# 7장 데이터 정제 및 준비

## 문자열 다루기

2020.06.26 2h

# 7.3 문자열 다루기

- **파이썬 인기 이유 중  하나**
  - 문자열이나 텍스트 처리가 용이
  - 기본 내장 함수 사용
- **정규 표현식**
  - Regular Expressions
    - **복잡한 매칭이나 텍스트 조작에 활용**

# 7.3.2 정규 표현식 regex

- **파이썬 모듈 re**
  - Regular Expression
    - **텍스트에서 문자열 패턴을 찾는 유연한 방법을 제공**
    - **3가지 부류**
      - **패턴 매칭, 치환, 분류**
  - 컴파일하고 split 메소드 실행
    - **re.split( ' ₩s+ ' , text)**
  - ₩s+
    - **하나 이상의 공백문자(탭, space, 개행문자 등)**

**Regular Expressions**

```
In [220]: import re
          text = "foo    bar₩t baz  ₩tqux"
          re.split('₩s+', text)
Out[220]: ['foo', 'bar', 'baz', 'qux']

In [223]: regex = re.compile('₩s+')
          regex
Out[223]: re.compile(r'₩s+', re.UNICODE)

In [224]: regex.split(text)
Out[224]: ['foo', 'bar', 'baz', 'qux']

In [225]: regex.findall(text)
Out[225]: ['    ', '₩t ', '  ₩t']
```



https://wikidocs.net/4308

# 정규 표현식

- **하나의 단원으로 구성 가능**
  - 내용이 많음

# 7장 데이터 정제 및 준비

# 정규 표현식 미니 강의

# 정규 표현식의 기초

- **메타 문자**
  - 정규 표현식에서 사용하는 메타 문자(meta characters)
    - **※ 메타 문자란 원래 그 문자가 가진 뜻이 아닌 특별한 용도로 사용하는 문자**
  - . ^ $ * + ? { } [ ] ₩ | ( )
    - **정규 표현식에 위 메타 문자를 사용하면 특별한 의미**

# 문자 클래스 []

- **메타 문자는 바로 문자 클래스(character class)인 [ ]**
  - 문자 클래스로 만들어진 정규식은 "[ ] 사이의 문자들과 매치"라는 의미
    - **두 문자 사이에 하이픈(-)을 사용하면 두 문자 사이의 범위(From - To)를 의미**
    - **[a-zA-Z] : 알파벳 모두**
    - **[0-9] : 숫자**
  - ₩s+
    - **하나 이상의 공백 문자(whitespace)가 있는 문자열 매칭**

**[자주 사용하는 문자 클래스]**

[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.

대문자로 사용된 것은 소문자의 반대임을 추측할 수 있다.

# 정규 표현식의 Dot(.) 메타 문자

- **Dot(.)**
  - 줄바꿈 문자인 ₩n을 제외한 모든 문자와 매치됨을 의미
- **a.b**
  - "a + 모든문자 + b"
    - **a와 b라는 문자 사이에 어떤 문자가 들어가도 모두 매치된다는 의미**
      - **axb, a0b: 매칭**
      - **abc: 비매칭**

        » "a"문자와 "b"문자 사이에 어떤 문자라도 하나는 있어야 하는 이 정규식과 일치하지 않으므로
- **그럼 a[.]b**
  - " a + .(마침표) + b"
    - **"a.b" 문자열과 매치**
    - **"a0b" 문자열과는 매치되지 않음**
  - 문자 클래스([]) 내에 .
    - **문자 . 그대로를 의미, 마침표를 의미**

# 0 또는 반복

- **반복 ***
  - ca*t
    - *** 바로 앞에 있는 문자 a가 0부터 무한대로 반복될 수 있다는 의미**

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

# 1 또는 이상 반복

- **반복 +**
  - +는 최소 1번 이상 반복될 때 사용
    - **즉 *가 반복 횟수 0부터라면 +는 반복 횟수 1부터인 것**
  - ca+t
    - **"c + a(1번 이상 반복) + t"**

| 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 ({m,n}, ?)

- **반복 횟수를 3회만 또는 1회부터 3회까지만으로 제한하고 싶을 수도 있지 않을까?**
  - {m, n} 정규식
    - **반복 횟수가 m부터 n까지 매치**
    - **또한 m 또는 n을 생략**
      - **{3,}처럼 사용: 반복 횟수가 3 이상**
      - **{,3}처럼 사용하면 반복 횟수가 3 이하를 의미**
      - **생략된 m은 0과 동일하며, 생략된 n은 무한대(2억 개 미만)의 의미**
  - {1,}은 +와 동일하고, {0,}은 *와 동일

# ?

- **? 메타 문자가 의미**
  - {0, 1}
  - ab?c
    - **"a + b(있어도 되고 없어도 된다) + c"**

| 정규식 | 문자열 | Match 여부 | 설명 |
|---|---|---|---|
| `ab?c` | abc | Yes | "b"가 1번 사용되어 매치 |
| `ab?c` | ac | Yes | "b"가 0번 사용되어 매치 |

# 파이썬에서 정규 표현식을 지원하는 re 모듈

- **re(regular expression의 약어) 모듈을 제공**
  - 파이썬을 설치할 때 자동으로 설치되는 기본 라이브러리
    - **import re**
    - **p = re.compile('ab*')**
      - **re.compile을 사용하여 정규 표현식(위 예에서는 ab*)을 컴파일**
      - **re.compile의 결과로 돌려주는 객체 p(컴파일된 패턴 객체)를 사용하여 그 이후의 작업을 수행**

# 정규식을 이용한 문자열 검색

- **컴파일된 패턴 객체를 사용하여 문자열 검색을 수행**
  - 패턴이란 정규식을 컴파일한 결과
- **컴파일된 패턴 객체는 다음과 같은 4가지 메서드 제공**
  - match, search는 정규식과 매치될 때는 match 객체를 돌려 주고, 매치되지 않을 때는 None을 돌려 줌
    - **※ match 객체란 정규식의 검색 결과로 돌려주는 객체**

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

| Method | 목적 |
|---|---|
| match() | 문자열의 처음부터 정규식과 매치되는지 조사한다. |
| search() | 문자열 전체를 검색하여 정규식과 매치되는지 조사한다. |
| findall() | 정규식과 매치되는 모든 문자열(substring)을 리스트로 돌려준다. |
| finditer() | 정규식과 매치되는 모든 문자열(substring)을 반복 가능한 객체로 돌려준다. |

# match

- **match 메서드**
  - 문자열의 처음부터 정규식과 매치되는지 조사
    - **"python" 문자열은 [a-z]+ 정규식에 부합되므로 match 객체를 반환**
    - **"3 python" 문자열은 처음에 나오는 문자 3이 정규식 [a-z]+에 부합되지 않으므로 None을 반환**
  - 다음과 같은 흐름으로 작성
    - **match의 결과로 match 객체 또는 None을 반환**
    - **즉 match의 결괏값이 있을 때만 그 다음 작업을 수행**

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

```
>>> m = p.match("python")
>>> print(m)
<_sre.SRE_Match object at 0x01F3F9F8>
```

```
>>> m = p.match("3 python")
>>> print(m)
None
```

```
p = re.compile(정규표현식)
m = p.match( 'string goes here' )
if m:
    print('Match found: ', m.group())
else:
    print('No match')
```

# search

- **컴파일된 패턴 객체 p**
  - search 메서드를 수행
    - **문자열 전체에서 검색**
    - **match 메서드를 수행했을 때와 동일하게 매치**
  - "3 " 이후의 "python" 문자열과 매치
    - **"3 python" 문자열의 첫 번째 문자는 "3"이지만 search는 문자열의 처음부터 검색하는 것이 아니라 문자열 전체를 검색하기 때문**
  - 패턴과 일치하는 첫번째 것을 반환
  - match 메서드와 search 메서드는 문자열의 처음부터 검색할 지의 여부에 따라 다르게 사용

```
>>> m = p.search("python")
>>> print(m)
<_sre.SRE_Match object at 0x01F3FA68>
```

```
>>> m = p.search("3 python")
>>> print(m)
<_sre.SRE_Match object at 0x01F3FA30>
```

# findall finditer

- **findall**
  - 패턴과 일치하는 모든 부분 문자열을 리스트로 반환
    - **"life is too short" 문자열의 'life', 'is', 'too', 'short' 단어를 각각 [a-z]+ 정규식과 매치해서 리스트로 반환**
- **finditer**
  - findall과 동일하지만 그 결과로 반복 가능한 객체(iterator object)를 반환
    - **반복 가능한 객체가 포함하는 각각의 요소는 match 객체**

```
>>> result = p.findall("life is too short")
>>> print(result)
['life', 'is', 'too', 'short']
```

```
>>> result = p.finditer("life is too short")
>>> print(result)
<callable_iterator object at 0x01F5E390>
>>> for r in result: print(r)
...
<_sre.SRE_Match object at 0x01F3F9F8>
<_sre.SRE_Match object at 0x01F3FAD8>
<_sre.SRE_Match object at 0x01F3FAA0>
<_sre.SRE_Match object at 0x01F3F9F8>
```

# match 객체의 메서드

- **match 메서드와 search 메서드를 수행한 결과로 돌려준 match 객체**
  - 어떤 문자열이 매치되었는가?
  - 매치된 문자열의 인덱스는 어디서부터 어디까지인가?

| method | 목적 |
|---|---|
| group() | 매치된 문자열을 돌려준다. |
| start() | 매치된 문자열의 시작 위치를 돌려준다. |
| end() | 매치된 문자열의 끝 위치를 돌려준다. |
| span() | 매치된 문자열의 (시작, 끝)에 해당하는 튜플을 돌려준다. |

```
>>> m = p.match("python")
>>> m.group()
'python'
>>> m.start()
0
>>> m.end()
6
>>> m.span()
(0, 6)
```

```
>>> m = p.search("3 python")
>>> m.group()
'python'
>>> m.start()
2
>>> m.end()
8
>>> m.span()
(2, 8)
```

# 모듈 단위로 수행하기

- **축약한 형태**
  - re.match(pattern, "검색할 문자열")
    - **컴파일과 match 메서드를 한 번에 수행**
    - **보통 한 번 만든 패턴 객체를 여러 번 사용해야 할 때는 이 방법보다 re.compile을 사용하는 것이 편함**

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

위 코드가 축약된 형태는 다음과 같다.

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

# 컴파일 옵션

- **정규식을 컴파일할 때 다음 옵션을 사용**
  - IGNORECASE(I)
    - **대소문자에 관계없이 매치할 수 있도록**
  - DOTALL(S)
    - **. 이 줄바꿈 문자를 포함하여 모든 문자와 매치할 수 있도록**
  - MULTILINE(M)
    - **여러 줄과 매치할 수 있도록 (^, $ 메타 문자의 사용과 관계가 있는 옵션)**
  - VERBOSE(X)
    - **verbose 모드를 사용할 수 있도록**
      - **정규식을 보기 편하게 만들 수 있고 주석 등을 사용**
  - 사용 방법
    - **re.DOTALL처럼 전체 옵션 이름을 써도 되고**
    - **re.S처럼 약어를 써도 가능**

# IGNORECASE, I

- **re.IGNORECASE 또는 re.I 옵션**
  - 대소문자 구별 없이 매치를 수행할 때 사용하는 옵션
  - [a-z] 정규식은 소문자 만을 의미하지만
    - **re.I 옵션으로 대소문자 구별 없이 매치**

```
>>> p = re.compile('[a-z]', re.I)
>>> p.match('python')
<_sre.SRE_Match object at 0x01FCFA30>
>>> p.match('Python')
<_sre.SRE_Match object at 0x01FCFA68>
>>> p.match('PYTHON')
<_sre.SRE_Match object at 0x01FCF9F8>
```

# 패턴의 이해

- **pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+₩.[A-Z]{2,4}'**
- **userid@server.domain**
  - Username 구성
    - **[A-Z0-9._%+-]+**
      - **[A-Z0-9._%+-]이 하나 이상**
      - **a 부터 z , 0 부터 9 까지의 문자, ._%+-가 1개 이상**
  - Domain 구성
    - **[A-Z0-9.-]+**
      - **[A-Z0-9.-]이 하나 이상**
  - Suffix 구성
    - **[A-Z]{2,4}**
      - **알파벳의 개수가 2에서 4까지만**

# 7장 데이터 정제 및 준비

## 이메일 주소를 검사하는 정규 표현식

# compile과 findall

- **Pattern**
  - pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+₩.[A-Z]{2,4}'
    - 이메일 주소를 검사하는 정규 표현식

```
In [300]: text = """Dave dave@google.com
          Steve steve@gmail.com
          Rob rob@gmail.com
          Ryan ryan@yahoo.com
          """

In [301]: pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+₩.[A-Z]{2,4}'
          pattern

Out[301]: '[A-Z0-9._%+-]+@[A-Z0-9.-]+₩₩.[A-Z]{2,4}'

In [302]: # re.IGNORECASE makes the regex case-insensitive
          regex = re.compile(pattern, flags=re.IGNORECASE)

In [303]: regex.findall(text)

Out[303]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

# search

- **텍스트에서 첫 번째 이메일 주소만을 찾아 줌**
  - 시작점과 큰 위치를 알 수 있음

```
In [304]: m = regex.search(text)
          m
Out[304]: <re.Match object; span=(5, 20), match='dave@google.com'>

In [305]: text[m.start():m.end()]
Out[305]: 'dave@google.com'

In [306]: print(m.start(), m.end())
          5 20
```

# group과 findall

- **이메일 주소를 찾아서 동시에**
  - 각 이메일 주소를 사용자이름, 도메인이름, 도메인 접미사 3가지 부분으로 나누기
    - **각 패턴에 괄호: 문자 그룹을 정의**
      - **괄호 내 쌍이 그룹을 형성**
    - **group() 메소드**
      - **각 패턴 콤포넌트의 튜플을 반환**
    - **findall() 메소드**
      - **튜플의 리스트를 반환**

```
In [309]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)₩.([A-Z]{2,4})'
          regex = re.compile(pattern, flags=re.IGNORECASE)

In [310]: m = regex.match('wesm@bright.net')
          m.groups()

Out[310]: ('wesm', 'bright', 'net')

In [311]: regex.findall(text)

Out[311]: [('dave', 'google', 'com'),
           ('steve', 'gmail', 'com'),
           ('rob', 'gmail', 'com'),
           ('ryan', 'yahoo', 'com')]
```

# sub의 특수 기호

- **패턴 그룹 참조 방법**
  - ₩1, ₩2
    - **각각 첫 번째, 두 번째 찾은 그룹**

```
In [312]: print(regex.sub(r'Username: ₩1, Domain: ₩2, Suffix: ₩3', text))

Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

# 7.3.3 pandas의 벡터화된 문자열 함수

- **Series 및 Index**
  - 배열의 각 요소에서 쉽게 조작 할 수 있는 일련의 문자열 처리 방법
    - **누락 NA 값을 자동으로 제외한다는 것**
    - **이들은 str속성을 통해 액세스**
      - **일반적으로 동등한 (스칼라) 내장 문자열 메소드와 일치하는 이름이 있음**

```
In [16]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'],
    ....:                dtype="string")
    ....:

In [17]: s.str.lower()
Out[17]:
0       a
1       b
2       c
3    aaba
4    baca
5    <NA>
6    caba
7     dog
8     cat
dtype: string

In [18]: s.str.upper()
Out[18]:
0       A
1       B
2       C
3    AABA
4    BACA
5    <NA>
6    CABA
7     DOG
8     CAT
dtype: string

In [19]: s.str.len()
Out[19]:
0       1
1       1
2       1
3       4
4       4
5    <NA>
6       4
7       3
8       3
dtype: Int64
```



https://pandas.pydata.org/pandas-docs/stable/user_guide/text.html