Regular Expression HOWTO

Simple Patterns

Matching Characters

- special *metacharacter* 들은 매치되지 않는다.
  - `. ^ $ * + ? { } [ ] \ | ( )`
- `[ ]`
  - 매치하고 싶은 캐릭터들의 *set*
  - `-` 를 이용하여 범위 지정한다
  - *metacharacter*들은 `[ ]` 안에서는 일반 문자로 매칭됨
  - `^`를 처음에 넣으면 해당 문자를 제외한 모든 문자에 매칭됨
    - ex. `[^5]` 는 5를 제외한 모든 문자에 매칭
- `\`
  - metacharacter 를 매칭하고 싶을 때 사용
  - 숫자 집합, 문자 집합 또는 공백이 아닌 집합 등 정의된 문자 집합을 나타냄
- `\w`
  - byte 패턴이면 [a-zA-Z0-9_] 와 동일
  - string 이면 유니코드 모든 문자와 일치
- `\d`
  - 숫자 집합과 매칭. `[0-9]`
- `\D`
  - 숫자가 아닌 집합과 매칭. `[^0-9]`
- `\s`
  - 공백 문자와 매칭. `[ \t\n\r\f\v]`
- `\S`
  - 공백이 아닌 문자와 매칭. `[^ \t\n\r\f\v]`
- `\w`
  - 모든 영숫자와 일치. `[a-zA-Z0-9]`
- `\W`
  - 모든 영숫자가 아닌 문자와 일치. `[^a-zA-Z0-9]`
- 위의 모든 시퀀스는 `[ ]` 안에 포함될 수 있다.
- `.`
  - 개행 문자를 제외한 모든 문자와 일치

Repeating Things

- `*`
  - 이전 문자가 0부터 무한번까지 매칭되는지?
  - *greedy*이다. 가능한 많은 반복까지 매칭한다.
  - 매칭 엔진은 가능한 멀리서부터 시작한다.

- `+`
  - 이전 문자가 **1**부터 무한번까지 매칭
- `?`
  - 이전 문자가 0번 또는 1번 매칭
- `{m, n}`
  - `m` 이상 `n`이하 반복되는지 매칭
  - `m` 이 생략되면 0번 이상
  - `n`이 생략되면 무한대까지

Using Regular Expressions

Compiling Regular Expressions

- 파이썬에서는 `re.compile()`를 거쳐야 정규표현식을 사용 가능
- re 모듈은 파이썬에 포함된 C 확장 모듈이다.

The Backslash Plague

- `\`는 정규표현식과 파이썬 간에 충돌 문제가 있다
  - 매칭해야할 텍스트가 `\section`이라고 하면
  - RE는 `\\section`이어야 하고
  - 이를 string으로 표현하려면 `"\\\\section"`이 되어야 한다
- Raw String을 사용하여 해결하자!
  - `r"\n"`은 `\`과 `n`을 나타낸다. 반면 일반 파이썬 string에서는 개행문자를 나타낸다.

Performing Matches

Method / Attribute	Purpose
`match()`	문자열 시작부터 매치되는지 결정
`search()`	문자열 어느 곳이든지 매치되는지 찾는다
`findall()`	일치하는 모든 substring을 찾아 리스트로 반환
`finditer()`	일치하는 모든 substring을 찾아 iterator로 반환

- match 객체가 반환되어 시작과 끝, 매치된 substring 등의 정보를 가진다.
- **tkinter**의 `redemo.py`를 사용하면 복잡한 RE를 디버깅할 수 있다.
- match 객체

Method / Attribute	Purpose
`group()`	RE 에 매치되는 문자열 반환
`start()`	매치의 시작지점 반환
`end()`	매치의 끝지점 반환
`span()`	(start, end) 형의 튜플 반환

- `match()` 메소드는 문자열의 시작지점부터 체크하므로, start()는 항상 0이다.

- `search()` 메소드는 모든 문자열을 스캔하므로 그렇지 않을 수도 있다.

```
>>> 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)
```

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

Module-Level Functions

- 모듈 레벨에도 여러 메서드가 있고, 한번 컴파일된 객체는 캐시에 저장된다.

Compilation Flags

Flag	Meaning
ASCII, A	아스키 문자 기준으로 매칭
DOTALL ,S	`.`을 개행 문자를 포함해서 매칭해라
IGNORECASE, I	대소문자 구분하지 마라
LOCALE, L	해당 언어를 인식해서 수행해라
MULTILINE, M	멀티라인 매칭, `^`와 `$`에 영향을 미친다
VERBOSE, X	말이 많은 RE를 가능하게 한다.

More Pattern Power

More Metacharacters

- `|`
  - OR을 의미한다. 우선순위가 가장 낮아서 각각의 패턴에 대해 매칭한다.
  - `'|'`를 매치하고 싶다면 `\|` 또는 `[|]`을 사용하자
- `^`
  - 라인의 시작과 매치한다.

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

  - `'^'`를 매치하고 싶다면 `\^` 를 사용하자
- `$`

    - 라인의 끝과 매치한다.

```
>>> 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='}'>
```

    - `'$'`를 매치하고 싶다면 `\$` 또는 `[$]`을 사용하자

- `\A`
    - 문자열의 시작과 매치한다.
    - MULTILINE 모드에서는 ^과 다르다. ^는 newline 문자 이후에도 일치한다.
- `\Z`
    - 문자열의 끝과만 매치한다.
- `\b`
    - word boundary. 단어의 처음 또는 끝에만 매치한다.
    - 단어의 끝은 공백 또는 비 영숫자 문자로 표시된다.

```
>>> 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
```

    - 파이썬 문자열 리터럴은 `\b`를 백스페이스로 봄에 주의!

```
>>> 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'>
```

- `\B`
    - word boundary가 아닌 위치에만 매치한다

Grouping

- 반복문에서도 가능하다

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

- group 0 은 전체 RE 이다.

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

- 그룹은 중첩될 수 있으며, 왼쪽에서부터 번호를 매긴다

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