I Know, Let's Use Regular Expressions!

So you're looking at words, which, at least in English, means you're looking at strings of characters. You have rules that say you need to find different combinations of characters, then do different things to them. This sounds like a job for regular expressions!

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):              #①
        return re.sub('$', 'es', noun)         #②
    elif re.search('[^aeioudgkprt]h$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

① This is a regular expression, but it uses a syntax you didn't see in Regular Expressions. The square brackets mean "match exactly one of these characters." So `[sxz]` means "`s`, or `x`, or `z`", but only one of them. The `$` should be familiar; it matches the end of string. Combined, this regular expression tests whether noun ends with `s`, `x`, or `z`.

② This `re.sub()` function performs regular expression-based string substitutions.

Let's look at regular expression substitutions in more detail.

```
import re
print (re.search('[abc]', 'Mark'))    #①
#<_sre.SRE_Match object at 0x001C1FA8>

print (re.sub('[abc]', 'o', 'Mark'))  #②
#Mork

print (re.sub('[abc]', 'o', 'rock'))  #③
#rook
```

```
print (re.sub('[abc]', 'o', 'caps'))  #④
#oops
```

① Does the string Mark contain `a`, `b`, or `c`? Yes, it contains `a`.

② OK, now find `a`, `b`, or `c`, and replace it with `o`. `Mark` becomes `Mork`.

③ The same function turns `rock` into `rook`.

④ You might think this would turn `caps` into `oaps`, but it doesn't. `re.sub` replaces all of the matches, not just the first one. So this regular expression turns `caps` into `oops`, because both the `c` and the `a` get turned into `o`.

And now, back to the `plural()` function…

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)          #①
    elif re.search('[^aeioudgkprt]h$', noun):   #②
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):         #③
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

① Here, you're replacing the end of the string (matched by `$`) with the string `es`. In other words, adding `es` to the string. You could accomplish the same thing with string concatenation, for example `noun + 'es'`, but I chose to use regular expressions for each rule, for reasons that will become clear later in the chapter.

② Look closely, this is another new variation. The `^` as the first character inside the square brackets means something special: negation. `[^abc]` means "any single character except `a`, `b`, or `c`". So `[^aeioudgkprt]` means any character except `a`, `e`, `i`, `o`, `u`, `d`, `g`, `k`, `p`, `r`, or `t`. Then that character needs to be followed by `h`, followed by end of string. You're looking for words that end in H where the H can be heard.

③ Same pattern here: match words that end in Y, where the character before the Y is *not* `a`, `e`, `i`, `o`, or `u`. You're looking for words that end in Y that

sounds like I.

Let's look at negation regular expressions in more detail.

```
import re
print (re.search('[^aeiou]y$', 'vacancy'))  #①
#<_sre.SRE_Match object at 0x001C1FA8>

print (re.search('[^aeiou]y$', 'boy'))      #②
#None

print (re.search('[^aeiou]y$', 'day'))
#None

print (re.search('[^aeiou]y$', 'pita'))     #③
#None
```

① `vacancy` matches this regular expression, because it ends in `cy`, and `c` is not `a`, `e`, `i`, `o`, or `u`.

② `boy` does not match, because it ends in `oy`, and you specifically said that the character before the `y` could not be `o`. `day` does not match, because it ends in `ay`.

③ `pita` does not match, because it does not end in `y`.

```
import re
print (re.sub('y$', 'ies', 'vacancy') )             #①
#vacancies

print (re.sub('y$', 'ies', 'agency'))
#agencies

print (re.sub('([^aeiou])y$', r'\1ies', 'vacancy'))  #②
#vacancies
```

① This regular expression turns `vacancy` into `vacancies` and `agency` into agencies, which is what you wanted. Note that it would also turn `boy` into `boies`, but that will never happen in the function because you did that `re.search` first to find out whether you should do this `re.sub`.

② Just in passing, I want to point out that it is possible to combine these two regular expressions (one to find out if the rule applies, and another to actually apply it) into a single regular expression. Here's what that would look like. Most of it should look familiar: you're using a remembered group, which you learned in Case study: Parsing Phone Numbers. The group is used to remember the character before the letter `y`. Then in the substitution string, you use a new syntax, `\1`, which means "hey, that first group you remembered? put it right here." In this case, you remember the `c` before the `y`; when you do the substitution, you substitute `c` in place of `c`, and ies in place of `y`. (If you have more than one remembered group, you can use `\2` and `\3` and so on.)

Regular expression substitutions are extremely powerful, and the `\1` syntax makes them even more powerful. But combining the entire operation into one regular expression is also much harder to read, and it doesn't directly map to the way you first described the pluralizing rules. You originally laid out rules like "if the word ends in S, X, or Z, then add ES". If you look at this function, you have two lines of code that say "if the word ends in S, X, or Z, then add ES". It doesn't get much more direct than that.