세상의 속도를
따라잡고 싶다면

점프 투 파이썬

박응용 지음(위키독스 운영자)

이지스퍼블리싱

08

정규 표현식

08-3 강력한 정규 표현식의 세계로

강력한 정규 표현식의 세계

문자열 소비가 없는 메타 문자

- +, *, [ ], { } 등의 메타 문자는 매치가 성사되면 문자열을 탐색하는 시작 위치가 변경됨
  - 보통 소비된다고 표현

- 이와 달리 문자열을 소비시키지 않는 메타 문자도 존재
- 문자열 소비가 없는(zerowidth assertions) 메타 문자

강력한 정규 표현식의 세계

- **문자열 소비가 없는 메타 문자**
  - |
    - or과 동일한 의미
    - A | B : A 또는 B라는 의미

```
>>> p = re.compile('Crow|Servo')
>>> m = p.match('CrowHello')
>>> print(m)
<re.Match object; span=(0, 4), match='Crow'>
```

강력한 정규 표현식의 세계

- **문자열 소비가 없는 메타 문자**
  - ^
    - 문자열의 맨 처음과 일치함을 의미

```
>>> print(re.search('^Life', 'Life is too short'))
<re.Match object; span=(0, 4), match='Life'>
>>> print(re.search('^Life', 'My Life'))
None
```

      - ^Life : Life 문자열이 처음에 온 경우에는 매치하지만, 처음 위치가 아닌 경우에는 매치되지 않음

강력한 정규 표현식의 세계

문자열 소비가 없는 메타 문자

$

- ^ 메타 문자와 반대의 경우
- 문자열의 끝과 매치함을 의미

```
>>> print(re.search('short$', 'Life is too short'))
<re.Match object; span=(12, 17), match='short'>
>>> print(re.search('short$', 'Life is too short, you need python'))
None
```

- short$: 검색할 문자열이 short로 끝난 경우에는 매치되지만 그 이외의 경우에는 매치되지 않음

강력한 정규 표현식의 세계

문자열 소비가 없는 메타 문자

- **₩A**
  - 문자열의 처음과 매치됨을 의미
  - ^ 메타 문자와 동일한 의미이지만 re.MULTILINE 옵션 사용 시 다르게 해석됨
    - ^ : 각 줄의 문자열의 처음과 매치
    - ₩A : 줄과 상관없이 전체 문자열의 처음하고만 매치

강력한 정규 표현식의 세계

- **문자열 소비가 없는 메타 문자**
  - **₩Z**
    - 문자열의 끝과 매치됨을 의미
    - ₩A와 동일하게 re.MULTILINE 옵션을 사용할 경우 $ 메타 문자와 다르게 해석됨
      - $: 각 줄의 문자열의 끝과 매치
      - ₩Z : 줄과 상관없이 전체 문자열의 끝하고만 매치

강력한 정규 표현식의 세계

문자열 소비가 없는 메타 문자

₩b

- 단어 구분자(word boundary), 보통 단어는 화이트스페이스에 의해 구분됨

```
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<re.Match object; span=(3, 8), match='class'>
```

  - ₩bclass₩b : 앞뒤가 화이트스페이스로 구분된 class라는 단어와 매치

- 파이썬 리터럴 규칙에 따르면 ₩b는 백스페이스(backspace)를 의미하므로, 정규식에서 사용할 때는 백스페이스가 아닌 단어 구분자라는 것을 알리기 위해 r'₩bclass₩b'처럼 raw string임을 알려 주는 기호 r을 반드시 붙여야 함

강력한 정규 표현식의 세계

문자열 소비가 없는 메타 문자

₩B

- ₩b 메타 문자와 반대의 경우
- 화이트스페이스로 구분된 단어가 아닌 경우에만 매치

```
>>> p = re.compile(r'\Bclass\B')
>>> print(p.search('no class at all'))
None
>>> print(p.search('the declassified algorithm'))
<re.Match object; span=(6, 11), match='class'>
>>> print(p.search('one subclass is'))
None
```

강력한 정규 표현식의 세계

그루핑

- ABC 문자열이 계속해서 반복되는지 조사하는 정규식을 작성하고 싶다면 그루핑(grouping)으로 해결 가능

```
(ABC)+
```

- ( ): 그룹을 만들어주는 메타 문자

강력한 정규 표현식의 세계

그루핑

- 예) '이름 + " " + 전화번호' 형태의 문자열을 찾는 정규식

```
>>> p = re.compile(r"\w+\s+\d+[-]\d+[-]\d+")
>>> m = p.search("park 010-1234-1234")
```

  - 매치된 문자열 중에서 특정 부분만 뽑아내고 싶을 때 그룹 활용
    - match 객체의 group(인덱스) 메서드 활용

```
>>> p = re.compile(r"(\w+)\s+\d+[-]\d+[-]\d+")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(1))
park
```

```
>>> p = re.compile(r"(\w+)\s+(\d+[-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(2))
010-1234-1234
```

강력한 정규 표현식의 세계

그루핑

group(인덱스) 메서드의 인덱스 의미

group(인덱스)	설명
group(0)	매치된 전체 문자열
group(1)	첫 번째 그룹에 해당되는 문자열
group(2)	두 번째 그룹에 해당되는 문자열
group(n)	n번째 그룹에 해당되는 문자열

강력한 정규 표현식의 세계

그루핑

그루핑된 문자열 재참조(backreferences)하기

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

- 정규식 (₩b₩w+)₩s+₩1은 (그룹) + " " + 그룹과_동일한_단어와 매치된다는 것을 의미
- 이렇게 정규식을 만들면 2개의 동일한 단어를 연속적으로 사용해야만 매치됨
- 재참조 메타 문자 ₩1

강력한 정규 표현식의 세계

그루핑

- 그루핑된 문자에 이름 붙이기
  - 그룹을 인덱스가 아닌 이름(named groups)으로 참조하는 방법

```
(?P<그룹명>...)
```

  - 예) 이름과 전화번호를 추출하는 정규식

```
(?P<name>\w+)\s+((\d+)[-]\d+[-]\d+)
```

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

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

강력한 정규 표현식의 세계

전방 탐색

- 정규식의 전방 탐색(lookahead assertions) 확장 구문
  - 암호문처럼 알아보기 어렵게 바뀜

- 전방 탐색의 종류와 표현

• 긍정형 전방 탐색((?=...)): ...에 해당하는 정규식과 매치되어야 하며 조건이 통과되어도 문자열이 소비되지 않는다. • 부정형 전방 탐색((?!...)): ...에 해당하는 정규식과 매치되지 않아야 하며 조건이 통과되어도 문자열이 소비되지 않는다.

강력한 정규 표현식의 세계

전방 탐색

- 긍정형(positive) 전방 탐색
  - **(?=:)**
  - 기존 정규식과 검색에서는 동일한 효과를 발휘하지만, :에 해당하는 문자열이 정규식 엔진에 의해 소비되지 않아(검색에는 포함되지만 검색 결과에서는 제외됨) 검색 결과에서는 :이 제거된 후 리턴해 주는 효과가 있음

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

강력한 정규 표현식의 세계

전방 탐색

- 부정형(negative) 전방 탐색
  - ...에 해당하는 정규식과 매치되지 않아야 하며
    조건이 통과되어도 문자열이 정규식 엔진에 의해 소비되지 않음

```
.*[.](?!bat$).*$
```

  - bat가 아닌 경우에만 통과된다는 의미
  - bat 문자열이 있는지 조사하는 과정에서 문자열이 소비되지 않으므로
    bat가 아니라고 판단되면 그 이후 정규식 매치가 진행됨

강력한 정규 표현식의 세계

문자열 바꾸기

- sub 메서드를 사용하면 정규식과 매치되는 부분을 다른 문자로 쉽게 바꿀 수 있음
  - sub(바꿀 문자열, 대상 문자열)

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

  - 바꾸기 횟수를 제어하려면 세 번째 인수에 count 값 설정

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

강력한 정규 표현식의 세계

문자열 바꾸기

- sub 메서드 사용 시 참조 구문 사용하기
  - **이름 + 전화번호**의 문자열을 **전화번호 + 이름**으로 바꿈

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

  - 그룹 이름 대신 참조 번호 사용 가능

```
>>> p = re.compile(r"(?P<name>\w+)\s+(?P<phone>(\d+)[-]\d+[-]\d+)")
>>> print(p.sub("\g<2> \g<1>", "park 010-1234-1234"))
010-1234-1234 park
```

강력한 정규 표현식의 세계

문자열 바꾸기

- sub 메서드의 매개변수로 함수 넣기
  - 첫 번째 인수에 함수를 전달할 수도 있음

```
>>> def hexrepl(match):
...     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.'
```

  - 해당 함수의 첫 번째 매개변수에는 정규식과 매치된 match 객체가 입력됨
  - 매치되는 문자열은 함수의 리턴값으로 바뀜

강력한 정규 표현식의 세계

greedy와 non-greedy

- greedy(탐욕스러운): 메타 문자가 매치할 수 있는 최대한의 문자열을 모두 소비하는 현상
- 예)<html>이 아닌 문자열 전체를 반환

```
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print(re.match('<.*>', s).span())
(0, 32)
>>> print(re.match('<.*>', s).group())
<html><head><title>Title</title>
```

탐욕을 제한

- non-greedy
  - ? 문자 사용
  - 되도록 최소한으로 반복을 수행

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

  - *?, +?, ??, [m,n]?와 같이 사용 가능

감사합니다.