A Plural Rule Iterator

Now it's time for the finale. Let's rewrite the plural rules generator as an iterator.

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
        self.pattern_file = open(self.rules_filename, encoding='utf-8')
        self.cache = []

    def __iter__(self):
        self.cache_index = 0
        return self

    def __next__(self):
        self.cache_index += 1
        if len(self.cache) >= self.cache_index:
            return self.cache[self.cache_index - 1]

        if self.pattern_file.closed:
            raise StopIteration

        line = self.pattern_file.readline()
        if not line:
            self.pattern_file.close()
            raise StopIteration

        pattern, search, replace = line.split(None, 3)
        funcs = build_match_and_apply_functions(
            pattern, search, replace)
        self.cache.append(funcs)
        return funcs

rules = LazyRules()
```

So this is a class that implements `__iter__()` and `__next__()`, so it can be used as an iterator. Then, you instantiate the class and assign it to `rules`. This happens just once, on import.

iter(f) calls `f.__iter__`. **next(f)** calls `f.__next__`

Let's take the class one bite at a time.

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
        self.pattern_file = open(self.rules_filename, encoding='utf-8')  #①
        self.cache = []                                                  #②
```

① When we instantiate the `LazyRules` class, open the pattern file but don't read anything from it. (That comes later.)

② After opening the patterns file, initialize the cache. You'll use this cache later (in the `__next__()` method) as you read lines from the pattern file.

Before we continue, let's take a closer look at `rules_filename`. It's not defined within the `__iter__()` method. In fact, it's not defined within any method. It's defined at the class level. It's a *class* variable, and although you can access it just like an instance variable (`self.rules_filename`), it is shared across all instances of the `LazyRules` class.

```
import plural6

r1 = plural6.LazyRules()
r2 = plural6.LazyRules()
print (r1.rules_filename)                                  #①
#plural6-rules.txt

print (r2.rules_filename)
#plural6-rules.txt

r2.rules_filename = 'r2-override.txt'                      #②
print (r2.rules_filename)
#r2-override.txt

print (r1.rules_filename)
#plural6-rules.txt

print (r2.__class__.rules_filename )                       #③
#plural6-rules.txt

r2.__class__.rules_filename = 'papayawhip.txt'  #④
print (r1.rules_filename)
#papayawhip.txt

print (r2.rules_filename)                                 #⑤
#r2-overridetxt
```

① Each instance of the class inherits the `rules_filename` attribute with the value defined by the class.

② Changing the attribute's value in one instance does not affect other instances…

③ …nor does it change the class attribute. You can access the class attribute (as opposed to an individual instance's attribute) by using the special `__class__` attribute to access the class itself.

④ If you change the class attribute, all instances that are still inheriting that value (like `r1` here) will be affected.

⑤ Instances that have overridden that attribute (like `r2` here) will not be affected.

And now back to our show.

```
def __iter__(self):          #①
      self.cache_index = 0
      return self            #②
```

① The `__iter__()` method will be called every time someone — say, a `for` loop — calls `iter(rules)`.

② The one thing that every `__iter__()` method must do is return an iterator. In this case, it returns `self`, which signals that this class defines a `__next__()` method which will take care of returning values throughout the iteration.

```
def __next__(self):                                  #①
      #.
      #.
      #.
      pattern, search, replace = line.split(None, 3)
      funcs = build_match_and_apply_functions(       #②
          pattern, search, replace)
      self.cache.append(funcs)                       #③
      return funcs
```

① The `__next__()` method gets called whenever someone — say, a `fo`r loop — calls `next(rules)`. This method will only make sense if we start at the end and

work backwards. So let's do that.

② The last part of this function should look familiar, at least. The `build_match_and_apply_functions()` function hasn't changed; it's the same as it ever was.

③ The only difference is that, before returning the match and apply functions (which are stored in the tuple `funcs`), we're going to save them in `self.cache`.

Moving backwards...

```
def __next__(self):
        #.
        #.
        #.
        line = self.pattern_file.readline()  #①
        if not line:                         #②
            self.pattern_file.close()
            raise StopIteration              #③
        #.
        #.
        #.
```

① A bit of advanced file trickery here. The `readline()` method (note: singular, not the plural `readlines()`) reads exactly one line from an open file. Specifically, the next line. (*File objects are iterators too! It's iterators all the way down...*)

② If there was a line for `readline()` to read, `line` will not be an empty string. Even if the file contained a blank line, `line` would end up as the one-character string `'\n'` (a carriage return). If `line` is really an empty string, that means there are no more lines to read from the file.

③ When we reach the end of the file, we should close the file and raise the magic `StopIteration` exception. Remember, we got to this point because we needed a match and apply function for the next rule. The next rule comes from the next line of the file... but there is no next line! Therefore, we have no value to return. The iteration is over. (♫ The party's over... ♫)

Moving backwards all the way to the start of the `__next__()` method...

```
def __next__(self):
        self.cache_index += 1
        if len(self.cache) >= self.cache_index:
```

```
            return self.cache[self.cache_index - 1]     #①

        if self.pattern_file.closed:

            raise StopIteration                         #②
        #.
        #.
        #.
```

① `self.cache` will be a list of the functions we need to match and apply individual rules. (At least that should sound familiar!) `self.cache_index` keeps track of which cached item we should return next. If we haven't exhausted the cache yet (i.e. if the length of `self.cache` is greater than s`elf.cache_index`), then we have a cache hit! Hooray! We can return the match and apply functions from the cache instead of building them from scratch.

② On the other hand, if we don't get a hit from the cache, and the file object has been closed (which could happen, further down the method, as you saw in the previous code snippet), then there's nothing more we can do. If the file is closed, it means we've exhausted it — we've already read through every line from the pattern file, and we've already built and cached the match and apply functions for each pattern. The file is exhausted; the cache is exhausted; I'm exhausted. Wait, what? Hang in there, we're almost done.

Putting it all together, here's what happens when:

- When the module is imported, it creates a single instance of the `LazyRules` class, called `rules`, which opens the pattern file but does not read from it.
- When asked for the first match and apply function, it checks its cache but finds the cache is empty. So it reads a single line from the pattern file, builds the match and apply functions from those patterns, and caches them.
- Let's say, for the sake of argument, that the very first rule matched. If so, no further match and apply functions are built, and no further lines are read from the pattern file.
- Furthermore, for the sake of argument, suppose that the caller calls the `plural()` function *again* to pluralize a different word. The `for` loop in the `plural()` function will call `iter(rules)`, which will reset the cache index but will not reset the open file object.

- The first time through, the `for` loop will ask for a value from `rules`, which will invoke its `__next__()` method. This time, however, the cache is primed with a single pair of match and apply functions, corresponding to the patterns in the first line of the pattern file. Since they were built and cached in the course of pluralizing the previous word, they're retrieved from the cache. The cache index increments, and the open file is never touched.
- Let's say, for the sake of argument, that the first rule does not match this time around. So the `for` loop comes around again and asks for another value from `rules`. This invokes the `__next__()` method a second time. This time, the cache is exhausted — it only contained one item, and we're asking for a second — so the `__next__()` method continues. It reads another line from the open file, builds match and apply functions out of the patterns, and caches them.
- This read-build-and-cache process will continue as long as the rules being read from the pattern file don't match the word we're trying to pluralize. If we do find a matching rule before the end of the file, we simply use it and stop, with the file still open. The file pointer will stay wherever we stopped reading, waiting for the next `readline()` command. In the meantime, the cache now has more items in it, and if we start all over again trying to pluralize a new word, each of those items in the cache will be tried before reading the next line from the pattern file.

We have achieved pluralization nirvana.

1. **Minimal startup cost**. The only thing that happens on `import` is instantiating a single class and opening a file (but not reading from it).
2. **Maximum performance**. The previous example would read through the file and build functions dynamically every time you wanted to pluralize a word. This version will cache functions as soon as they're built, and in the worst case, it will only read through the pattern file once, no matter how many words you pluralize.
3. **Separation of code and data**. All the patterns are stored in a separate file. Code is code, and data is data, and never the twain shall meet.

> *Is this really nirvana? Well, yes and no. Here's something to consider with the `LazyRules` example: the pattern file is opened (during `__init__()`), and*

it remains open until the final rule is reached. Python will eventually close the file when it exits, or after the last instantiation of the `LazyRules` class is destroyed, but still, that could be a long time. If this class is part of a long-running Python process, the Python interpreter may never exit, and the `LazyRules` object may never get destroyed.

There are ways around this. Instead of opening the file during `__init__()` and leaving it open while you read rules one line at a time, you could open the file, read all the rules, and immediately close the file. Or you could open the file, read one rule, save the file position with the `tell()` method, close the file, and later re-open it and use the `seek()` method to continue reading where you left off. Or you could not worry about it and just leave the file open, like this example code does. Programming is design, and design is all about trade-offs and constraints. Leaving a file open too long might be a problem; making your code more complicated might be a problem. Which one is the bigger problem depends on your development team, your application, and your runtime environment.