Case Study: Street Addresses

This series of examples was inspired by a real-life problem I had in my day job several years ago, when I needed to scrub and standardize street addresses exported from a legacy system before importing them into a newer system. (See, I don't just make this stuff up; it's actually useful.) This example shows how I approached the problem.

```
s = '100 NORTH MAIN ROAD'
print(s.replace('ROAD', 'RD.'))              #①
#'100 NORTH MAIN RD.'

s = '100 NORTH BROAD ROAD'
print(s.replace('ROAD', 'RD.'))              #②
#'100 NORTH BRD. RD.'

print(s[:-4] + s[-4:].replace('ROAD', 'RD.')) #③
#'100 NORTH BROAD RD.'

import re                                    #④
print(re.sub('ROAD$', 'RD.', s))             #⑤
#'100 NORTH BROAD RD.'
```

① My goal is to standardize a street address so that `'ROAD'` is always abbreviated as `'RD.'`. At first glance, I thought this was simple enough that I could just use the string method `replace()`. After all, all the data was already uppercase, so case mismatches would not be a problem. And the search string, `'ROAD'`, was a constant. And in this deceptively simple example, `s.replace()` does indeed work.

② Life, unfortunately, is full of counterexamples, and I quickly discovered this one. The problem here is that `'ROAD'` appears twice in the address, once as part of the street name `'BROAD'` and once as its own word. The `replace()` method sees these two occurrences and blindly replaces both of them; meanwhile, I see my addresses getting destroyed.

③ To solve the problem of addresses with more than one ‘ROAD’ substring, you could resort to something like this: only search and replace `'ROAD'` in the last four characters of the address `(s[-4:])`, and leave the string alone (`s[:-4]`). But you can see that this is already getting unwieldy. For example, the pattern is dependent on the length of the string you’re replacing. (If you were replacing `'STREET'` with `'ST.'`, you would need to use `s[:-6]` and `s[-6:].replace(...)`.) Would you like to come back in six months and debug this? I know I wouldn’t.

④ It’s time to move up to regular expressions. In Python, all functionality related to regular expressions is contained in the `re` module.

⑤ Take a look at the first parameter: `'ROAD$'`. This is a simple regular expression that matches `'ROAD'` only when it occurs at the end of a string. The `$` means “end of the string.” (There is a corresponding character, the caret `^`, which means “beginning of the string.”) Using the `re.sub()` function, you search the string s for the regular expression `'ROAD$'` and replace it with `'RD.'`. This matches the `ROAD` at the end of the string `s`, but does *not* match the `ROAD` that’s part of the word `BROAD`, because that’s in the middle of `s`.

^ matches the start of a string. $ matches the end of a string.

Continuing with my story of scrubbing addresses, I soon discovered that the previous example, matching `'ROAD'` at the end of the address, was not good enough, because not all addresses included a street designation at all. Some addresses simply ended with the street name. I got away with it most of the time, but if the street name was `'BROAD'`, then the regular expression would match `'ROAD'` at the end of the string as part of the word `'BROAD'`, which is not what I wanted.

```
import re

s = '100 BROAD'
print (re.sub('ROAD$', 'RD.', s))
#'100 BRD.'

print (re.sub('\\bROAD$', 'RD.', s))   #①
#'100 BROAD'
```

```
print (re.sub(r'\bROAD$', 'RD.', s))   #②
#'100 BROAD'


s = '100 BROAD ROAD APT. 3'
print (re.sub(r'\bROAD$', 'RD.', s))   #③
#'100 BROAD ROAD APT. 3'

print (re.sub(r'\bROAD\b', 'RD.', s))  #④
#'100 BROAD RD. APT 3'
```

① What I really wanted was to match `'ROAD'` when it was at the end of the string *and* it was its own word (and not a part of some larger word). To express this in a regular expression, you use `\b`, which means "a word boundary must occur right here." In Python, this is complicated by the fact that the `'\'` character in a string must itself be escaped. This is sometimes referred to as the backslash plague, and it is one reason why regular expressions are easier in Perl than in Python. On the down side, Perl mixes regular expressions with other syntax, so if you have a bug, it may be hard to tell whether it's a bug in syntax or a bug in your regular expression.

② To work around the backslash plague, you can use what is called a *raw string*, by prefixing the string with the letter `r`. This tells Python that nothing in this string should be escaped; `'\t'` is a tab character, but `r'\t'` is really the backslash character `\` followed by the letter `t`. I recommend always using raw strings when dealing with regular expressions; otherwise, things get too confusing too quickly (and regular expressions are confusing enough already).

③ *sigh* Unfortunately, I soon found more cases that contradicted my logic. In this case, the street address contained the word `'ROAD'` as a whole word by itself, but it wasn't at the end, because the address had an apartment number after the street designation. Because `'ROAD'` isn't at the very end of the string, it doesn't match, so the entire call to `re.sub()` ends up replacing nothing at all, and you get the original string back, which is not what you want.

④ To solve this problem, I removed the `$` character and added another `\b`. Now the regular expression reads "match `'ROAD'` when it's a whole word by itself anywhere in the string," whether at the end, the beginning, or somewhere in the middle.