

2025-1 한국어 정보 처리

Python 기본 사용법 3: 정규 표현식

전태희

taeheejeon22@gmail.com

목차

1. 정규 표현식이란?
2. 정규 표현식의 기본 문법 1
3. re 모듈
4. 정규 표현식의 기본 문법 2
5. re 모듈의 컴파일 옵션
6. 정규 표현식의 심화 문법

1. 정규 표현식이란?

정규 표현식이란?

- 정의 (Wikipedia, '[Regular expression](#)')
 - 텍스트에서 나타나는 패턴을 명시하는 문자들의 연쇄

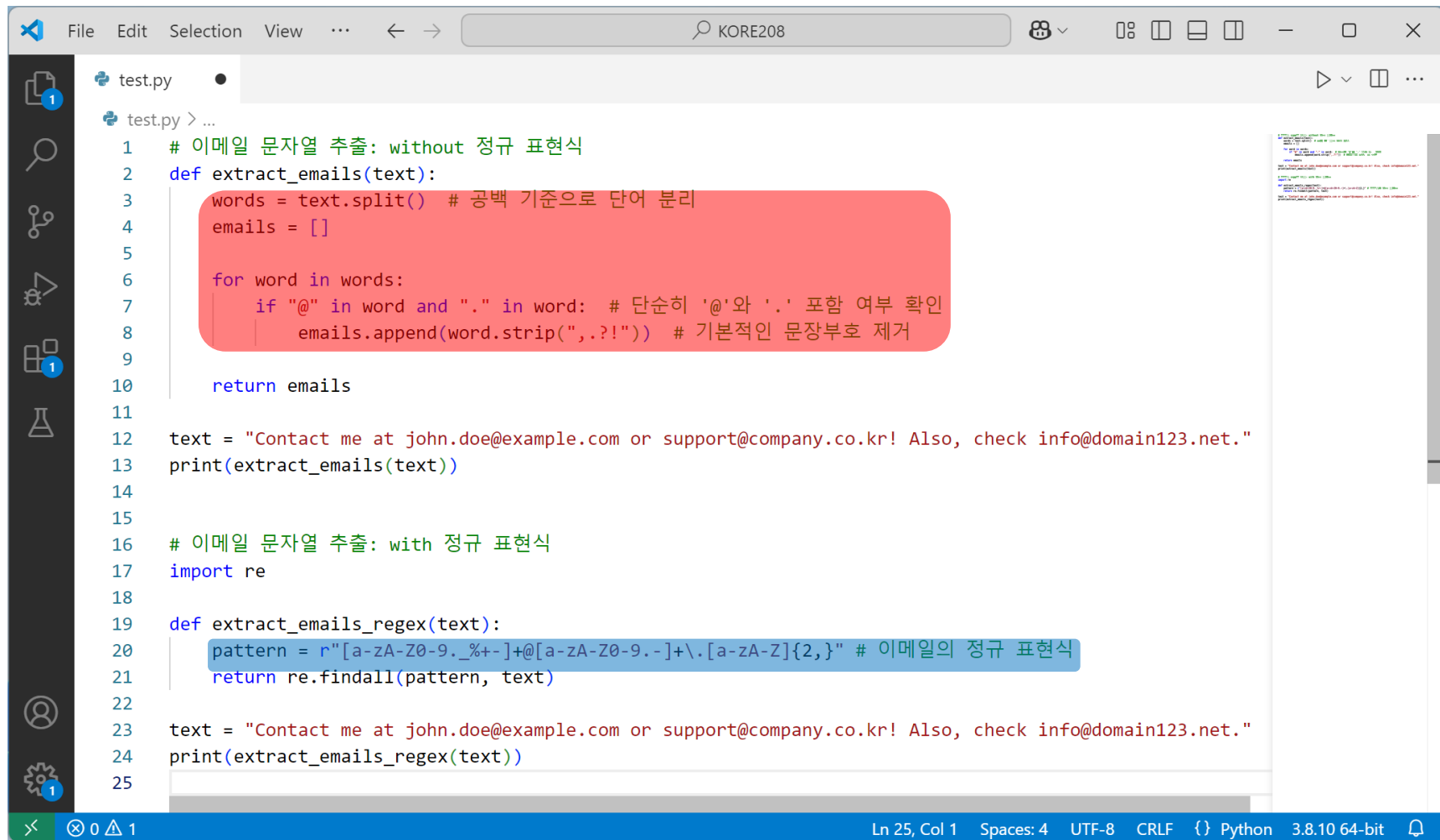
```
Friends, Romans, countrymen, lend me your ears;  
I come to bury Caesar, not to praise him.  
The evil that men do lives after them;  
The good is oft interred with their bones;  
So let it be with Caesar. The noble Brutus  
Hath told you Caesar was ambitious:  
If it were so, it was a grievous fault,  
And grievously hath Caesar answer'd it.
```

- 소문자 r 뒤에 1개 이상의 소문자 모음 문자가 이어지는 패턴

정규 표현식은 왜 필요한가? [1/2]

- 문서 편집 중 ctrl + f를 통해 단순 개별 문자열을 검색하는 것만으로는
부족함을 느끼는 경우들이 많음
 - 숫자가 포함되어 있는 모든 단어들을 검색할 수 없을까?
 - 1999년, 010-1234-5678, 90°F, ...
 - 모음 문자가 2개 이상 포함되어 있는 모든 단어들을 검색할 수 없을까?
 - number , cooler, real, ...

정규 표현식은 왜 필요한가? [2/2]



```
test.py
test.py > ...
1 # 이메일 문자열 추출: without 정규 표현식
2 def extract_emails(text):
3     words = text.split() # 공백 기준으로 단어 분리
4     emails = []
5
6     for word in words:
7         if "@" in word and "." in word: # 단순히 '@'와 '.' 포함 여부 확인
8             emails.append(word.strip(",.?!")) # 기본적인 문장부호 제거
9
10    return emails
11
12 text = "Contact me at john.doe@example.com or support@company.co.kr! Also, check info@domain123.net."
13 print(extract_emails(text))
14
15
16 # 이메일 문자열 추출: with 정규 표현식
17 import re
18
19 def extract_emails_regex(text):
20     pattern = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}" # 이메일의 정규 표현식
21     return re.findall(pattern, text)
22
23 text = "Contact me at john.doe@example.com or support@company.co.kr! Also, check info@domain123.net."
24 print(extract_emails_regex(text))
25
```

Ln 25, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

2. 정규 표현식의 기본 문법

메타 문자 (Meta-characters)

- 문자 그 자체를 가리키지 않고, 정규 표현식에서 특별한 문법적 의미를 지니며 쓰이는 문자
- 다양한 문자열 패턴을 표현하는 데에 필수적으로 쓰임
- 유형
 - 유형 1: 그 자체로 실제 문자에 대응되는 메타 문자
 - `[] . + * { } ?` 등
 - 유형 2: 실제 문자에 대응되지 않고, 문자열의 구조를 표현하는 데 쓰이는 메타 문자
 - `| ^ $ \wA \wZ \wb \WB ()` 등

[] [1/5]

▪ 문자 클래스

- 1개의(길이 1) 문자에 대응되는, 1개 이상의 문자의 집합을 표현
- [] 안에 나열되는 문자들의 순서는 아무런 의미가 없음

▪ 예

- [abc]
 - 'a'
 - 'before'
 - 'abc'
- [123456789]
 - '321'
 - 'abcZ89'

[] [2/5]

▪하이픈(-)을 이용해 문자 사이의 범위를 표현할 수 있음

○*[start-end]*

- *start*: 범위의 시작 문자; *end*: 범위의 끝 문자

○인덱싱, 슬라이싱에서와 달리 *end* (범위의 끝 문자) 까지 포함됨

○문자의 범위는 인코딩 체계를 따라 정해짐

○예

정규 표현식	의미
[a-c]	a, b, c 중 하나에 대응되는 문자 1개
[a-zA-Z]	a, b, ..., y, z, A, B, ..., Y, Z 중 하나에 대응되는 문자 1개

```
>>> ord('a')
97
>>> ord('c')
99
```

```
>>> ord('a')
97
>>> ord('z')
122
```

```
>>> ord('A')
65
>>> ord('Z')
90
```

[] [3/5]

■ 문자 클래스 내에서의 ^의 사용

○ ^은 정규 표현식에서 기본적으로 문자열 처음을 표시하는 기호이나,
문자 클래스 [] 내에서는 다른 기능을 함

○ not

제시되는 문자 범위들에 포함되지 않는, 나머지 모든 것을 가리킴을 의미

○ 예

정규 표현식	의미	대응 문자열
[^123]	1, 2, 3이 아닌 문자 중 하나와 대응되는 문자 1개	abc54321
[^A-Z]	영문자 대문자가 아닌 문자 중 하나와 대응되는 문자 1개	ABCDE12345가나다

[] [4/5]

▪ 자주 사용하는 문자 클래스를 위한 이스케이프 시퀀스 (Escape Sequence)

	[]를 이용한 표기	의미
\d	[0-9]	숫자 중 하나에 대응되는 문자 1개
\D	[^0-9]	숫자가 아닌 문자 중 하나와 대응되는 문자 1개
\s	[\t\n\r\f\v]	whitespace 문자 중 하나에 대응되는 문자 1개
\S	[^ \t\n\r\f\v]	whitespace 문자가 아닌 문자 중 하나에 대응되는 문자 1개
\w	[a-zA-Z0-9_]	문자와 숫자 (alphanumeric) 중 하나에 대응되는 문자 1개
\W	[^a-zA-Z0-9_]	문자와 숫자 (alphanumeric)가 아닌 문자 중 하나에 대응되는 문자 1개

[] [5/5]

■연습

- 1) 영문자의 모든 모음 문자 중 하나에 대응되는 문자 1개
- 2) 영문자의 모든 자음 문자 중 하나에 대응되는 문자 1개
- 3) 현대 한글의 모든 자음 및 모음 문자 중 하나에 대응되는 문자 1개
- 4) 현대 한글의 모든 음절 문자 중 하나에 대응되는 문자 1개

. [1/2]

- \Wn을 제외한 문자 중 하나 (사실상 모든 문자)와 대응되는 문자 1개
 - 메타 문자가 아닌, 문자 그대로의 .를 나타내려면 \W. 사용
 - 예

정규 표현식	의미	대응 문자열
a.b	'a' + \n_제외_모든_문자 + 'b'	aab !@#a0b456
199.	'199' + \n_제외_모든_문자	1990년 199쪽
..	\n_제외_모든_문자 + \n_제외_모든_문자	az 23 사람이

. [2/2]

▪ 문자 클래스 [] 내에서 쓰이는 경우

○ 메타 문자가 아닌, 문자 그대로의 . 가리킴

○ 예

정규 표현식	의미	대응 문자열
[.!?]	.! ? 중 하나의 문자에 대응되는 문자 1개	Yes! 안녕하세요.
[^.!?]	.! ? 가 아닌 문자 중 하나에 대응되는 문자 1개	!Yes! ?안녕하세요.

✱

■ 앞의 문자가 0회 1회 이상(무한대까지) 반복된 문자열

○ 앞의 문자가 없는 경우 (0회) 가 포함됨에 유의할 것

○ 예

정규식	문자열	매치 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)

+

■ 앞의 문자가 1회 이상(무한대까지) 반복된 문자열

○ *와 달리, 앞의 문자가 반드시 1회 이상 등장해야 함

○ 예

정규식	문자열	매치 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)

{ } [1 / 3]

- 앞의 문자가 임의의 횟수만큼 반복된 문자열
 - *, +와 달리 반복 횟수를 사용자가 직접 설정할 수 있음
- 기본 사용법
 - {m, n}
 - m: 최소 반복 횟수
 - n: 최대 반복 횟수

{ } [2/3]

▪ 다른 사용법

- {m, }

- m회 이상 - 무한대

- {,n}

- 0회 혹은 1회 이상 - n회 이하

- {1,}

- + 과 같은 의미

- {0,}

- * 과 같은 의미

{ } [3/3]

▪ 예

정규식	문자열	매치 여부	설명
<code>ca{2,5}t</code>	cat	No	"a"가 1번만 반복되어 매치되지 않음.
<code>ca{2,5}t</code>	caat	Yes	"a"가 2번 반복되어 매치
<code>ca{2,5}t</code>	caaaaat	Yes	"a"가 5번 반복되어 매치

?

- 앞의 문자가 0회 혹은 1회 나타난 문자열
- {0,1}과 같은 의미
- 예

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

메타 문자의 조합 [1 / 3]

▪ 메타 문자들을 조합함으로써 무수한 수의 문자열 패턴을 표현할 수 있음

▪ 다음 텍스트에서 아래를 추출하려면?

○이메일

○전화번호

○일시

○주소

...

직원 명단:

김현수: 나이 52, 남성, 이메일: hyunsoo.kim@industry.co.kr, 전화번호: 010-2222-3333

박지은: 나이 34, 여성, 이메일: jieun.park@techhub.com, 전화번호: 010-4444-5555

서태지: 나이 46, 남성, 이메일: taegi.seo@musicentertainment.kr, 전화번호: 010-6666-7777

회사 주소록:

인천시 남동구 구월동 987번지 654호, 인천제조회사, 대표전화: 032-1234-5678

대전시 유성구 지족동 321번지 432호, 대전연구소, 대표전화: 042-8765-4321

이벤트 일정:

2025년 07월 15일 16:00 - 데이터 사이언스 워크샵, 장소: 대전 컨벤션 센터

2026년 08월 22일 09:00 - 창업 경진대회, 장소: 인천 송도 컨벤션

메모:

이민호: 010-8888-9999, 이메일: minho.lee@cinemakorea.kr

기록: 2023년 05월 10일, 총 결제 금액: 450,000원

메모: '2023년 05월 전략회의', 핵심 이슈는 인공지능 통합.

HTML 코드 조각:

```
<div class="profile">
  <p>이 페이지는 직원 정보를 상세히 나타내고 있습니다.</p>
  <a href="http://www.techindustry.kr">자세한 정보 보기</a>
</div>
```

로그 데이터:

정보: 2023년 06월 05일 10:30:35 - 파일 업로드 완료

경고: 2023년 06월 06일 12:00:00 - 네트워크 연결 지연

메타 문자의 조합 [2/3]

■ 앞 텍스트의 문자열 패턴 분석

○ 이메일

- 계정@도메인.최상위_도메인

○ 전화번호

- 3자리_숫자 - 3-4자리_숫자 - 4자리_숫자

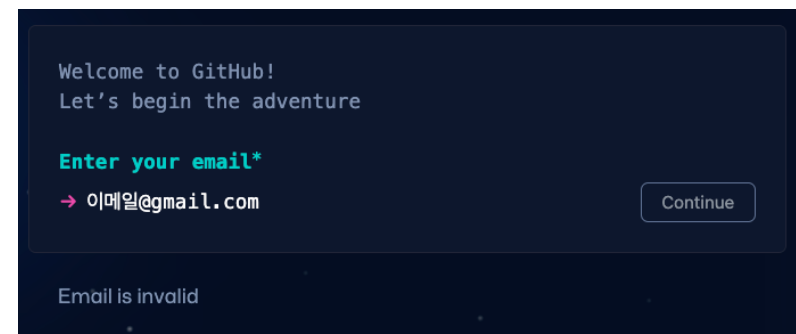
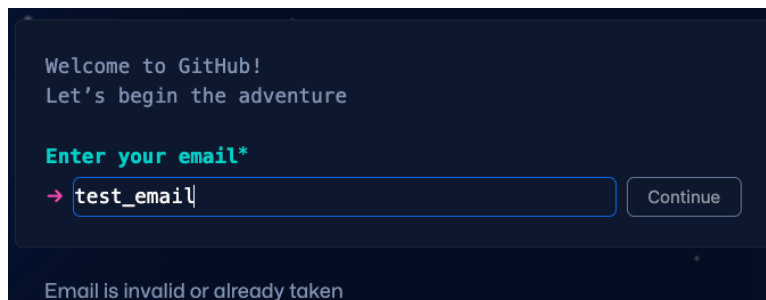
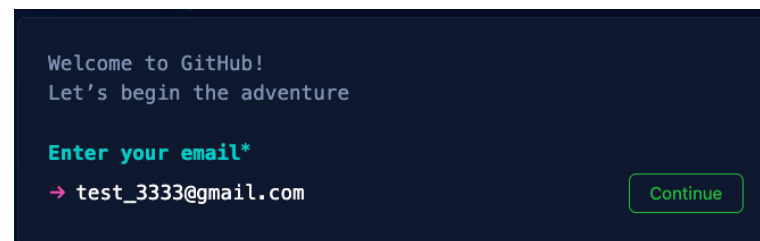
○ 날짜

- 4자리_숫자 년 1-2자리_숫자 월 1-2자리_숫자 일

메타 문자의 조합 [3/3]

■ 이메일 정규 표현식의 활용 예

○ $^{\circ}[a-zA-Z0-9._\%+-]+\@[a-zA-Z0-9.-]+\W.[a-zA-Z]{2,}\$$



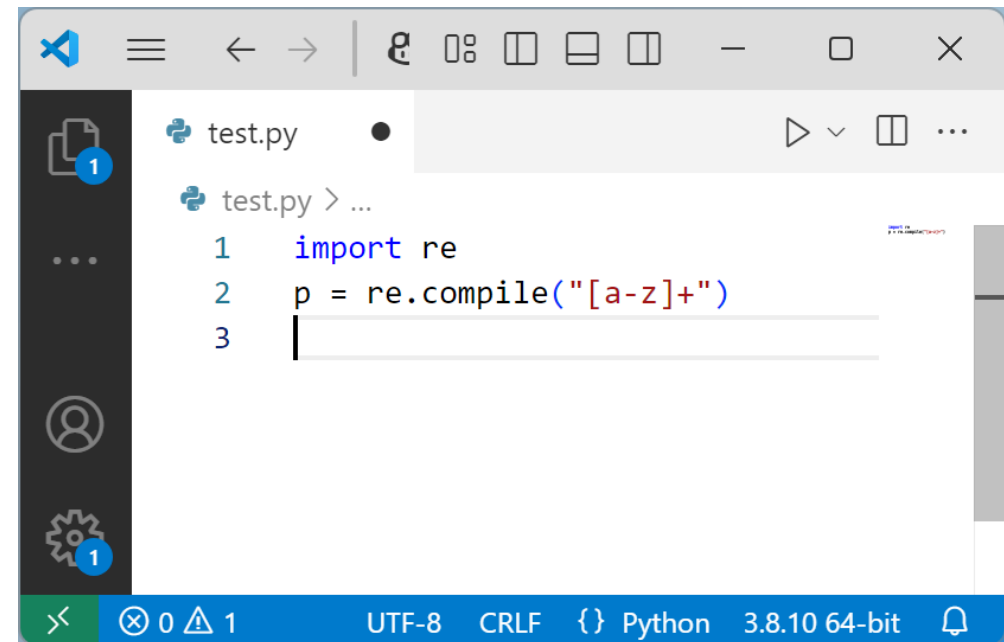
3. re 모듈

Python 표준 라이브러리 re

- 정규 표현식 (regular expression) 지원을 위한 라이브러리
- Python 설치 시 자동으로 설치됨
- import 키워드를 이용하여 불러옴

.compile()

- 정규 표현식을 컴파일
- 본 맥락에서의 컴파일(compile)
 - 입력된 문자열을 컴퓨터가 문자열이 아닌, '정규 표현식'으로 인식할 수 있도록 만드는 행위



```
test.py
test.py > ...
1 import re
2 p = re.compile("[a-z]+")
3
```

The screenshot shows a code editor window with a file named 'test.py'. The code inside the file is as follows:

```
1 import re
2 p = re.compile("[a-z]+")
3
```

The status bar at the bottom indicates the file is encoded in UTF-8, uses CRLF line endings, and is a Python 3.8.10 64-bit file. There is also a notification icon on the right.

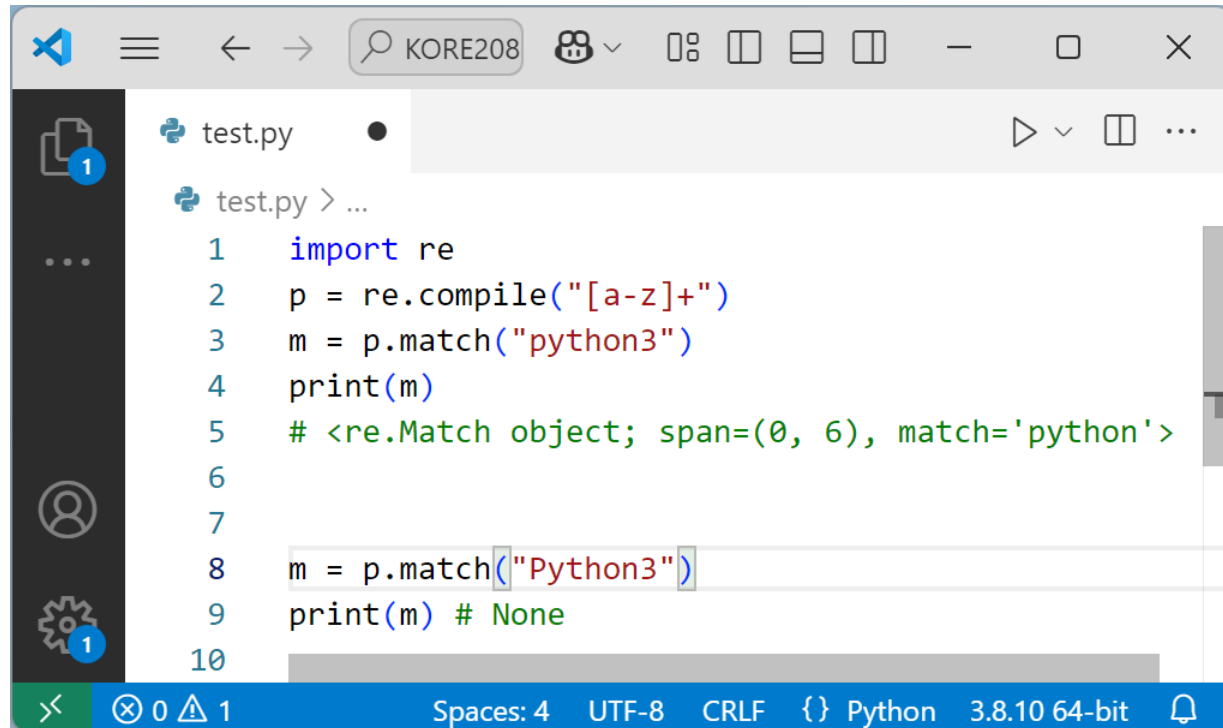
.match() [1 / 2]

- 문자열의 처음부터 정규 표현식과 매치되는 부분 문자열이 있는지 검색
- 매치되는 부분 문자열이 있으면 Match 객체 리턴
 - 매치되는 부분 문자열을 찾으면 검색을 바로 종료함
- 매치되는 부분 문자열이 없으면 아무것도 리턴하지 않음 (즉 None 리턴)

.match() [2/2]

■ 사용 예

- span: 정규표현식과 매치되는 문자열의 시작 인덱스, 끝 인덱스
 - 실제로는 끝_인덱스 - 1까지 포함됨
- match: 정규 표현식과 매치되는 문자열
- 첫 부분부터 매치가 되지 않으면, 뒤에 매치되는 문자열이 있더라도 None 리턴



```
test.py
test.py > ...
1  import re
2  p = re.compile("[a-z]+")
3  m = p.match("python3")
4  print(m)
5  # <re.Match object; span=(0, 6), match='python'>
6
7
8  m = p.match("Python3")
9  print(m) # None
10
```

Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

.search() [1 / 2]

- 문자열 전체에서 정규 표현식과 매치되는 부분 문자열이 있는지 검색
- 기본적으로는 .match()와 비슷하게, 매치되는 부분 문자열이 있으면 그에 대한 Match 객체를, 그렇지 않으면 None 리턴
- .match()와 달리, 첫 부분에서 매치되는 부분 문자열이 나타나지 않더라도 뒤에 나타난다면 Match 객체 리턴

.search() [2/2]

■ 사용 예

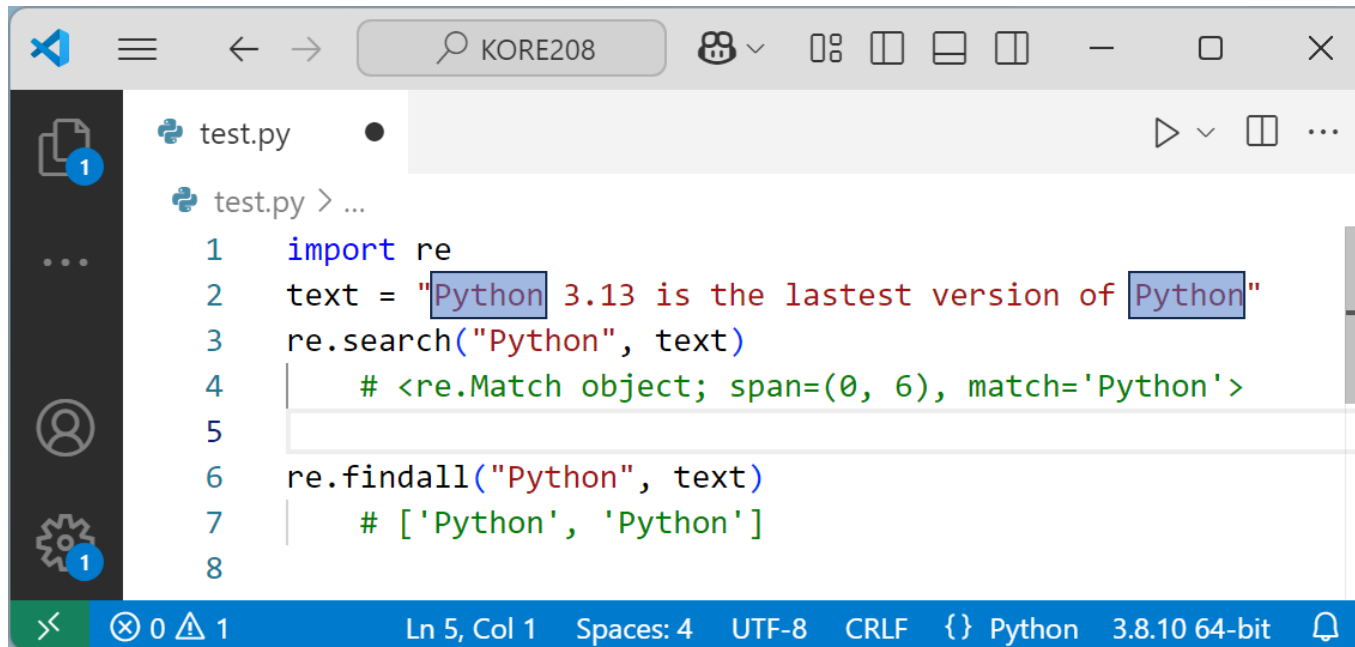
```
test.py
test.py > ...
1  import re
2  p = re.compile("[a-z]+")
3  m = p.search("python3")
4  print(m)
5      # <re.Match object; span=(0, 6), match='python'>
6
7  m = p.search("Python3")
8  print(m)
9      # <re.Match object; span=(1, 6), match='ython'>
10
```

Ln 7, Col 24 Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

- .match()에서와 달리 Python3에 대해서도 Match 객체 리턴

.findall() [1/3]

- 정규 표현식과 매치되는 모든 부분 문자열을 요소로 갖는 리스트 리턴
- .findall()은 문자열 끝까지 계속 검색함

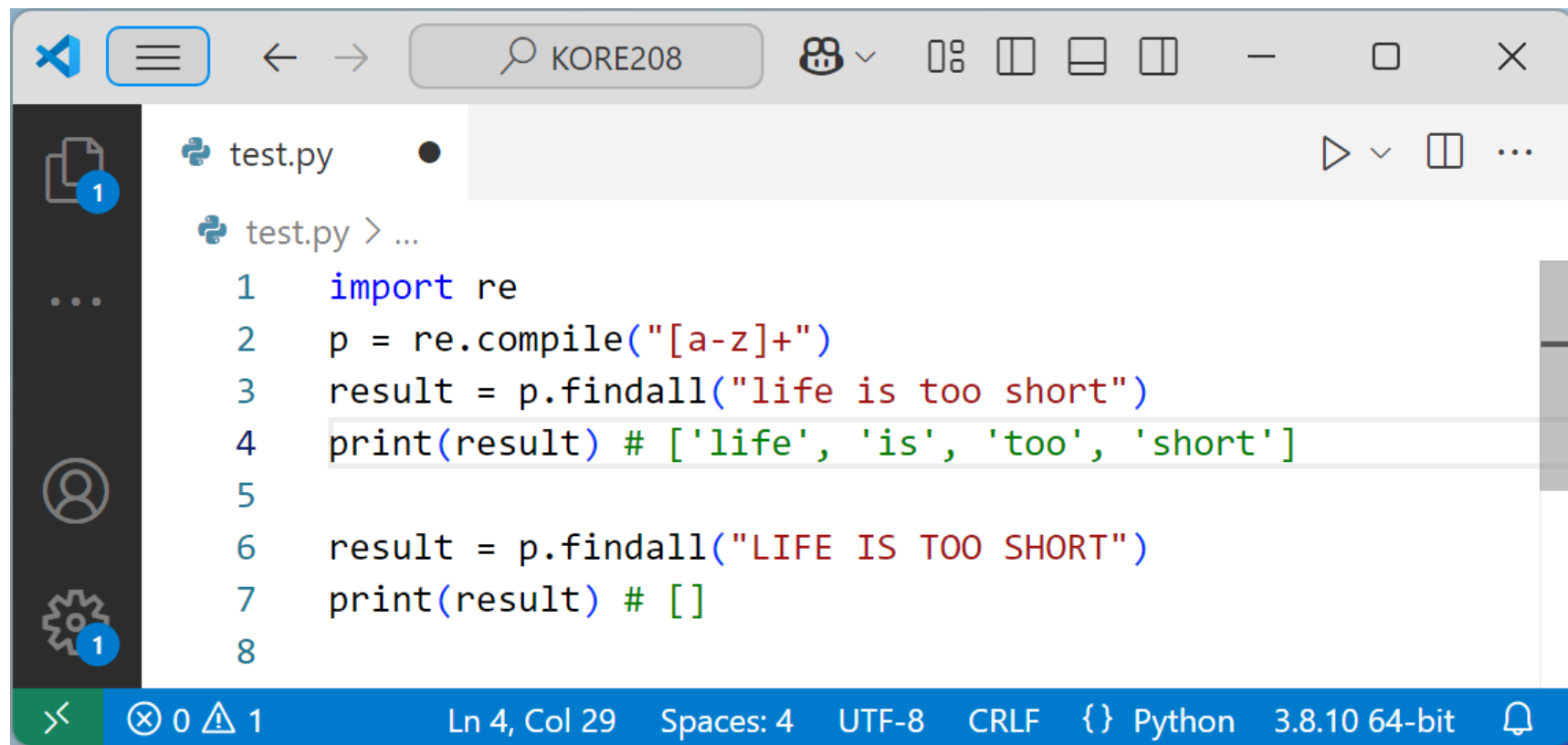


```
test.py
test.py > ...
1 import re
2 text = "Python 3.13 is the latest version of Python"
3 re.search("Python", text)
4 # <re.Match object; span=(0, 6), match='Python'>
5
6 re.findall("Python", text)
7 # ['Python', 'Python']
8
```

Ln 5, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

.findall() [2/3]

■ 사용 예

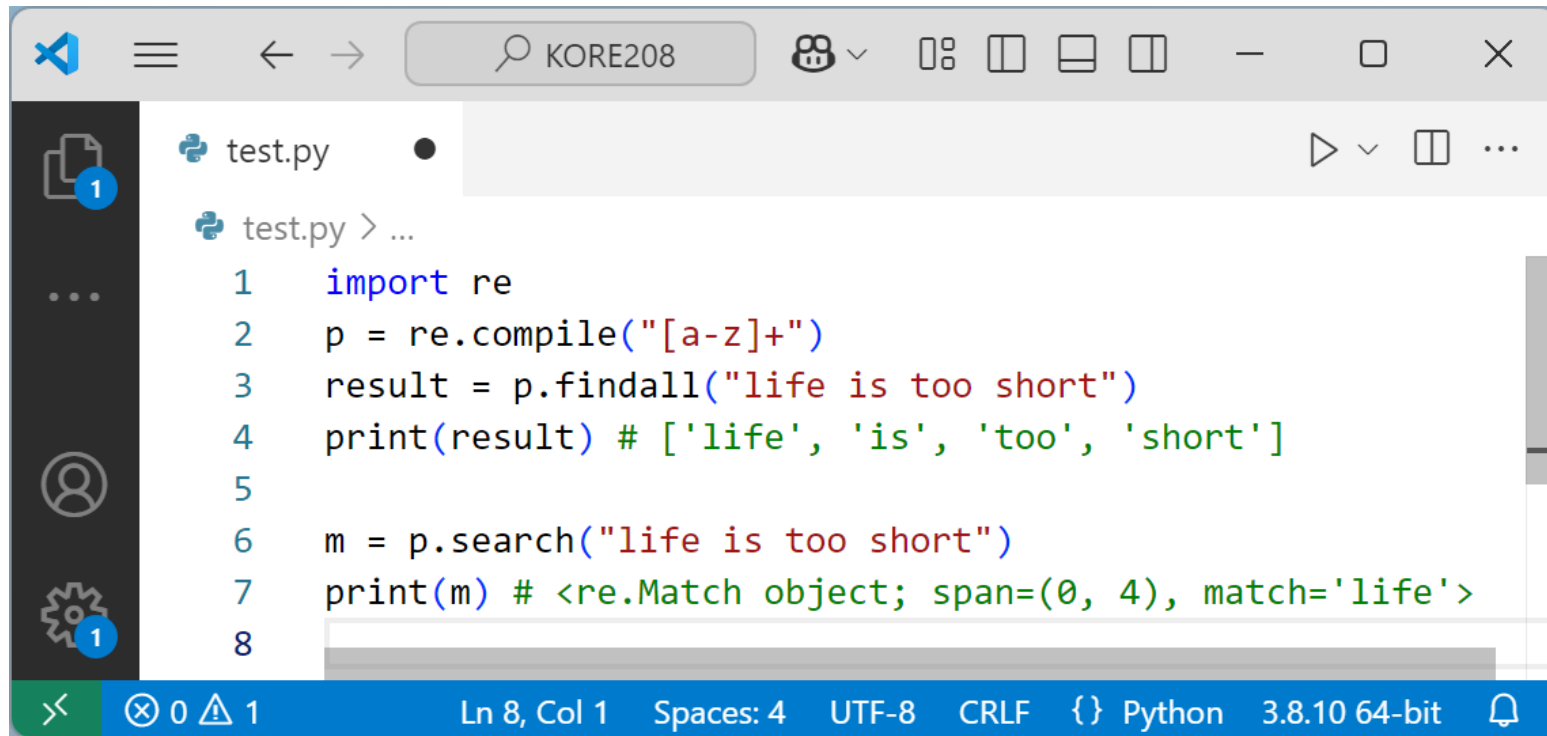


```
test.py
test.py > ...
1  import re
2  p = re.compile("[a-z]+")
3  result = p.findall("life is too short")
4  print(result) # ['life', 'is', 'too', 'short']
5
6  result = p.findall("LIFE IS TOO SHORT")
7  print(result) # []
8
```

VS Code interface details: The window title is 'KORE208'. The status bar at the bottom shows 'Ln 4, Col 29', 'Spaces: 4', 'UTF-8', 'CRLF', '{} Python 3.8.10 64-bit'. There are also icons for search, run, and other editor functions.

.findall() [3/3]

- .match(), .search()는 매치되는 부분 문자열을 찾으면 즉시 검색을 종료하나, .findall()은 매치되는 부분 문자열을 찾더라도 문자열 끝까지 검색함

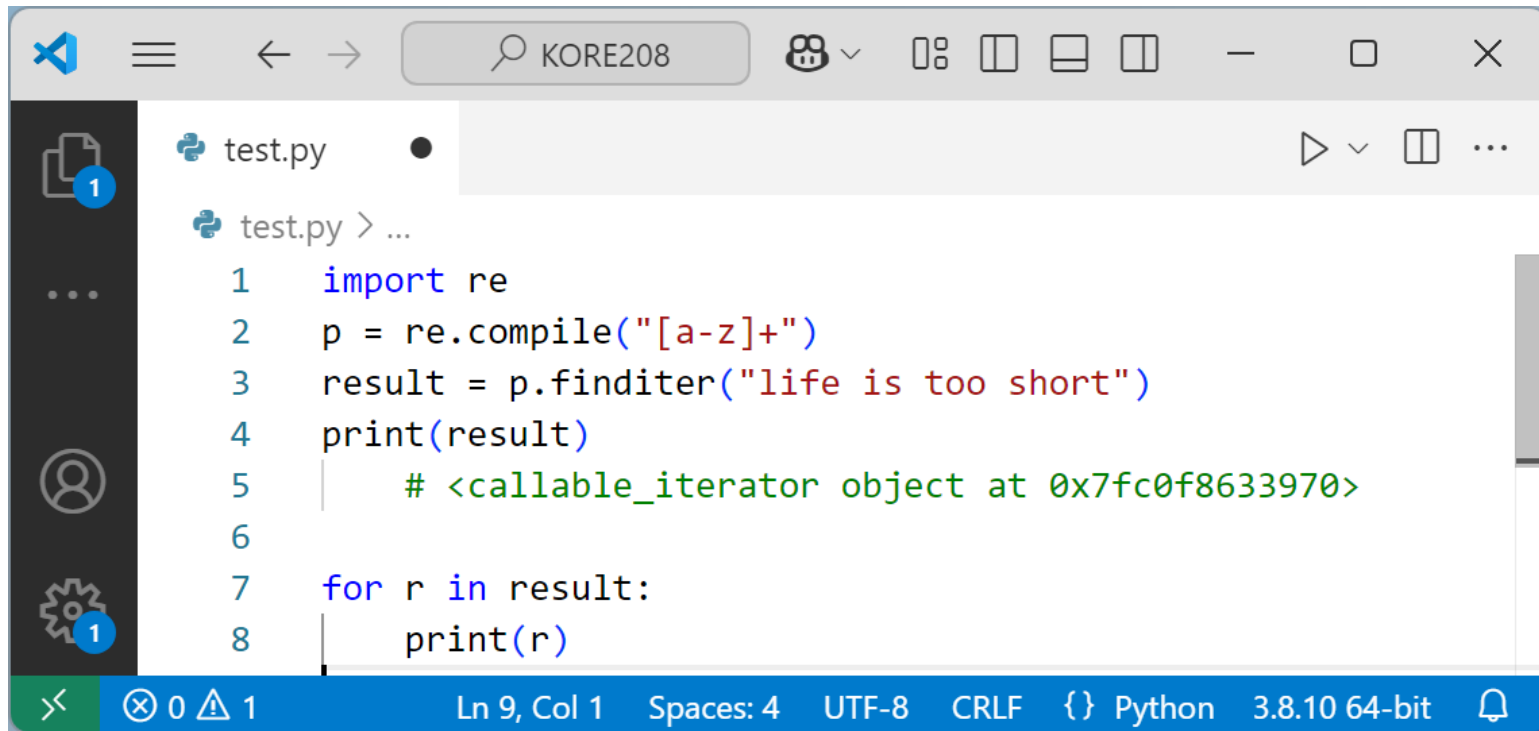


```
test.py
test.py > ...
1  import re
2  p = re.compile("[a-z]+")
3  result = p.findall("life is too short")
4  print(result) # ['life', 'is', 'too', 'short']
5
6  m = p.search("life is too short")
7  print(m) # <re.Match object; span=(0, 4), match='life'>
8
```

Ln 8, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

.finditer()

- .findall()과 기능은 거의 동일함
- 리턴 값이 리스트가 아니라, iterable 객체라는 점에서 다름



```
test.py
test.py > ...
1  import re
2  p = re.compile("[a-z]+")
3  result = p.finditer("life is too short")
4  print(result)
5  # <callable_iterator object at 0x7fc0f8633970>
6
7  for r in result:
8      print(r)
```

The screenshot shows a Visual Studio Code window with a file named 'test.py'. The code in the editor uses the `re.finditer()` method to find all non-overlapping matches of the regular expression `[a-z]+` in the string `"life is too short"`. The output of `print(result)` is a `<callable_iterator object at 0x7fc0f8633970>`. A `for` loop then iterates over this iterator, printing each match. The status bar at the bottom indicates the file is encoded in UTF-8, uses CRLF line endings, and is a Python 3.8.10 64-bit file.

.sub() [1 / 3]

- 어떤 문자열 내에서 정규 표현식과 매치되는 부분을 다른 문자열로 바꿈
- 문자열 메서드 .replace()와 유사하나,
정규 표현식을 사용할 수 있다는 점에서 훨씬 강력함
- 기본 사용법
 - 정규_표현식.sub(새_부분_문자열, 전체_문자열)

.sub() [2/3]

▪ 문자열 수정이 필요한 문제 상황

○ 다음 텍스트에서 철자가 틀린

committee를 모두 바르게 수정하려면?

- 실제 쓰인 것들을 보면 committee, comittee, comitte, committtee 등 다양한 표기가 나타남
- 실수가 일어나는 부분은 m, t, e이며 모두 2개를 쓰는 것이 맞음

○ 정규 표현식을 쓰지 않는다면 어떻게 수정할 수 있을까?

During the recent board **committee** meeting, it was decided that the steering **comittee** would oversee the new project initiatives. The finance **committte** has been tasked with reviewing the budget proposals submitted last week. Members of the organizing **committtee** are expected to provide a detailed report on event logistics by the end of this month.

Additionally, the outreach **comitte** has proposed a partnership with local universities to foster community engagement. The executive **comittee**, however, raised concerns regarding the oversight and measurable outcomes of such collaborations. Feedback from the advisory **committte** will be crucial in finalizing the partnership terms.

.sub() [3/3]

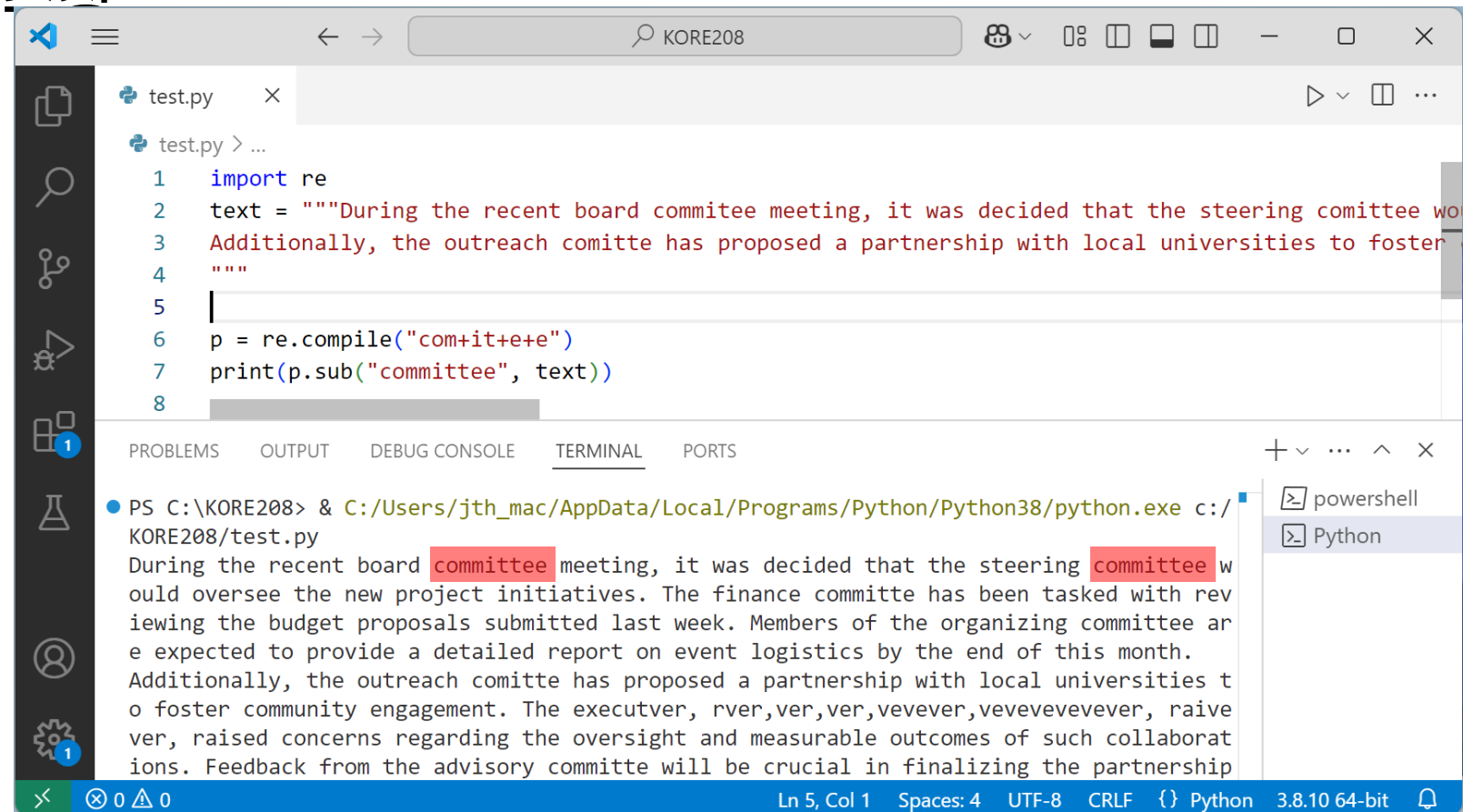
■.sub()를 이용한 사전

○정규 표현식

com+it+e+

○그 의미는?

- co
- m+
- i
- †+
- e+



The screenshot shows a VS Code editor window with a file named 'test.py'. The code in the editor is as follows:

```
1 import re
2 text = """During the recent board committee meeting, it was decided that the steering committee wo
3 Additionally, the outreach comitte has proposed a partnership with local universities to foster
4 """
5 |
6 p = re.compile("com+it+e+")
7 print(p.sub("committee", text))
8
```

The terminal output shows the result of running the script:

```
PS C:\KORE208> & C:/Users/jth_mac/AppData/Local/Programs/Python/Python38/python.exe c:/KORE208/test.py
During the recent board committee meeting, it was decided that the steering committee w
ould oversee the new project initiatives. The finance committee has been tasked with rev
iewing the budget proposals submitted last week. Members of the organizing committee ar
e expected to provide a detailed report on event logistics by the end of this month.
Additionally, the outreach comitte has proposed a partnership with local universities t
o foster community engagement. The executver, rver,ver,ver,vevever,vevevevevever, raive
ver, raised concerns regarding the oversight and measurable outcomes of such collaborat
ions. Feedback from the advisory committe will be crucial in finalizing the partnership
```

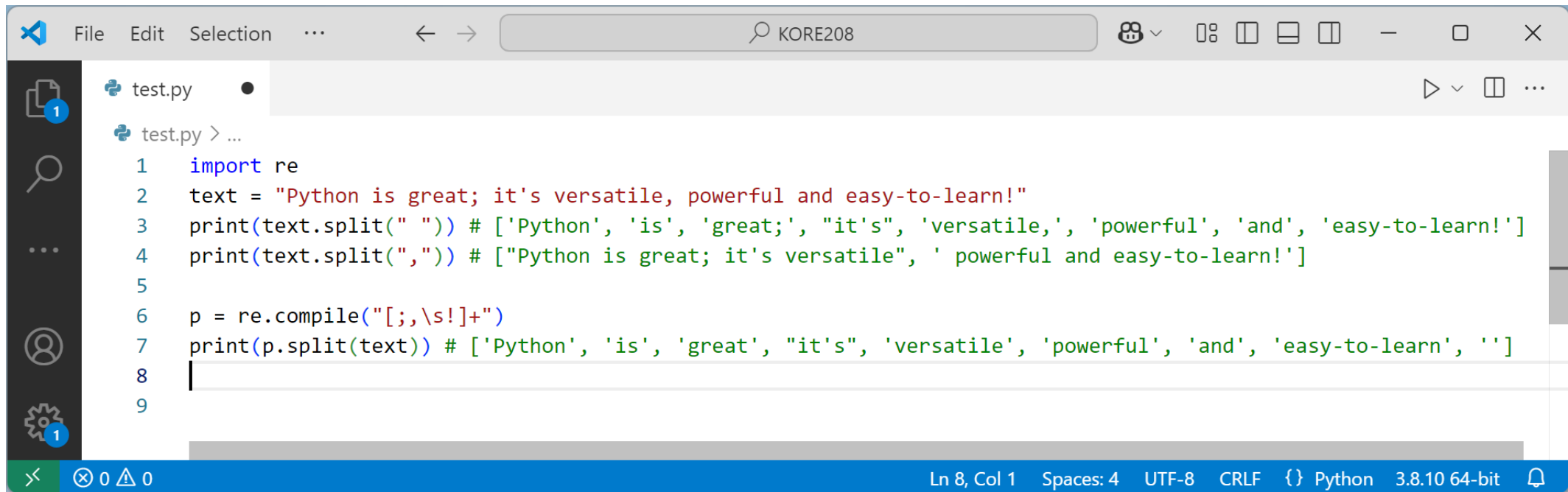
The terminal output shows the result of running the script, with the word 'committee' appearing in red in the original image, indicating a match with the regular expression.

.split() [1 / 2]

- 문자열의 .split() 메서드에서 구분자를 정규 표현식으로 쓰도록 한 버전
 - 문자열 메서드 .split()은 단 하나의 문자열만을 구분자로 사용함
 - re 라이브러리의 .split()은 무수한 문자열을 아우를 수 있는 패턴을 구분자로 사용하므로 훨씬 유연하게 활용할 수 있음
- 기본 사용법
 - *정규_표현식.split(문자열)*

.split() [2/2]

▪ 사용 예: 텍스트의 단어 추출



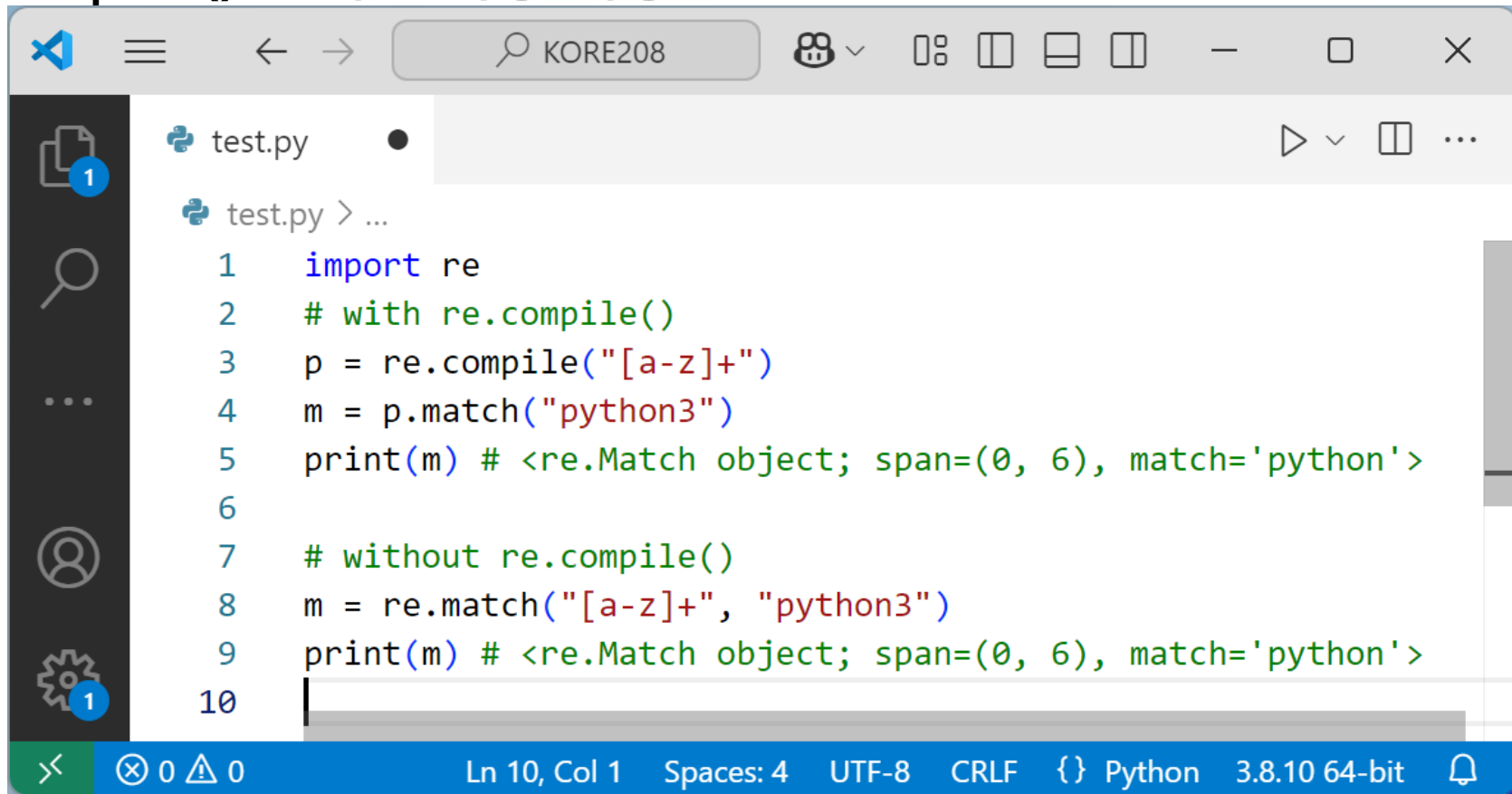
```
File Edit Selection ... KORE208
test.py
test.py > ...
1 import re
2 text = "Python is great; it's versatile, powerful and easy-to-learn!"
3 print(text.split(" ")) # ['Python', 'is', 'great;', "it's", 'versatile,', 'powerful', 'and', 'easy-to-learn!']
4 print(text.split(",")) # ["Python is great; it's versatile", ' powerful and easy-to-learn!']
5
6 p = re.compile("[;,\s!]+")
7 print(p.split(text)) # ['Python', 'is', 'great', "it's", 'versatile', 'powerful', 'and', 'easy-to-learn', '']
8
9
```

Ln 8, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

- 문자열의 기본 메서드 split으로는
구분자 목록을 제시하고 경우에 따라 그중 하나를 구분자로 사용하는 것은 불가능

re 모듈의 다른 사용법 [1/2]

- re.compile() 없이도 사용 가능함



```
test.py
test.py > ...
1  import re
2  # with re.compile()
3  p = re.compile("[a-z]+")
4  m = p.match("python3")
5  print(m) # <re.Match object; span=(0, 6), match='python'>
6
7  # without re.compile()
8  m = re.match("[a-z]+", "python3")
9  print(m) # <re.Match object; span=(0, 6), match='python'>
10
```

Ln 10, Col 1 Spaces: 4 UTF-8 CRLF { } Python 3.8.10 64-bit

re 모듈의 다른 사용법 [2/2]

- `re.compile()`을 사용하는 것이 유리한 경우
 - 만든 정규 표현식을 여러 번 사용해야 하는 경우
 - 정규 표현식이 복잡하여, 다른 코드 내에 포함시키면 코드가 번잡해지는 경우
- `re.compile()`을 사용하지 않는 것이 유리한 경우
 - 만든 정규 표현식을 1-2회 정도 사용한 후 더 이상 사용하지 않는 경우
 - 정규 표현식이 간단하여, 다른 코드 내에 포함시키는 것이 더 간결한 경우

Match 객체의 활용 [1 / 4]

- re 모듈의 메서드를 사용할 시 드는 의문

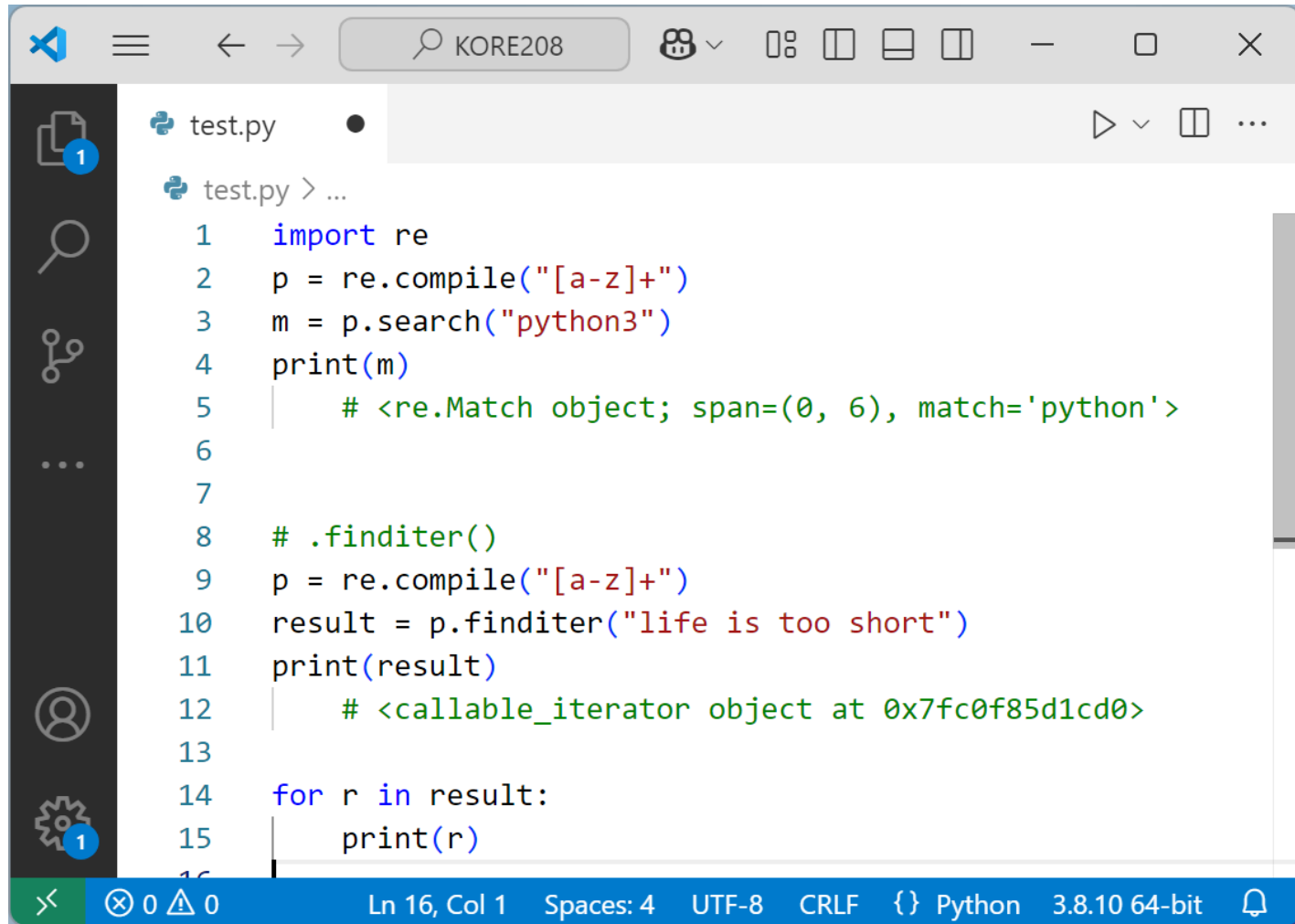
- 정규 표현식과 매치되는 부분 문자열이 정확히 무엇인가?
- 매치된 부분 문자열은, 원래 문자열의 인덱스를 기준으로 어디서부터 어디까지인가?

Match 객체의 활용 [2/4]

▪ Match 객체

- `.match()`, `.search()`이 리턴하는 Match 객체

- `.finditer()`이 리턴하는 iterable 객체에 속한 개별 Match 객체



```
test.py
test.py > ...
1  import re
2  p = re.compile("[a-z]+")
3  m = p.search("python3")
4  print(m)
5      # <re.Match object; span=(0, 6), match='python'>
6
7
8  # .finditer()
9  p = re.compile("[a-z]+")
10 result = p.finditer("life is too short")
11 print(result)
12      # <callable_iterator object at 0x7fc0f85d1cd0>
13
14 for r in result:
15     print(r)
```

Ln 16, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

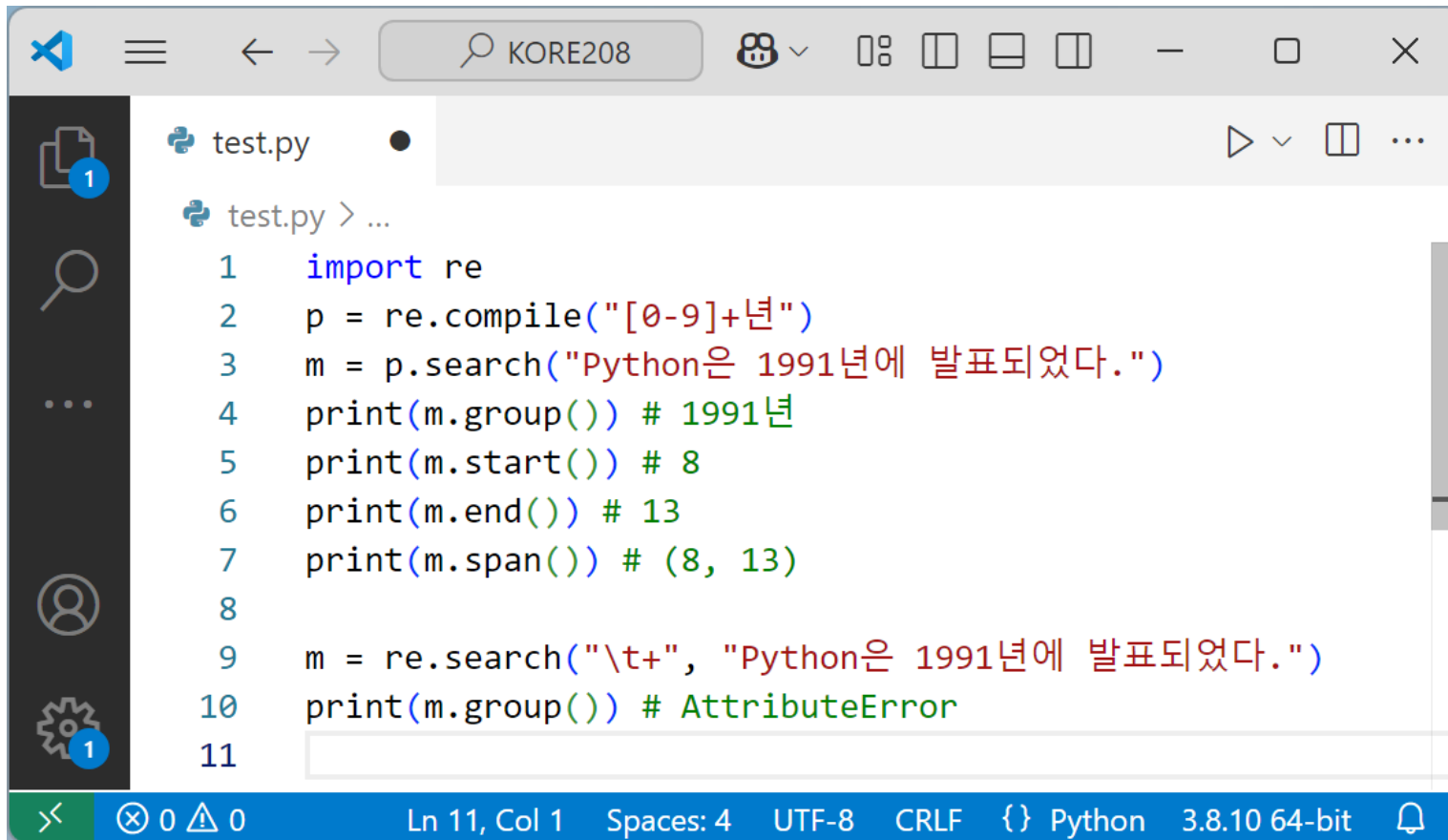
Match 객체의 활용 [3/4]

▪ Match 객체의 메서드

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

Match 객체의 활용 [4/4]

○사용 예



```
test.py
test.py > ...
1  import re
2  p = re.compile("[0-9]+년")
3  m = p.search("Python은 1991년에 발표되었다.")
4  print(m.group()) # 1991년
5  print(m.start()) # 8
6  print(m.end()) # 13
7  print(m.span()) # (8, 13)
8
9  m = re.search("\t+", "Python은 1991년에 발표되었다.")
10 print(m.group()) # AttributeError
11
```

4. 정규 표현식의 기본 문법 2

문자열의 구조를 표현하는 메타 문자

- 그 자체로는 문자에 대응되지 않으면서, 문자열의 시작과 끝을 표시하는 등의 방식으로 문자열의 구조를 나타낼 수 있는 메타 문자들이 존재함

- 앞서 다룬 [], + 등은 그 자체로 문자열에 쓰인 문자(들)에 대응됨

- 실제 문자열 sp~~eeeeee~~d abc~~1~~

- 정규 표현식 sp~~e~~+d [0-9]

- 이와 같이 실제 문자열에 대응되는 것을 ‘소비’라고 하며,

소비되지 않으면서도 문자열 패턴을 표현하는 방식을 Zero-width Assertion 이라고 함

|

▪ or의 의미

- Alice|Bob은 Alice 혹은 Bob 중 하나에 대응됨

▪ 2개 이상의 부분 문자열 중 하나에 대응될 수 있음

▪ 예

- ..이|가

- ‘사람이’

- ‘학교가’

- 0|2|4|6|8+

- ‘208원’

- ‘685m’



- 문자열의 맨 처음을 표시함
- 문자 클래스 내에서 쓰이는 ^과는 다르므로 주의할 것

○ `[^0-9]`

○ `^[0-9]`

- 예

```
test.py
1 import re
2 re.findall("^나는", "나는 하늘을 나는 꿈을 꾸었다.")
3 # ['나는']
```

re.match() vs. re.search() [1/2]

▪ re.match()의 예: re.match("[a-z]+")

- 전체 문자열: "123_Python"

- 실제 검색에 쓰이는 정규 표현식: **^[a-z]+**

- .match()는 전체 문자열의 처음부터 검색하기 때문에 실제로는 정규 표현식에 ^가 위치하는 것과 마찬가지로 됨

- 일치 여부: False

- 시작 위치에 영문자 소문자가 아닌, **1**이 위치하므로 전체 문자열과 정규 표현식과 일치하지 않음

re.match() vs. re.search() [2/2]

▪ re.search()의 예: re.search("[a-z]+")

- .match()와 동일하게 문자열 처음부터 검색하나, 검색되는 문자열이 계속 바뀜
- 검색 시작 위치를 1칸씩 뒤로 옮겨 가면서, 일치 여부 확인

iteration	검색 문자열	정규 표현식	일치 여부
1	"123_Python"	^[a-z]+	False
2	"23_Python"	^[a-z]+	False
3	"3_Python"	^[a-z]+	False
4	"_Python"	^[a-z]+	False
5	"Python"	^[a-z]+	False
6	"ython"	^[a-z]+	True

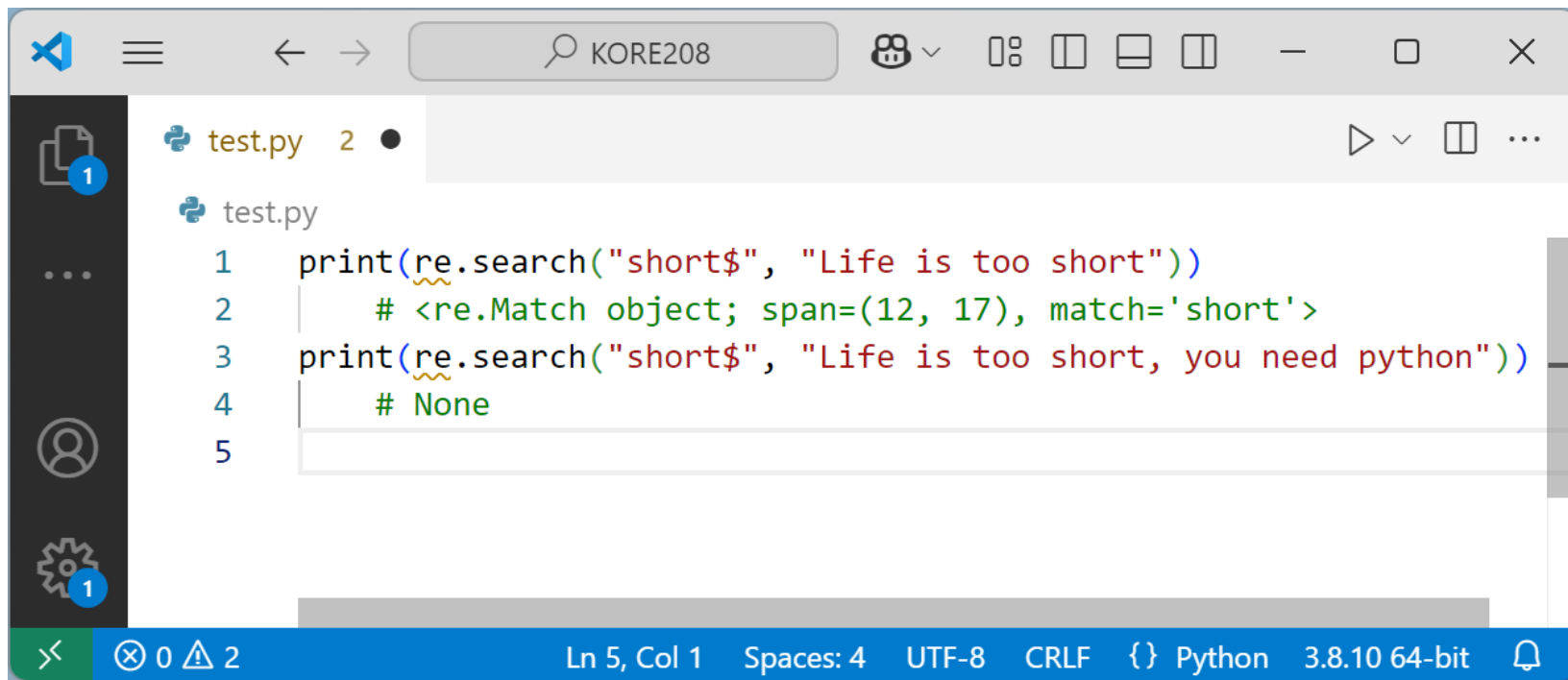
- 일치하는 경우가 발견될 때까지 계속 진행
- 일치하는 경우가 발견되면 즉시 종료

\$

■ 문자열의 맨 끝을 표시함

○ 즉 ^의 반대

■ 예

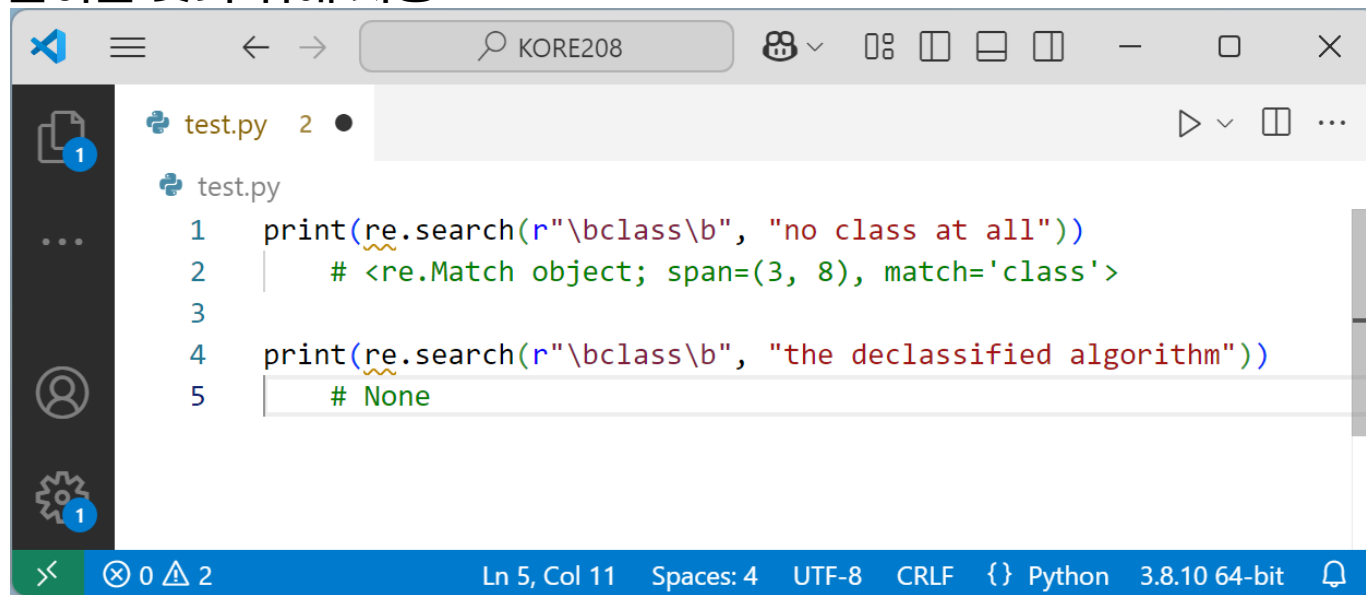


```
test.py 2 ●
test.py
1 print(re.search("short$", "Life is too short"))
2     # <re.Match object; span=(12, 17), match='short'>
3 print(re.search("short$", "Life is too short, you need python"))
4     # None
5
```

Ln 5, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

\b [1/2]

- 단어 경계 (word boundary)
 - 일반적으로 단어는 Whitespace Characters로 구분됨
 - 문자열의 처음과 끝도 단어 경계에 포함됨
- 어떤 패턴과 부분 일치하는 단어가 아니라, 완전 일치되는 단어를 찾기 위해 사용



```
test.py 2
test.py
1 print(re.search(r"\bclass\b", "no class at all"))
2     # <re.Match object; span=(3, 8), match='class'>
3
4 print(re.search(r"\bclass\b", "the declassified algorithm"))
5     # None
```

Ln 5, Col 11 Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

\b [2/2]

- 컴파일 시에 raw string임을 컴퓨터에게 알려 주기 위해 r 기호를 붙임
 - \b는 Python에서 백스페이스 문자를 가리키므로 백스페이스가 아니라 메타 문자임을 명시해야 함
 - `print(re.search(r"\bclass\b", "no class at all"))`
 - raw string에 대해서는 뒤에서 다시 다룸

\B

▪ \wb의 반대 의미

- 즉 \WB로 둘러 싸인 부분 문자열이, 어떤 단어 문자열의 내부에 있음을 표시
- r 기호를 사용하지 않아도 작동하나, Python 3.12부터는 경고 표시됨

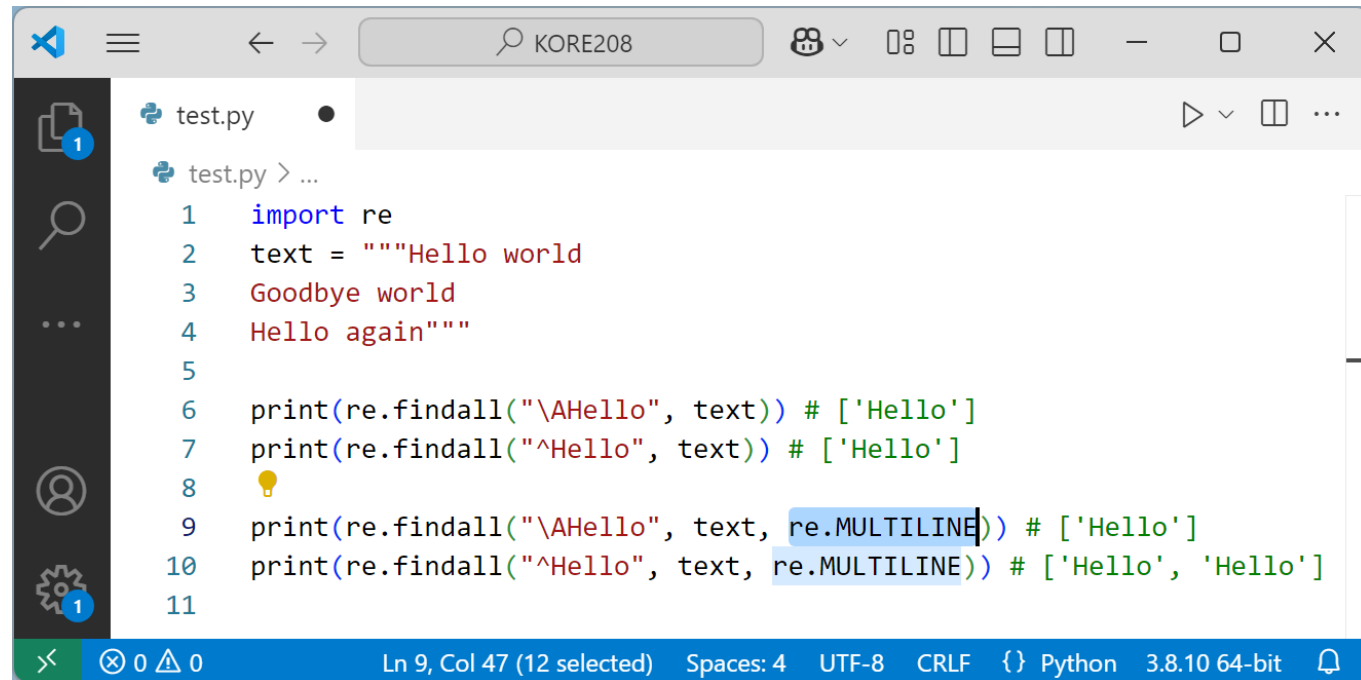
```
test.py
1  import re
2  print(re.search(r"\bclass\b", "no class at all"))
3      # <re.Match object; span=(3, 8), match='class'>
4  print(re.search(r"\Bclass\B", "no class at all"))
5      # None
6
7  print(re.search(r"\bclass\b", "the declassified algorithm"))
8      # None
9  print(re.search(r"\Bclass\B", "the declassified algorithm"))
10     # <re.Match object; span=(6, 11), match='class'>
11
12
```


\A

- 기본적으로 문자열 처음을 표현한다는 점에서 ^와 같음

- re 모듈의 re.MULTILINE 옵션을 사용하면 ^과 다른 의미를 갖도록 할 수 있음

- ^은 각 행의 처음을 가리키고, \A은 전체 문자열의 처음을 가리킴



```
test.py
test.py > ...
1  import re
2  text = """Hello world
3  Goodbye world
4  Hello again"""
5
6  print(re.findall("\AHello", text)) # ['Hello']
7  print(re.findall("^Hello", text)) # ['Hello']
8
9  print(re.findall("\AHello", text, re.MULTILINE)) # ['Hello']
10 print(re.findall("^Hello", text, re.MULTILINE)) # ['Hello', 'Hello']
11
```

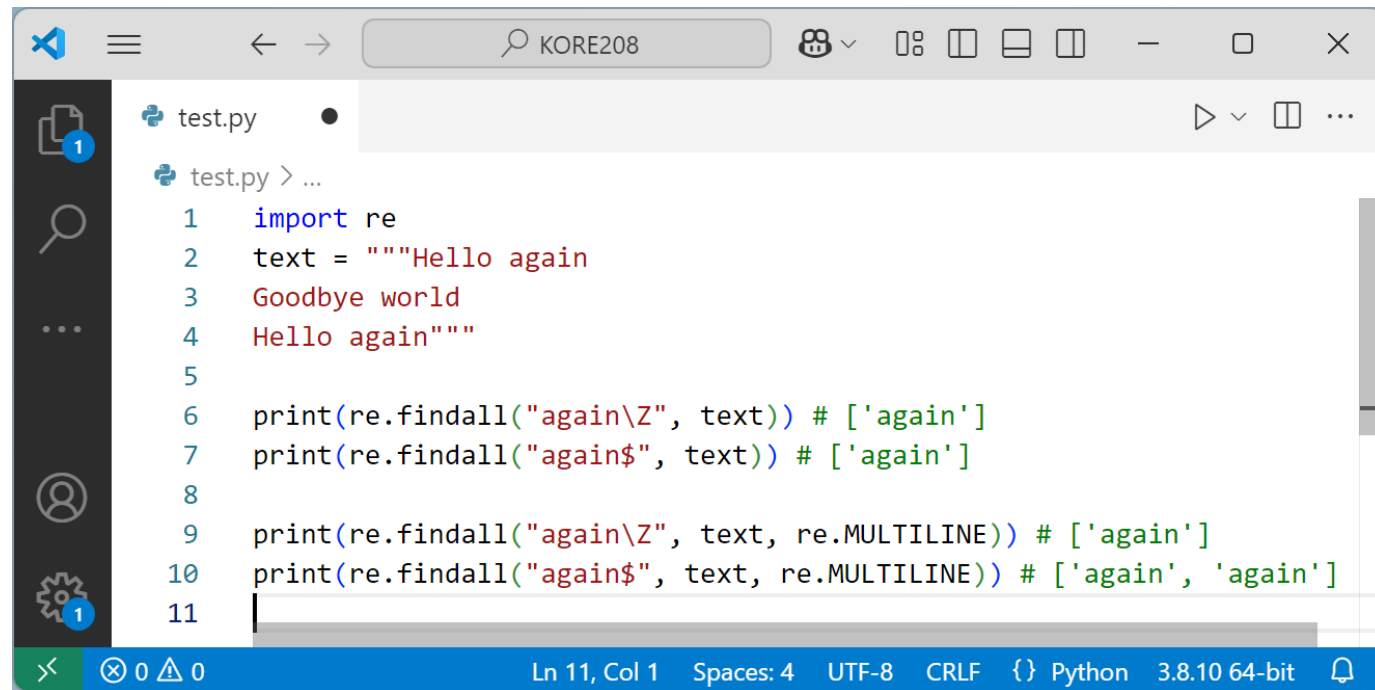
\Z

- \WA가 ^에 대응된다면,
WZ는 \$에 대응됨

- 즉 문자열 끝 표현함

- re 모듈의 re.MULTILINE
옵션을 사용하면 \$과 다른
의미를 갖도록 할 수 있음

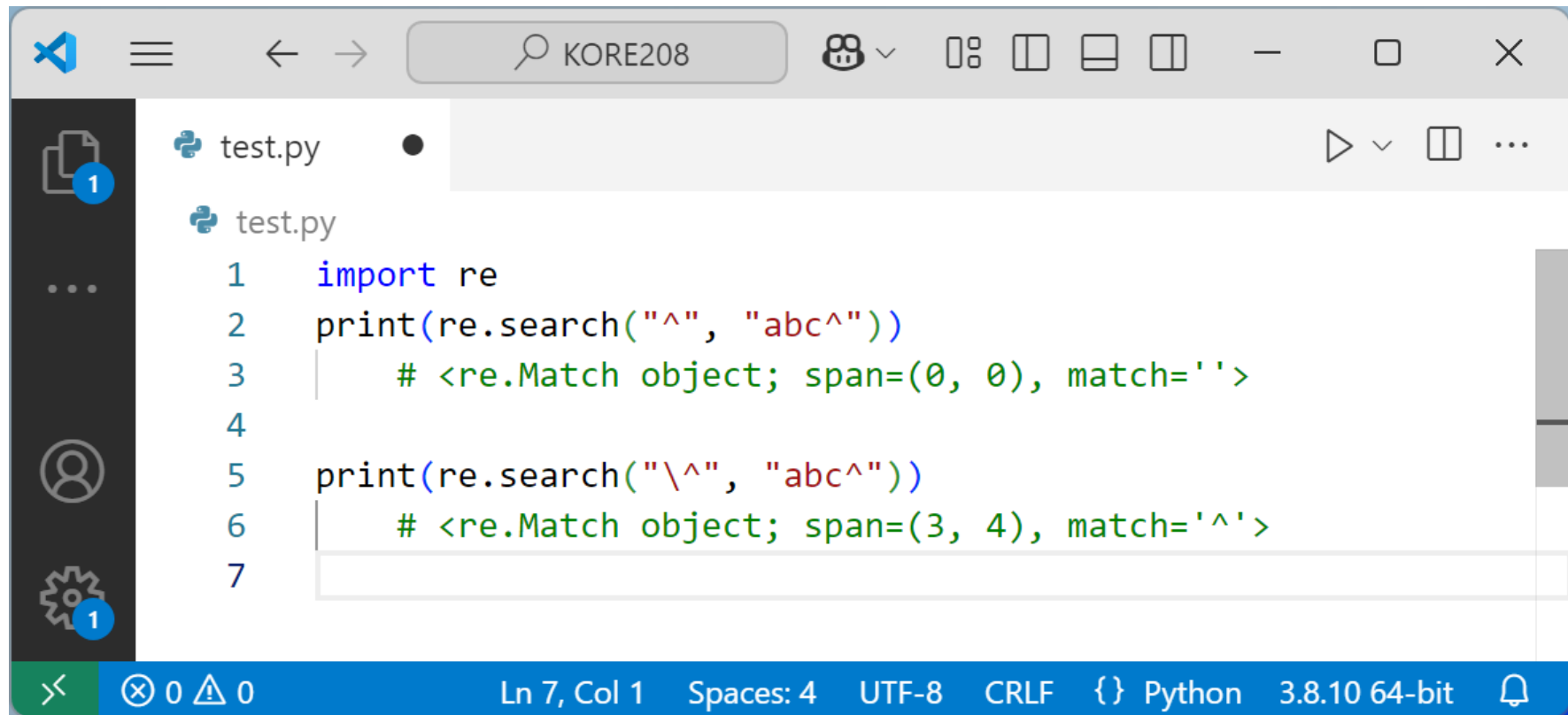
- ^은 각 행의 처음을 가리키고,
WA은 전체 문자열의 처음을
가리킴



```
test.py
1  import re
2  text = """Hello again
3  Goodbye world
4  Hello again"""
5
6  print(re.findall("again\Z", text)) # ['again']
7  print(re.findall("again$", text)) # ['again']
8
9  print(re.findall("again\Z", text, re.MULTILINE)) # ['again']
10 print(re.findall("again$", text, re.MULTILINE)) # ['again', 'again']
11
```

메타 문자를 문자 그 자체로 쓰는 방법

- 문자열의 이스케이프 코드에서처럼 앞에 `\` (백 슬래시)를 붙이면 됨

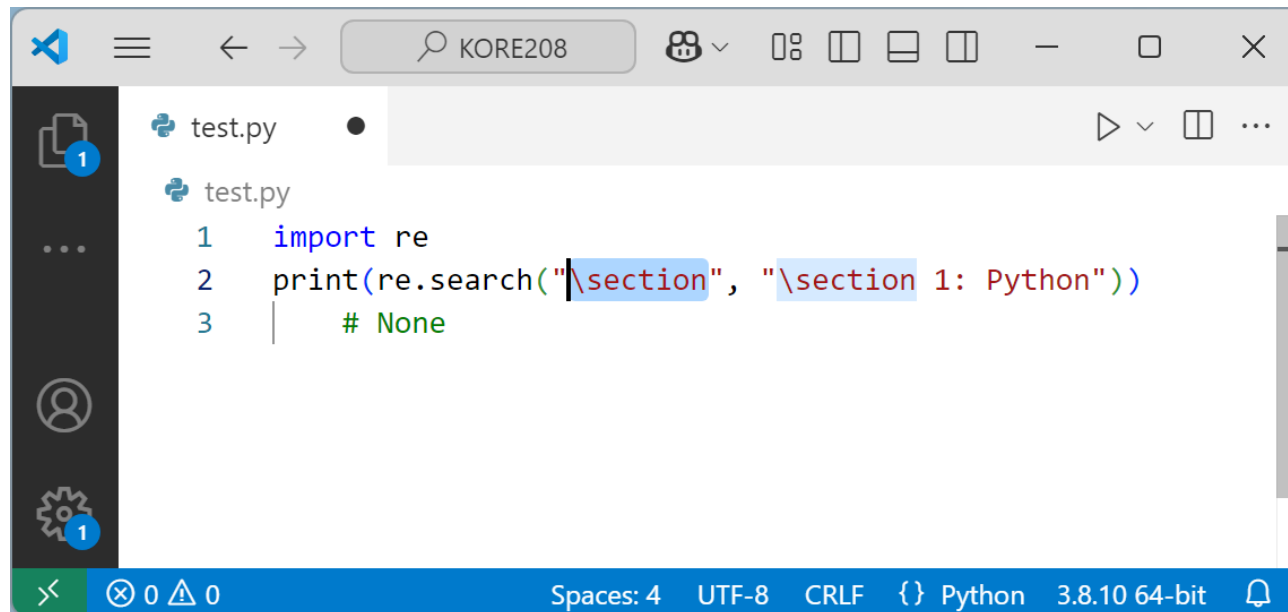


```
test.py
1  import re
2  print(re.search("^", "abc^"))
3      # <re.Match object; span=(0, 0), match=''>
4
5  print(re.search("\^", "abc^"))
6      # <re.Match object; span=(3, 4), match='^'>
7
```

Ln 7, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

백슬래시 문제 [1 / 3]

- 부분 문자열 'Wsection'을 찾기 위한 정규 표현식



```
test.py
1 import re
2 print(re.search("\\section", "\\section 1: Python"))
3 # None
```

- 정규 표현식 자체에 백슬래시(W)가 들어가면, 의도와 다르게 동작하는 경우가 있음

- 정규 표현식은 Wsection은 다음과 같은 의미임

- [\t\n\r\f\v]ection

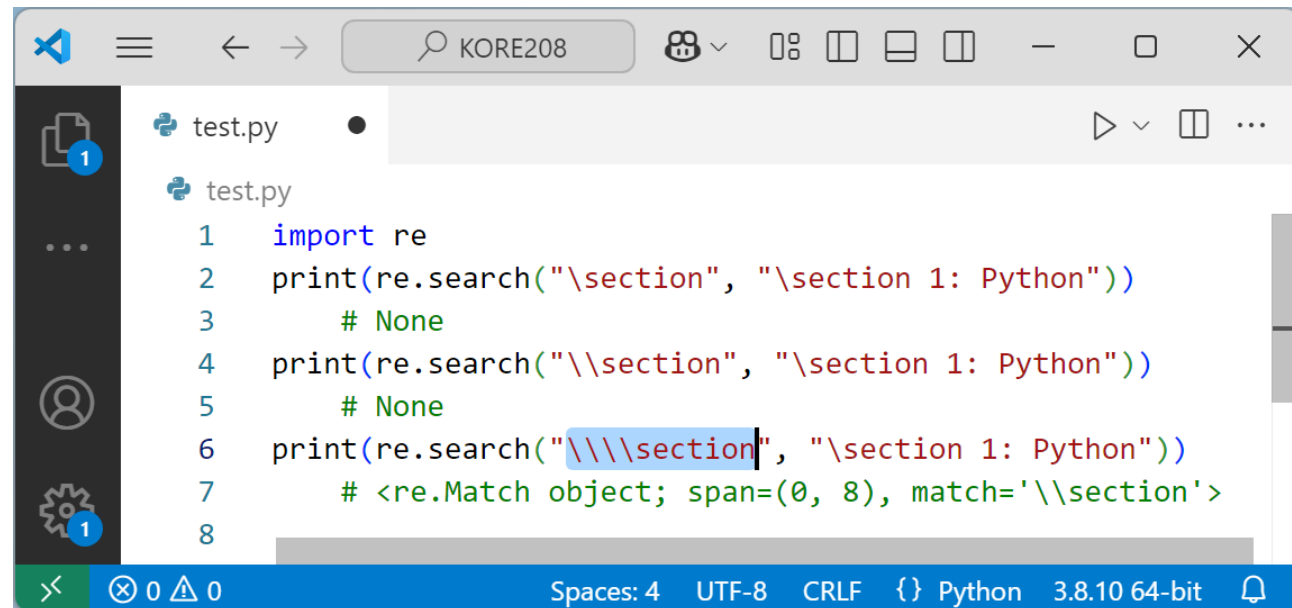
백슬래시 문제 [2/3]

▪ ₩를 2개 사용하면?

- 첫 번째 ₩를 이스케이프 문자, 두 번째 ₩를 문자 그대로의 백슬래시로 인식시키기
- Python 내부 규칙에 따라 ₩₩이 ₩로 바뀌어 컴파일이 이루어져 ₩₩로 쓰더라도 ₩가 하나 쓰인 것으로 인식함

▪ ₩₩₩₩와 같이 ₩를 4개 사용해야 ₩₩를 인식시킬 수 있음

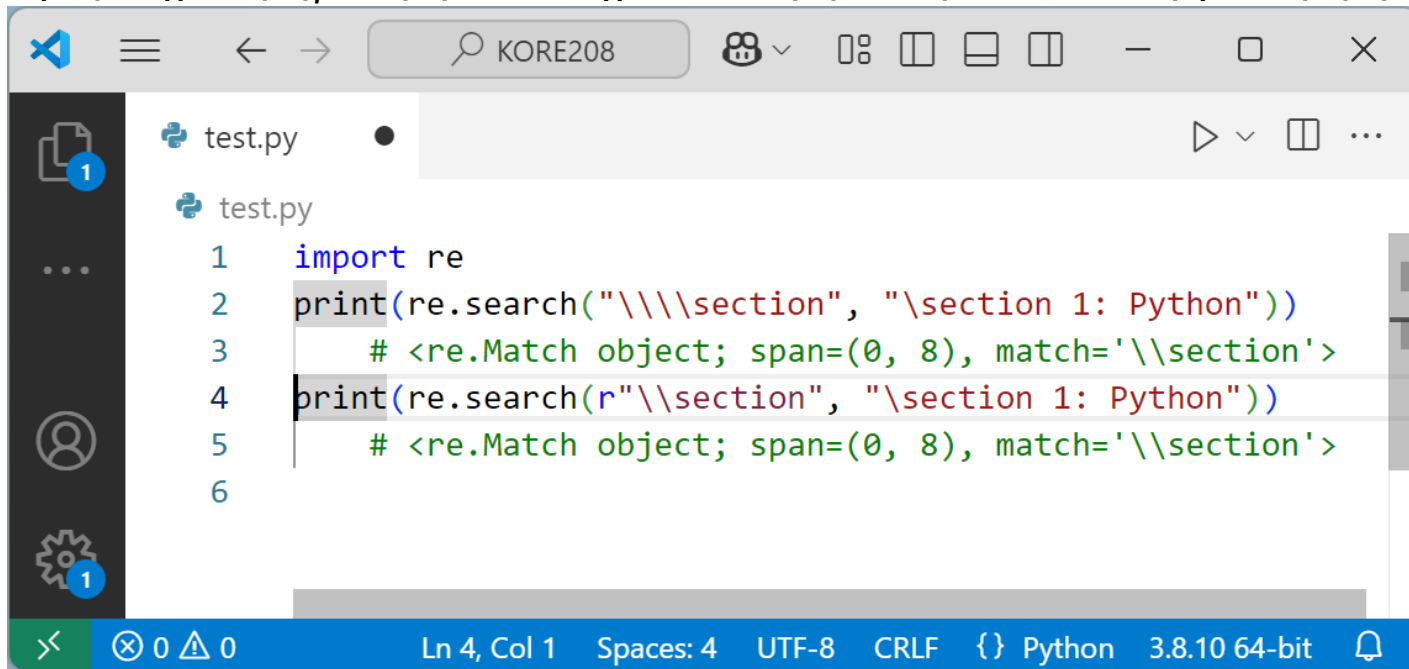
- 앞의 2개는 이스케이프 문자 ₩, 두 번째 2개는 문자 그대로의 ₩에 대응됨



```
test.py
1  import re
2  print(re.search("\section", "\section 1: Python"))
3      # None
4  print(re.search("\\section", "\section 1: Python"))
5      # None
6  print(re.search("\\\\section", "\section 1: Python"))
7      # <re.Match object; span=(0, 8), match='\\section'>
8
```

백슬래시 문제 [3/3]

- 앞의 방식은 `\\` 하나를 위해 너무 번잡한 정규 표현식을 쓰게 됨
 - `\\` 하나를 표기하기 위해 `\\`를 두 개 써야 함
- Python 내부 규칙을 무시하고 `\\`을 `r\\` 하나로 표기할 수 있도록 하려면, 정규식 앞에 `r` 기호를 쓰면 됨
 - `\\`가 쓰인 경우에만 유효하며, 쓰이지 않은 정규 표현식이라면 `r`이 있든 없든 아무런 차이가 없음



```
test.py
1 import re
2 print(re.search("\\\\section", "\\section 1: Python"))
3     # <re.Match object; span=(0, 8), match='\\section'>
4 print(re.search(r"\\section", "\\section 1: Python"))
5     # <re.Match object; span=(0, 8), match='\\section'>
6
```

5. re 모듈의 컴파일 옵션

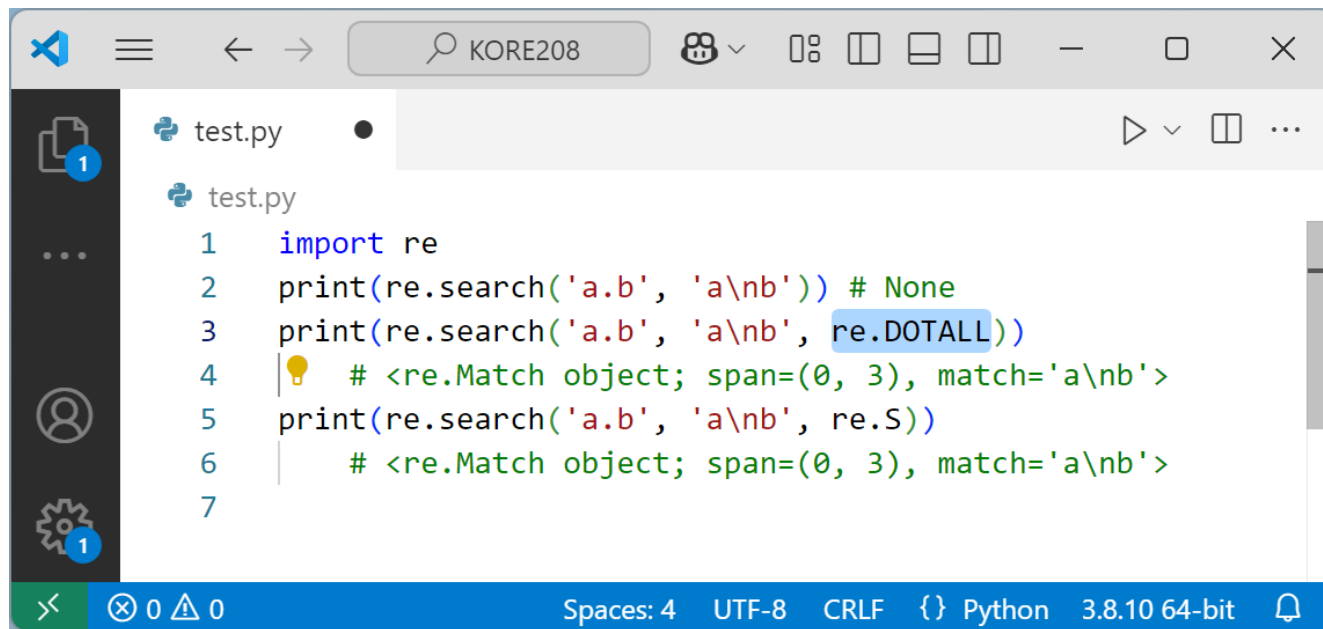
re.DOTALL

▪ 메타 문자 `.`가 줄바꿈 문자 `\n`까지 포함하도록 하는 옵션

○ 즉 모든 문자와 매칭되도록 함

▪ `re.S`로 축약할 수 있음

▪ 예



```
test.py
1  import re
2  print(re.search('a.b', 'a\nb')) # None
3  print(re.search('a.b', 'a\nb', re.DOTALL))
4  # <re.Match object; span=(0, 3), match='a\nb'>
5  print(re.search('a.b', 'a\nb', re.S))
6  # <re.Match object; span=(0, 3), match='a\nb'>
7
```

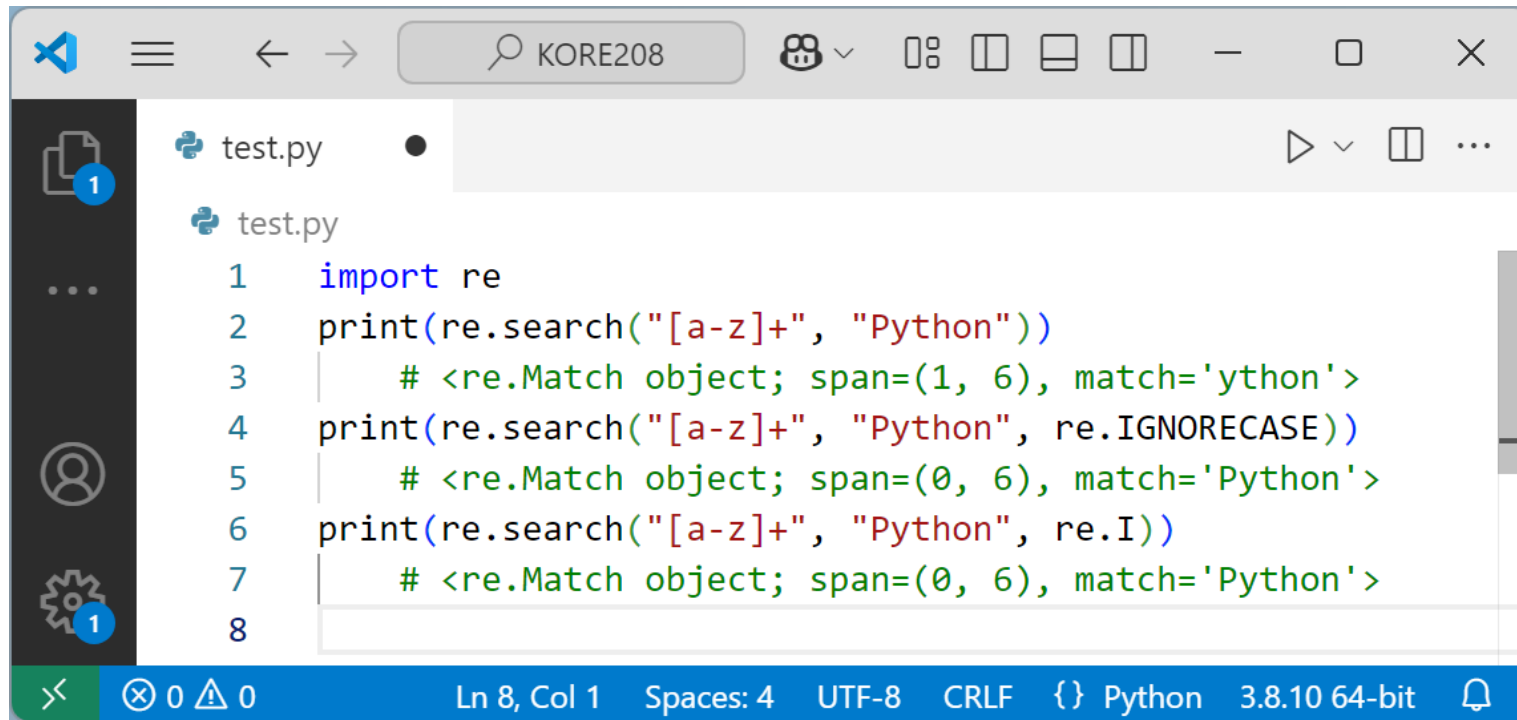
The screenshot shows a VS Code editor window with a file named 'test.py'. The code in the file demonstrates the use of the `re.DOTALL` and `re.S` flags. Line 2 shows a search for 'a.b' in 'a\nb' which returns `None`. Line 3 shows a search for 'a.b' in 'a\nb' with the `re.DOTALL` flag, which returns a match object. Line 5 shows a search for 'a.b' in 'a\nb' with the `re.S` flag, which also returns a match object. The status bar at the bottom indicates the editor is configured for Python 3.8.10 64-bit, with 4 spaces, UTF-8 encoding, and CRLF line endings.

re.IGNORECASE

- 대소문자 구별 없이 검색하는 옵션

- re.I로 축약할 수 있음

- 예



The screenshot shows a Visual Studio Code editor window with a file named 'test.py'. The code in the file demonstrates the use of the `re.IGNORECASE` flag (also known as `re.I`) for regular expression matching. The code is as follows:

```
1 import re
2 print(re.search("[a-z]+", "Python"))
3     # <re.Match object; span=(1, 6), match='ython'>
4 print(re.search("[a-z]+", "Python", re.IGNORECASE))
5     # <re.Match object; span=(0, 6), match='Python'>
6 print(re.search("[a-z]+", "Python", re.I))
7     # <re.Match object; span=(0, 6), match='Python'>
8
```

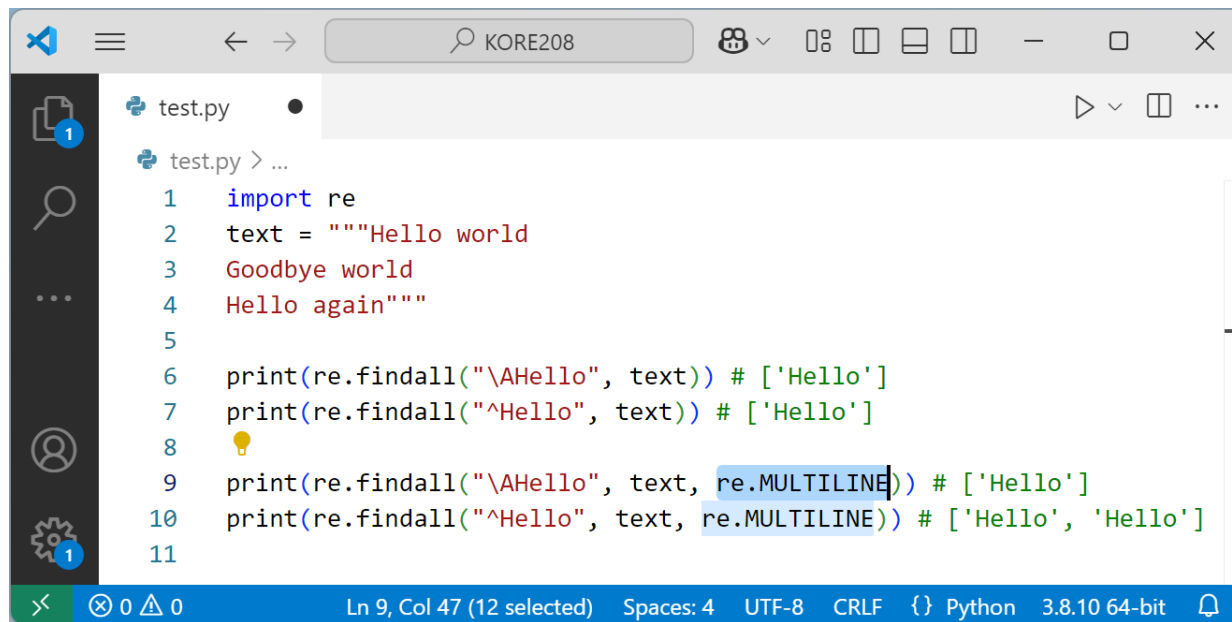
The status bar at the bottom of the editor indicates the current position is Line 8, Column 1, with 4 spaces, UTF-8 encoding, CRLF line endings, and the Python 3.8.10 64-bit interpreter.

re.MULTILINE

- 메타 문자 `^`과 `$`이 줄바꿈 문자로 구분되는 행의 시작과 끝을 표현하도록 하는 옵션

- `re.M`으로 축약할 수 있음

- 예



```
test.py
test.py > ...
1  import re
2  text = """Hello world
3  Goodbye world
4  Hello again"""
5
6  print(re.findall("\AHello", text)) # ['Hello']
7  print(re.findall("^Hello", text)) # ['Hello']
8
9  print(re.findall("\AHello", text, re.MULTILINE)) # ['Hello']
10 print(re.findall("^Hello", text, re.MULTILINE)) # ['Hello', 'Hello']
11
```

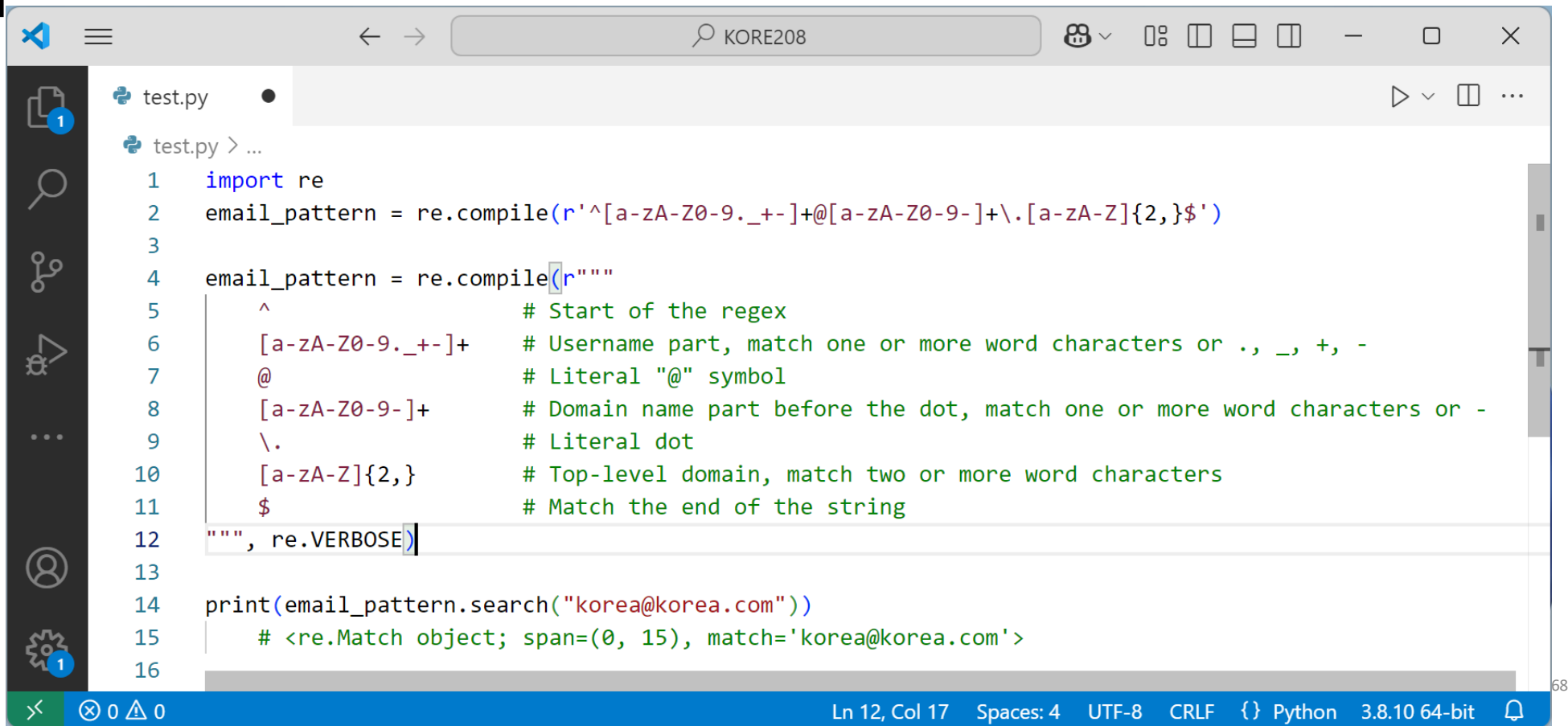
Ln 9, Col 47 (12 selected) Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

re.VERBOSE [1/2]

- 정규 표현식에 주석을 적고,
그것을 여러 행으로 나누어 작성할 수 있도록 하는 옵션
 - 정규 표현식이 분명 강력하고 편리하기는 하나,
표현하는 패턴이 복잡해질수록 독자가 이해하기 어려워짐
- re.X로 축약할 수 있음
- 정규 표현식 작성 시 사용된 Whitespace character는 컴파일 시 제거됨
 - 단, 문자 클래스 [] 안에 쓰인 것은 유지됨

re.VERBOSE [2/2]

■ 예



```
test.py
test.py > ...
1 import re
2 email_pattern = re.compile(r'^[a-zA-Z0-9._+-]+@[a-zA-Z0-9-]+\.[a-zA-Z]{2,}$')
3
4 email_pattern = re.compile(r"""
5     ^                # Start of the regex
6     [a-zA-Z0-9._+-]+  # Username part, match one or more word characters or ., _, +, -
7     @                # Literal "@" symbol
8     [a-zA-Z0-9-]+     # Domain name part before the dot, match one or more word characters or -
9     \.               # Literal dot
10    [a-zA-Z]{2,}       # Top-level domain, match two or more word characters
11    $                 # Match the end of the string
12 """, re.VERBOSE)
13
14 print(email_pattern.search("korea@korea.com"))
15     # <re.Match object; span=(0, 15), match='korea@korea.com'>
16
```

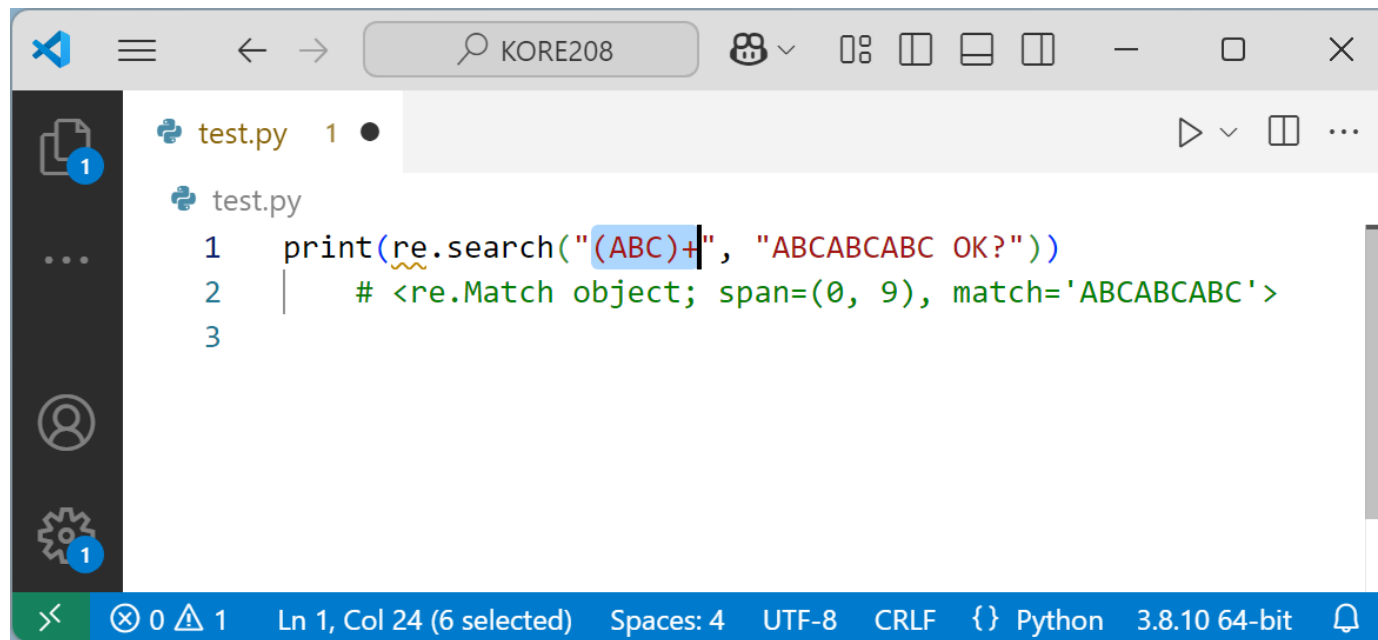
Ln 12, Col 17 Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

6. 정규 표현식 심화 학습

그루핑 [1/5]

- 복수의 문자로 이루어진 부분 문자열을 정규 표현식 내에서 다루기 위해 사용하는 기법
- 소괄호 ()로 묶음으로써 정규 표현식 내의 그룹을 만들 수 있음

▪ 예



The screenshot shows a Visual Studio Code editor window with a file named 'test.py'. The code in the editor is as follows:

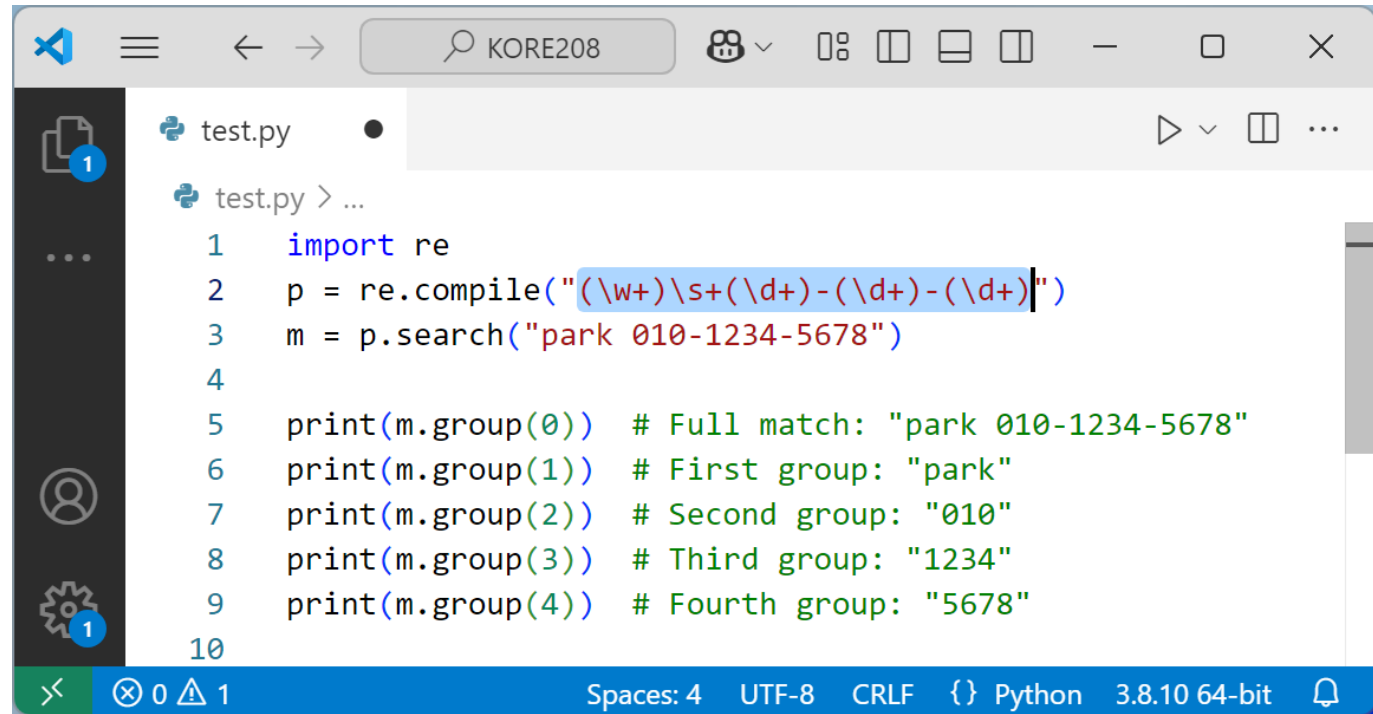
```
1 print(re.search("(ABC)+", "ABCABCABC OK?"))
2     # <re.Match object; span=(0, 9), match='ABCABCABC'>
3
```

The regular expression `(ABC)+` is highlighted in blue in the original image, indicating the use of a group. The status bar at the bottom shows the cursor is at line 1, column 24, with 6 characters selected. The status bar also indicates the file is a Python 3.8.10 64-bit script.

그루핑 [2/5]

■ 그룹 인덱싱

- 그룹 개수에는 제한이 없으며
각 그룹에 인덱싱 가능함
- 그룹 인덱스는 0이 아닌
1부터 시작함
- 인덱스 0은 그룹과 무관한,
정규 표현식에 매치된 부분
문자열 전체에 대응됨

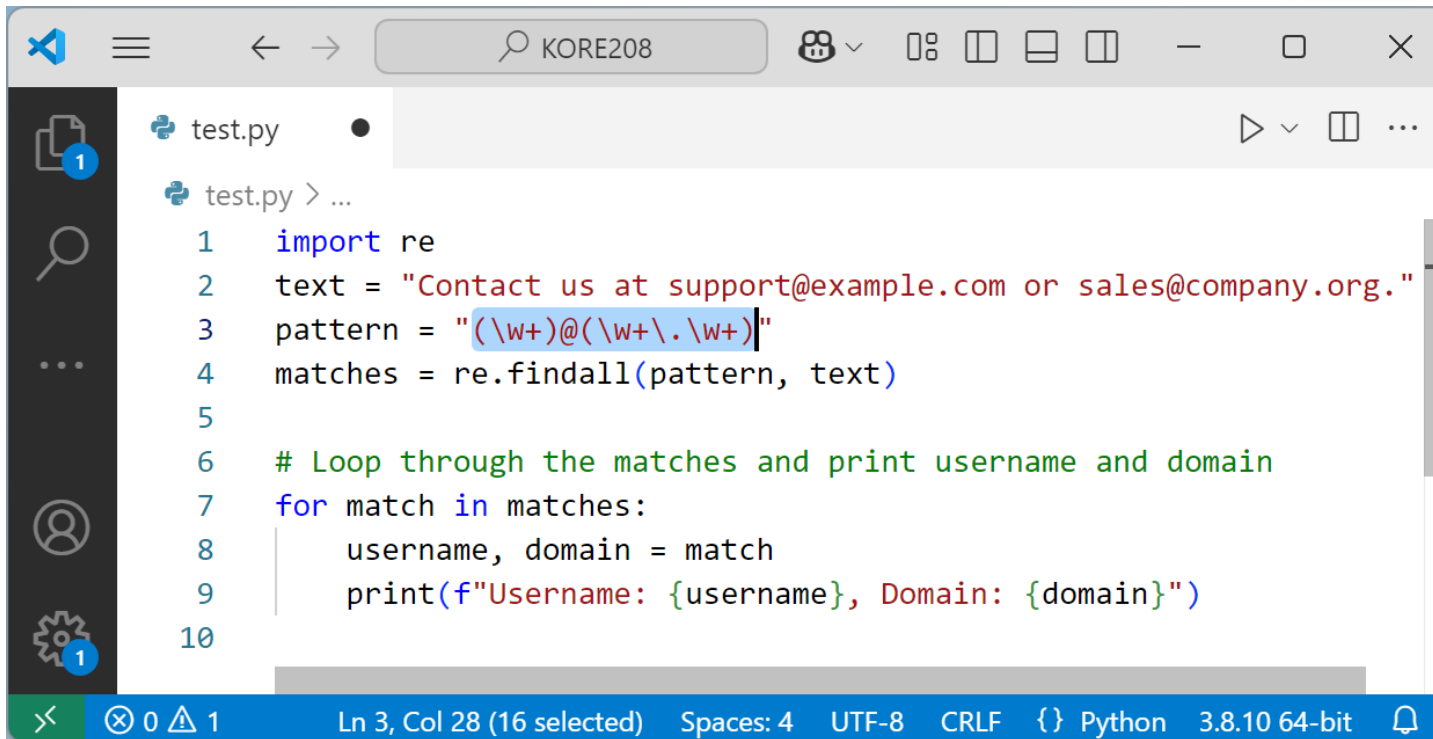


```
test.py
test.py > ...
1  import re
2  p = re.compile("(\w+)\s+(\d+)-(\d+)-(\d+)")
3  m = p.search("park 010-1234-5678")
4
5  print(m.group(0)) # Full match: "park 010-1234-5678"
6  print(m.group(1)) # First group: "park"
7  print(m.group(2)) # Second group: "010"
8  print(m.group(3)) # Third group: "1234"
9  print(m.group(4)) # Fourth group: "5678"
10
```

Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

그루핑 [3/5]

- 그루핑의 유용함 1: 매치된 문자열 중 특정 부분에 대한 접근 가능성
- 예: 이메일에서 username과 domain 부분 추출



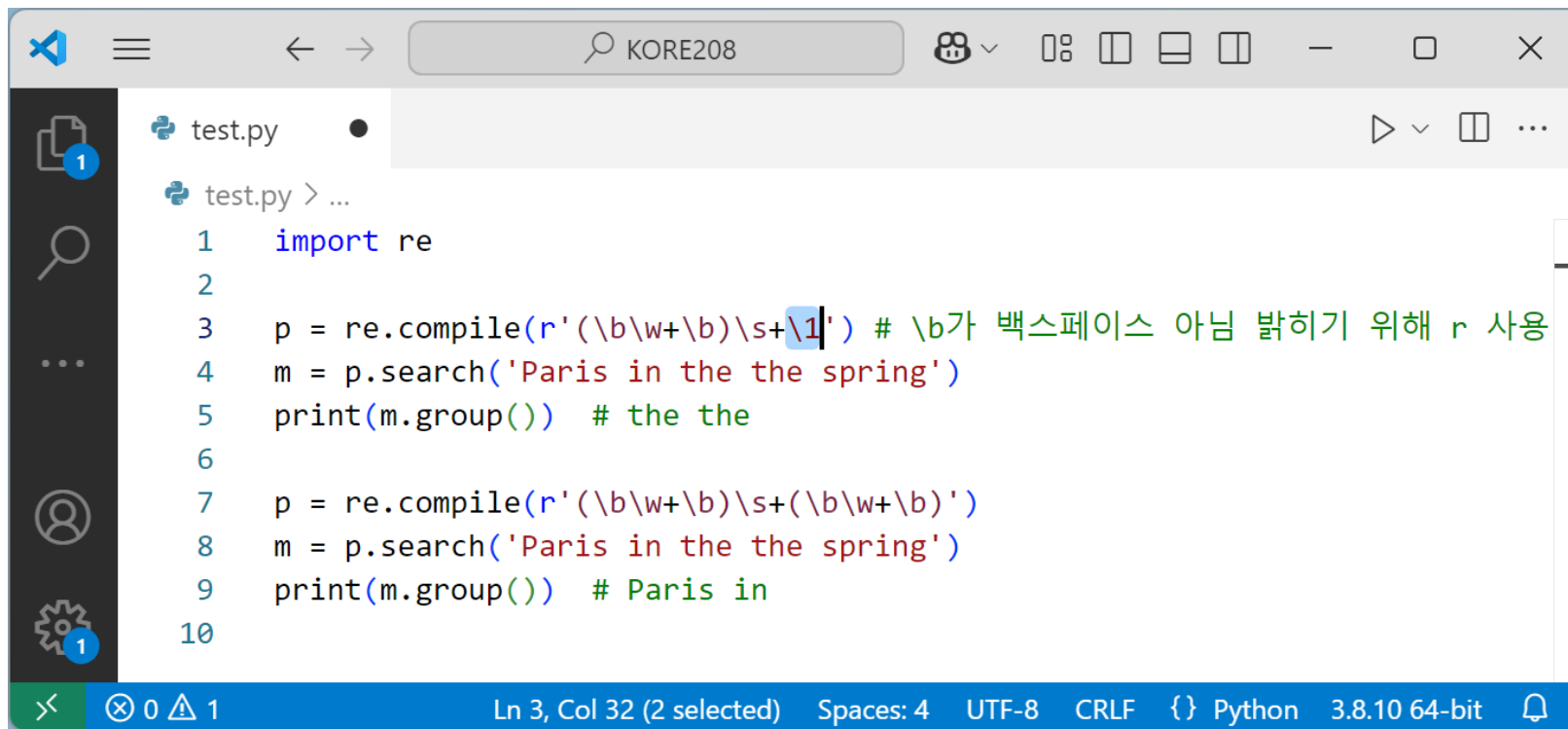
The screenshot shows a Visual Studio Code editor window with a file named 'test.py'. The code is a Python script that uses the 're' module to find all email addresses in a given text. The text is 'Contact us at support@example.com or sales@company.org.'. The script defines a regular expression pattern '(\w+)@(\w+\.\w+)' and uses 're.findall()' to find all matches. It then loops through the matches and prints the username and domain for each match. The status bar at the bottom indicates the current position is Line 3, Column 28, with 16 characters selected. The editor is running Python 3.8.10 64-bit.

```
1 import re
2 text = "Contact us at support@example.com or sales@company.org."
3 pattern = "(\w+)@(\w+\.\w+)"
4 matches = re.findall(pattern, text)
5
6 # Loop through the matches and print username and domain
7 for match in matches:
8     username, domain = match
9     print(f"Username: {username}, Domain: {domain}")
10
```


그루핑 [4/5]

■ 그루핑의 유용함 2: 그룹 인덱싱을 이용한 코드 간결화 및 고도화

○예: 그룹을 `\1`로 인덱싱함으로써, 동일한 단어가 연달아 나타나는 패턴을 표현

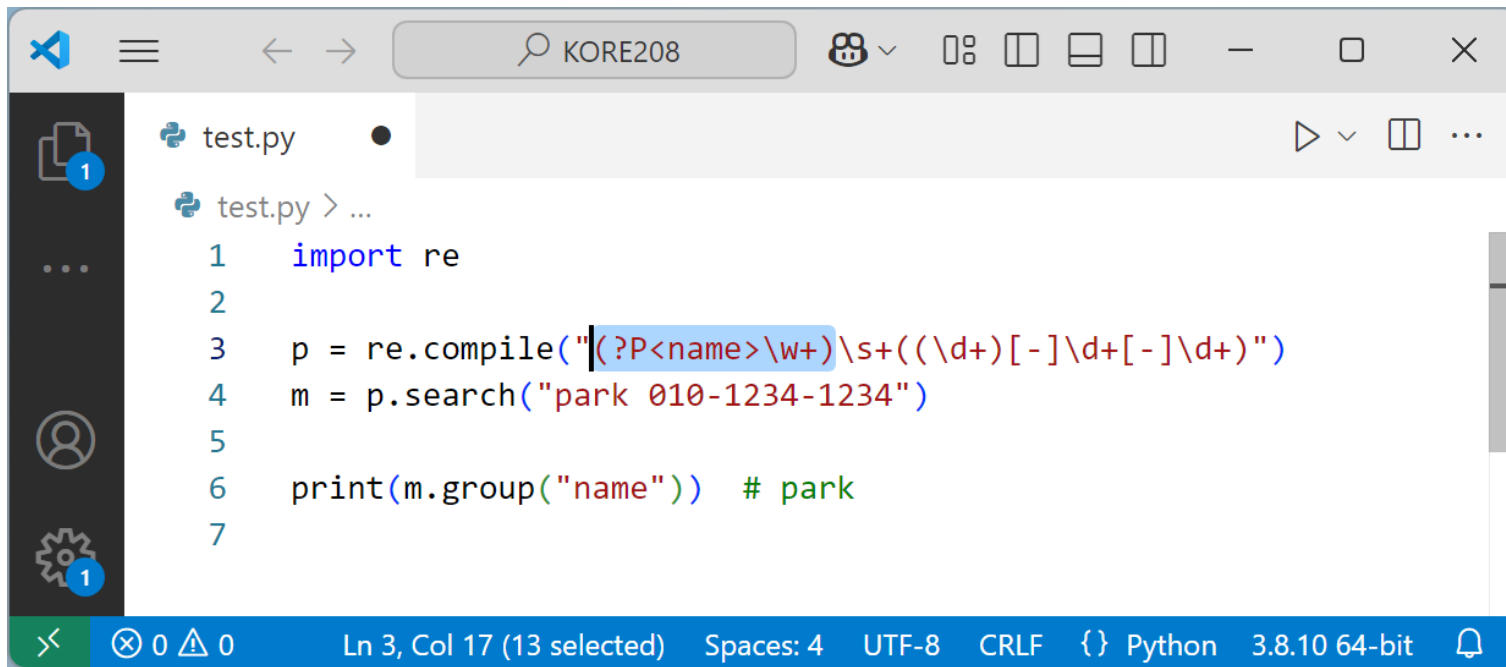


```
test.py
test.py > ...
1  import re
2
3  p = re.compile(r'(\b\w+\b)\s+\1') # \b가 백스페이스 아님 밝히기 위해 r 사용
4  m = p.search('Paris in the the spring')
5  print(m.group()) # the the
6
7  p = re.compile(r'(\b\w+\b)\s+(\b\w+\b)')
8  m = p.search('Paris in the the spring')
9  print(m.group()) # Paris in
10
```

Ln 3, Col 32 (2 selected) Spaces: 4 UTF-8 CRLF {} Python 3.8.10 64-bit

그루핑 [5/5]

- 그루핑의 유용함 3: 그룹에 이름을 붙임으로써 가독성 높일 수 있음
 - (?P<그룹명>패턴)
- 예: 앞서 단순히 (\w+)로 표현했던 이름 그룹에 ?P<name>을 통해 이름을 붙임



```
test.py
test.py > ...
1  import re
2
3  p = re.compile("(?P<name>\w+)\s+((\d+)[-]\d+[-]\d+)"
4  m = p.search("park 010-1234-1234")
5
6  print(m.group("name")) # park
7
```

The screenshot shows a Visual Studio Code editor window with a file named 'test.py'. The code in the editor is a Python script that demonstrates the use of a named group in a regular expression. The script imports the 're' module, compiles a regular expression pattern, searches for it in a string, and prints the result. The regular expression pattern is `(?P<name>\w+)\s+((\d+)[-]\d+[-]\d+)`, where `(?P<name>\w+)` is a named group for the name. The string being searched is `"park 010-1234-1234"`. The output of the script is `park`. The status bar at the bottom indicates the current position is Line 3, Column 17, with 13 characters selected. The editor is configured for Python 3.8.10 64-bit.

전방 탐색: 긍정형 전방 탐색 [1 / 3]

- 패턴 1 뒤에 패턴 2가 이어져야 하나, 매치 문자열에서는 패턴 1만 포함시키고 패턴 2는 배제하고 싶은 경우에 사용
 - 여기서 패턴 1은 소비, 패턴 2는 소비되지 않음
- 사용 방법
 - (?=패턴)

전방 탐색: 긍정형 전방 탐색 [2/3]

■문제 상황 예

○url에서 프로토콜 문자열 부분만 추출하고자 하는 경우

- `http://google.com`

○프로토콜은 url 처음부터 나타나며 이후 부분의 경계에는 `://...`이 위치함

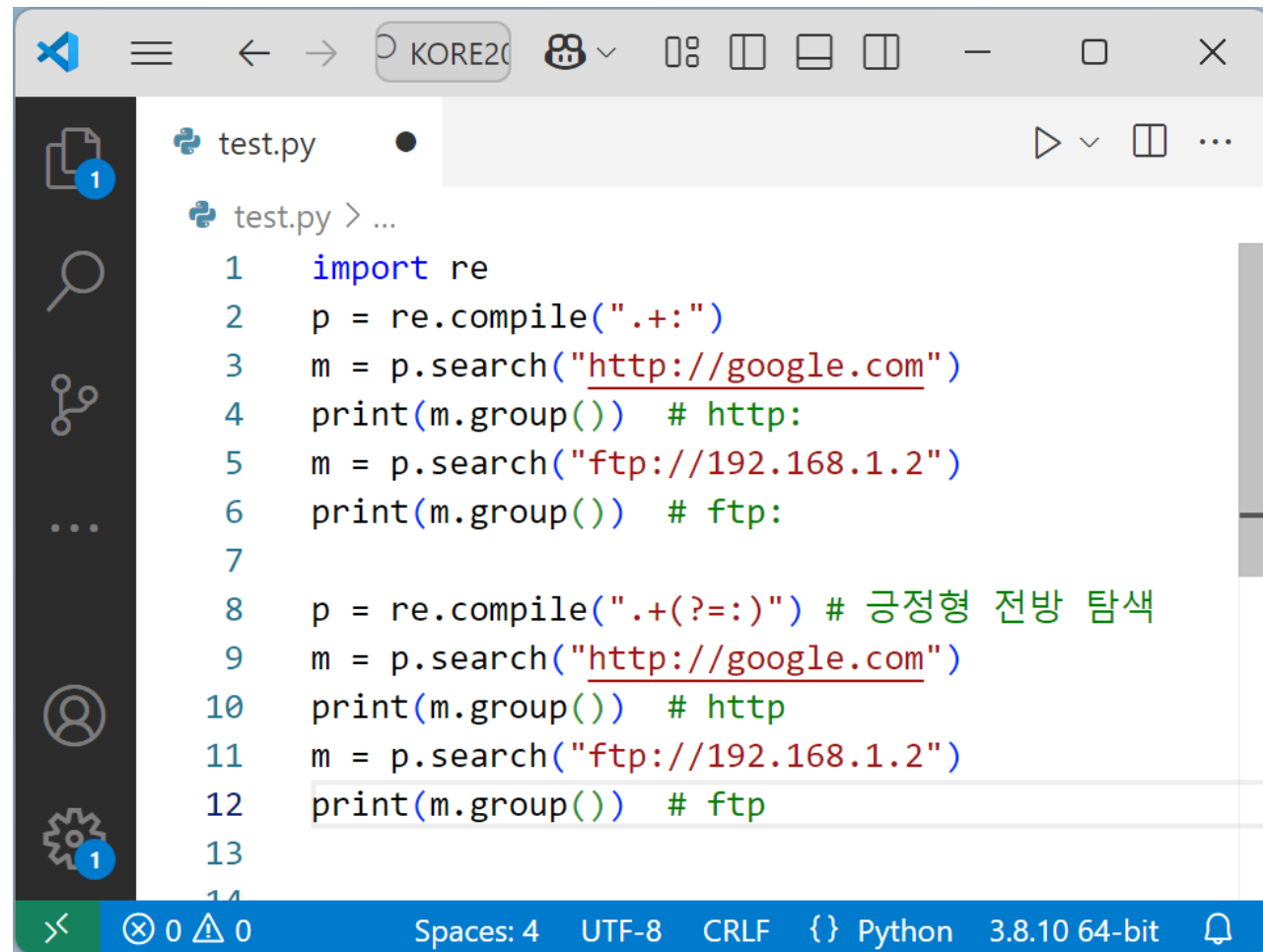
○여기서 `:`은 프로토콜의 끝부분임을 파악할 수 있게 하는 표지이나,

프로토콜 문자열 자체에 포함되지는 않으므로 매치 문자열에서 배제해야 함

전방 탐색: 긍정형 전방 탐색 [3/3]

■ 긍정형 전방 탐색을 통한 해결

- 긍정형 전방 탐색을 사용하면
검색 시에는 :을 활용하나
매치 문자열에서는 배제할 수
있음



```
test.py
test.py > ...
1  import re
2  p = re.compile(".*+:")
3  m = p.search("http://google.com")
4  print(m.group()) # http:
5  m = p.search("ftp://192.168.1.2")
6  print(m.group()) # ftp:
7
8  p = re.compile(".*+(?=:)") # 긍정형 전방 탐색
9  m = p.search("http://google.com")
10 print(m.group()) # http
11 m = p.search("ftp://192.168.1.2")
12 print(m.group()) # ftp
13
14
```

VS Code interface showing the file explorer on the left with a search icon and a settings icon. The editor displays the Python code. The status bar at the bottom indicates 'Spaces: 4', 'UTF-8', 'CRLF', '{ }', 'Python', '3.8.10 64-bit', and a bell icon.

전방 탐색: 부정형 전방 탐색 [1 / 3]

- 패턴 1 뒤에 패턴 2가 이어지지 않아야 하며, 매치 문자열에서는 패턴 1만 포함시키고자 하는 경우에 사용
 - 긍정형에서와 마찬가지로 패턴 1은 소비, 패턴 2는 소비되지 않음
- 사용 방법
 - (?!패턴)

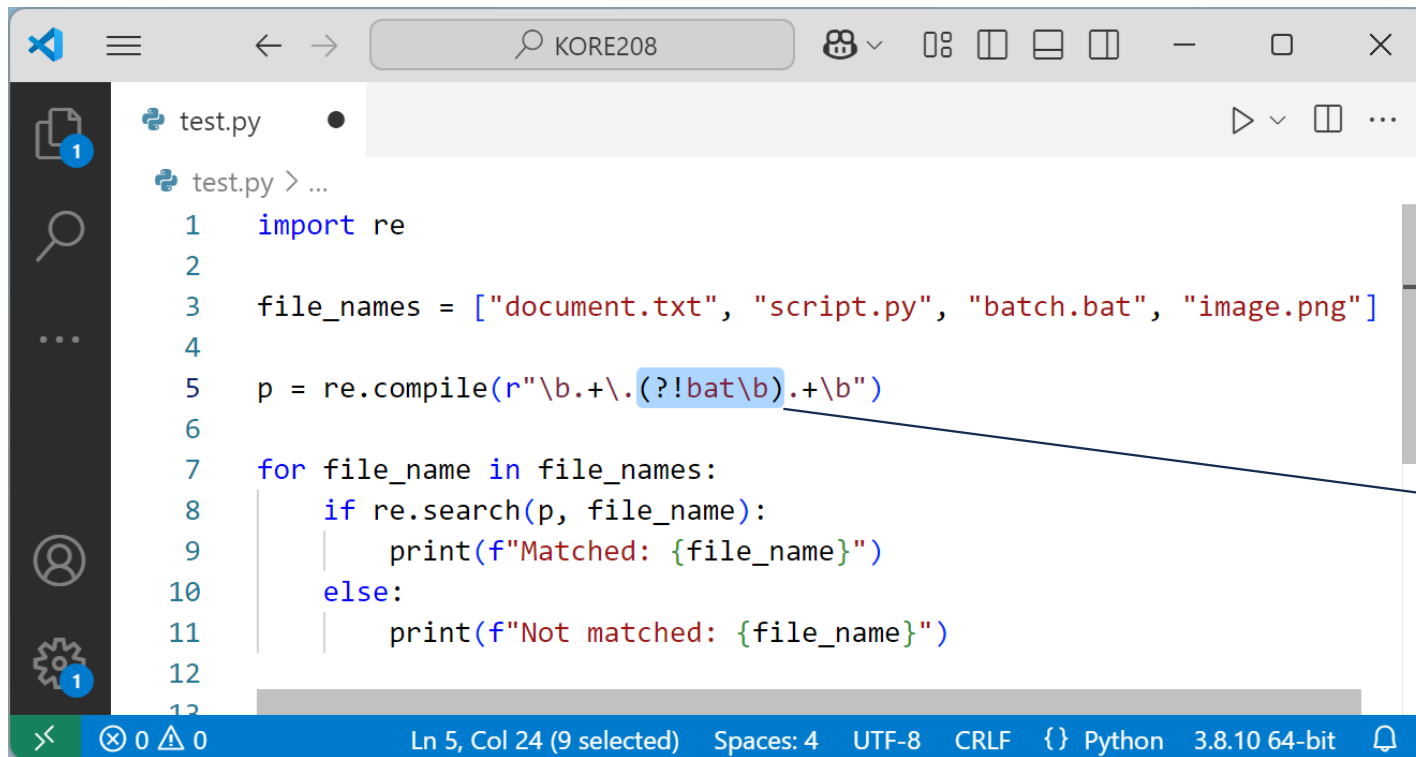
전방 탐색: 부정형 전방 탐색 [2/3]

■ 문제 상황 예

- 파일 확장자가 bat인 것을 제외한 것들에 대해서만 파일명 추출하고자 하는 경우
 - `file_names = ["document.txt", "script.py", "batch.bat", "image.png"]`
- 파일명의 패턴
 - *파일_이름 + . + 확장자*
 - `p = re.compile(r"\b.+\.+\.b")`

전방 탐색: 부정형 전방 탐색 [3/3]

■ 부정형 전방 탐색을 통한 해결



```
test.py
1 import re
2
3 file_names = ["document.txt", "script.py", "batch.bat", "image.png"]
4
5 p = re.compile(r"\b.+\. (? !bat\b) .+ \b")
6
7 for file_name in file_names:
8     if re.search(p, file_name):
9         print(f"Matched: {file_name}")
10    else:
11        print(f"Not matched: {file_name}")
12
```

- 어떤 문자열에서 이 패턴이 나타나면, 그 부분은 매치되지 않는 것으로 취급됨
- 어떤 문자열에서 이 패턴이 나타나지 않으면, 이 정규 표현식 부분은 소비되지 않으므로 없는 것과 마찬가지임
즉 이 경우 전체 정규 표현식은 `\b.+\. .+ \b`과 같은 기능을 함

Greedy와 Non-Greedy [1 / 2]

- 복수의 문자들과 매치될 수 있는 메타 문자는 기본적으로 가능한 한 탐욕적으로 문자열을 소비함
 - 이러한 메타 문자들을 Quantifier라고 함
 - * + ? {m,n}
 - <가 나타난 후, 최초의 >은 <html>에서 나타나나, 문자열 뒤쪽에도 >이 계속 나타나므로, 다음의 정규 표현식은 최후의 >이 나타나는 부분까지 매치됨



```
test.py
test.py > ...
1  import re
2
3  s = '<html><head><title>Title</title>'
4  print(re.match('<.*>', s))
5  # <re.Match object; span=(0, 32), match='<html><head><title>Title</title>'>
6
```

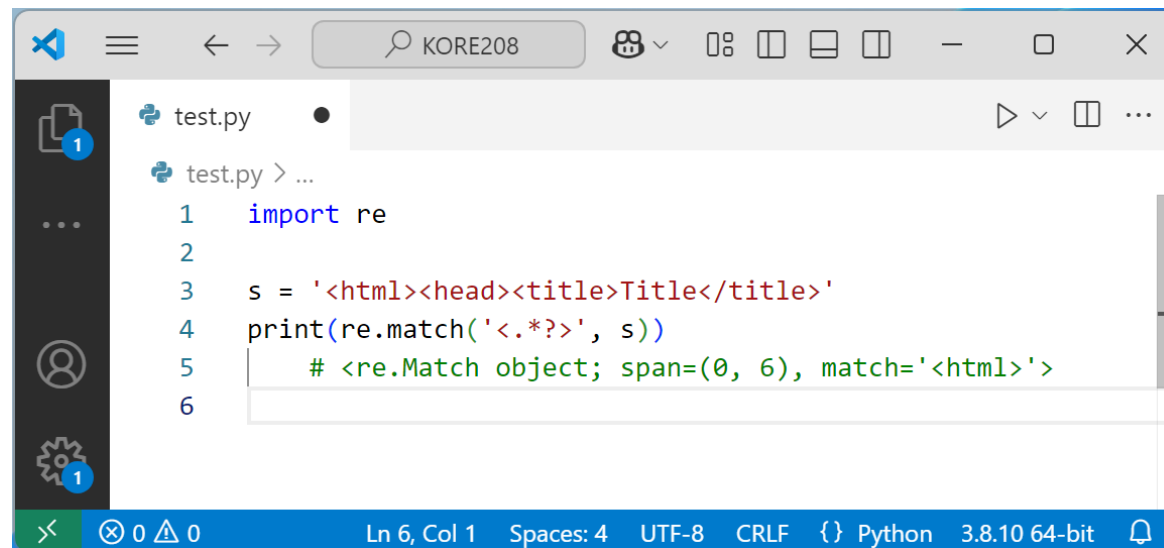
Ln 6, Col 5 Spaces: 4 UTF-8 CRLF { } Python 3.8.10 64-bit

Greedy와 Non-Greedy [2/2]

- Non-Greedy 문자인 ?를 사용하면 이러한 탐욕을 억제하고 소비를 최소한으로 할 수 있음

- Quantifier 뒤에 ?를 붙임

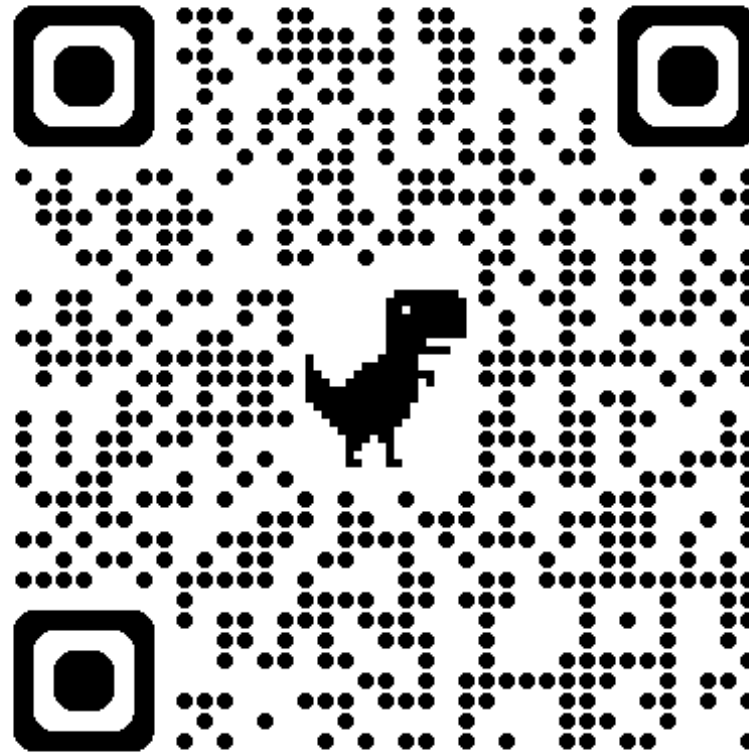
- *? +? ?? {m,n}?



```
test.py
test.py > ...
1  import re
2
3  s = '<html><head><title>Title</title>'
4  print(re.match('<.*?>', s))
5  # <re.Match object; span=(0, 6), match='<html>'>
6
```

The screenshot shows a Visual Studio Code window with a file named 'test.py'. The code in the editor is a Python script that imports the 're' module, defines a string 's' containing HTML tags, and uses 're.match' with a non-greedy regex pattern '<.*?>'. The output comment shows that the match successfully finds the first opening tag '<html>'.

설문 (미응답자만 응답하기)



- <https://forms.gle/boJBKd51uM6WbdfL8>