
정규식 HOWTO

출시 버전 3.10.18

Guido van Rossum
and the Python development team

7월 08, 2025

Contents

1	소개	2
2	단순한 패턴	2
2.1	문자 일치	2
2.2	반복하기	3
3	정규식 사용하기	4
3.1	정규식 컴파일하기	4
3.2	백 슬래시 전염병	5
3.3	일치 수행하기	6
3.4	모듈 수준 함수	8
3.5	컴파일 플러그	8
4	더 많은 패턴 기능	10
4.1	메타 문자 더 보기	10
4.2	그룹	11
4.3	비 포착 그룹과 이름 있는 그룹	12
4.4	미리 보기 어서션	14
5	문자열 수정하기	15
5.1	문자열 분할하기	15
5.2	검색과 치환	16
6	일반적인 문제들	17
6.1	문자열 메서드를 사용하십시오	17
6.2	match()와 search() 비교	18
6.3	탐욕적 과 비 탐욕적 비교	18
6.4	re.VERBOSE 사용하기	19
7	피드백	19

저자 A.M. Kuchling <amk@amk.ca>

요약

이 설명서는 파이썬에서 `re` 모듈로 정규식을 사용하는 방법을 소개하는 입문서입니다. 라이브러리 레퍼런스의 해당 절보다 더 부드러운 소개를 제공합니다.

1 소개

정규식(RE, *regexes* 또는 *regex* 패턴이라고 불립니다)은 본질적으로 파이썬에 내장된 매우 작고 고도로 특수화된 프로그래밍 언어이며, `re` 모듈을 통해 사용할 수 있습니다. 이 작은 언어를 사용하여, 일치시키려는 가능한 문자열 집합에 대한 규칙을 지정합니다; 이 집합은 영어 문장, 전자 메일 주소, TeX 명령 또는 원하는 어떤 것이건 포함 할 수 있습니다. 그런 다음 “이 문자열이 패턴과 일치합니까?”, 또는 “이 문자열의 어느 부분에 패턴과 일치하는 것이 있습니까?”와 같은 질문을 할 수 있습니다. 또한 RE를 사용하여 문자열을 수정하거나 여러 방법으로 분할할 수 있습니다.

정규식 패턴은 일련의 바이트 코드로 컴파일된 다음 C로 작성된 일치 엔진에 의해 실행됩니다. 고급 사용을 위해서는, 엔진이 지정된 RE를 실행하는 방법에 주의를 기울이고, 더 빠르게 실행되는 바이트 코드를 생성하기 위해 특정한 방법으로 RE를 작성하는 것이 필요할 수 있습니다. 최적화는 일치 엔진의 내부를 잘 이해하고 있어야 하므로 이 설명서에서 다루지 않습니다.

정규식 언어는 비교적 작고 제한적이므로, 정규식을 사용하여 가능한 모든 문자열 처리 작업을 수행할 수 있는 것은 아닙니다. 정규식으로 수행할 수는 있지만, 표현이 아주 복잡해지는 작업도 있습니다. 이럴 때, 처리하기 위한 파이썬 코드를 작성하는 것이 더 나을 수 있습니다; 파이썬 코드는 정교한 정규식보다 느리겠지만, 아마도 더 이해하기 쉬울 겁니다.

2 단순한 패턴

우리는 가능한 가장 단순한 정규식에 대해 배우는 것으로 시작합니다. 정규식은 문자열에 대한 연산에 사용되므로, 가장 일반적인 작업으로 시작하겠습니다: 문자 일치.

정규식의 기초가 되는 컴퓨터 과학(결정적인 혹은 비결정적인 유한 오토마타)에 대한 자세한 설명은, 컴파일러 작성에 관한 거의 모든 교과서를 참조 할 수 있습니다.

2.1 문자 일치

대부분 글자와 문자는 단순히 자신과 일치합니다. 예를 들어, 정규식 `test`는 문자열 `test`와 정확히 일치합니다. (이 RE가 `Test` 나 `TEST`와 일치하도록 대/소문자를 구분하지 않는 모드를 활성화할 수 있습니다; 나중에 자세히 설명합니다.)

이 규칙에는 예외가 있습니다; 일부 문자는 특수한 메타 문자(*metacharacters*)이며, 자신과 일치하지 않습니다. 그 대신, 그들은 일반적으로 그렇지 않은 것을 일치시켜야 한다는 신호를 보냅니다. 또는 반복하거나 의미를 바꾸어 RE의 다른 부분에 영향을 줍니다. 이 설명서의 많은 부분은 다양한 메타 문자와 그 기능에 대해 논의하는데 할애하고 있습니다.

다음은 메타 문자의 전체 목록입니다; 이것들의 의미는 이 HOWTO의 나머지 부분에서 논의될 것입니다.

```
. ^ $ * + ? { } [ ] \ | ( )
```

우리가 살펴볼 첫 번째 메타 문자는 `[와]` 입니다. 일치시키려는 문자 집합인 문자 클래스를 지정하는 데 사용됩니다. 문자는 개별적으로 나열되거나, 두 문자를 주고 '-'로 구분하여 문자의 범위를 나타낼 수 있습니다. 예를 들어, `[abc]`는 `a`, `b` 또는 `c` 문자와 일치합니다; 이것은 `[a-c]`와 같은데, 같은 문자 집합을 표현하기 위해 범위를 사용합니다. 소문자들만 일치시키려면, RE가 `[a-z]`가 됩니다.

메타 문자(\ 제외)는 클래스 내부에서는 활성화되지 않습니다. 예를 들어, `[akm$]`는 'a', 'k', 'm' 또는 '\$' 문자와 일치합니다; '\$'는 대개 메타 문자이지만, 문자 클래스 안에서는 특수한 특성이 없어집니다.

여집합 (*complement set*)을 사용해서 클래스에 나열되지 않은 문자를 일치시킬 수 있습니다. 이것은 클래스의 첫 번째 문자로 '^'를 포함하는 것으로 나타냅니다. 예를 들어, `[^5]`는 '5'를 제외한 모든 문자와 일치합니다. 캐럿이 문자 클래스의 다른 곳에 나타나면, 특별한 의미가 없습니다. 예를 들어, `[5^]`는 '5'나 '^'와 일치합니다.

아마도 가장 중요한 메타 문자는 백 슬래시(\)입니다. 파이썬 문자열 리터럴에서와 같이, 백 슬래시 다음에 다양한 특수 시퀀스를 알리는 다양한 문자가 따라올 수 있습니다. 또한, 모든 메타 문자를 이스케이프 처리하여 패턴으로 일치시킬 수 있도록 합니다. 예를 들어, `[나\]`와 일치시켜야 할 때, 특별한 의미를 제거하기 위해 앞에 백 슬래시를 붙일 수 있습니다: `[나\\]`.

'\'로 시작하는 특수 시퀀스 중 일부는 숫자(digit) 집합, 글자(letter) 집합 또는 공백이 아닌 모든 것의 집합과 같이 종종 유용한 미리 정의된 문자 집합을 나타냅니다.

예를 들어 보겠습니다: `\w`는 모든 영숫자(alphanumeric character)와 일치합니다. 정규식 패턴을 바이트열로 표현하면, 이것은 `[a-zA-Z0-9_]` 클래스와 동등합니다. 정규식 패턴이 문자열이면, `\w`는 `unicodedata` 모듈이 제공하는 유니코드 데이터베이스에서 글자(letter)로 표시된 모든 문자를 일치시킵니다. 정규식을 컴파일할 때 `re.ASCII` 플래그를 제공하여 문자열 패턴에서 `\w`의 더 제한된 정의를 사용할 수 있습니다.

다음 특수 시퀀스 목록은 완전하지 않습니다. 유니코드 문자열 패턴에 대한 시퀀스와 확장 클래스 정의의 전체 목록은, 표준 라이브러리 레퍼런스에서 정규식 문법의 마지막 부분을 참조하십시오. 일반적으로, 유니코드 버전은 유니코드 데이터베이스의 적절한 범주에 있는 모든 문자와 일치합니다.

`\d` 모든 십진 숫자와 일치합니다; 이것은 클래스 `[0-9]`와 동등합니다.

`\D` 모든 비 숫자 문자와 일치합니다; 이것은 클래스 `[^0-9]`와 동등합니다.

`\s` 모든 공백 문자와 일치합니다; 이것은 클래스 `[\t\n\r\f\v]`와 동등합니다.

`\S` 모든 비 공백 문자와 일치합니다; 이것은 클래스 `^[^ \t\n\r\f\v]`와 동등합니다.

`\w` 모든 영숫자(alphanumeric character)와 일치합니다; 이것은 클래스 `[a-zA-Z0-9_]`와 동등합니다.

`\W` 모든 비 영숫자와 일치합니다; 이것은 클래스 `^[a-zA-Z0-9_]`와 동등합니다.

이 시퀀스들은 문자 클래스 내에 포함될 수 있습니다. 예를 들어, `[\s,.]`는 모든 공백 문자, ',', ' 또는 '.'와 일치하는 문자 클래스입니다.

이 절의 마지막 메타 문자는 .입니다. 개행 문자를 제외한 모든 문자와 일치하며, 개행 문자와도 일치하는 대체 모드(`re.DOTALL`)가 있습니다. .은 “모든 문자”와 일치시키려고 할 때 자주 사용됩니다.

2.2 반복하기

다양한 문자 집합을 일치시킬 수 있다는 것이 문자열에서 사용할 수 있는 메서드로 이미 가능하지 않은 것을 정규식이 수행할 수 있는 첫 번째 것입니다. 그러나, 이것이 정규식의 유일한 추가 기능이라면, 그다지 진보했다고 할 수 없습니다. 또 다른 기능은 RE의 일부가 특정 횟수만큼 반복되어야 한다고 지정할 수 있다는 것입니다.

우리가 살펴볼 반복을 위한 첫 번째 메타 문자는 *입니다. *는 리터럴 문자 '*'와 일치하지 않습니다; 대신 이전 문자를 정확히 한 번이 아닌 0번 이상 일치시킬 수 있도록 지정합니다.

예를 들어, `ca*t`는 'ct' (0개의 'a' 문자), 'cat' (1개의 'a'), 'caaat' (3개의 'a' 문자) 등과 일치합니다.

*와 같은 반복은 탐욕스럽습니다 (*greedy*); RE를 반복할 때, 일치 엔진은 가능한 한 여러 번 반복하려고 시도합니다. 패턴의 뒷부분이 일치하지 않으면, 일치 엔진은 되돌아가서 더 작은 반복으로 다시 시도합니다.

단계별 예제를 통해 더 명확하게 알 수 있습니다. 정규식 `a[bcd]*b`를 생각해 봅시다. 이 문자는 'a' 문자와 일치하고, 0개 이상의 `[bcd]` 클래스 문자가 뒤따르고, 마지막에 'b'로 끝납니다. 이제 이 RE를 문자열 'abcdb'와 일치시킨다고 상상해보십시오.

단계	일치된 것	설명
1	a	RE의 a가 일치합니다.
2	abcbcd	엔진은 가능한 한 길게 [bcd]*와 일치시키려고 문자열의 끝까지 갑니다.
3	실패	엔진은 b를 일치하려고 시도하지만, 현재 위치가 문자열의 끝이므로 실패합니다.
4	abcb	물려서서, [bcd]*가 하나 적은 문자와 일치합니다.
5	실패	b를 다시 시도하지만, 현재 위치는 'd' 인 마지막 문자에 있습니다.
6	abc	다시 물려서서, [bcd]*가 bc하고 만 일치합니다.
6	abcb	b를 다시 시도합니다. 이번에는 현재 위치의 문자가 'b'이므로 성공합니다.

RE의 끝에 도달했으며, 'abcb'와 일치했습니다. 이것은 일치 엔진이 처음에는 갈 수 있는 데까지 가본 다음, 일치하는 것이 발견되지 않으면 점진적으로 물려서고, 나머지 RE의 나머지 부분을 반복해서 다시 시도하는 것을 보여줍니다. [bcd]*에 대한 일치 항목의 길이가 0이 될 때까지 물려서고, 그것마저도 실패하면, 엔진은 문자열이 RE와 전혀 일치하지 않는다고 결론을 내립니다.

또 다른 반복 메타 문자는 +인 데, 하나 이상과 일치합니다. *와 +의 차이점에 주의하십시오; *는 0 이상과 일치하므로, 반복되는 내용이 전혀 표시되지 않을 수 있습니다. 반면 +는 적어도 1번 이상 나타날 것을 요구합니다. 비슷한 예제를 사용하면, ca+t는 'cat' (1 'a'), 'caaat' (3 'a')와 일치하지만 'ct'와 일치하지는 않습니다.

There are two more repeating qualifiers. The question mark character, ?, matches either once or zero times; you can think of it as marking something as being optional. For example, home-?brew matches either 'homebrew' or 'home-brew'.

The most complicated repeated qualifier is {m, n}, where m and n are decimal integers. This qualifier means there must be at least m repetitions, and at most n. For example, a/{1, 3}b will match 'a/b', 'a//b', and 'a///b'. It won't match 'ab', which has no slashes, or 'a////b', which has four.

m이나 n을 생략 할 수 있습니다; 이때, 빠진 값에 대해 합리적인 값이 가정됩니다. m을 생략하면 0 하한으로 해석하는 반면, n을 생략하면 무한대의 상한을 뜻합니다.

Readers of a reductionist bent may notice that the three other qualifiers can all be expressed using this notation. {0, *} is the same as *, {1, *} is equivalent to +, and {0, 1} is the same as ?. It's better to use *, +, or ? when you can, simply because they're shorter and easier to read.

3 정규식 사용하기

이제 간단한 정규식을 살펴보았습니다. 실제로 파이썬에서 어떻게 사용해야 할까요? re 모듈은 정규식 엔진에 대한 인터페이스를 제공해서, RE를 객체로 컴파일한 다음 일치를 수행 할 수 있도록 합니다.

3.1 정규식 컴파일하기

정규식은 패턴 객체로 컴파일되는데, 패턴 일치를 검색하거나 문자열 치환을 수행하는 등의 다양한 작업을 위한 메서드를 갖고 있습니다.

```
>>> import re
>>> p = re.compile('ab*')
>>> p
re.compile('ab*')
```

re.compile()은 다양한 특수 기능과 문법 변형을 가능하게 하는 선택적 flags 인자도 받아들입니다. 나중에 사용할 수 있는 설정을 살펴보도록 하겠지만, 지금은 한 가지 예 만 보겠습니다:

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

RE는 문자열로 `re.compile()`에 전달됩니다. 정규식이 핵심 파이썬 언어의 일부가 아니고, 정규식을 표현하기 위한 특수 문법이 만들어지지 않았기 때문에 RE는 문자열로 다뤄집니다. (RE를 전혀 필요로 하지 않는 응용 프로그램이 있기 때문에, 이를 포함해서 언어 사양을 부풀릴 필요가 없습니다.) 대신, `re` 모듈은 `socket`이나 `zlib` 모듈과 마찬가지로 파이썬에 포함된 C 확장 모듈일 뿐입니다.

RE를 문자열에 넣는 것은 파이썬 언어가 더 간단하게 유지되도록 하지만, 다음 절의 주제인 한 가지 단점이 있습니다.

3.2 백 슬래시 전염병

앞에서 언급한 것처럼, 정규식은 백 슬래시 문자('\')를 사용하여 특수 형식을 나타내거나 특수 문자 특별한 의미를 갖지 않고 사용되도록 합니다. 이것은 파이썬이 문자열 리터럴에서 같은 목적으로 같은 문자를 사용하는 것과 충돌합니다.

LaTeX 파일에서 발견되는 문자열 `\section`과 일치하는 RE를 작성한다고 가정해 봅시다. 프로그램 코드에 무엇을 쓸지 알아내기 위해, 일치시키고자 하는 문자열로 시작하십시오. 그런 다음, 백 슬래시와 다른 메타 문자 앞에 백 슬래시를 붙여 이스케이프 처리하면, 문자열 `\\section`을 얻게 됩니다. `re.compile()`에 전달되어야 하는 결과 문자열은 `\\section`이어야 합니다. 그러나, 이를 파이썬 문자열 리터럴로 표현하려면, 두 개의 백 슬래시를 모두 다시 이스케이프 처리해야 합니다.

문자	단계
<code>\section</code>	일치시킬 텍스트 문자열
<code>\\section</code>	<code>re.compile()</code> 을 위해 이스케이프 처리된 백 슬래시
<code>\\\\section"</code>	문자열 리터럴을 위해 이스케이프 처리된 백 슬래시

즉, 리터럴 백 슬래시와 일치시키려면, RE 문자열로 `\\\\`을 작성해야 하는데, 정규식은 `\\`이어야 하고, 일반 파이썬 문자열 리터럴 안에서 각 백 슬래시를 `\\`로 표현해야 하기 때문입니다. 백 슬래시를 반복적으로 사용하는 RE에서는, 수없이 반복되는 백 슬래시로 이어져, 결과 문자열을 이해하기 어렵게 만듭니다.

해결책은 정규식에 파이썬의 날 문자열 표기법을 사용하는 것입니다; 백 슬래시는 'r' 접두사가 붙은 문자열 리터럴에서 특별한 방법으로 처리되지 않아서, `r"\n"`는 '\'과 'n'을 포함하는 두 문자 문자열이지만, `"\n"`는 줄 넘김을 포함하는 한 문자 문자열입니다. 정규식은 종종 이 날 문자열 표기법을 사용하여 파이썬 코드로 작성됩니다.

또한, 정규식에서는 유효하지만, 파이썬 문자열 리터럴에서는 유효하지 않은 특수 이스케이프 시퀀스는 이제 `DeprecationWarning`을 발생시키고 결국에는 `SyntaxError`가 될 것입니다. 이는 날 문자열 표기법이나 백 슬래시 이스케이핑이 사용되지 않으면 시퀀스가 유효하지 않게 됨을 뜻합니다.

일반 문자열	날 문자열
<code>"ab*"</code>	<code>r"ab*"</code>
<code>\\\\section"</code>	<code>r"\\section"</code>
<code>\\w+\\s+\\1"</code>	<code>r"\\w+\\s+\\1"</code>

3.3 일치 수행하기

일단 컴파일된 정규식을 나타내는 객체가 있으면, 이것으로 무엇을 할까요? 패턴 객체에는 여러 가지 메서드와 어트리뷰트가 있습니다. 가장 중요한 것만 여기서 다루어집니다; 전체 목록을 보려면 `re` 설명서를 참조하십시오.

메서드/어트리뷰트	목적
<code>match()</code>	문자열의 시작 부분에서 RE가 일치하는지 판단합니다.
<code>search()</code>	이 RE가 일치하는 위치를 찾으면서, 문자열을 훑습니다.
<code>findall()</code>	RE가 일치하는 모든 부분 문자열을 찾아 리스트로 반환합니다.
<code>finditer()</code>	RE가 일치하는 모든 부분 문자열을 찾아 이터레이터로 반환합니다.

`match()`와 `search()`는 일치하는 항목이 없으면 `None`을 반환합니다. 성공하면, 일치 객체 인스턴스가 반환되고, 일치에 대한 정보가 들어 있습니다: 어디에서 시작하고 끝나는지, 일치하는 부분 문자열 등입니다.

You can learn about this by interactively experimenting with the `re` module. If you have `tkinter` available, you may also want to look at [Tools/demo/redemo.py](#), a demonstration program included with the Python distribution. It allows you to enter REs and strings, and displays whether the RE matches or fails. `redemo.py` can be quite useful when trying to debug a complicated RE.

이 HOWTO는 예제에 표준 파이썬 인터프리터를 사용합니다. 먼저, 파이썬 인터프리터를 실행하고, `re` 모듈을 импорт 한 다음, RE를 컴파일하십시오:

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

이제, 다양한 문자열을 RE `[a-z]+`와 일치시켜볼 수 있습니다. `+`는 ‘하나 이상의 반복’을 의미하므로, 빈 문자열은 전혀 일치하지 않아야 합니다. 이때 `match()`는 `None`을 반환해야 하며, 인터프리터는 출력을 인쇄하지 않습니다. 분명하게 하기 위해 `match()`의 결과를 명시적으로 인쇄 할 수 있습니다.

```
>>> p.match("")
>>> print(p.match(""))
None
```

이제, 일치해야 하는 문자열을 시도해 봅시다, 가령 `tempo`. 이때, `match()`는 일치 객체를 반환하므로, 나중에 사용할 수 있도록 결과를 변수에 저장해야 합니다.

```
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

이제 `match object`를 조회하여 일치하는 문자열에 대한 정보를 얻을 수 있습니다. 일치 객체 인스턴스에는 여러 메서드와 어트리뷰트가 있습니다; 가장 중요한 것들은 다음과 같습니다:

메서드/어트리뷰트	목적
<code>group()</code>	RE와 일치하는 문자열을 반환합니다.
<code>start()</code>	일치의 시작 위치를 반환합니다
<code>end()</code>	일치의 끝 위치를 반환합니다
<code>span()</code>	일치의 (시작, 끝) 위치를 포함하는 튜플을 반환합니다.

이 메서드를 실험해보면 곧 그 의미가 분명해질 것입니다:

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

group() 은 RE에 의해 일치된 부분 문자열을 반환합니다. start() 와 end() 는 일치의 시작과 끝 인덱스를 반환합니다. span() 은 시작과 끝 인덱스를 단일 튜플로 반환합니다. match() 메서드는 문자열의 시작 부분에서 RE가 일치하는지 검사하므로, start() 는 항상 0입니다. 그러나, 패턴의 search() 메서드는 문자열을 훑기 때문에, 일치가 0에서 시작하지 않을 수 있습니다.

```
>>> print(p.match('::: message'))
None
>>> m = p.search('::: message'); print(m)
<re.Match object; span=(4, 11), match='message'>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

실제 프로그램에서, 가장 일반적인 스타일은 일치 객체를 변수에 저장한 다음, None 인지 확인하는 것입니다. 보통 이런 식입니다:

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

두 패턴 메서드는 패턴에 대한 모든 일치를 반환합니다. findall() 은 일치하는 문자열 리스트를 반환합니다:

```
>>> p = re.compile(r'\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
```

이 예제에서는 리터럴을 날 문자열 리터럴로 만드는 r 접두어가 필요한데, 일반적인 “요리된(cooked)” 문자열 리터럴에 있는, 정규식에서는 허락되지만, 파이썬에서 인식하지 못하는, 이스케이프 시퀀스가 이제 DeprecationWarning을 발생시키고, 결국에는 결국 SyntaxError가 될 것이기 때문입니다. 백 슬래시 전염병을 참조하십시오.

findall() 은 결과로 반환하기 전에 전체 리스트를 만들어야 합니다. finditer() 메서드는 매치 객체 인스턴스의 시퀀스를 이터레이터로 반환합니다:

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
...     print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```


3.4 모듈 수준 함수

패턴 객체를 생성하고 메서드를 호출해야만 하는 것은 아닙니다; re 모듈은 `match()`, `search()`, `findall()`, `sub()` 등의 최상위 수준 함수도 제공합니다. 이 함수들은 첫 번째 인자로 RE 문자열이 추가된 해당 패턴 메서드와 같은 인자를 취하고, 여전히 None이나 일치 객체 인스턴스를 반환합니다.

```
>>> print(re.match(r'From\s+', 'Fromage amk'))
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<re.Match object; span=(0, 5), match='From ' >
```

내부적으로, 이 함수는 단순히 패턴 객체를 만들고 적절한 메서드를 호출합니다. 또한 컴파일된 객체를 캐시에 저장하므로, 같은 RE를 사용하는 이후의 호출에서는 패턴을 반복해서 구문 분석할 필요가 없습니다.

이러한 모듈 수준의 함수를 사용해야 할까요, 아니면 패턴을 얻어서 메서드를 직접 호출해야 할까요? 루프 내에서 정규식에 액세스하고 있다면, 사전 컴파일이 몇 번의 함수 호출을 절약할 수 있습니다. 루프 바깥에서는, 내부 캐시 덕분에 큰 차이가 없습니다.

3.5 컴파일 플래그

컴파일 플래그를 사용하면 정규식의 작동 방식을 수정할 수 있습니다. 플래그는 re 모듈에서 IGNORECASE와 같은 긴 이름과 I와 같은 간단한 한 글자 형식의 두 가지 이름으로 사용 가능합니다. (Perl의 패턴 수정자에 익숙하다면, 한 글자 형식은 같은 글자를 사용합니다; 예를 들어, re.VERBOSE의 짧은 형식은 re.X입니다.) 다중 플래그는 비트별 OR 하여 지정할 수 있습니다; 예를 들어, re.I | re.M은 I와 M 플래그를 모두 설정합니다.

다음은 사용 가능한 플래그와 각 플래그에 대한 자세한 설명입니다.

플래그	의미
ASCII, A	\w, \b, \s 및 \d와 같은 여러 이스케이프가 해당 속성이 있는 ASCII 문자에만 일치하도록 합니다.
DOTALL, S	.가 개행 문자를 포함한 모든 문자와 일치하도록 합니다.
IGNORECASE, I	대소 문자 구분 없는 일치를 수행합니다.
LOCALE, L	로케일을 고려하는 일치를 수행합니다.
MULTILINE, M	다중 행 일치, ^와 \$에 영향을 줍니다.
VERBOSE, X ('확장' 용)	더 명확하고 이해하기 쉽게 정리될 수 있는 상세한 RE를 활성화합니다.

I

IGNORECASE

대소 문자를 구분하지 않는 일치를 수행합니다; 문자 클래스와 리터럴 문자열은 대소 문자를 무시하여 문자와 일치합니다. 예를 들어 [A-Z]는 소문자와도 일치합니다. ASCII 플래그로 ASCII가 아닌 일치를 막지 않는 한, 전체 유니코드 일치도 작동합니다. 유니코드 패턴 [a-z]나 [A-Z]가 IGNORECASE 플래그와 함께 사용되면, 52 ASCII 문자와 4개의 추가 비 ASCII 문자와 일치합니다: 'İ' (U+0130, 위에 점이 있는 라틴 대문자 I), 'ı' (U+0131, 라틴 소문자 점 없는 i), 'ŀ' (U+017F, 라틴 소문자 긴 s) 및 'K' (U+212A, 켈빈 기호). Spam은 'Spam', 'spam', 'spAM' 또는 'İpam'과 일치합니다 (마지막은 유니코드 모드에서만 일치합니다). 이 소문자화는 현재 로케일을 고려하지 않습니다; 고려하려면 LOCALE 플래그를 설정하면 됩니다.

L

LOCALE

\w, \W, \b, \B 및 대소 문자를 구분하지 않는 일치를 유니코드 데이터베이스 대신 현재 로케일에 의존하도록 만듭니다.

로케일은 언어 차이를 고려한 프로그램을 작성하는 데 도움이 되는 C 라이브러리의 기능입니다. 예를 들어, 인코딩된 프랑스어 텍스트를 처리할 때, 단어와 일치하도록 `\w+`를 쓰고 싶습니다, 하지만 `\w`는 바이트열 패턴에서 `[A-Za-z]` 문자 클래스하고만 일치합니다; `é`나 `ç`에 해당하는 바이트는 일치하지 않습니다. 시스템이 올바르게 구성되고 프랑스어 로케일이 선택되면, 특정 C 함수는 `é`에 해당하는 바이트도 문자로 간주하여야 함을 프로그램에 알립니다. 정규식을 컴파일할 때 `LOCALE` 플래그를 설정하면, 컴파일된 결과 객체가 `\w`에 대해 이러한 C 함수를 사용하게 됩니다; 더 느리기는 하지만, 기대하는 대로 `\w+`가 프랑스어 단어를 일치시킬 수 있습니다. 이 플래그의 사용은 파이썬 3에서는 권장하지 않는데, 로케일 메커니즘이 매우 신뢰성이 떨어지고, 한 번에 하나의 “컬처(culture)” 만 처리하고, 8비트 로케일에서 만 작동하기 때문입니다. 파이썬 3에서 유니코드 (str) 패턴에 대해 유니코드 일치가 기본적으로 이미 활성화되어 있으며, 다른 로케일/언어를 처리할 수 있습니다.

M

MULTILINE

(^과 \$는 아직 설명되지 않았습니니다; 메타 문자 더 보기 절에서 소개될 예정입니다.)

보통 ^는 문자열의 시작 부분에서만 일치하고, \$는 문자열의 끝부분과 문자열 끝에 있는 줄 바꿈 (있다면) 바로 앞에서 일치합니다. 이 플래그를 지정하면 ^는 문자열 시작 부분과 문자열 내의 각 줄 시작 부분(각 줄 바꿈의 바로 뒤)에서 일치합니다. 비슷하게, \$ 메타 문자는 문자열 끝과 각 줄의 끝(각 줄 바꿈 바로 앞)에서 일치합니다.

S

DOTALL

'.' 특수 문자가 개행 문자를 포함하는 모든 문자와 일치하도록 만듭니다; 이 플래그가 없으면, '.'는 개행 문자를 제외한 모든 문자와 일치합니다.

A

ASCII

`\w`, `\W`, `\b`, `\B`, `\s` 및 `\S`가 전체 유니코드 일치 대신 ASCII 전용 일치를 수행하도록 만듭니다. 유니코드 패턴에서만 의미가 있으며, 바이트열 패턴에서는 무시됩니다.

X

VERBOSE

이 플래그는 정규식을 포매팅하는 더 유연한 방법을 제공해서 더 가독성 있는 정규식을 작성할 수 있도록 합니다. 이 플래그가 지정되면, 문자 클래스에 있거나 이스케이프 되지 않은 백 슬래시 뒤에 있을 때를 제외하고, RE 문자열 내의 공백을 무시합니다; 이것은 RE를 보다 명확하게 구성하고 들여쓰기 할 수 있도록 합니다. 이 플래그는 RE 내에 엔진이 무시하는 주석을 넣을 수도 있게 합니다; 주석은 문자 클래스나 이스케이프 처리되지 않은 백 슬래시 뒤에 있지 않은 '#'로 표시됩니다.

예를 들어, 여기에 `re.VERBOSE`를 사용하는 RE가 있습니다; 얼마나 더 읽기 쉬워지는지 보이십니까?

```
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)
```

상세 설정이 없으면, RE는 이렇게 됩니다:

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

위의 예에서, 파이썬의 문자열 리터럴 자동 이어붙이기를 사용해서 RE를 더 작은 조각으로 나누었지만, `re.VERBOSE`를 사용하는 버전보다 여전히 이해하기가 어렵습니다.

4 더 많은 패턴 기능

지금까지 정규식의 일부 기능에 관해서만 설명했습니다. 이 절에서는, 몇 가지 새로운 메타 문자와 그룹을 사용하여 일치하는 텍스트의 부분을 꺼내는 방법을 다룹니다.

4.1 메타 문자 더 보기

우리가 아직 다루지 않은 몇 가지 메타 문자가 있습니다. 대부분 이 절에서 다룰 것입니다.

논의할 나머지 메타 문자 중 일부는 폭이 없는 어서션(*zero-width assertions*)입니다. 이들은 엔진이 문자열을 통해 앞으로 나아가도록 하지 않습니다; 대신, 문자를 전혀 소비하지 않고, 단순히 성공하거나 실패합니다. 예를 들어, `\b`는 현재 위치가 단어 경계에 위치한다는 어서션입니다; 위치는 `\b`에 의해 전혀 변경되지 않습니다. 이것은 폭이 없는 어서션을 반복해서는 안 된다는 뜻인데, 주어진 위치에서 일단 일치하면 명백히 무한한 횟수만큼 일치 할 수 있기 때문입니다.

| 대안, 또는 “or” 연산자. *A*와 *B*가 정규식이면, *A|B*는 *A*나 *B*와 일치하는 문자열과 일치합니다. |는 여러 문자로 된 문자열의 대안을 사용할 때 합리적으로 작동하도록 하기 위해 우선순위가 매우 낮습니다. `Crow|Servo`는 'Crow'나 'Servo'와 일치합니다, 'Cro', 'w'나 'S' 그리고 'ervo'가 아닙니다.

리터럴 '|'를 일치시키려면, `\|`를 사용하거나 `[]` 처럼 문자 클래스 안에 넣으십시오.

^ 줄의 시작 부분에 일치합니다. MULTILINE 플래그가 설정되어 있지 않은 한, 문자열 시작 부분에서만 일치합니다. MULTILINE 모드에서는, 문자열 내의 각 줄 바꿈 바로 뒤에서도 일치합니다.

예를 들어, `From`이라는 단어를 줄의 시작 부분에서만 일치시키려면, 사용할 RE는 `^From`입니다.

```
>>> print(re.search('^From', 'From Here to Eternity'))
<re.Match object; span=(0, 4), match='From'>
>>> print(re.search('^From', 'Reciting From Memory'))
None
```

리터럴 '^'를 일치시키려면, `\^`를 사용하십시오.

\$ 줄의 끝부분과 일치하는데, 문자열의 끝이나 줄 바꿈 문자 다음에 오는 모든 위치로 정의됩니다.

```
>>> print(re.search('{}$', '{block}'))
<re.Match object; span=(6, 7), match='}'>
>>> print(re.search('{}$', '{block} '))
None
>>> print(re.search('{}$', '{block}\n'))
<re.Match object; span=(6, 7), match='}'>
```

리터럴 '\$'를 일치시키려면, `\$`를 사용하거나 `[$]` 처럼 문자 클래스 안에 넣으십시오.

\A 문자열의 시작 부분에서만 일치합니다. MULTILINE 모드가 아닐 때, `\A`와 `^`는 실질적으로 같습니다. MULTILINE 모드에서는, 다릅니다: `\A`는 여전히 문자열의 시작 부분에서만 일치하지만, `^`는 문자열 내의 줄 바꿈 문자 뒤에 오는 모든 위치에서 일치 할 수 있습니다.

\Z 문자열 끝부분에서만 일치합니다.

\b 단어 경계. 이것은 단어(word)의 시작이니 끝부분에서만 일치하는 폭이 없는 어서션입니다. 단어는 영숫자 문자의 시퀀스로 정의되므로, 단어의 끝은 공백이나 영숫자가 아닌 문자로 표시됩니다.

다음 예제는 완전한 단어일 때만 `class`와 일치합니다. 다른 단어 안에 포함되어 있으면 일치하지 않습니다.

```
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<re.Match object; span=(3, 8), match='class'>
>>> print(p.search('the declassified algorithm'))
None
>>> print(p.search('one subclass is'))
None
```

이 특수 시퀀스를 사용할 때 기억해야 할 두 가지 미묘한 점이 있습니다. 첫째, 이것은 파이썬의 문자열 리터럴과 정규식 시퀀스 간의 최악의 충돌입니다. 파이썬의 문자열 리터럴에서 `\b`는 ASCII 값 8을 갖는 백스페이스 문자입니다. 날 문자열을 사용하지 않으면, 파이썬이 `\b`를 백스페이스로 변환하고, 여러분의 RE는 예상대로 일치하지 않습니다. 다음 예제는 앞의 RE와 같아 보이지만, RE 문자열 앞의 'r'가 빠졌습니다.

```
>>> p = re.compile('\bclass\b')
>>> print(p.search('no class at all'))
None
>>> print(p.search('\b' + 'class' + '\b'))
<re.Match object; span=(0, 7), match='\x08class\x08'>
```

둘째, 이 어서션이 사용되지 않는 문자클래스 내에서, `\b`는 파이썬의 문자열 리터럴과의 호환성을 위해 백스페이스 문자를 나타냅니다.

`\B` 또 다른 쪽이 없는 어서션, 이것은 `\b`의 반대이며, 현재 위치가 단어 경계에 있지 않을 때만 일치합니다.

4.2 그룹

종종 단지 RE가 일치하는지보다 많은 정보를 얻을 필요가 있습니다. 정규식은 종종 관심 있는 다른 구성 요소와 일치하는 몇 개의 서브 그룹으로 나누어진 RE를 작성하여 문자열을 해부하는 데 사용됩니다. 예를 들어, RFC-822 헤더 행은 다음과 같이 ':'로 구분된 헤더 이름과 값으로 나뉩니다:

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
MIME-Version: 1.0
To: editor@example.com
```

이것은 전체 헤더 행과 일치하는 정규식을 작성하고, 헤더 이름과 일치하는 그룹 하나와 헤더 값과 일치하는 다른 그룹을 가짐으로써 처리할 수 있습니다.

Groups are marked by the '(', ') ' metacharacters. ' (' and ') ' have much the same meaning as they do in mathematical expressions; they group together the expressions contained inside them, and you can repeat the contents of a group with a repeating qualifier, such as *, +, ?, or {m,n}. For example, (ab) * will match zero or more repetitions of ab.

```
>>> p = re.compile('(ab)*')
>>> print(p.match('ababababab').span())
(0, 10)
```

' (, ') '로 표시된 그룹은 일치하는 텍스트의 시작과 끝 인덱스도 포착합니다; 이것은 `group()`, `start()`, `end()` 및 `span()`에 인자를 전달하여 꺼낼 수 있습니다. 그룹은 0부터 시작하여 번호가 매겨집니다. 그룹 0은 항상 존재합니다; 이것은 전체 RE이므로 일치 객체 메서드는 모두 그룹 0을 기본 인자로 사용합니다. 나중에 일치하는 텍스트 범위를 포착하지 않는 그룹을 표현하는 방법을 살펴보겠습니다.

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
```

(다음 페이지에 계속)

```
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

서브 그룹은 왼쪽에서 오른쪽으로 1부터 위로 번호가 매겨집니다. 그룹은 중첩될 수 있습니다; 숫자를 결정하려면, 왼쪽에서 오른쪽으로 가면서 여는 괄호 문자를 세십시오.

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

`group()` 으로는 한 번에 여러 개의 그룹 번호를 전달할 수 있으며, 이때 해당 그룹에 해당하는 값을 포함하는 튜플을 반환합니다.

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

`groups()` 메서드는 모든 서브 그룹에 대한 문자열을 포함하는 튜플을 반환합니다, 1에서 최대까지.

```
>>> m.groups()
('abc', 'b')
```

패턴의 역참조를 사용하면 이전 포착 그룹의 내용이 문자열의 현재 위치에서도 발견되어야 한다고 지정할 수 있습니다. 예를 들어, `\1`은 그룹 1의 정확한 내용이 현재 위치에서 발견되면 성공하고, 그렇지 않으면 실패합니다. 파이썬의 문자열 리터럴은 백 슬래시 뒤에 숫자를 붙여 문자열에 임의의 문자를 포함할 수 있기 때문에, RE에 역참조를 포함할 때 날 문자열을 사용해야 합니다.

예를 들어, 다음 RE는 문자열에서 중복 단어를 감지합니다.

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

이와 같은 역참조는 단순히 문자열을 검색하는 데는 별로 유용하지 않습니다 — 이런 식으로 데이터를 반복하는 텍스트 형식은 거의 없습니다 — 하지만 곧 문자열 치환을 수행할 때 아주 유용하다는 것을 알게 될 것입니다.

4.3 비 포착 그룹과 이름 있는 그룹

정교한 RE는 관심 있는 부분 문자열을 포착하고 RE 자체를 그룹화하고 구조화하기 위해 많은 그룹을 사용할 수 있습니다. 복잡한 RE에서는, 그룹 번호를 추적하기가 어려워집니다. 이 문제를 해결하는 데 도움이 되는 두 가지 기능이 있습니다. 둘 다 정규식 확장에 같은 문법을 사용하므로, 그것부터 살펴보겠습니다.

Perl 5는 표준 정규식에 대한 강력한 추가 기능으로 유명합니다. 이러한 새로운 기능을 위해 Perl 개발자는 Perl의 정규식을 표준 RE와 혼란스러울 만큼 다르게 만들지 않으면서 한 글자 메타 문자나 `\`로 시작하는 새로운 특수 시퀀스를 선택할 수 없었습니다. 예를 들어, `&`를 새로운 메타 문자로 선택하면, 예전 정규식은 `&`가 일반 문자라고 가정하고 `\&`나 `[&]`로 작성하여 이스케이프 하지 않을 것입니다.

Perl 개발자가 선택한 해법은 `(?...)`를 확장 문법으로 사용하는 것입니다. 괄호 바로 뒤에 있는 `?`는 `?`가 반복할 것이 없기 때문에 문법 에러였습니다. 따라서 이것은 어떤 호환성 문제도 일으키지 않습니다. `?` 다음에

나오는 문자는 어떤 확장이 사용되는지 나타내므로, (?=foo) 는 한가지 확장이고 (긍정적인 미리 보기 어서션), (?:foo) 는 또 다른 것입니다 (서브 정규식 foo를 포함하는 비 포착 그룹).

파이썬은 여러 Perl의 확장을 지원하고 Perl의 확장 문법에 확장 문법을 추가합니다. 물음표 뒤의 첫 번째 문자가 P이면, 파이썬에 특유한 확장임을 알 수 있습니다.

이제 일반적인 확장 문법을 살펴보았으므로, 복잡한 RE에서 그룹 작업을 단순화하는 기능으로 돌아갈 수 있습니다.

때로 그룹을 사용하여 정규식의 일부를 나타내고 싶지만, 그룹의 내용을 꺼내는 데는 관심이 없습니다. 이 사실을 비 포착 그룹을 사용해서 명시적으로 만들 수 있습니다: (?:...), 여기서 ...을 다른 정규식으로 바꿀 수 있습니다.

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

그룹과 일치하는 내용을 꺼낼 수 없다는 점을 제외하면, 비 포착 그룹은 포착 그룹과 정확히 같게 작동합니다; 안에 어떤 것이든 넣을 수 있고, *와 같은 반복 메타 문자로 반복할 수 있고, 다른 그룹(포착이나 비 포착) 내에 중첩할 수 있습니다. (?:...) 는 기존 패턴을 수정할 때 특히 유용합니다. 다른 모든 그룹의 번호가 매겨지는 방식을 변경하지 않고 새 그룹을 추가 할 수 있기 때문입니다. 포착 그룹과 비 포착 그룹을 검색할 때 성능 차이가 없다는 점을 짚고 넘어가야 할 것 같습니다; 두 형태 중 어느 것도 다른 것보다 빠르지 않습니다.

더 중요한 기능은 이름 있는 그룹입니다: 번호로 참조하는 대신, 이름으로 그룹을 참조 할 수 있습니다.

이름 있는 그룹의 문법은 파이썬 특정 확장 중 하나입니다: (?P<name>...). *name*은, 당연히, 그룹의 이름입니다. 이름 있는 그룹은 포착 그룹과 똑같이 동작하며, 추가로 이름을 그룹과 연관시킵니다. 포착 그룹을 다루는 일치 객체 메서드는 모두 숫자로 그룹을 가리키는 정수나 원하는 그룹의 이름을 포함하는 문자열을 받아들입니다. 이름 있는 그룹에는 여전히 번호가 매겨지므로, 두 가지 방법으로 그룹에 대한 정보를 꺼낼 수 있습니다:

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search('((( Lots of punctuation )))')
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

또한, groupdict() 로 이름 있는 그룹을 딕셔너리로 꺼낼 수 있습니다:

```
>>> m = re.match(r'(?P<first>\w+) (?P<last>\w+)', 'Jane Doe')
>>> m.groupdict()
{'first': 'Jane', 'last': 'Doe'}
```

이름 있는 그룹은 숫자를 기억하는 대신 쉽게 기억할 수 있는 이름을 사용할 수 있어서 편리합니다. 다음은 imaplib 모듈에서 온 예제 RE입니다:

```
InternalDate = re.compile(r'INTERNALDATE "'
    r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
    r'(?P<year>[0-9][0-9][0-9][0-9])'
    r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9])'
    r' (?P<zonen>[-+]) (?P<zoneh>[0-9][0-9]) (?P<zonom>[0-9][0-9])'
    r'")')
```

그룹 9를 꺼내는 것을 기억하는 대신, m.group('zonom') 을 꺼내기가 훨씬 쉽습니다.

(...)\1과 같은 정규식에서 역참조 문법은 그룹 번호를 나타냅니다. 자연스럽게 번호 대신 그룹 이름을 사용하는 변형이 있습니다. 이것은 다른 파이썬 확장입니다: (?P=name)은 *name*이라는 그룹의 내용이 현재 위치에서 다시 일치해야 함을 나타냅니다. 중복된 단어를 찾는 정규식인 \b(\w+)\s+\1\b는 \b(?P<word>\w+)\s+(?P=word)\b로 표현할 수도 있습니다:

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

4.4 미리 보기 어서션

또 다른 쪽이 없는 어서션은 미리 보기 어서션(lookahead assertion)입니다. 미리 보기 어서션은 긍정과 부정 형식 모두 제공되며, 다음과 같이 표시됩니다:

(?=...) 긍정적인 미리 보기 어서션. 포함된 정규식(여기에서는 ...로 표시되었습니다)이 현재 위치에서 성공적으로 일치하면 성공하고, 그렇지 않으면 실패합니다. 그러나, 일단 포함된 정규식이 시도되면, 일치 엔진은 전혀 앞으로 나아가지 않습니다; 어서션이 시작한 곳에서 나머지 패턴을 시도합니다.

(?!...) 부정적인 미리 보기 어서션. 이것은 긍정적인 어서션의 반대입니다; 포함된 정규식이 문자열의 현재 위치에서 일치하지 않으면 성공합니다.

이를 구체적으로 설명하기 위해, 미리 보기가 유용한 경우를 살펴보겠습니다. 파일 이름을 일치시키고 .로 구분된 기본 이름과 확장자로 분리하는 간단한 패턴을 생각해봅시다. 예를 들어, `news.rc`에서, `news`는 기본 이름이고 `rc`는 파일명의 확장자입니다.

이것과 일치하는 패턴은 매우 간단합니다:

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

.는 메타 문자이므로 특수하게 처리해야 하므로, 해당 문자에만 일치하기 위해 문자 클래스 내에 있습니다. 또한 후행 \$도 유의하십시오; 나머지 문자열이 확장에 포함되어야 함을 보장하기 위해 추가됩니다. 이 정규식은 `foo.bar`와 `autoexec.bat`와 `sendmail.cf`와 `printers.conf`와 일치합니다.

자, 문제를 조금 복잡하게 생각해봅시다; 확장자가 `bat`이 아닌 파일명을 일치시키려면 어떻게 해야 할까요? 몇 가지 잘못된 시도:

.*[.][^b].*\$ 위의 첫 번째 시도는 확장자의 첫 번째 문자가 `b`가 아니도록 요구하여 `bat`를 제외하려고 시도합니다. 패턴이 `foo.bar`와도 일치하지 않기 때문에, 이것은 잘못된 것입니다.

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

다음에서 하나를 요구하여 첫 번째 해결 방법을 패치할 때 정규식이 더 복잡해집니다: 확장자의 첫 번째 문자가 `b`가 아닙니다; 두 번째 문자가 `a`가 아닙니다; 또는 세 번째 문자가 `t`가 아닙니다. 이것은 `foo.bar`를 받아들이고 `autoexec.bat`를 거부하지만, 세 문자 확장자를 요구하고 `sendmail.cf`와 같은 두 문자로 된 확장자를 가진 파일명을 허용하지 않습니다. 문제를 해결하기 위해 패턴을 다시 복잡하게 만들 것입니다.

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

세 번째 시도에서, `sendmail.cf`와 같이 세 문자보다 짧은 확장자를 허용하기 위해 두 번째와 세 번째 문자는 모두 선택적입니다.

이제 패턴이 정말 복잡해져서, 읽고 이해하기 어렵습니다. 더욱이, 문제가 변경되어 확장자 `bat`와 `exe`를 모두 제외하려면, 패턴이 훨씬 복잡하고 혼란스러워집니다.

부정적인 미리 보기는 이 모든 혼란을 제거합니다:

.*[.](?!bat\$)[^.]*\$ 부정적인 미리 보기는 다음과 같은 의미입니다: `bat` 정규식이 이 지점에서 일치하지 않으면, 나머지 패턴을 시도합니다; `bat$`가 일치하면, 전체 패턴이 실패합니다. 후행 \$는 `sample.batch`와 같이 `bat`로 시작하기만 하는 확장자를 허용하기 위해 필요합니다. `[^.]`*는 파일명에 여러 점이 있을 때 패턴이 작동하도록 합니다.

다른 파일명 확장자를 제외하는 것이 이제는 쉽습니다; 단순히 어서션 안에 대안으로 추가하십시오. 다음 패턴은 bat 나 exe로 끝나는 파일명을 제외합니다:

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

5 문자열 수정하기

지금까지는, 정적 문자열에 대한 검색만 수행했습니다. 정규식은 다음과 같은 패턴 메서드를 사용하여 다양한 방법으로 문자열을 수정하는 데 흔히 사용됩니다:

메서드/어트리뷰트	목적
<code>split()</code>	RE가 일치하는 모든 곳에서 분할하여, 문자열을 리스트로 분할합니다
<code>sub()</code>	RE가 일치하는 모든 부분 문자열을 찾고, 다른 문자열로 대체합니다.
<code>subn()</code>	<code>sub()</code> 와 같은 일을 하지만, 새로운 문자열과 치환 횟수를 반환합니다

5.1 문자열 분할하기

패턴의 `split()` 메서드는 RE가 일치하는 모든 곳에서 문자열을 분할하여, 조각의 리스트를 반환합니다. 이것은 `split()` 문자열 메서드와 비슷하지만 분리하는 데 사용되는 구분자에 훨씬 더 일반성을 제공합니다; 문자열 `split()` 는 공백이나 고정 문자열로의 분할 만 지원합니다. 예상대로, 모듈 수준의 `re.split()` 함수도 있습니다.

.split (string[, maxsplit=0])

정규식과 일치하는 것으로 *string*을 분할합니다. RE에서 포착하는 괄호가 사용되면, 해당 내용도 결과 리스트의 일부로 반환됩니다. *maxsplit*가 0이 아니면, 최대 *maxsplit* 번 분할만 수행됩니다.

maxsplit 값을 전달하여, 분할 수를 제한 할 수 있습니다. *maxsplit*가 0이 아니면, 최대 *maxsplit* 번 분할만 이루어지고, 나머지 문자열은 리스트의 마지막 요소로 반환됩니다. 다음 예제에서, 구분자는 영숫자가 아닌 문자 시퀀스입니다.

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

때로는 구분자 사이의 텍스트가 무엇인지에 관심이 있을 뿐만 아니라, 구분자가 무엇인지도 알아야 할 필요가 있습니다. RE에서 포착하는 괄호가 사용되면, 해당 값도 리스트의 일부로 반환됩니다. 다음 호출을 비교하십시오:

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '...', 'is', ' ', 'a', ' ', 'test', '.', '']
```

모듈 수준 함수 `re.split()` 는 첫 번째 인자로 사용할 RE를 추가하지만, 이를 제외하고는 같습니다.

```
>>> re.split(r'[W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'([W]+)', 'Words, words, words.')
['Words', ' ', ' ', 'words', ' ', ' ', 'words', '.', '']
```

(다음 페이지에 계속)

```
>>> re.split(r'[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

5.2 검색과 치환

또 다른 흔한 작업은 패턴에 대한 모든 일치를 찾아 다른 문자열로 치환하는 것입니다. `sub()` 메서드는 치환 값(문자열이나 함수일 수 있습니다)과 처리할 문자열을 취합니다.

.sub(replacement, string[, count=0])

`string`에서 가장 왼쪽에 나타나는 겹쳐지지 않은 RE의 일치를 `replacement`로 치환한 문자열을 반환합니다. 패턴이 없으면, `string`이 변경 없이 반환됩니다.

선택적 인자 `count`는 치환될 패턴 일치의 최대 수입니다; `count`는 음수가 아닌 정수여야 합니다. 기본값 0은 모든 일치를 치환하는 것을 의미합니다.

다음은 `sub()` 메서드를 사용하는 간단한 예제입니다. 색상 이름을 `colour`라는 단어로 바꿉니다:

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

`subn()` 메서드는 같은 작업을 수행하지만, 새 문자열 값과 수행된 치환 수가 포함된 2-튜플을 반환합니다:

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

빈 일치는 이전의 빈 일치와 인접하지 않은 경우에만 치환됩니다.

```
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b--d-'
```

`replacement`가 문자열이면, 그 안에 있는 모든 역슬래시 이스케이프가 처리됩니다. 즉, `\n`은 단일 개행 문자로 변환되고, `\r`은 캐리지 리턴으로 변환되고, 나머지도 마찬가지입니다. `\&`와 같은 알 수 없는 이스케이프는 그대로 남습니다. `\6`과 같은 역참조는 RE의 해당 그룹과 일치하는 부분 문자열로 치환됩니다. 이렇게 하면 결과 치환 문자열에 원본 텍스트의 일부를 통합할 수 있습니다.

이 예제는 뒤에 `{, }`로 묶인 문자열이 오는 단어 `section`과 일치하고, `section`을 `subsection`으로 변경합니다:

```
>>> p = re.compile('section{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}', 'section{First} section{second}')
'subsection{First} subsection{second}'
```

또한 `(?P<name>...)` 문법으로 정의된 이름 있는 그룹을 참조하기 위한 문법이 있습니다. `\g<name>`은 `name` 그룹과 일치하는 부분 문자열을 사용하고, `\g<number>`는 해당 그룹 번호를 사용합니다. 따라서 `\g<2>`는 `\2`와 동등하지만, `\g<2>0`과 같은 치환 문자열에서 모호하지 않습니다. (`\20`은 '0'이 뒤에 오는 그룹 2에 대한 참조가 아닌, 그룹 20에 대한 참조로 해석됩니다.) 다음 치환은 모두 동등하지만, 치환 문자열의 세 가지 변형을 모두 사용합니다.

```
>>> p = re.compile('section{ (?P<name> [^}]*) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}', 'section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}', 'section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}', 'section{First}')
'subsection{First}'
```

*replacement*는 함수일 수도 있는데, 더 많은 제어를 제공합니다. *replacement*가 함수면, *pattern*의 겹쳐지지 않는 모든 일치에 대해 함수가 호출됩니다. 각 호출에서, 함수는 그 일치에 대한 일치 객체 인자를 전달받고, 이 정보를 사용하여 원하는 치환 문자열을 계산하고 이를 반환 할 수 있습니다.

다음 예제에서, 치환 함수는 십진수를 16진수로 변환합니다:

```
>>> def hexrepl(match):
...     "Return the hex string for a decimal number"
...     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.'
```

모듈 수준의 `re.sub()` 함수를 사용할 때, 패턴은 첫 번째 인자로 전달됩니다. 패턴은 객체나 문자열로 제공될 수 있습니다; 정규식 플래그를 지정해야 하면, 패턴 객체를 첫 번째 매개 변수로 사용하거나 패턴 문자열에 포함된 수정자를 사용해야 합니다, 예를 들어 `sub("(?i)b+", "x", "bbbb BBBB")` 는 'x x'를 반환합니다.

6 일반적인 문제들

정규식은 일부 응용을 위한 강력한 도구이지만, 어떤 면에서는 동작이 직관적이지 않고 때때로 예상대로 동작하지 않을 수도 있습니다. 이 절에서는 가장 일반적인 함정 중 일부를 지적합니다.

6.1 문자열 메서드를 사용하십시오

때때로 `re` 모듈을 사용하는 것은 실수입니다. 고정된 문자열이나 단일 문자 클래스와 일치시키려고 하고, `IGNORECASE` 플래그와 같은 `re` 기능을 사용하지 않는다면, 정규식의 모든 기능이 필요하지 않을 수 있습니다. 문자열은 고정 문자열을 사용하는 연산을 수행하는 몇 가지 메서드를 가지고 있으며, 대개 훨씬 빠릅니다. 더 크고, 더 일반화된 정규식 엔진 대신, 구현이 목적에 맞게 최적화된 단일하고 작은 C 루프이기 때문입니다.

한가지 예는 하나의 고정 된 문자열을 다른 것으로 치환하는 것일 수 있습니다; 예를 들어, `word`를 `deed`로 바꿀 수 있습니다. `re.sub()`가 이를 위한 함수인 것처럼 보이지만, `replace()` 메서드를 고려하십시오. `replace()`는 또한 단어 안에 있는 `word`를 치환해서, `swordfish`를 `sdeedfish`로 바꾸지만, 나이브한 `RE` `word`도 그렇게 했을 것입니다. (단어의 일부에 대한 치환을 수행하는 것을 피하고자, 패턴은 `word` 양쪽에 단어 경계가 있도록 `\bword\b`여야 합니다. 이 작업은 `replace()`의 능력을 넘어섭니다.)

또 다른 일반적인 작업은 문자열에서 단일 문자를 모두 삭제하거나 다른 단일 문자로 바꾸는 것입니다. `re.sub('\n', ' ', s)`와 같은 방식으로 이 작업을 수행 할 수 있지만, `translate()`는 두 가지 작업을 모두 수행 할 수 있으며 정규식 연산보다 빠릅니다.

정리하면, `re` 모듈을 사용하기 전에, 더 빠르고 간단한 문자열 메서드로 문제를 해결할 수 있는지 고려하십시오.

6.2 match()와 search() 비교

`match()` 함수는 문자열 시작 부분에서 RE가 일치하지만 확인하는 반면, `search()` 는 일치를 찾기 위해 문자열을 정방향으로 검색합니다. 이 차이를 염두에 두는 것이 중요합니다. 기억하십시오, `match()` 는 0에서 시작하는 성공적인 일치만을 보고합니다; 일치이 0에서 시작하지 않으면, `match()` 는 이를 보고하지 않습니다.

```
>>> print(re.match('super', 'superstition').span())
(0, 5)
>>> print(re.match('super', 'insuperable'))
None
```

반면에, `search()` 는 문자열을 정방향으로 검색하여, 발견된 첫 번째 일치를 보고합니다.

```
>>> print(re.search('super', 'superstition').span())
(0, 5)
>>> print(re.search('super', 'insuperable').span())
(2, 7)
```

때로 `re.match()` 를 계속 사용하면서, 단지 `*`를 RE 앞에 추가하고 싶을 수 있습니다. 이 유혹에 저항하고, 대신 `re.search()` 를 사용하십시오. 정규식 컴파일러는 일치를 찾는 프로세스의 속도를 높이기 위해 RE에 대한 분석을 수행합니다. 그러한 분석의 하나는 일치점의 첫 번째 문자가 무엇인지 알아내는 것입니다; 예를 들어, Crow로 시작하는 패턴은 'C'로 시작하는 것과 일치해야 합니다. 이 분석을 통해 엔진은 시작 문자를 찾기 위해 문자열을 빠르게 검색하고, 'C'가 발견될 때만 전체 일치를 시도합니다.

`*`를 추가하면 이 최적화가 실패하고, 문자열의 끝부분까지 스캔한 다음, RE의 나머지에 대한 일치를 찾기 위해 역추적합니다. 대신 `re.search()` 를 사용하십시오.

6.3 탐욕적 과 비 탐욕적 비교

`a*`에서와같이 정규식을 반복할 때, 결과 동작은 가능한 한 많은 패턴을 소비하는 것입니다. 이 사실은 HTML 태그를 둘러싼 화살 괄호(angle brackets)와 같이 쌍을 이루는 구분 기호 쌍을 일치시키려고 할 때 여러분을 물어뜯을 수 있습니다. 하나의 HTML 태그를 일치하는 나이트 패턴은 `*`의 탐욕스러운 성격 때문에 작동하지 않습니다.

```
>>> 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>
```

RE는 '`<html>`'의 '`<`'와 일치하고, `*`는 나머지 문자열을 소비합니다. RE에는 여전히 남아있는 것이 있고, `>`는 문자열의 끝에서 일치할 수 없기 때문에, 정규식 엔진은 `>`와 일치하는 것을 찾을 때까지 문자 단위로 역추적해야 합니다. 최종 일치점 '`<html>`'의 '`<`'에서 '`</title>`'의 '`>`'까지 확장되는데, 이는 여러분이 원하는 것이 아닙니다.

In this case, the solution is to use the non-greedy qualifiers `*?`, `+?`, `??`, or `{m, n}?`, which match as *little* text as possible. In the above example, the '`>`' is tried immediately after the first '`<`' matches, and when it fails, the engine advances a character at a time, retrying the '`>`' at every step. This produces just the right result:

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

(정규식으로 HTML이나 XML을 구문 분석하기는 쉽지 않습니다. 빠르지만 지저분한 패턴은 일반적인 경우를 처리할 것이지만, HTML과 XML에는 명확한 정규식을 깨뜨릴 특수 사례가 있습니다; 모든 가능한 경우를 처리하도록 정규식을 작성하면, 패턴이 아주 복잡해질 수 있습니다. 이러한 작업에는 HTML이나 XML 구문 분석 모듈을 사용하십시오.)

6.4 re.VERBOSE 사용하기

지금까지 정규식이 매우 콤팩트한 표기법이라는 사실을 눈치챈 것입니다만, 극단적으로 읽기 어렵지는 않았습니니다. 중간 정도의 복잡성을 가진 RE는 역슬래시, 괄호 및 메타 문자의 긴 모음이 되어 읽고 이해하기 어려울 수 있습니다.

이러한 RE의 경우, 정규식을 컴파일할 때 re.VERBOSE 플래그를 지정하면 정규식을 보다 명확하게 포맷할 수 있어서 도움이 됩니다.

re.VERBOSE 플래그는 여러 가지 효과가 있습니다. 문자 클래스 안에 있지 않은 공백이 무시됩니다. 이것은, `dog | cat`과 같은 정규식은 가독성이 떨어지는 `dog|cat`과 동등하지만, `[a b]`는 여전히 'a', 'b' 또는 스페이스 문자와 일치함을 뜻합니다. 또한, 주석을 RE에 넣을 수도 있습니다; 주석은 # 문자에서 다음 줄 바꿈까지 확장됩니다. 삼중 따옴표로 묶은 문자열과 함께 사용하면, RE를 더 깔끔하게 포맷할 수 있습니다:

```
pat = re.compile(r"""
\s*           # Skip leading whitespace
(?:P<header>[^\:]+) # Header name
\s* :         # Whitespace, and a colon
(?:P<value>.*?)   # The header's value -- *? used to
                  # lose the following trailing whitespace
\s*$          # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

이것이 다음보다 훨씬 읽기 쉽습니다:

```
pat = re.compile(r"\s*(?:P<header>[^\:]+)\s*:(?:P<value>.*?)\s*$")
```

7 피드백

정규식은 복잡한 주제입니다. 이 문서가 도움이 되었습니까? 불분명 한 부분이 있거나, 여기에서 다루지 않은 문제가 있습니까? 그렇다면 저자에게 개선을 위한 제안을 보내주십시오.

정규식에 대한 가장 완벽한 책은 거의 확실히 O'Reilly가 출판한 Jeffrey Friedl의 *Mastering Regular Expressions*입니다. 불행히도, 이 책은 Perl과 Java의 정규식에만 집중하고, 파이썬 자료를 전혀 포함하지 않아서 파이썬 프로그래밍에 대한 참조로는 유용하지 않습니다. (첫 번째 판은 지금은 제거된 파이썬의 regex 모듈을 다뤘습니다만, 큰 도움은 되지 못합니다.) 여러분의 도서관에서 확인해보십시오.