A List Of Functions

Now you're going to add a level of abstraction. You started by defining a list of rules: if this, do that, otherwise go to the next rule. Let's temporarily complicate part of the program so you can simplify another part.

```
import re

def match_sxz(noun):
    return re.search('[sxz]$', noun)

def apply_sxz(noun):
    return re.sub('$', 'es', noun)

def match_h(noun):
    return re.search('[^aeioudgkprt]h$', noun)

def apply_h(noun):
    return re.sub('$', 'es', noun)

def match_y(noun):                              #①
    return re.search('[^aeiou]y$', noun)

def apply_y(noun):                              #②
    return re.sub('y$', 'ies', noun)

def match_default(noun):
    return True

def apply_default(noun):
    return noun + 's'

rules = ((match_sxz, apply_sxz),                #③
         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
         )

def plural(noun):
    for matches_rule, apply_rule in rules:      #④
        if matches_rule(noun):
            return apply_rule(noun)
```

① Now, each match rule is its own function which returns the results of calling the `re.search()` function.

② Each apply rule is also its own function which calls the `re.sub()` function to apply the appropriate pluralization rule.

③ Instead of having one function (`plural()`) with multiple rules, you have the `rules` data structure, which is a sequence of pairs of functions.

④ Since the rules have been broken out into a separate data structure, the new `plural()` function can be reduced to a few lines of code. Using a `for` loop, you can pull out the match and apply rules two at a time (one match, one apply) from the rules structure. On the first iteration of the `for` loop, `matches_rule` will get `match_sxz`, and `apply_rule` will get `apply_sxz`. On the second iteration (assuming you get that far), `matches_rule` will be assigned `match_h`, and `apply_rule` will be assigned `apply_h`. The function is guaranteed to return something eventually, because the final match rule (`match_default`) simply returns `True`, meaning the corresponding apply rule (`apply_default`) will always be applied.

The “rules” variable is a sequence of pairs of functions.

The reason this technique works is that everything in Python is an object, including functions. The `rules` data structure contains functions — not names of functions, but actual function objects. When they get assigned in the `for` loop, then `matches_rule` and `apply_rule` are actual functions that you can call. On the first iteration of the `for` loop, this is equivalent to calling `matches_sxz(noun)`, and if it returns a match, calling `apply_sxz(noun)`.

If this additional level of abstraction is confusing, try unrolling the function to see the equivalence. The entire `for` loop is equivalent to the following:

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

The benefit here is that the `plural()` function is now simplified. It takes a sequence of rules, defined elsewhere, and iterates through them in a generic fashion.

1. Get a match rule
2. Does it match? Then call the apply rule and return the result.
3. No match? Go to step 1.

The rules could be defined anywhere, in any way. The `plural()` function doesn't care.

Now, was adding this level of abstraction worth it? Well, not yet. Let's consider what it would take to add a new rule to the function. In the first example, it would require adding an if statement to the `plural()` function. In this second example, it would require adding two functions, `match_foo()` and `apply_foo()`, and then updating the `rules` sequence to specify where in the order the new match and apply functions should be called relative to the other rules.

But this is really just a stepping stone to the next section. Let's move on…