UNIT 43

정규표현식 사용하기

43 정규표현식 사용하기

정규표현식 사용하기

- 정규표현식(regular expression)은 일정한 규칙(패턴)을 가진 문자열을 표현하는 방법
- 복잡한 문자열 속에서 특정한 규칙으로 된 문자열을 검색한 뒤 추출하거나 바꿀 때 사용함
- 문자열이 정해진 규칙에 맞는지 판단할 때도 사용함

43.1 문자열 판단하기

» 문자열 판단하기

- 정규표현식은 re 모듈을 가져와서 사용하며 match 함수에 정규표현식 패턴과 판단할 문자열을 넣음(re는 regular expression의 약자)

```
• re.match('패턴', '문자열')
```

- 다음은 'Hello, world!' 문자열에 'Hello'와 'Python'이 있는지 판단함

```
>>> import re
>>> re.match('Hello', 'Hello, world!')     # 문자열이 있으므로 정규표현식 매치 객체가 반환됨
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> re.match('Python', 'Hello, world!')    # 문자열이 없으므로 아무것도 반환되지 않음
```

- 여기서는 'Hello'가 있으므로 match='Hello'와 같이 패턴에 매칭된 문자열이 표시됨
- 'Hello, world!'.find('Hello')처럼 문자열 메서드로도 충분히 가능함

43.1 문자열 판단하기

» 문자열이 맨 앞에 오는지 맨 뒤에 오는지 판단하기

- 문자열 앞에 ^를 붙이면 문자열이 맨 앞에 오는지 판단하고, 문자열 뒤에 $를 붙이면 문자열이 맨 뒤에 오는지 판단함(특정 문자열로 끝나는지)

  - ^문자열
  - 문자열$

- 이때는 match 대신 search 함수를 사용해야 함
- match 함수는 문자열 처음부터 매칭되는지 판단하지만, search는 문자열 일부분이 매칭되는지 판단함

  - re.search('패턴', '문자열')

```
>>> re.search('^Hello', 'Hello, world!')     # Hello로 시작하므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> re.search('world!$', 'Hello, world!')    # world!로 끝나므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(7, 13), match='world!'>
```

43.1 문자열 판단하기

» 지정된 문자열이 하나라도 포함되는지 판단하기

- 기본 개념은 OR 연산자와 같음

```
• 문자열|문자열
• 문자열|문자열|문자열|문자열
```

- 'hello|world'는 문자열에서 'hello' 또는 'world'가 포함되는지 판단함

```
>>> re.match('hello|world', 'hello')    # hello 또는 world가 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 5), match='hello'>
```

43.2 범위 판단하기

» 범위 판단하기

- 다음과 같이 [ ](대괄호) 안에 숫자 범위를 넣으며 * 또는 +를 붙임
- 숫자 범위는 0-9처럼 표현하며 *는 문자(숫자)가 0개 이상 있는지, +는 1개 이상 있는지 판단함

- [0-9]+

```
>>> re.match('[0-9]*', '1234')    # 1234는 0부터 9까지 숫자가 0개 이상 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 4), match='1234'>
>>> re.match('[0-9]+', '1234')    # 1234는 0부터 9까지 숫자가 1개 이상 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 4), match='1234'>
>>> re.match('[0-9]+', 'abcd')    # 1234는 0부터 9까지 숫자가 1개 이상 없으므로 패턴에 매칭되지 않음
```

- *와 + 활용은 다음과 같이 a*b와 a+b를 확인해보면 쉽게 알 수 있음

```
>>> re.match('a*b', 'b')      # b에는 a가 0개 이상 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 1), match='b'>
>>> re.match('a+b', 'b')      # b에는 a가 1개 이상 없으므로 패턴에 매칭되지 않음
>>> re.match('a*b', 'aab')    # aab에는 a가 0개 이상 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 3), match='aab'>
>>> re.match('a+b', 'aab')    # aab에는 a가 1개 이상 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 3), match='aab'>
```

43.2 범위 판단하기

» 문자가 한 개만 있는지 판단하기

- ?와 .을 사용함
- ?는 ? 앞의 문자(범위)가 0개 또는 1개인지 판단하고, .은 .이 있는 위치에 아무 문자(숫자)가 1개 있는지 판단합니다.

- 문자?
- [0-9]?
- .

```
>>> re.match('abc?d', 'abd')          # abd에서 c 위치에 c가 0개 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 3), match='abd'>
>>> re.match('ab[0-9]?c', 'ab3c')     # [0-9] 위치에 숫자가 1개 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 4), match='ab3c'>
>>> re.match('ab.d', 'abxd')          # .이 있는 위치에 문자가 1개 있으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 4), match='abxd'>
```

43.2 범위 판단하기

» 문자 개수 판단하기

- 문자열의 경우에는 문자열을 괄호로 묶고 뒤에 {개수} 형식을 지정함

- 문자{개수}
- (문자열){개수}

- h{3}은 h가 3개 있는지 판단하고, (hello){3}은 hello가 3개 있는지 판단함

```
>>> re.match('h{3}', 'hhhello')
<_sre.SRE_Match object; span=(0, 3), match='hhh'>
>>> re.match('(hello){3}', 'hellohellohelloworld')
<_sre.SRE_Match object; span=(0, 15), match='hellohellohello'>
```

43.2 범위 판단하기

» 문자 개수 판단하기

- 특정 범위의 문자(숫자)가 몇 개 있는지 판단할 수도 있음
- 이때는 범위 [ ] 뒤에 {개수} 형식을 지정함
  - `[0-9]{개수}`
- 다음은 휴대전화의 번호 형식에 맞는지 판단함

```
>>> re.match('[0-9]{3}-[0-9]{4}-[0-9]{4}', '010-1000-1000')    # 숫자 3개-4개-4개 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 13), match='010-1000-1000'>
>>> re.match('[0-9]{3}-[0-9]{4}-[0-9]{4}', '010-1000-100')   # 숫자 3개-4개-4개 패턴에 매칭되지 않음
```

43.2 범위 판단하기

» 문자 개수 판단하기

- 이 기능은 문자(숫자)의 개수 범위도 지정할 수 있음
- {시작개수,끝개수} 형식으로 시작 개수와 끝 개수를 지정해주면 특정 개수 사이에 들어가는지 판단함

- (문자){시작개수,끝개수}
- (문자열){시작개수,끝개수}
- [0-9]{시작개수,끝개수}

- 다음은 일반전화의 번호 형식에 맞는지 판단함

```
>>> re.match('[0-9]{2,3}-[0-9]{3,4}-[0-9]{4}', '02-100-1000')    # 2~3개-3~4개-4개 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 11), match='02-100-1000'>
>>> re.match('[0-9]{2,3}-[0-9]{3,4}-[0-9]{4}', '02-10-1000')  # 2~3개-3~4개-4개 패턴에 매칭되지 않음
```

43.2 범위 판단하기

» 숫자와 영문 문자를 조합해서 판단하기

- 영문 문자 범위는 a-z, A-Z와 같이 표현함

```
• a-z
• A-Z
```

```
>>> re.match('[a-zA-Z0-9]+', 'Hello1234')    # a부터 z, A부터 Z, 0부터 9까지 1개 이상 있으므로
<_sre.SRE_Match object; span=(0, 9), match='Hello1234'>                    # 패턴에 매칭됨
>>> re.match('[A-Z0-9]+', 'hello')    # 대문자, 숫자는 없고 소문자만 있으므로 패턴에 매칭되지 않음
```

- 한글은 영문 문자와 방법이 같음
- 가-힣처럼 나올 수 있는 한글 조합을 정해주면 됨

43.2 범위 판단하기

» 특정 문자 범위에 포함되지 않는지 판단하기

- 특정 문자 범위에 포함되지 않는지 판단하려면 다음과 같이 문자(숫자) 범위 앞에 ^를 붙이면 해당 범위를 제외함

  - **[^범위]***
  - **[^범위]+**

- '[^A-Z]+'는 대문자를 제외한 모든 문자(숫자)가 1개 이상 있는지 판단함

```
>>> re.match('[^A-Z]+', 'Hello')    # 대문자를 제외. 대문자가 있으므로 패턴에 매칭되지 않음
>>> re.match('[^A-Z]+', 'hello')    # 대문자를 제외. 대문자가 없으므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 5), match='hello'>
```

43.2 범위 판단하기

» 특정 문자 범위에 포함되지 않는지 판단하기

- 범위를 제외할 때는 '[^A-Z]+'와 같이 [ ] 안에 넣어주고, 특정 문자 범위로 시작할 때는 '^[A-Z]+'와 같이 [ ] 앞에 붙여줌
- 다음과 같이 '^[A-Z]+'는 영문 대문자로 시작하는지 판단함

- ^[범위]*
- ^[범위]+

```
>>> re.search('^[A-Z]+', 'Hello')        # 대문자로 시작하므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 1), match='H'>
```

43.2 범위 판단하기

» 특정 문자 범위에 포함되지 않는지 판단하기

- 특정 문자(숫자) 범위로 끝나는지 확인할 때는 정규표현식 뒤에 $를 붙이면 됨

- [범위]*$
- [범위]+$

```
>>> re.search('[0-9]+$', 'Hello1234')    # 숫자로 끝나므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(5, 9), match='1234'>
```

43.2 범위 판단하기

» 특수 문자 판단하기

- 정규표현식에 사용하는 특수 문자 *, +, ?, ., ^, $, (,) [,], – 등을 판단하려면 특수 문자를 판단할 때는 특수 문자 앞에 \를 붙이면 됨
- 단, [ ] 안에서는 \를 붙이지 않아도 되지만 에러가 발생하는 경우에는 \를 붙임

• \특수문자

```
>>> re.search('\*+', '1 ** 2')                    # *이 들어있는지 판단
<_sre.SRE_Match object; span=(2, 4), match='**'>
>>> re.match('[$()a-zA-Z0-9]+', '$(document)')    # $, (, )와 문자, 숫자가 들어있는지 판단
<_sre.SRE_Match object; span=(0, 11), match='$(document)'>
```

43.2 범위 판단하기

» 특수 문자 판단하기

- 단순히 숫자인지 문자인지 판단할 때는 \d, \D, \w, \W를 사용하면 편리함

- \d: [0-9]와 같음. 모든 숫자
- \D: [^0-9]와 같음. 숫자를 제외한 모든 문자
- \w: [a-zA-Z0-9_]와 같음. 영문 대소문자, 숫자, 밑줄 문자
- \W: [^a-zA-Z0-9_]와 같음. 영문 대소문자, 숫자, 밑줄 문자를 제외한 모든 문자

```
>>> re.match('\d+', '1234')          # 모든 숫자이므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 4), match='1234'>
>>> re.match('\D+', 'Hello')         # 숫자를 제외한 모든 문자이므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> re.match('\w+', 'Hello_1234')    # 영문 대소문자, 숫자, 밑줄 문자이므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 10), match='Hello_1234'>
>>> re.match('\W+', '(:)')    # 영문 대소문자, 숫자, 밑줄문자를 제외한 모든 문자이므로 패턴에 매칭됨
<_sre.SRE_Match object; span=(0, 3), match='(:)'>
```

43.2 범위 판단하기

» 공백 처리하기

- 공백은 ' '처럼 공백 문자를 넣어도 되고, \s 또는 \S로 표현할 수도 있음

- \s: [\t\n\r\f\v]와 같음. 공백(스페이스), \t(탭) \n(새 줄, 라인 피드), \r(캐리지 리턴), \f(폼피드), \v(수직 탭)을 포함
- \S: [^ \t\n\r\f\v]와 같음. 공백을 제외하고 \t, \n, \r, \f, \v만 포함

```
>>> re.match('[a-zA-Z0-9 ]+', 'Hello 1234')     # ' '로 공백 표현
<_sre.SRE_Match object; span=(0, 10), match='Hello 1234'>
>>> re.match('[a-zA-Z0-9\s]+', 'Hello 1234')    # \s로 공백 표현
<_sre.SRE_Match object; span=(0, 10), match='Hello 1234'>
```

43.3 그룹 사용하기

» 그룹 사용하기

- 정규표현식 그룹은 해당 그룹과 일치하는 문자열을 얻어올 때 사용함

- **(정규표현식) (정규표현식)**

- 다음은 공백으로 구분된 숫자를 두 그룹으로 나누어서 찾은 뒤 각 그룹에 해당하는 문자열(숫자)을 가져옴

- **매치객체.group(그룹숫자)**

```
>>> m = re.match('([0-9]+) ([0-9]+)', '10 295')
>>> m.group(1)    # 첫 번째 그룹(그룹 1)에 매칭된 문자열을 반환
'10'
>>> m.group(2)    # 두 번째 그룹(그룹 2)에 매칭된 문자열을 반환
'295'
>>> m.group()     # 매칭된 문자열을 한꺼번에 반환
'10 295'
>>> m.group(0)    # 매칭된 문자열을 한꺼번에 반환
'10 295'
```

43.3 그룹 사용하기

» 그룹 사용하기

- 매치객체.groups()

```
>>> m.groups()    # 각 그룹에 해당하는 문자열을 튜플 형태로 반환
('10', '295')
```

- 그룹 개수가 많아지면 숫자로 그룹을 구분하기가 힘들어짐
- 그룹에 이름을 지으면 편리함

- (?P<이름>정규표현식)

- 다음은 함수를 호출하는 코드 print(1234)에서 함수의 이름 print와 인수 1234를 추출함

- 매치객체.group('그룹이름')

```
>>> m = re.match('(?P<func>[a-zA-Z_][a-zA-Z0-9_]+)\((?P<arg>\w+)\)', 'print(1234)')
>>> m.group('func')    # 그룹 이름으로 매칭된 문자열 출력
'print'
>>> m.group('arg')     # 그룹 이름으로 매칭된 문자열 출력
'1234'
```

43.3 그룹 사용하기

» 패턴에 매칭되는 모든 문자열 가져오기

- 그룹 지정 없이 패턴에 매칭되는 모든 문자열을 가져오려면 findall 함수를 사용하며 매칭된 문자열을 리스트로 반환함

```
re.findall('패턴', '문자열')
```

- 다음은 문자열에서 숫자만 가져옴

```
>>> re.findall('[0-9]+', '1 2 Fizz 4 Buzz Fizz 7 8')
['1', '2', '4', '7', '8']
```

43.3 그룹 사용하기

» 문자열 바꾸기

- 문자열을 바꿀 때는 sub 함수를 사용하며 패턴, 바꿀 문자열, 문자열, 바꿀 횟수를 넣어줌
- 바꿀 횟수를 넣으면 지정된 횟수만큼 바꾸며 바꿀 횟수를 생략하면 찾은 문자열을 모두 바꿈

• re.sub('패턴', '바꿀문자열', '문자열', 바꿀횟수)

```
>>> re.sub('apple|orange', 'fruit', 'apple box orange tree')    # apple 또는 orange를 fruit로 바꿈
'fruit box fruit tree'
```

```
>>> re.sub('[0-9]+', 'n', '1 2 Fizz 4 Buzz Fizz 7 8')    # 숫자만 찾아서 n으로 바꿈
'n n Fizz n Buzz Fizz n n'
```

43.3 그룹 사용하기

» 문자열 바꾸기

- sub 함수는 바꿀 문자열 대신 교체 함수를 지정할 수도 있음
- 교체 함수는 매개변수로 매치 객체를 받으며 바꿀 결과를 문자열로 반환하면 됨
- 다음은 문자열에서 숫자를 찾은 뒤 숫자를 10배로 만듦

- 교체함수(매치객체)
- re.sub('패턴', 교체함수, '문자열', 바꿀횟수)

```
>>> def multiple10(m):          # 매개변수로 매치 객체를 받음
...     n = int(m.group())      # 매칭된 문자열을 가져와서 정수로 변환
...     return str(n * 10)      # 숫자에 10을 곱한 뒤 문자열로 변환해서 반환
...
>>> re.sub('[0-9]+', multiple10, '1 2 Fizz 4 Buzz Fizz 7 8')
'10 20 Fizz 40 Buzz Fizz 70 80'
```

- 교체 함수의 내용이 간단하다면 다음과 같이 람다 표현식을 만들어서 넣어도 됨

```
>>> re.sub('[0-9]+', lambda m: str(int(m.group()) * 10), '1 2 Fizz 4 Buzz Fizz 7 8')
'10 20 Fizz 40 Buzz Fizz 70 80'
```

43.3 그룹 사용하기

» 찾은 문자열을 결과에 다시 사용하기

- 정규표현식을 그룹으로 묶음
- 바꿀 문자열에서 \\숫자 형식으로 매칭된 문자열을 가져와서 사용할 수 있음

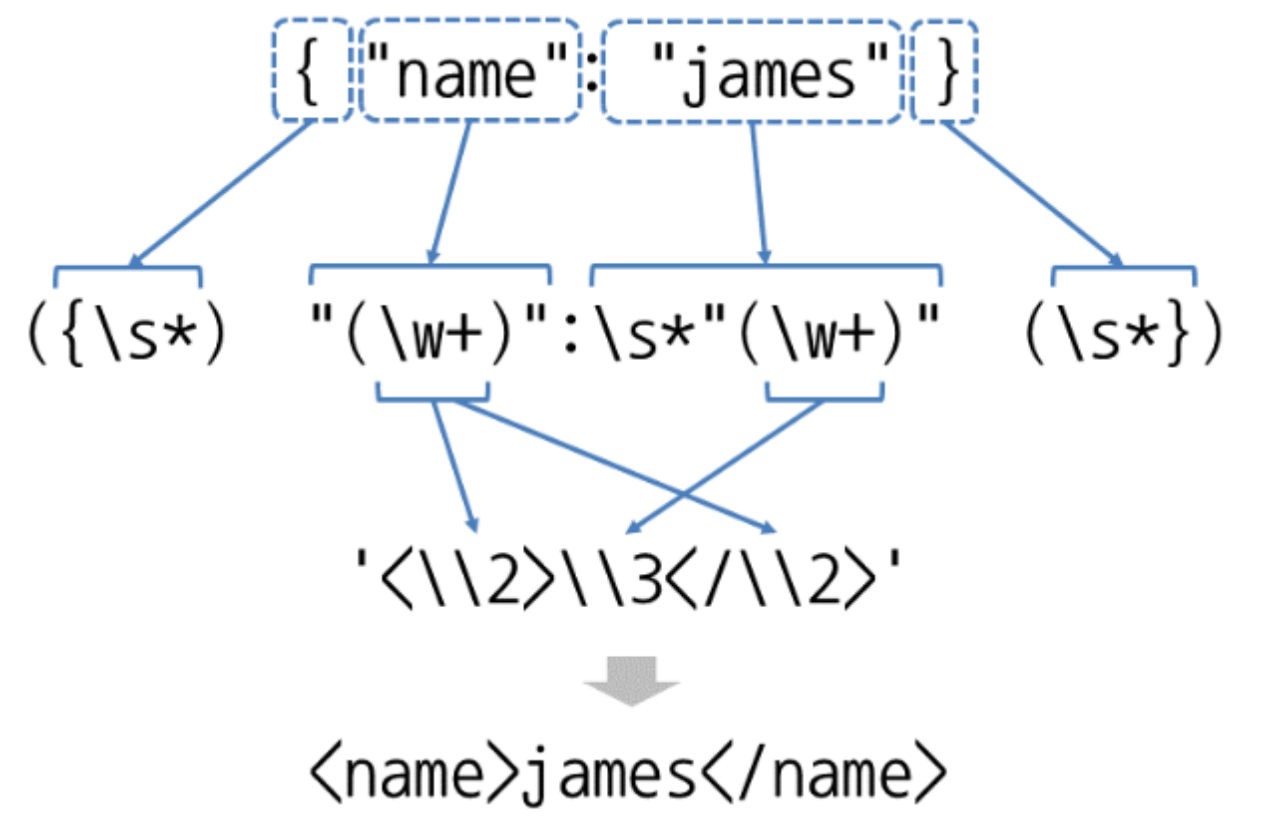
• \\숫자

```
>>> re.sub('([a-z]+) ([0-9]+)', '\\2 \\1 \\2 \\1', 'hello 1234')    # 그룹 2, 1, 2, 1 순으로 바꿈
'1234 hello 1234 hello'
```

```
>>> re.sub('({\s*)"(\w+)":\s*"(\w+)"(\s*})', '<\\2>\\3</\\2>', '{ "name": "james" }')
'<name>james</name>'
```

43.3 그룹 사용하기

▼ 그림 정규 표현식으로 찾은 문자열을 사용



43.3 그룹 사용하기

» 찾은 문자열을 결과에 다시 사용하기

- 만약 그룹에 이름을 지었다면 \\g〈이름〉 형식으로 매칭된 문자열을 가져올 수 있음(\\g〈숫자〉 형식으로 숫자를 지정해도 됨)

- \\g〈이름〉
- \\g〈숫자〉

```
>>> re.sub('({\s*)"(?P<key>\w+)":\s*"(?P<value>\w+)"(\s*})', '<\\g<key>>\\g<value></\\g<key>>', '{ "name": "jam
es" }')
'<name>james</name>'
```