A List Of Patterns

Defining separate named functions for each match and apply rule isn't really necessary. You never call them directly; you add them to the `rules` sequence and call them through there. Furthermore, each function follows one of two patterns. All the match functions call `re.search()`, and all the apply functions call `re.sub()`. Let's factor out the patterns so that defining new rules can be easier.

```
import re

def build_match_and_apply_functions(pattern, search, replace):
    def matches_rule(word):                                          #①
        return re.search(pattern, word)
    def apply_rule(word):                                            #②
        return re.sub(search, replace, word)
    return (matches_rule, apply_rule)                                #③
```

① `build_match_and_apply_functions()` is a function that builds other functions dynamically. It takes pattern, search and replace, then defines a `matches_rule()` function which calls `re.search()` with the pattern that was passed to the `build_match_and_apply_functions()` function, and the word that was passed to the `matches_rule()` function you're building. Whoa.

② Building the apply function works the same way. The apply function is a function that takes one parameter, and calls `re.sub()` with the `search` and `replace` parameters that were passed to the `build_match_and_apply_functions()` function, and the word that was passed to the `apply_rule()` function you're building. This technique of using the values of outside parameters within a dynamic function is called *closures*. You're essentially defining constants within the apply function you're building: it takes one parameter (`word`), but it then acts on that plus two other values (`search` and `replace`) which were set when you defined the apply function.

③ Finally, the `build_match_and_apply_functions()` function returns a tuple of

two values: the two functions you just created. The constants you defined within those functions (`pattern` within the `matches_rule()` function, and `search` and `replace` within the `apply_rule()` function) stay with those functions, even after you return from `build_match_and_apply_functions()`. That's insanely cool.

If this is incredibly confusing (and it should be, this is weird stuff), it may become clearer when you see how to use it.

```
import re
def build_match_and_apply_functions(pattern, search, replace):
  pass

patterns = (('[sxz]$','$','es'),('[^aeioudgkprt]h$','$','es'),('(qu|[^aeiou])y$','y$','ies'),
           #①
rules = [build_match_and_apply_functions(pattern, search, replace)
         for (pattern, search, replace) in patterns]
```

① Our pluralization "rules" are now defined as a tuple of tuples of *strings* (not functions). The first string in each group is the regular expression pattern that you would use in `re.search()` to see if this rule matches. The second and third strings in each group are the search and replace expressions you would use in `re.sub()` to actually apply the rule to turn a noun into its plural.

② There's a slight change here, in the fallback rule. In the previous example, the `match_default()` function simply returned `True`, meaning that if none of the more specific rules matched, the code would simply add an `s` to the end of the given word. This example does something functionally equivalent. The final regular expression asks whether the word has an end (`$` matches the end of a string). Of course, every string has an end, even an empty string, so this expression always matches. Thus, it serves the same purpose as the `match_default()` function that always returned `True`: it ensures that if no more specific rule matches, the code adds an `s` to the end of the given word.

③ This line is magic. It takes the sequence of strings in `patterns` and turns them into a sequence of functions. How? By "mapping" the strings to the `build_match_and_apply_functions()` function. That is, it takes each triplet of strings and calls the `build_match_and_apply_functions()` function with those three strings as arguments. The `build_match_and_apply_functions()` function returns a tuple of two functions. This means that `rules` ends up being functionally equivalent to the previous example: a list of tuples, where each

tuple is a pair of functions. The first function is the match function that calls

`re.search()`, and the second function is the apply function that calls `re.sub()`.

Rounding out this version of the script is the main entry point, the `plural()` function.

```
def plural(noun):
    for matches_rule, apply_rule in rules:  #①
        if matches_rule(noun):
            return apply_rule(noun)
```

① Since the `rules` list is the same as the previous example (really, it is), it should come as no surprise that the `plural()` function hasn't changed at all. It's completely generic; it takes a list of rule functions and calls them in order. It doesn't care how the rules are defined. In the previous example, they were defined as separate named functions. Now they are built dynamically by mapping the output of the `build_match_and_apply_functions()` function onto a list of raw strings. It doesn't matter; the `plural()` function still works the same way.