:Solve the EQ1')
result.group()
```

```
'problem1'
```

```
result = re.search('(?<=:)\w+', 'problem1:Solve the EQ1')
result.group()
```

```
'Solve'
```

실습

- **실습1**: 전방 탐색과 후방탐색을 함께 사용해서 결과를 보이시요.
- 예문:

```
<HEAD>
<TITLE>Seo Maria's Homepage</TITLE>
</HEAD>
```

<Title> , <title>등도 해결하도록 하시오

- 결과:

Seo Maria's Homepage

- **실습2**: 파일이름 중에서 확장자가 bat인 파일을 제외한 파일을 정규식을 이용해 찾으시오.
  - 예: input: foo.bar, autoexec.bat, sendmail.cf, checksum.exe
    결과 : foo.bar, sendmail.cf, checksum.exe

- 파일이름 중에서 확장자가 bat 또는 exe인 파일을 제외한 파일을 정규식을 이용해 찾으시오.
  - 결과 : foo.bar, sendmail.cf

Grouping

Group을 만들어주는 메타문자 ( )
: 괄호안의 정규식을 그룹으로 만듦

( ) 괄호: 그룹을 만들어 주는 메타문자

- ABC라는 문자열이 **계속해서 반복되는지** 조사하는 정규식 : (ABC)+

```
import re
p = re.compile('(ABC)+')
m = p.search("ABCABCABC OK?")
print(m)
```

```
<_sre.SRE_Match object; span=(0, 9), match='ABCABCABC'>
```

```
print(m.group())
```

```
ABCABCABC
```

- 매치된 문자열 중에서 **특정 부분의 문자열만 뽑아내기 위해서** 그룹이용

이름 + " " + 전화번호 형태의 문자열을 찾는 정규표현식 **\w+\s+\d+[-]\d+[-]\d+**이다.
그런데 이렇게 매치된 문자열 중에서 이름만 혹은 전화번호만 뽑아내고 싶다면 어떻게 해야 할까?

- ₩w+₩s+₩d+[-]₩d+[-]₩d+에서 이름에 해당되는 ₩w+ 부분을 그룹((₩w+))으로 만들면 match object의 group(index) 메서드를 이용하여 그룹핑된 부분의 문자열만 뽑아낼 수 있다.
- 그룹이 중첩되어 사용되는 경우는 바깥쪽부터 시작하여 안쪽으로 들어갈수록 인덱스가 증가한다.

```
p = re.compile(r"(\w+)\s((\d+)[-]\d+[-]\d+)")
m = p.search("Seo 010-1234-5678")
print(m.group(0)) #매치된 전체 문자열
print("이름:", m.group(1)) #첫번째 그룹에 해당되는 문자열
print("전화번호:", m.group(2))
print("국번:", m.group(3))
```

```
Seo 010-1234-5678
이름: Seo
전화번호: 010-1234-5678
국번: 010
```

Grouping ( )

- ( ) 내에 정규 표현식을 정의하고 특정 단어나 특정 그룹을 표시

Example	Description
(re)	Groups regular expressions and remembers matched text.
(?imx)	Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?-imx)	Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?: re)	Groups regular expressions without remembering matched text.
(?imx: re)	Temporarily toggles on i, m, or x options within parentheses.
(?-imx: re)	Temporarily toggles off i, m, or x options within parentheses.
(?#...)	Comment.
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using pattern negation. Doesn't have a range.
(?< re)	Matches independent pattern without backtracking.

그룹핑된 문자열 재참조하기

정규식 (\b\w+)\s+\1은 (그룹1) + " " + "그룹1과 동일한 단어" 와 매치됨을 의미한다.

재 참조 메타문자인 \1은 정규식의 그룹 중 첫 번째 그룹을 지칭한다.

```
p = re.compile(r"(\b\w+)\s+\1")
p.search('Paris in the the spring').group()
```

```
'the the'
```

```
p = re.compile(r"([Pp])ython&\1ails")
p.search('Python&Pails').group()
```

```
'Python&Pails'
```

그룹핑된 문자열에 이름 붙이기

- 그룹에 이름을 지어주기 위해서는 다음과 같은 확장구문을 사용해야 한다.

  (?P<그룹명>...)

  이름과 전화번호를 추출하는 정규식에서

  (\w+) --> (?P<name>\w+)

```
p = re.compile(r"(?P<name>\w+)\s(?P<TelNum>(\d+)[-]\d+[-]\d+)")
m = p.search("Seo 010-1234-5678")
print(m.group("name"))
```

```
Seo
```

```
print(m.group("TelNum"))
```

```
010-1234-5678
```

```
p = re.compile(r"(?P<word>\b\w+)\s+(?P=word)") #\1대신에
p.search('Paris in the the spring').group()
```

```
'the the'
```

```
import re
str = "DDpotato1 potato2 FFpotato3 SSpotato4 SSpotato5"
re.findall(r"[S]{2}potato\d", str)
```

```
['SSpotato4', 'SSpotato5']
```

```
re.search(r"[S]{2}potato\d", str)
```

```
<_sre.SRE_Match object; span=(28, 37), match='SSpotato4'>
```

```
m =re.search(r"[S]{2}potato\d", str)
print(m.group())
```

```
SSpotato4
```

```
m1 =re.search(r"[S]{2}(potato\d)", str)
print(m1.group()) #group을 만들어 이용하면
print(m1.group(1))
```

```
SSpotato4
potato4
```

```
re.findall(r"[S]{2}(potato\d)", str)
```

```
['potato4', 'potato5']
```

정규식 정의 및 실행

- **re** 모듈은 파이썬이 설치될 때 자동으로 설치되는 기본 라이브러리
- p = re.compile('ab*', re.IGNORECASE) : **re.IGNORECASE** 옵션이 의미하는 것은 대소문자를 구분하지 않음

```
>>> import  re
>>> mat = re.compile("[a-z]+")
>>> mat
<_sre.SRE_Pattern object at 0x063D2C60>
>>>
```

Compile() 함수가 실행되면 컴파일된 패턴 객체가 리턴됨.

패턴은 정규식을 컴파일한 결과

```
>>> m = re.match('[a-z]+', "python")
>>> m.group()
'python'
>>> mo2 = re.match("[a-z]+","abc")
>>> mo2.group()
'abc'
>>>
```

함수를 이용한 모듈 단위로 수행
직접 re 패키지 내의 함수(match() 함수 등)를 사용하여 실행

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

```
<_sre.SRE_Match object; span=(0, 6), match='python'>
```

```
m = p.match("1 python")
print(m)
```

```
None
```

```
mo = re.match('[a-z]+', "python")
mo.group()
```

```
'python'
```

```
mo2 = re.match('[a-z]+', "1 python")
print(mo2)
```

```
None
```

```
mo2 = re.match('[a-z]+', "1 python")
mo2.group()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-b590e9debdaa> in <module>()
      1 mo2 = re.match('[a-z]+', "1 python")
----> 2 mo2.group()

AttributeError: 'NoneType' object has no attribute 'group'
```

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

```
<_sre.SRE_Match object; span=(0, 6), match='python'>
```

```
m = p.match("1 python")
print(m)
```

```
None
```

```
mo = re.match('[a-z]+', "python")
mo.group()
```

```
'python'
```

```
mo2 = re.match('[a-z]+', "1 python")
print(mo2)
```

```
None
```

```
mo2 = re.match('[a-z]+', "Fortran", re.IGNORECASE)
mo2.group()
```

```
'Fortran'
```

```
mo2 = re.match('[a-z]+', "Fortran")
print(mo2)
```

```
None
```

```
m = p.search("python")
print(m)
```

```
<_sre.SRE_Match object; span=(0, 6), match='python'>
```

```
m = p.search("1 python")
print(m)
```

```
<_sre.SRE_Match object; span=(2, 8), match='python'>
```

함수:문자열 검색

- 문자열에 패턴을 찾아 검색이 필요한 경우 처리
- match, search는 정규식과 매치될 때에는 match object를 리턴하고 매치되지 않을 경우에는 None을 리턴

함수	목적
match(패턴,문자열,플래그)	문자열의 처음부터 정규식과 매치되는지 조사한다.
search(패턴,문자열,플래그)	문자열 전체를 검색하여 정규식과 매치되는지 조사한다.

```
line = "Cats are smarter than dogs"
matchObj = re.match( '(.*) are (.*?) (.*)', line, re.M|re.I)
if matchObj:
    print ("matchObj.group() : ", matchObj.group())
    print ("matchObj.group(1) : ", matchObj.group(1))
    print ("matchObj.group(2) : ", matchObj.group(2))
    print ("matchObj.group(3) : ", matchObj.group(3))
else:
    print ("No match!!" )
```

(.*) 패턴은 문자숫자가 연속
(.*?) 패턴은 문자숫자가 연속된 것이 0또는 1

group(숫자)는 각 패턴매칭된 결과

line = "Cats are smarter than dogs"

matchObj = re.match(**r'(.*) are (.*?) (.*)',** line, re.M|re.I)

- Cats 는 (.*) 매칭 , smarter는 (.*?)와 매칭, than dogs는 (.*)와 매칭
- re.I : 대소문자에 관계없이 매치, re.M : 여러 줄과 매치

```
line = "Cats are smarter than dogs"
matchObj = re.match( '(.*) are (.*?) (.*)', line, re.M|re.I)
if matchObj:
    print("matchObj.group() : ", matchObj.group())
    print( "matchObj.group(1) : ", matchObj.group(1))
    print("matchObj.group(2) : ", matchObj.group(2))
    print( "matchObj.group(3) : ", matchObj.group(3))
else:
    print( "No match!!"  )
```

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
matchObj.group(3) :  than dogs
```

```
import re
line = "Cats are smarter than dogs"
mO = re.match(r'(.*) are (.*) (.*)', line, re.M|re.I)
if mO:
    print("matchObj.group() :", mO.group())
    print("matchObj.group(1) :", mO.group(1))
    print("matchObj.group(2) :", mO.group(2))
    print("matchObj.group(3) :", mO.group(3))
else:
    print("No match!")
```

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) : Cats
matchObj.group(2) : smarter than
matchObj.group(3) : dogs
```

```
import re
line = "Cats are smarter than dogs"
mO = re.match(r'(.*) are (.*)', line, re.M|re.I)
if mO:
    print("matchObj.group() :,", mO.group())
    print("matchObj.group(1) :,", mO.group(1))
    print("matchObj.group(2) :,", mO.group(2))
else:
    print("No match!")
```

```
matchObj.group() :, Cats are smarter than dogs
matchObj.group(1) :, Cats
matchObj.group(2) :, smarter than dogs
```

```
import re
line = "Cats are smarter than dogs"
mO = re.match(r'(.*) are (.*?) (.*?) (.*?$)', line, re.M|re.I)
if mO:
    print("matchObj.group() :", mO.group())
    print("matchObj.group(1) :", mO.group(1))
    print("matchObj.group(2) :", mO.group(2))
    print("matchObj.group(3) :", mO.group(3))
    print("matchObj.group(4) :", mO.group(4))
else:
    print("No match!")
```

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) : Cats
matchObj.group(2) : smarter
matchObj.group(3) : than
matchObj.group(4) : dogs
```

```
mO = re.match(r'(.*) are (.*?) (.*?) (.*?)', line, re.M|re.I)
if mO:
    print("matchObj.group() :", mO.group())
    print("matchObj.group(1) :", mO.group(1))
    print("matchObj.group(2) :", mO.group(2))
    print("matchObj.group(3) :", mO.group(3))
    print("matchObj.group(4) :", mO.group(4))
else:
    print("No match!")
```

```
matchObj.group() : Cats are smarter than
matchObj.group(1) : Cats
matchObj.group(2) : smarter
matchObj.group(3) : than
matchObj.group(4) :
```

함수:Greedy(*) vs Non-Greedy(?)

- * 메타문자는 매우 탐욕스러워서 매치할 수 있는 최대한의 문자열인 <html><head><title>Title</title> 문자열을 모두 소모시켜 버렸다. 어떻게 하면 이 탐욕스러움을 제한하고 <html> 이라는 문자열까지만 소모되도록 막을 수 있을까?
- non-greedy 문자인 ?을 사용하면 *의 탐욕을 제한할 수 있다.
- non-greedy 문자인 ?은 ***?, +?, ??, {m,n}?**과 같이 사용할 수 있다.
- 가능한 한 **가장 최소한의 반복을 수행하도록** 도와주는 역할을 한다.

```
s = '<html><head><title>Title</title>'
print(len(s))

print(re.match('<.*>', s).span())
print(re.match('<.*>', s).group())

print(re.match('<.*?>', s).span())
print(re.match('<.*?>', s).group())
```

<.*> 패턴은 모든 매칭을 다 처리해서 결과는
<html><head><title>Title</title>'

<.*?> 패턴 첫 번째만 처리해서 결과는 <html>

```
s = '<html><head><title>Title</title>'
print(len(s))

print(re.match('<.*>', s).span())
print(re.match('<.*>', s).group())

print(re.match('<.*?>', s).span())
print(re.match('<.*?>', s).group())
```

```
32
(0, 32)
<html><head><title>Title</title>
(0, 6)
<html>
```

sub( ): 문자열 수정

- 문자열에 패턴을 찾아 변경이 필요한 경우 처리

함수	목적
sub(pattern, replace, string)	정규식에 매칭되는 것을 변경.

```
import re
phone = "010-1234-1234 #This is Phone Number"
# 주석제거하자
num = re.sub(r"#.*$", "", phone)
print("Phone Number:", num)
# 숫자이외의 모든 문자를 제거하자
num = re.sub(r"\D", "",phone)
print("Phone Number:", num)
```

```
Phone Number: 010-1234-1234
Phone Number: 01012341234
```

패턴 #.*$는 #으로 시작하는 모든 문자를 $(문자열의 끝)까지 매칭

패턴 ₩D는 [^0-9] 즉 숫자가 아닌 문자를 매칭

subn( ) 메서드

- subn : sub와 동일한 기능을 하지만 리턴되는 결과를 튜플로 리턴한다는 차이가 있다.
  - 리턴된 튜플의 첫 번째 요소는 변경된 문자열이고, 두 번째 요소는 바꾸기가 발생한 횟수이다.

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

```
'colour socks and colour socks'
```

```
p.subn('colour', 'blue socks and red socks')
```

```
('colour socks and colour socks', 2)
```

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

```
'colour socks and red socks'
```

```
p.sub('colour', 'blue socks and red socks', count=2)
```

```
'colour socks and colour socks'
```

sub 메서드 사용 시 참조 구문 사용하기

- **이름 + 전화번호**의 문자열을 **전화번호 + 이름**으로 바꾸는 예이다.
- sub의 바꿀 문자열 부분에 ₩g<그룹명>을 이용하면 정규식의 그룹명을 참조할 수 있게된다.
- 그룹명 대신 참조번호를 이용해도 마찬가지 결과가 리턴된다.

```
p = re.compile(r"(?P<name>\w+)\s(?P<phone>(\d+)[-]\d+[-]\d+)")
m = p.sub("\g<phone> \g<name>", "Seo 010-1234-5678")
print(m)
```

```
010-1234-5678 Seo
```

```
print( p.sub("\g<2> \g<1>", "Seo 010-1234-5678"))
```

```
010-1234-5678 Seo
```

sub 메서드의 입력 인수로 함수 넣기

- sub 메서드의 입력 인수로 함수를 넣을 수도 있다.

```
def hexrepl(match):
    "Return the hex string for a decimal number"
    value = int(match.group())
    return hex(value)
p = re.compile(r'\d+')
p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
```

```
'Call 0xffd2 for printing, 0xc000 for user code.'
```

- hexrepl 함수는 match 객체(위에서 숫자에 매치되는)를 입력으로 받아 16진수로 변환하여 리턴하는 함수이다.
- sub의 첫 번째 입력 인수로 함수를 사용할 경우 해당 함수의 첫 번째 입력 인수에는 정규식과 매치된 match 객체가 입력된다. 그리고 매치되는 문자열은 함수의 리턴값으로 바뀌게 된다.

실습 : sub()이용

- 아래 문장을 정규식을 이용해서 해결하시오.

- 입력: "Please, square the following numbers, 3 7 11 13 17 19"
- 출력: Please, square the following numbers, 9 49 121 169 289 361

Compile

정규표현식 -Compile

- 정규표현식 패턴 객체 생성한 후 매칭을 시키는 **객체를 생성하여** 처리하는 방법
  - Compile options
    - ✓ **DOTALL, S** - . 이 줄바꿈 문자를 포함하여 모든 문자와 매치
    - ✓ **IGNORECASE, I** - 대소문자에 관계없이 매치
    - ✓ **MULTILINE, M** – 여러 줄과 매치 (^, $ 메타문자의 사용과 관계가 있는 옵션)
    - ✓ **VERBOSE, X** - verbose 모드를 사용(정규식을 보기 편하게 만들 수 있고 주석 등을 사용)

```
>>> re.compile('.*')
<_sre.SRE_Pattern object at 0x064AB2A8>
>>>
```

Compile Options- DOTALL, S

- . 메타문자는 줄 바꿈 문자(₩n)를 제외한 모든 문자와 매치되는 규칙이 있다.
- ₩n 문자도 포함하여 매치하고 싶은 경우에 **re.DOTALL** 또는 **re.S** 옵션으로 정규식을 컴파일한다.

```
>>> SS = re.compile('.ake')
>>> RR = SS.match('₩nake')
>>> RR
>>> RR == None
True
```

```
>>> SS = re.compile('.ake', re.S)
>>> RR = SS.match('₩nake')
>>> RR == None
False
>>> RR.group()
'₩nake'
>>>
```

₩n은 .과 매치되지 않기 때문이다.
이것이 가능하려면 다음과 같
이 re.S 옵션을 사용해야 한다.

```
import re
ss =re.compile(".ake")
rr =ss.match('\nake')
print(rr == None)
```

```
True
```

```
ss =re.compile(".ake", re.S)
rr =ss.match('\nake')
print(rr == None)
```

```
False
```

```
rr.group()
```

```
'\nake'
```

Compile Options-IGNORECASE, I

- **re.IGNORECASE** 또는 **re.I** 는 대소문자 구분 없이 매치를 수행하고자 할 경우에 사용하는 옵션이다.

```
>>> II = re.compile('[a-z]+')
>>> ii = II.match('Python')
>>> ii == None
True
>>>
```

```
>>> II = re.compile('[a-z]+',re.I)
>>> ii = II.match('Python')
>>> ii == None
False
>>> ii.group()
'Python'
>>>
```

[a-z] 정규식은 소문자만을 의미하지만 re.I 옵션에 의해서 대소문자에 관계없이 매치된다.

```
II = re.compile('[a-z]+')
ii = II.match('Python')
print(ii == None)
```

```
True
```

```
II = re.compile('[a-z]+',re.I)
ii = II.match('Python')
print(ii == None)
```

```
False
```

```
ii.group()
```

```
'Python'
```

Compile Options-MULTILINE, M

- **re.MULTILINE** 또는 **re.M** 옵션은 ^, $ 메타 문자를 문자열의 각 라인마다 적용해준다.
- **^** - 문자열의 처음, **$** - 문자열의 마지막
  - ^python 인 경우 처음은 항상 "python"으로 시작, python$라면 마지막은 항상 "python"으로 끝나야 매치

```
>>> MM = re.compile('^Hello₩s₩w+')
>>> data = """Hello World
... Hello Dahl
... Hello Moon"""
>>> mm = MM.match(data)
>>> mm.group()
'Hello World'
>>>
```

정규식 ^Hello\s\w+ 은 "Hello"라는 문자열로 시작하고 그 후에 whitespace, 그 후에 단어가 와야 한다는 의미이다.

^ 메타 문자에 의해 Hello라는 문자열이 사용된 첫 번째 라인만 매치가 된 것이다.

```
>>> MM = re.compile('^Hello₩s₩w+',re.M)
>>> MM.findall(data)
['Hello World', 'Hello Dahl', 'Hello Moon']
```

re.MULTILINE 옵션으로 인해 ^ 메타문자가 문자열 전체가 아닌 각 라인의 처음이라는 의미를 갖게 되어 다음과 같은 결과가 출력될 것이다.

```
MM = re.compile('^Hello\s\w+')
data = """Hello World
Hello Dahl
Hello Moon"""
mm = MM.match(data)
mm.group()
```

```
'Hello World'
```

```
MM = re.compile('^Hello\s\w+',re.M)
MM.findall(data)
```

```
['Hello World', 'Hello Dahl', 'Hello Moon']
```

Compile Options-VERBOSE, X

- 이해하기 어려운 정규식에 주석 또는 라인단위로 구분을 하여 표시할 수 있도록 처리

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

=

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

정규식이 복잡할 경우 주석과 여러 줄로 표현하여 가독성을 높일 수 있다.

re.VERBOSE 옵션을 사용하면 문자열에 사용된 **whitespace 는 컴파일 시 제거된다. (단 [] 내에 사용된 whitespace는 제외)**

그리고 라인단위로 # 을 이용하여 주석문을 작성하는 것이 가능하게 된다.

```
charref = re.compile(r'&[#](0[0-7]+|[0-9]+|x[0-9a-fA-F]+);')
cr = charref.match('&#x56A;')
if (cr is not None):
    print(cr.group())
else:
    print(cr)
```

```
&#x56A;
```

```
cr = charref.match('&#106;')
if (cr is not None):
    print(cr.group())
else:
    print(cr)
```

```
&#106;
```

```
charref = re.compile(r"""
&[#] #start of a numeric entity reference
(
0[0-7]+ # Octal form
|[0-9]+ # Decimal form
|x[0-9a-fA-F]+ #Hexadecimal form
)
; # Trailing semicolon
""", re.X)
cr = charref.match('&#106;')
if (cr is not None):
    print(cr.group())
else:
    print(cr)
```

```
&#106;
```

```
import re
charref=re.compile(r'&[#](0[0-7]+|[0-9]+|x[0-9a-fA-F]+);')
cr=charref.match('&#076')
if (cr is not None):
    print(cr.group())
else:
    print(cr)
```

```
None
```

```
import re
charref=re.compile(r'&[#](0[0-7]+|[0-9]+|x[0-9a-fA-F]+);')
cr=charref.match('&#076;')
if (cr is not None):
    print(cr.group())
else:
    print(cr)
```

```
&#076;
```

Compile 후 검색

정규표현식 –Compile 후 검색

- match, search는 정규식과 매치될 때에는 match object를 리턴하고 매치되지 않을 경우에는 None을 리턴
- match - 문자열의 처음부터 검색
- search – 문자열 내에서 일치하는 것이 있는지 검색

Method	목적
match()	문자열의 처음부터 정규식과 매치되는지 조사한다.
search()	문자열 전체를 검색하여 정규식과 매치되는지 조사한다.
findall()	정규식과 매치되는 모든 라인의 문자열(substring)을 **리스트로 리턴한다**
finditer()	정규식과 매치되는 모든 라인의 문자열(substring)을 **iterator 객체로** 리턴한다
sub()	정규식과 매치되면 변경시킴
split()	매칭되면 패턴별로 쪼개서 리턴

Parameter	Description
pattern	정규표현식
string	패턴매칭될 문자열
flags	패턴 매칭할 때 필요한 컴파일 options 들로 or(|)연산자로 묶어서 표현

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

```
result = p.findall("life is too short")
print(result)
```

```
['life', 'is', 'too', 'short']
```

```
result = p.finditer("life is too short")
print(result)
for r in result:
    print(r)
```

```
<callable_iterator object at 0x0000020CD406C160>
<_sre.SRE_Match object; span=(0, 4), match='life'>
<_sre.SRE_Match object; span=(5, 7), match='is'>
<_sre.SRE_Match object; span=(8, 11), match='too'>
<_sre.SRE_Match object; span=(12, 17), match='short'>
```

finditer는 findall과 동일하지만 그 결과로 반복 가능한 객체를 리턴한다.
반복 가능한 객체가 포함하는 각각의 요소는 match 객체이다.

```
result = p.findall("life is too short")
print(result)
```

```
['life', 'is', 'too', 'short']
```

```
result = p.finditer("life is too short")
print(result)
for r in result:
    print(r)
```

```
<callable_iterator object at 0x0000020CD406C4A8>
<_sre.SRE_Match object; span=(0, 4), match='life'>
<_sre.SRE_Match object; span=(5, 7), match='is'>
<_sre.SRE_Match object; span=(8, 11), match='too'>
<_sre.SRE_Match object; span=(12, 17), match='short'>
```

```
result = p.finditer("life is too short")
print(result)
for r in result:
    print(r.group())
```

```
<callable_iterator object at 0x0000020CD3FC7320>
life
is
too
short
```

match

- match는 **첫번째 자리부터 동일한 패턴이 발생할 때만** Object가 만들어 짐
- match() 메소드에서 리턴하는 객체를 이용해서 메소드를 가지고 확인

```
>>> mat = re.compile("[a-z]+")
>>> mat
<_sre.SRE_Pattern object at 0x063D2C60>
>>>
>>> mo = mat.match('abc')
>>> mo
<_sre.SRE_Match object at 0x065567C8>
>>>
>>> mo.group()
'abc'
>>>
>>> mo.start()
0
>>> mo.end()
3
>>> mo.span()
(0, 3)
>>> #동일한 패턴이 아니면 미스매칭
>>> if mat.match("  abc") == None :
...     print("mismatch")
...
mismatch
```

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

```
import re
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)
```

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

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

search

- 내부에 있는 패턴을 검색하여 처음부터 매칭되는 것을 검색하여 매칭시킴

```
>>> mat = re.compile("[a-z]+")
>>> mat
<_sre.SRE_Pattern object at
0x063D2C60>
>>>
>>> so1 = mat.search("123abc")
>>> so1
<_sre.SRE_Match object at 0x06556608>
>>> so1.group()
'abc'
>>> so1.start()
3
>>> so1.end()
6
>>> so1.span()
(3, 6)
>>>
```

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

정규식 처리하기

정규식 처리하기(1/2)

1. 정규식 객체 생성

```
>>> import re
>>> p = re.compile("[a-zA-Z0-9]+")
>>> p
<_sre.SRE_Pattern object at 0x06506570>
>>> p.__class__
<type '_sre.SRE_Pattern'>
>>> p.__class__.__name__
'SRE_Pattern'
>>>
```

```
import re
p = re.compile("[a-zA-Z0-9]+")
p
```

```
re.compile(r'[a-zA-Z0-9]+', re.UNICODE)
```

```
p.__class__
```

```
_sre.SRE_Pattern
```

```
p.__class__.__name__
```

```
'SRE_Pattern'
```

2. 정규식 패턴 매칭을 위한 Match object 생성

```
>>> m = p.match("python")
>>> m
<_sre.SRE_Match object at 0x06556AA0>
>>> m.__class__.__name__
'SRE_Match'
>>>
```

```
m=p.match("python")
m
```

```
<_sre.SRE_Match object; span=(0, 6), match='python'
```

```
m.__class__.__name__
```

```
'SRE_Match'
```

```
m.group()
```

```
'python'
```

```
m.span()
```

```
(0, 6)
```

정규식 처리하기(2/2)

3. 정규식 패턴 매칭 결과를 확인

```
>>> m.group()
'python'
>>> m.span()
(0, 6)
>>>
```

실습 1

- 아래와 같은 data에서 찾은 부품들의 총 가격을 구하기 위해, 각 부품의 가격을 추출하시오.

```
data = """
ABC01: $23.45
HGG42: $5.01
CFXE1: $889.00
XTC99: $69.89
Total items found: 4
"""
```

- 출력

```
23.45
5.01
889.00
69.89
```

실습 2

- 아래와 같은 예문에서 가격과 수량을 나타내는 숫자들을 추출하시오.

```
data = """
I paid $30 for 100 apples,
50 oranges, and 60 pears.
I saved $5 on this order.
"""
```

- 출력

```
가격을 나타내는 숫자들
30
5
```

```
수량을 나타내는 숫자들
100
50
60
```