스터디 7회차

파이썬

정규 표현식

정규표현식은 복잡한 문자열을 처리할 때 사용하는 기법이다.

```
주민등록번호를 포함하고 있는 텍스트가 있다. 이 텍스트에 포함된 모든 주민등록번호의 뒷자리를 *문자로 변경해보자
```

다음과 같은 문제가 주어졌을때

1. 전체 텍스트를 공백 문자로 나눈다.(split)
2. 나뉜 단어가 주민등록 형식인지 조사.
3. 단어가 주민등록번호 형식이라면 뒷자리를 *로 변환
4. 나뉜 단어를 다시 조합

이렇게 코드를 구현한다.

```
data = """
park 800905-1049118
kim  700905-1059119
"""

result = []
for line data.split("\n"):
    word_result = []
    for word in line.split(" "): #공백 문자마다 나누기
 if len(word) == 14 and word[:6].isdigit() and word[7:].isdigit():  # word의 길이가 14이고 6번째 전까지 숫자          word = word[:6] + "-" + "*******"
리고 7번째 이후부터도 숫자이면.
        word_result.append(word)    # 결괏값 word를 result에 넣는다.
    result.append(" ".join(word_result))    #나눈 단어들을 조립
print("\n".join(result))
```

이것의 결괏값은

```
결과값:
park 800905-*******
kim  700905-*******
```

정규 표현식을 사용하면 코드를 간단하게 할 수 있다.

정규표현식의 기초, 메타문자

정규 표현식에 사용하는 메타 문자는 다음과 같다

```
. ^ $ * + ? { } [ ] \ | ( )  #메타문자: 원래 그 문자가 가진 뜻이 아닌 다른 뜻으로 사용한는 문자
```

정규 표현식에서 위 문자들은 특별한 의미를 가진다.

문자 클래스[ ]

우리가 가장 먼저 살펴볼 메타 문자는 바로 문자 클래스(character class)인 [ ]이다. 문자 클래스로 만들어진 정규식은 `"[ ] 사이의 문자들과 매치"`라는 의미를 갖는다.

> ※ 문자 클래스를 만드는 메타 문자인 [ ] 사이에는 어떤 문자도 들어갈 수 있다.

즉 정규 표현식이 `[abc]`라면 이 표현식의 의미는 `"a, b, c 중 한 개의 문자와 매치"`를 뜻한다. 이해를 돕기 위해 문자열 "a", "before", "dude"가 정규식 [abc]와 어떻게 매치되는지 살펴보자.

[abc]

문자열	매치 여부	설명
a	Yes	"a"는 정규식과 일치하는 문자"a"가 있으므로 매치
before	Yes	"before"는 정규식과 일치하는 문자인 "b"가 있으므로 매치
dude	No	"dude"는 정규식과 일치하는 문자인 a, b, c 중 어느 하나도 포함하고 있지 않으므로 매치되지 않음

[ ] 안의 두 문자 사이에 하이픈(-)을 사용하면 두 문자 사이의 범위(From - To)를 의미한다. 예를 들어 [a-c]라는 정규 표현식은 [abc]와 동일하고 [0-5]는 [012345]와 동일하다.

다음은 하이픈(-)을 사용한 문자 클래스의 사용 예이다.

- [a-zA-Z] : 알파벳 모두
- [0-9] : 숫자

문자 클래스([ ]) 안에는 어떤 문자나 메타 문자도 사용할수 있지만 주의해야 할 메타 문자가 1가지 있다. 그것은 바로 `^`인데, 문자 클래스 안에 `^` 메타 문자를 사용할 경우에는 반대(not)라는 의미를 갖는다. 예를 들어 `[^0-9]`라는 정규 표현식은 숫자가 아닌 문자만 매치된다.

정규 표현식	설명
\d	숫자와 매치[0-9]와 동일
\D	숫자가 아닌것과 매치 `[^0-9]`
\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)와 매치, `[a-zA-Z0-9_]`와 동일한 표현식이다

Dot(.)

정규 표현식의 Dot(.) 메타 문자는 줄바꿈 문자인 `\n`을 제외한 모든 문자와 매치됨을 의미한다.

```
a.b     #a와 b 사이에 줄바꿈 문자를 제외한 어떤 문자가 들어가도 모두 매치
```

위 정규식의 의미는 a와 b 사이에 어떤 문자가 들어가도 모두 매치된다는 의미이다.

문자열	매치여부	설명
aab	o	"aab"는 가운데 문자 "a"가 모든 문자를 의미하는 `.`과 일치하므로 정규식과 매치된다."
a0b	o	"a0b"는 가운데 문자 "0"가 모든 문자를 의미하는 `.`과 일치하므로 정규식과 매치된다."
abc	x	"abc"는 "a"문자와 "b"문자 사이에 어떤 문자라도 하나는있어야 하는 이 정규식과 일치하지 않으므로 매치되지 않는다."

반복(*)

다음 정규식을 보면

```
ca*t
```

이 정규식에는 반복을 나타내는 *메타 문자가 쓰였다. 바로 앞에 있는 a가 0부터 무한대까지 반복될 수 있다는 의미이다.

ca*t

문자열	매치여부	설명
ct	o	a가 0번 반복되어매치
cat	o	a가 1번 반복되어 매치
caaat	o	a가 0번 이상 반복되어 매치

반복+

반복을 나타내는 또다른 매치문자+ 위에것이랑 무엇이 다른가

+는 최소 1번이상 반복될 때 사용되는게 차이

ca+t

문자열	매치여부	설명
ct	x	a가 0번 반복되어 매치x
cat	o	a가 1번 반복되어 매치
caaat	o	a가 0번 이상 반복되어 매치

반복({m,n},?)

반복 횟수를 제한하고 싶을때 사용

{ }메타 문자를 사용하면 반복 횟수를 고정할 수 있다

3회만, 혹은 1회부터 3회까지만 등등

```
1. {m}

ca{2}t     #a가 두번 반복되면 매치
cat은 a가 1번 반복되므로 매치x
```

```
2. {m,n}

ca{2,5}t   #a가 2~5번 반복되면 매치
caat 매치
caaaaat 매치
```

```
3. ?

반복은 아니지만 비슷한 개념으로 ?가 있다. 메타문자가 의미하는건 {0,1}을 의미한다.

ab?c    #b가 0~1번 사용되면 매치
ac      b가 0번 사용되어 매치
abc     매치
```

*,+,?는 모두 {m,n} 형태로 고쳐쓰는것이 가능하지만 가급적 표현이 쉬운 *,+,? 를 권장한다.

정규 표현식을 지원하는 re 모듈

파이썬은 정규 표현식을 지원하기 위해 re(regular expression)모듈을 제공한다 re 모듈은 파이썬을 설치할 때 자동으로 설치되는 기본 라이브러리이다. 사용 방법은

```
import re
p = re.compile('ab*')   #ab*(b가 0번 이상 반복)
```

re.complie 을 사용하여 정규표현식을 컴파일한다. 위 예에서는 ab*(b가 0번 이상 반복)를 컴파일 한것

re.complie의 결과로 돌려주는 객체 p를 사용하여 그 이후 작업을 수행할 수 있다.

정규식을 사용한 문자열 검색

메서드	목적
match()	문자열의 처음부터 정규식과 매치되는지 조사
search()	문자열 전체를 검색하여 정규식과 매치되는지 조사
findall()	정규식과 매치되는 모든 문자열(substring)을 리스트로 돌려준다.
finditer()	정규식과 매치되는 모든 문자열(substring)을 반복가능한 객체로 돌려준다.

match, search는 정규식과 매치될때 match객체를 돌려주고 매치되지 않을때 none을 돌려준다.

우선 예시

```
import re
p = re.complie('[a-z]+')     `#[a-z]+ 사이의 문자들과 매치
```

1. match : 문자열의 처음부터 정규식과 매치되는지 조사 위 예시에 match 메서드 수행하면

```
>>> m = p.match("python")
>>> print(m)
<re.Match object; span=(0,6), match='python'>   #match 객체를 돌려줌
```

python문자열은 a-z사이의 문자들로 구성되어 [a-z]+정규식에 부합한다. match 객체를 돌려줌

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

3 python 문자는 처음에 나오는 문자3이 정규식을 만족하지 않으므로 None을 돌려준다

match의 결과로 match 객체 또는 None을 돌려주기 때문에 파이썬 정규식 프로그램은 보통 다음과 같은 흐름으로 작성한다.

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

즉 match의 결괏값이 있을 때만 수행하고싶으시다는거지~

2 search

컴파일된 객체p를 가지고 이번에는 search메서드를 수행

```
>>> m = p.search("python")
>>> print(m)
<re.Match object; span=(0,6), match='python'>
```

match 메서드하고 동일하게 매치된다. 그렇다면 차이점은?

```
>>> m = p.search("3 python")
>>> print(m)
<re.Match object; span=(2,8), match='python'>
```

3 python 문자열의 첫 번째 문자는 3이지만 search는 문자열의 처음부터 검색하는 것이 아니라 문자열 전체를 검색하기 때문에 3 이후의 python 문자열과 매치된다. 즉 match 쓰면 앞에서 틀리면 그냥 none을 뱉는데 search는 조금 더 성의있게 전체를 훑고 매치 조사한다.

이렇듯 match 메서드와 search 메서드는 문자열의 처음부터 검색할지의 여부에 따라 다르게 사용해야 한다.

3. findall

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

"life is too short" 문자열의 'life', 'is', 'too', 'short' 단어를 각각 `[a-z]+` 정규식과 매치해서 리스트로 돌려준다

4.finditer

```
>>> result = p.finditer("life is too short")
>>> print(result)
<callable_iterator object at 0x01F5E390>
>>> for r in result: print(r)
...
<re.Match object; span=(0,4), match='life'>
<re.Match object; span=(5,7), match='is'>
<re.Match object; span=(8,11), match='too'>
<re.Match object; span=(12,17), match='short'>
```

finditer는 findall과 동일하지만 그 결과로 리스트 아니고 `반복 가능한 객체(iterator object)`를 돌려준다. 반복 가능한 객체가 포함하는 각각의 요소는 match 객체이다.

match 객체의 매서드

자, 이제 match 메서드와 search 메서드를 수행한 결과로 돌려준 match 객체에 대해 알아보자. 앞에서 정규식을 사용한 문자열 검색을 수행하면서 아마도 다음과 같은 궁금증이 생겼을 것이다.

- 어떤 문자열이 매치되었는가?
- 매치된 문자열의 인덱스는 어디서부터 어디까지인가?

match 객체의 메서드를 사용하면 이 같은 궁금증을 해결할 수 있다. 다음 표를 보자.

메서드	목적
group()	매치된 문자열을 돌려준다
start()	매치된 문자열의 시작 위치를 돌려준다
end()	매치된 문자열의 끝 위치를 돌려준다
span()	매치된 문자열의 (시작, 끝)에 해당되는 튜플을 돌려준다.

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

예상한 대로 결괏값이 출력되는 것을 확인할 수 있다. match 메서드를 수행한 결과로 돌려준 match 객체의 start()의 결괏값은 항상 0일 수밖에 없다. 왜냐하면 match 메서드는 항상 문자열의 시작부터 조사하기 때문이다.

만약 search 메서드를 사용했다면 start() 값은 다음과 같이 다르게 나올 것이다.

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

3 다음에 python부터 매치하기 때문

모듈 단위로 수행하기

```
지금까지 우리는 re.compile을 사용하여 컴파일된 패턴 객체로 그 이후의 작업을 수행했다. re
모듈은 이것을 좀 축약한 형태로 사용할 수 있는 방법을 제공한다. 다음 예를 보자.

>>> p = re.compile('[a-z]+')
>>> m = p.match("python")
위 코드가 축약된 형태는 다음과 같다.

>>> m = re.match('[a-z]+', "python")
위 예처럼 사용하면 컴파일과 match 메서드를 한 번에 수행할 수 있다. 보통 한 번 만든 패턴 객
체를 여러번 사용해야 할 때는 이 방법보다 re.compile을 사용하는 것이 편하다.
```

컴파일 옵션

정규식을 컴파일 할 때 다음 옵션을 사용할 수 있다.

옵션 이름	약어	설명
DOTALL	S	dot 문자가 줄바꿈 문자를 포함하여 모든 문자와 매치한다
IGNORECASE	I	대 소문자 관계 없이 매치
MULTILINE	M	여러 줄과 매치(^,$ 메타 문자의 사용과 관계있음)
VERBOSE	X	verbose 모드를 사용한다.(정규식을 보기 편하게 만들 수도 있고 주석 등을 사용 가능)

옵션을 사용할 때는 `re.DOTALL`처럼 전체 옵션 이름을 써도 되고 `re.S`처럼 약어를 써도 된다.

DOTALL, S

`.` 메타 문자는 줄바꿈 문자(`\n`)를 제외한 모든 문자와 매치되는 규칙이 있다. 만약 `\n` 문자도 포함하여 매치하고 싶다면 `re.DOTALL` 또는 `re.S` 옵션을 사용해 정규식을 컴파일하면 된다.

다음 예를 보자.

```
>>> import re
>>> p = re.compile('a.b')
>>> m = p.match('a\nb')
>>> print(m)
None
```

정규식이 `a.b`인 경우 문자열 `a\nb`는 매치되지 않음을 알 수 있다. 왜냐하면 `\n`은 `.` 메타 문자와 매치되지 않기 때문이다. `\n` 문자와도 매치되게 하려면 다음과 같이 `re.DOTALL` 옵션을 사용해야 한다.

```
>>> p = re.compile('a.b', re.DOTALL)
>>> m = p.match('a\nb')
>>> print(m)
<re.Match object; span=(0,3), match='a\nb'>
```

보통 `re.DOTALL` 옵션은 여러 줄로 이루어진 문자열에서 `\n`에 상관없이 검색할 때 많이 사용한다.

IGNORECASE, I

`re.IGNORECASE` 또는 `re.I` 옵션은 대소문자 구별 없이 매치를 수행할 때 사용하는 옵션이다. 다음 예제를 보자.

```
>>> p = re.compile('[a-z]', re.I)
>>> p.match('python')
<re.Match object; span=(0,1), match='p'>
>>> p.match('Python')   #P가 대문자
<re.Match object; span=(0,1), match='P'>
>>> p.match('PYTHON')   #전부 대문자
<re.Match object; span=(0,1), match='P'>
```

`[a-z]` 정규식은 소문자만을 의미하지만 re.I 옵션으로 대소문자 구별 없이 매치된다.

MULTILINE, M

`re.MULTILINE` 또는 `re.M` 옵션은 조금 후에 설명할 메타 문자인 `^`, `$`와 연관된 옵션이다. 이 메타 문자에 대해 간단히 설명하자면 `^`는 문자열의 처음을 의미하고, `$`는 문자열의 마지막을 의미한다. 예를 들어 정규식이 `^python`인 경우 문자열의 처음은 항상 python으로 시작해야 매치되고, 만약 정규식이 `python$`이라면 문자열의 마지막은 항상 python으로 끝나야 매치된다는 의미이다.

다음 예를 보자.

```
import re
p = re.compile("^python\s\w+")

data = """python one
life is too short
python two
you need python
python three"""

print(p.findall(data))
```

정규식 `^python\s\w+`은 python이라는 문자열로 시작하고 그 뒤에 whitespace, 그 뒤에 단어가 와야 한다는 의미이다. 검색할 문자열 data는 여러 줄로 이루어져 있다.

이 스크립트를 실행하면 다음과 같은 결과를 돌려준다.

```
['python one']
```

`^` 메타 문자에 의해 python이라는 문자열을 사용한 첫 번째 줄만 매치된 것이다.

하지만 `^` 메타 문자를 문자열 전체의 처음이 아니라 각 라인의 처음으로 인식시키고 싶은 경우도 있을 것이다. 이럴 때 사용할 수 있는 옵션이 바로 `re.MULTILINE` 또는 `re.M`이다. 위 코드를 다음과 같이 수정해 보자.

```
import re
p = re.compile("^python\s\w+", re.MULTILINE)

data = """python one
life is too short
python two
you need python
python three"""

print(p.findall(data))
```

`re.MULTILINE` 옵션으로 인해 `^` 메타 문자가 문자열 전체가 아닌 각 줄의 처음이라는 의미를 갖게 되었다. 이 스크립트를 실행하면 다음과 같은 결과가 출력된다.

```
['python one', 'python two', 'python three']
```

즉 `re.MULTILINE` 옵션은 `^`, `$` 메타 문자를 문자열의 각 줄마다 적용해 주는 것이다.

VERBOSE, X

지금껏 알아본 정규식은 매우 간단하지만 정규식 전문가들이 만든 정규식을 보면 거의 암호수준이다. 정규식을 이해하려면 하나하나 조심스럽게 뜯어보아야만 한다. 이렇게 이해하기 어려운 정규식을 주석 또는 줄 단위로 구분할 수 있다면 얼마나 보기 좋고 이해하기 쉬울까? 방법이 있다. 바로 `re.VERBOSE` 또는 `re.X` 옵션을 사용하면 된다.

다음 예를 보자.

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

다음과 같이 보기 쉽게 끊어보면

```
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)
```

첫 번째와 두 번째 예를 비교해 보면 컴파일된 패턴 객체인 charref는 모두 동일한 역할을 한다. 하지만 정규식이 복잡할 경우 두 번째처럼 주석을 적고 여러 줄로 표현하는 것이 훨씬 가독성이 좋다는 것을 알 수 있다.

`re.VERBOSE` 옵션을 사용하면 문자열에 사용된 whitespace는 컴파일할 때 제거된다(단 [ ] 안에 사용한 whitespace는 제외). 그리고 줄 단위로 #기호를 사용하여 주석문을 작성할 수 있다.

백슬래시 문제

정규 표현식을 파이썬에서 사용할 때 혼란을 주는 요소가 한 가지 있는데, 바로 백슬래시(`\`)이다.

예를 들어 어떤 파일 안에 있는 `"\section"` 문자열을 찾기 위한 정규식을 만든다고 가정해 보자.

```
\section
```

이 정규식은 `\s` 문자가 whitespace로 해석되어 의도한 대로 매치가 이루어지지 않는다.

의도한 대로 매치하고 싶다면 다음과 같이 변경해야 한다.

```
\\section
```

즉 위 정규식에서 사용한 `\` 문자가 문자열 자체임을 알려 주기 위해 백슬래시 2개를 사용하여 이스케이프 처리를 해야 한다.

따라서 위 정규식을 컴파일하려면 다음과 같이 작성해야 한다.

```
>>> p = re.compile('\\section')
```

그런데 여기에서 또 하나의 문제가 발견된다. 위처럼 정규식을 만들어서 컴파일하면 실제 파이썬 정규식 엔진에는 파이썬 문자열 리터럴 규칙에 따라 `\\`이 `\`로 변경되어 `\section`이 전달된다.

> ※ 이 문제는 위와 같은 정규식을 파이썬에서 사용할 때만 발생한다(파이썬의 리터럴 규칙). 유닉스의 grep, vi 등에서는 이러한 문제가 없다.

결국 정규식 엔진에 `\\` 문자를 전달하려면 파이썬은 `\\\\`처럼 백슬래시를 4개나 사용해야 한다.

> ※ 정규식 엔진은 정규식을 해석하고 수행하는 모듈이다.

```
>>> p = re.compile('\\\\section')
```

이렇게 해야만 원하는 결과를 얻을 수 있다. 하지만 너무 복잡하지 않은가?

만약 위와 같이 `\`를 사용한 표현이 계속 반복되는 정규식이라면 너무 복잡해서 이해하기 쉽지않을 것이다. 이러한 문제로 인해 파이썬 정규식에는 Raw String 규칙이 생겨나게 되었다. 즉 컴파일해야 하는 정규식이 Raw String임을 알려 줄 수 있도록 파이썬 문법을 만든 것이다. 그 방법은 다음과 같다.

```
>>> p = re.compile(r'\\section')
```

위와 같이 정규식 문자열 앞에 r 문자를 삽입하면 이 정규식은 Raw String 규칙에 의하여 백슬래시 2개 대신 1개만 써도 2개를 쓴 것과 동일한 의미를 갖게 된다.

> ※ 만약 백슬래시를 사용하지 않는 정규식이라면 r의 유무에 상관없이 동일한 정규식이 될 것이다.

메타문자

아직 살펴보지 않은 메타 문자에 대해서 모두 살펴보자. 여기에서 다룰 메타 문자는 앞에서 살펴본 메타 문자와 성격이 조금 다르다. 앞에서 살펴본 `+`, `*`, `[]`, `{}` 등의 메타문자는 매치가 진행될 때 현재 매치되고 있는 문자열의 위치가 변경된다(보통 소비된다고 표현한다).

하지만 이와 달리 문자열을 소비시키지 않는 메타 문자도 있다. 이번에는 이런 문자열 소비가 없는 (zerowidth assertions) 메타 문자에 대해 살펴보자.

|

`|` 메타 문자는 or과 동일한 의미로 사용된다. `A|B`라는 정규식이 있다면 A 또는 B라는 의미가 된다.

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

^

`^` 메타 문자는 문자열의 맨 처음과 일치함을 의미한다. 앞에서 살펴본 컴파일 옵션 `re.MULTILINE`을 사용할 경우에는 여러 줄의 문자열일 때 각 줄의 처음과 일치하게 된다.

다음 예를 보자.

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

`^Life` 정규식은 Life 문자열이 처음에 온 경우에는 매치하지만 처음 위치가 아닌 경우에는 매치되지 않음을 알 수 있다.

$

`$` 메타 문자는 `^` 메타 문자와 반대의 경우이다. 즉 `$`는 문자열의 끝과 매치함을 의미한다.

다음 예를 보자.

```
>>> 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$` 정규식은 검색할 문자열이 short로 끝난 경우에는 매치되지만 그 이외의 경우에는 매치되지 않음을 알 수 있다.

> ※ `^` 또는 `$` 문자를 메타 문자가 아닌 문자 그 자체로 매치하고 싶은 경우에는 `\^`, `\$` 로 사용하면 된다.

\A

`\A` 는 문자열의 처음과 매치됨을 의미한다. `^` 메타 문자와 동일한 의미이지만 `re.MULTILINE` 옵션을 사용할 경우에는 다르게 해석된다. `re.MULTILINE` 옵션을 사용할 경우 `^` 은 각 줄의 문자열의 처음과 매치되지만 `\A` 는 줄과 상관없이 전체 문자열의 처음하고만 매치된다.

\Z

`\Z` 는 문자열의 끝과 매치됨을 의미한다. 이것 역시 `\A` 와 동일하게 `re.MULTILINE` 옵션을 사용할 경우 `$` 메타 문자와는 달리 전체 문자열의 끝과 매치된다.

\b

`\b` 는 단어 구분자(Word boundary)이다. 보통 단어는 whitespace에 의해 구분된다.

다음 예를 보자.

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

`\bclass\b` 정규식은 앞뒤가 whitespace로 구분된 class라는 단어와 매치됨을 의미한다. 따라서 no class at all의 class라는 단어와 매치됨을 확인할 수 있다.

```
>>> print(p.search('the declassified algorithm'))
None
```

위 예의 the declassified algorithm 문자열 안에도 class 문자열이 포함되어 있긴 하지만 whitespace로 구분된 단어가 아니므로 매치되지 않는다.

```
>>> print(p.search('one subclass is'))
None
```

subclass 문자열 역시 class 앞에 sub 문자열이 더해져 있으므로 매치되지 않음을 알 수 있다.

`\b` 메타 문자를 사용할 때 주의해야 할 점이 있다. `\b` 는 파이썬 리터럴 규칙에 의하면 백스페이스(BackSpace)를 의미하므로 백스페이스가 아닌 단어 구분자임을 알려 주기 위해 `r'\bclass\b'` 처럼 Raw string임을 알려주는 기호 r을 반드시 붙여 주어야 한다.

\B

`\B` 메타 문자는 `\b` 메타 문자와 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'))
<re.Match object; span=(6, 11), match='class'>
>>> print(p.search('one subclass is'))
None
```

class 단어의 앞뒤에 whitespace가 하나라도 있는 경우에는 매치가 안 되는 것을 확인할 수 있다.

그루핑

ABC 문자열이 계속해서 반복되는지 조사하는 정규식을 작성하고 싶다고 하자. 어떻게 해야할까? 지금까지 공부한 내용으로는 위 정규식을 작성할 수 없다. 이럴 때 필요한 것이 바로 그루핑(Grouping) 이다.

위 경우는 다음처럼 그루핑을 사용하여 작성할 수 있다.

```
(ABC)+
```

그룹을 만들어 주는 메타 문자는 바로 `( )` 이다.

```
>>> p = re.compile('(ABC)+')
>>> m = p.search('ABCABCABC OK?')
>>> print(m)
<re.Match object; span=(0, 9), match='ABCABCABC'>
>>> print(m.group())
ABCABCABC
```

다음 예를 보자.

\w : [a-zA-Z0-9_]와 동일한 기능

\s : 공백문자와 동일한 기능

\d : [0-9]와 동일한 기능

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

`\w+\s+\d+[-]\d+[-]\d+` 은 `이름 + " " + 전화번호` 형태의 문자열을 찾는 정규식이다. 그런데 이렇게 매치된 문자열 중에서 이름만 뽑아내고 싶다면 어떻게 해야 할까?

보통 반복되는 문자열을 찾을 때 그룹을 사용하는데, 그룹을 사용하는 보다 큰 이유는 위에서 볼 수 있듯이 매치된 문자열 중에서 특정 부분의 문자열만 뽑아내기 위해서인 경우가 더 많다.

위 예에서 만약 '이름' 부분만 뽑아내려 한다면 다음과 같이 할 수 있다.

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

이름에 해당하는 `\w+` 부분을 그룹 `(\w+)` 으로 만들면 match 객체의 group(인덱스) 메서드를 사용하여 그루핑된 부분의 문자열만 뽑아낼 수 있다. group 메서드의 인덱스는 다음과 같은 의미를 갖는다.

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

다음 예제를 계속해서 보자.

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

이번에는 전화번호 부분을 추가로 그룹 `(\d+[-]\d+[-]\d+)`로 만들었다. 이렇게 하면 `group(2)`처럼 사용하여 전화번호만 뽑아낼 수 있다.

만약 전화번호 중에서 국번만 뽑아내고 싶으면 어떻게 해야 할까? 다음과 같이 국번 부분을 또 그루핑하면 된다.

```
>>> p = re.compile(r"(\w+)\s+((\d+)[-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(3))
010
```

위 예에서 볼 수 있듯이 `(\w+)\s+((\d+)[-]\d+[-]\d+)`처럼 그룹을 중첩되게 사용하는 것도 가능하다. 그룹이 중첩되어 있는 경우는 바깥쪽부터 시작하여 안쪽으로 들어갈수록 인덱스가 증가한다.

그루핑된 문자열 재참조하기

그룹의 또 하나 좋은 점은 한 번 그루핑한 문자열을 재참조(Backreferences)할 수 있다는 점이다. 다음 예를 보자.

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

정규식 `(\b\w+)\s+\1`은 `(그룹) + " " + 그룹과 동일한 단어`와 매치됨을 의미한다. 이렇게 정규식을 만들게 되면 2개의 동일한 단어를 연속적으로 사용해야만 매치된다. 이것을 가능하게 해주는 것이 바로 재참조 메타 문자인 `\1`이다. `\1`은 정규식의 그룹 중 첫 번째 그룹을 가리킨다.

> ※ 두 번째 그룹을 참조하려면 `\2`를 사용하면 된다.

그루핑된 문자열에 이름 붙이기

정규식 안에 그룹이 무척 많아진다고 가정해 보자. 예를 들어 정규식 안에 그룹이 10개 이상만 되어도 매우 혼란스러울 것이다. 거기에 더해 정규식이 수정되면서 그룹이 추가, 삭제되면 그 그룹을 인덱스로 참조한 프로그램도 모두 변경해 주어야 하는 위험도 갖게 된다.

만약 그룹을 인덱스가 아닌 이름(Named Groups)으로 참조할 수 있다면 어떨까? 그렇다면 이런 문제에서 해방되지 않을까?

이러한 이유로 정규식은 그룹을 만들 때 그룹 이름을 지정할 수 있게 했다. 그 방법은 다음과 같다.

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

위 정규식은 앞에서 본 이름과 전화번호를 추출하는 정규식이다. 기존과 달라진 부분은 다음과 같다.

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

대단히 복잡해진 것처럼 보이지만 `(\w+)` 라는 그룹에 name이라는 이름을 붙인 것에 불과하다. 여기에서 사용한 `(?...)` 표현식은 정규 표현식의 확장 구문이다. 이 확장 구문을 사용하기 시작하면 가독성이 상당히 떨어지긴 하지만 반면에 강력함을 갖게 된다.

그룹에 이름을 지어 주려면 다음과 같은 확장 구문을 사용해야 한다.

```
(?P<그룹명>...)
```

그룹에 이름을 지정하고 참조하는 다음 예를 보자.

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

위 예에서 볼 수 있듯이 name이라는 그룹 이름으로 참조할 수 있다.

그룹 이름을 사용하면 정규식 안에서 재참조하는 것도 가능하다.

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

위 예에서 볼 수 있듯이 재참조할 때에는 `(?P=그룹이름)` 이라는 확장 구문을 사용해야 한다.

전방 탐색

정규식에 막 입문한 사람들이 가장 어려워하는 것이 바로 전방 탐색(Lookahead Assertions) 확장 구문이다. 정규식 안에 이 확장 구문을 사용하면 순식간에 암호문처럼 알아보기 어렵게 바뀌기 때문이다. 하지만 이 전방 탐색이 꼭 필요한 경우가 있으며 매우 유용한 경우도 많으니 꼭 알아 두자.

다음 예를 보자.

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

정규식 `.+:` 과 일치하는 문자열로 http:를 돌려주었다. 만약 http:라는 검색 결과에서 :을 제외하고 출력하려면 어떻게 해야 할까? 위 예는 그나마 간단하지만 훨씬 복잡한 정규식이어서 그루핑은 추가로 할 수 없다는 조건까지 더해진다면 어떻게 해야 할까?

이럴 때 사용할 수 있는 것이 바로 전방 탐색이다. 전방 탐색에는 긍정(Positive)과 부정(Negative)의 2종류가 있고 다음과 같이 표현한다.

- 긍정형 전방 탐색(`(?=...)`) - `...` 에 해당되는 정규식과 매치되어야 하며 조건이 통과되어도 문자열이 소비되지 않는다.

- 부정형 전방 탐색(`(?!...)`) - `...`에 해당되는 정규식과 매치되지 않아야 하며 조건이 통과되어도 문자열이 소비되지 않는다.

긍정형 전방 탐색

긍정형 전방 탐색을 사용하면 http:의 결과를 http로 바꿀 수 있다. 다음 예를 보자.

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

정규식 중 `:`에 해당하는 부분에 긍정형 전방 탐색 기법을 적용하여 `(?=:)`으로 변경하였다. 이렇게 되면 기존 정규식과 검색에서는 동일한 효과를 발휘하지만 `:` 에 해당하는 문자열이 정규식 엔진에 의해 소비되지 않아(검색에는 포함되지만 검색 결과에는 제외됨) 검색 결과에서는 `:`이 제거된 후 돌려주는 효과가 있다.

자, 이번에는 다음 정규식을 보자.

```
.*[.].*$
```

이 정규식은 `파일 이름 + . + 확장자`를 나타내는 정규식이다. 이 정규식은 foo.bar, autoexec.bat, sendmail.cf 같은 형식의 파일과 매치될 것이다.

이 정규식에 확장자가 "bat인 파일은 제외해야 한다"는 조건을 추가해 보자. 가장 먼저 생각할 수 있는 정규식은 다음과 같다.

```
.*[.][^b].*$
```

이 정규식은 확장자가 b라는 문자로 시작하면 안 된다는 의미이다. 하지만 이 정규식은 foo.bar라는 파일마저 걸러 낸다. 정규식을 다음과 같이 수정해 보자.

```
.*[.]([^b]..|.[^a].|..[^t])$
```

이 정규식은 `|` 메타 문자를 사용하여 확장자의 첫 번째 문자가 b가 아니거나 두 번째 문자가 a가 아니거나 세 번째 문자가 t가 아닌 경우를 의미한다. 이 정규식에 의하여 foo.bar는 제외되지 않고 autoexec.bat은 제외되어 만족스러운 결과를 돌려준다. 하지만 이 정규식은 아쉽게도 sendmail.cf처럼 확장자의 문자 개수가 2개인 케이스를 포함하지 못하는 오동작을 하기 시작한다.

따라서 다음과 같이 바꾸어야 한다.

```
.*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$
```

확장자의 문자 개수가 2개여도 통과되는 정규식이 만들어졌다. 하지만 정규식은 점점 더 복잡해지고 이해하기 어려워진다.

그런데 여기에서 bat 파일말고 exe 파일도 제외하라는 조건이 추가로 생긴다면 어떻게 될까? 이 모든 조건을 만족하는 정규식을 구현하려면 패턴은 더욱더 복잡해질 것이다.

부정형 전방 탐색

이러한 상황의 구원 투수는 바로 부정형 전방 탐색이다. 위 예는 부정형 전방 탐색을 사용하면 다음과 같이 간단하게 처리된다.

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

확장자가 bat가 아닌 경우에만 통과된다는 의미이다. bat 문자열이 있는지 조사하는 과정에서 문자열이 소비되지 않으므로 bat가 아니라고 판단되면 그 이후 정규식 매치가 진행된다.

exe 역시 제외하라는 조건이 추가되더라도 다음과 같이 간단히 표현할 수 있다.

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

문자열 바꾸기

sub 메서드를 사용하면 정규식과 매치되는 부분을 다른 문자로 쉽게 바꿀 수 있다.

다음 예를 보자.

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

sub 메서드의 첫 번째 매개변수는 "바꿀 문자열(replacement)"이 되고, 두 번째 매개변수는 "대상 문자열"이 된다. 위 예에서 볼 수 있듯이 blue 또는 white 또는 red라는 문자열이 colour라는 문자열로 바뀌는 것을 확인할 수 있다.

그런데 딱 한 번만 바꾸고 싶은 경우도 있다. 이렇게 바꾸기 횟수를 제어하려면 다음과 같이 세 번째 매개변수로 count 값을 넘기면 된다.

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

처음 일치하는 blue만 colour라는 문자열로 한 번만 바꾸기가 실행되는 것을 알 수 있다.

[sub 메서드와 유사한 subn 메서드]

subn 역시 sub와 동일한 기능을 하지만 반환 결과를 튜플로 돌려준다는 차이가 있다. 돌려준 튜플의 첫 번째 요소는 변경된 문자열이고, 두 번째 요소는 바꾸기가 발생한 횟수이다.

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn( 'colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
```

sub 메서드 사용 시 참조 구문 사용하기

sub 메서드를 사용할 때 참조 구문을 사용할 수 있다. 다음 예를 보자.

```
>>> 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
```

위 예는 `이름 + 전화번호`의 문자열을 `전화번호 + 이름`으로 바꾸는 예이다. sub의 바꿀 문자열 부분에 `\g<그룹이름>`을 사용하면 정규식의 그룹 이름을 참조할 수 있게 된다.

다음과 같이 그룹 이름 대신 참조 번호를 사용해도 마찬가지 결과를 돌려준다.

```
>>> 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
```

sub 메서드의 매개변수로 함수 넣기

sub 메서드의 첫 번째 매개변수로 함수를 넣을 수도 있다. 다음 예를 보자.

```
>>> 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.'
```

hexrepl 함수는 match 객체(위에서 숫자에 매치되는)를 입력으로 받아 16진수로 변환하여 돌려주는 함수이다. sub의 첫 번째 매개변수로 함수를 사용할 경우 해당 함수의 첫 번째 매개변수에는 정규식과 매치된 match 객체가 입력된다. 그리고 매치되는 문자열은 함수의 반환 값으로 바뀌게 된다.

Greedy vs Non-Greedy

정규식에서 Greedy(탐욕스러운)란 어떤 의미일까? 다음 예제를 보자.

```
>>> 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>
```

`<.*>` 정규식의 매치 결과로 `<html>` 문자열을 돌려주기를 기대했을 것이다. 하지만 * 메타 문자는 매우 탐욕스러워서 매치할 수 있는 최대한의 문자열인 `<html><head><title>Title</title>` 문자열을 모두 소비해 버렸다. 어떻게 하면 이 탐욕스러움을 제한하고 `<html>` 문자열까지만 소비하도록 막을 수 있을까?

다음과 같이 non-greedy 문자인 `?`를 사용하면 `*`의 탐욕을 제한할 수 있다.

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

non-greedy 문자인 `?`는 `*?`, `+?`, `??`, `{m,n}?`와 같이 사용할 수 있다. 가능한 한 가장 최소한의 반복을 수행하도록 도와주는 역할을 한다.

통계

회귀분석

회귀분석이란?

통계학을 이용하여 변수들간의 연관성을 파악하고 변수들간의 영향이 미치는 정도를 파악할 수 있어서 결국 어떤 결과를 가져오는지, 그 규칙을 수식으로 제기 하기 위해

회귀선으로 돌아가서 회귀분석

단순히 차이가 있다고 알아내는 것을 넘어서 어떤 원인이 어떠한 결과를 가져오는지 확인 할 수 있는 방법

한 변수의 변화가 원인이 되어 다른 변수에 변화에 미치는 영향을 측정하는 방법

두 변수간의 인과관계를 분석할 때 사용

현재까지의 데이터로 미래 예측

독립변수와 종속변수

- 독립변수는 변수에 일어나는 현상을 설명하거나 변수가 원인이 되어 다른 변수에 영향을 주는 변수 x
- 종속변수는 연구모델에서 설명되거나 다른 변수로부터 영향을 받는 변수 y

회귀분석의 결과로 나온 독립변수와 종속변수간의 관계는 아주 정확하지는 않고 확률적이다. 즉 추정치와 예측치를 제공하며 수식으로 나타낼 수 있다. 명목척도와 서열척도로 측정된 경우에는 더미변수 회귀분석과 로지스틱 회귀분석을 이용하지만 책에서는 다루지 않는다.

회귀분석을 왜 하는가?

1. 독립변수와 종속변수의 관계 정도와 강도를 파악하기 위해. 상관분석에서 변수간의 연관관계를 확인했지만 회귀분석에서는 더욱 정확한 관계를 제시할 수 있고 연관성의 강도까지 파악 가능
2. 독립변수와 종속변수의 관계 양(+)인지 음(-)인지 파악하기 위해서이다. 회귀분석을 통해 변수간 기울기를 파악하여 양이나 음으로 구분해서 더욱 정확한 수치 제시가능
3. 독립변수와 종속변수의 관계를 파악했다면 그 크기의 정도가 어느정도인지 확인하기 위해

회귀분석의 가정

1. 독립변수와 종속변수가 나타내는 값의 분포는 선형
2. 수집된 데이터의 분산이 정규분포를 가정 : 표본이 많아지면 당연히 정규성을 띄겠지만 이런 정규성을 몰라도 회귀분석 하려면 정규분포라는 가정이 필요
3. 오차항의 평균은 0이다 : 산포하는 측정치를 대상으로 정확하게 평균을 구성하는 회귀선을 그으면 모든 잔차의 합은 0이 나와야 한다.

4. 독립변수는 사전에 주어진 고정변수이다 : 회귀분석에서 원인이 되는 변수는 정해져있다. 왜냐면 회귀분석은 독립변수의 투입에 따라 종속변수가 어떻게 변하는지 확인하는 과정. 독립변수를 변화시키면 안된다는것. 정해진 독립변수에 따른 종속변수의 변화를 보고싶은거니까.
5. 데이터는 독립적으로 추출되어야 한다 : 데이터들끼리 서로 영향이 있으면 독립변수와 종속변수의 관계가 오염될수도 있다.
6. 각 오차는 서로 독립적이어야한다. :5번과 같은 이유

단순회귀분석

단순회귀분석은 원인이 되는 요인을 단 하나로 제한해서 영향력을 판단. 하나의 영향력을 판단하고 다른 원인들로 확장시키면 파악이 더 편하니까.

단순회귀분석의 개념과 특징

단순회귀분석의 개념

독립변수 X가 종속변수 Y에 미치는 영향을 회귀식을 이용하여 분석하는 방법이다. 단순회귀분석에서 x와 y는 회귀식이라는 수학적인 방정식(1차방정식)으로 표현된다.

단순회귀분석의 특징

선형이다.

직선으로 표현된다. 산포도를 보고 선형 여부를 판단해야 한다.

찍힌 점들의 산포를 보고 선형이다. 선형이 아니다 라고 판단하는것은 연구자의 몫이지만

추세를 보았을때 X값이 증가하면 Y값이 증가하므로 선형이라고 판단할 수 있다.

회귀선의 절편과 기울기

X의 변화로 인해 Y가 변화하는게 1대 1로 짝을 이루므로 회귀식을 1차식으로 표현 가능하다.

인문/사회과학에서는 이렇게 단순하지 않고 고려해야할 변수가 많다. 원인이 하나가 아닌 경우가 대부분 그래서 정확한 식으로 도출하기가 어렵기에 당연히 직선으로 예쁘게 나오지 않고, 조건부적으로 평균 회귀선을 표현한다.

잔차

실제 데이터를 측정하면 독립변수에 따라 종속변수가 변화하는 정도가 다르게 나타나기도 한다. 회귀선에 완벽하게 일치하지 않는다는것

그래서 개별 측정치들간에 차이가 존재하는데 이것이 잔차이다

잔차는 회귀식에 반영한다.

잔차는 특정한 패턴이 없어야 한다. 잔차가 패턴이 있다면 회귀식에 변수 하나를 더 추가해야 하는데 그러면 회귀식이 더 복잡

최소자승법(최소제곱법)

회귀분석을 할 때 최소제곱법과 최대우도법이 가장 널리 사용된다

책에 나온 방법은 최소제곱법

측정된 표본에서 표본 회귀식을 구하는법

잔차를 제곱하여 모두 더한 제곱합이 최소가 되게 하는 함수를 구하면 이것이 바로 회귀선이다(최소자승법)

1. 잔차를 구한다
2. 잔차들의 제곱합을 구한다
3. 기울기와 상수항 구한다.

최대 우도법은 어떤 하나의 함수가 최대한 모수를 갖는 함수로 접근하도록 하는 방법

SSE 설명된 값의 제곱합

SSR 설명되지 않은 값의 제곱합

SST 전체 y의 값의 제곱합

y hat : 회귀선의 y값

단순 회귀식의 계수 도출

적합도 분석과 분산분석

회귀식이 얼마나 표본을 잘 나타내는지,

회귀선의 설명력은 R^2 로 나타낸다 (회귀계수)

R은 저번시간에 배운 상관계수다. 상관계수는 부호(방향, 음으로 상관인지 양으로 상관인지)가 있지만 회귀계수는 없다.

상관계수가 0이면 선형관계가 아닌것 다른 어떠한 관계가 있을수도

회귀계수가 0이란건 정말 쓸모없는 수치(상관계수가 0인데 왜 회귀분석?)

회귀선이 자료와 100퍼센트 일치하는 경우 설명력은 100%이다 (사실상 불가능)

보통은 0과 1 사이에 R^2이 있다.

회귀선이 하나도 표본을 설명하지 못하면 회귀계수는 0이다.

결정계수는 회귀 제곱합/총 제곱합

총 제곱합중에 얼마나 회귀 제곱합이 설명하는지 나타내므로 이러한 비로 결정계수를 구할 수 있다

유의성 검정과 회귀계수의 신뢰구간

회귀식의 유의성

f검정을 사용한다. f값과 임계치를 비교하여 임계치보다 f값이 크면 귀무 기각 대립 채택 분산비율 f는 평균 회귀제곱을 평균 오차제곱으로 나눈 값이다

귀무가설 : 회귀식이 유의하지 않다

대립가설 : 회귀식이 유의하다

회귀 계수의 유의성

각각 기울기와 상수항의 유의성을 확인하는것

실제 연구에서는 상수항의 값은 유의하지 않아도 최종 보고서에 사용한다.

기울기가 실제적으로 독립변수와 종속변수간의 인과관계를 설명하는데 훨씬 더 중요하기 때문

회귀분석은 주어진 데이터를 통해 독립변수가 한 단위 변화했을때 종속변수가 얼마나 변하는지 확인하는 분석 방법이므로 독립변수의 변화 이전에 주어진 상황은 큰 의미가 없다

이때 T검정을 사용!

표본의 회귀계수에 대한 분포를 확인해야 하는데 표본에 잔차에서 구한 분산을 활용한다. 모수를 알지 못하기 때문에 t분포를 이용한다

다만 표본이 많아지는경우 z분포의 값과 비슷해진다

기울기의 검정은

기울기가 0이다 : 귀무가설

기울기가 0이 아니다: 대립가설

상수항의 검정은

상수항이 0이다: 귀무가설

상수항이 0이 아니다: 대립가설

식을 통해 검정통계량 구하고

마찬가지로 식을 통해 신뢰구간 구할 수 있다.

여기서 자유도는 n-2 기울기와 상수항을 구하기 위해 관측값을 두개 썼기 때문이다.

엑셀님이 다 해주신다

다중회귀분석

단순회귀분석은 한 요인만 가지고 분석 한요인만 영향을 미치는것으로 가정

하지만 실생활에서는 다양한 요인이 얽혀있다.

다양한 독립변수들을 요인으로 투입하여 종속변수에 영향을 끼치는 요인을 분석하는것이 다중회귀분석

다중회귀분석의 개념

독립변수가 여러개일때 사용

스마트폰을 사용하는 소비자의 만족감에 영향을 주는 요소는?

다중회귀분석의 특징

독립변수마다 계수가 붙는다.

다중회귀분석에서도 개별 측정치들 사이의 차이인 잔차가 발생 할 수 있으며 이를 회귀식에 반영하면

기본적으로 회귀 분석은 독립변수가 1개 이상이며 종속변수는 1개로 정한다. 따라서 회귀분석은 종속변수가 하나인 경우만 실시 가능

종속변수가 여러개이면 종속변수마다 회귀분석을 실시해야 한다

다중회귀분석은 회귀계수를 계산 하는 방법은 독립변수가 늘었다는 것만 빼면 단순 회귀분석과 동일하게 최소제곱법을 사용한다.

이건 손으로 하기 매우 어렵습니다 ㅠㅠ 보통 행렬을 이용해서 식을 정리해서 합니다

다중 회귀분석의 가정

1. 독립변수는 서로 독립
2. 데이터의 분산이 정규분포
3. 오차항의 기댓값은 0이다
4. 잔차의 분산은 모두 등분산성을 가진다
5. 독립변수는 사전에 주어진 값

적합도 검정과 분산분석

단순회귀분석과 다르다

독립변수가 늘어나기때문에 회귀식의 종속변수에 대한 설명력은 단순보다 높아지고 그러므로 독립변수가 늘어난 만큼 R2이 변화하는 정도를 수정해야 한다. 변화 정도를 반영하면서 각 자유도로 나누어 불편추정량을 계산하면 수정된R^2을 계산 가능하다

SSR을 i 로 나누고

SSE를 n-i-1로 나눈다 (SST의 자유도는 제약이 1개니까 n-1이고 SSR의 자유도는 독립변수의 개수가 제약이 되니까 i 그리고 SSE의 자유도는 상수항 + 기울기 개수가 i+1개이니까 n-(i+1))

유의성 검정

이것 또한 f검정을 통해서 회귀식의 유의성을 검정해야 한다. f값을 구하고 f분포포의 값보다 크다면 귀무가설 기각

귀무가설 : 회귀식이 유의하지 않다

연습문제 풀이

Q4.

Q6

Q8

Q9

Q10

Q12

Q13

Q14

Q15