CHAPTER 20

Project 1: Instant Markup

In this project, you see how to use Python's excellent text-processing capabilities, including the capability to use regular expressions to change a plain-text file into one marked up in a language such as HTML or XML. You need such skills if you want to use text written by people who don't know these languages in a system that requires the contents to be marked up.

Don't speak fluent XML? Don't worry about that—if you have only a passing acquaintance with HTML, you'll do fine in this chapter. If you need an introduction to HTML, I suggest you take a look at Dave Raggett's excellent guide "Getting Started with HTML" at the World Wide Web Consortium's site (`http://www.w3.org/MarkUp/Guide`). For an example of XML use, see Chapter 22.

Let's start by implementing a simple prototype that does the basic processing, and then extend that program to make the markup system more flexible.

What's the Problem?

You want to add some formatting to a plain-text file. Let's say you've been handed the file from someone who can't be bothered with writing in HTML, and you need to use the document as a web page. Instead of adding all the necessary tags manually, you want your program to do it automatically.

Note In recent years, this sort of "plain-text markup" has, in fact, become quite common, probably mainly because of the explosion of wiki and blog software with plain-text interfaces. See the section "Further Exploration" at the end of this chapter for more information.

Your task is basically to classify various text elements, such as headlines and emphasized text, and then clearly mark them. In the specific problem addressed here, you add HTML markup to the text, so the resulting document can be displayed in a web browser and used as a web page. However, once you have built your basic engine, there is no reason why you can't add other kinds of markup (such as various forms of XML or perhaps $\LaTeX$ codes). After analyzing a text file, you can even perform other tasks, such as extracting all the headlines to make a table of contents.

Note LaTeX is another markup system (based on the TeX typesetting program) for creating various types of technical documents. I mention it here only as an example of other uses for your program. If you want to know more, you can visit the TeX Users Group web site at `http://www.tug.org`.

The text you're given may contain some clues (such as emphasized text being marked `*like this*`), but you'll probably need some ingenuity in making your program guess how the document is structured.

Before starting to write your prototype, let's define some goals:

- The input shouldn't be required to contain artificial codes or tags.
- You should be able to deal with both different blocks, such as headings, paragraphs, and list items, and in-line text, such as emphasized text or URLs.
- Although this implementation deals with HTML, it should be easy to extend it to other markup languages.

You may not be able to reach these goals fully in the first version of your program, but that's the point of the prototype, You write the prototype to find flaws in your original ideas and to learn more about how to write a program that solves your problem.

Tip If you can, it's probably a good idea to modify your original program incrementally rather than beginning from scratch. In the interest of clarity, I give you two completely separate versions of the program here.

Useful Tools

Consider what tools might be needed in writing this program:

- You certainly need to read from and write to files (see Chapter 11), or at least read from standard input (`sys.stdin`) and output with `print`.
- You probably need to iterate over the lines of the input (see Chapter 11).
- You need a few string methods (see Chapter 3).
- Perhaps you'll use a generator or two (see Chapter 9).
- You probably need the `re` module (see Chapter 10).

If any of these concepts seem unfamiliar to you, you should perhaps take a moment to refresh your memory.

Preparations

Before you start coding, you need some way of assessing your progress; you need a test suite. In this project, a single test may suffice: a test *document* (in plain text). Listing 20-1 contains sample text that you want to mark up automatically.

Listing 20-1. *A Sample Plain-Text Document (test_input.txt)*

```
Welcome to World Wide Spam, Inc.


These are the corporate web pages of *World Wide Spam*, Inc. We hope
you find your stay enjoyable, and that you will sample many of our
products.

A short history of the company

World Wide Spam was started in the summer of 2000. The business
concept was to ride the dot-com wave and to make money both through
bulk email and by selling canned meat online.

After receiving several complaints from customers who weren't
satisfied by their bulk email, World Wide Spam altered their profile,
and focused 100% on canned goods. Today, they rank as the world's
13,892nd online supplier of SPAM.

Destinations

From this page you may visit several of our interesting web pages:

  - What is SPAM? (http://wwspam.fu/whatisspam)

  - How do they make it? (http://wwspam.fu/howtomakeit)

  - Why should I eat it? (http://wwspam.fu/whyeatit)

How to get in touch with us

You can get in touch with us in *many* ways: By phone (555-1234), by
email (wwspam@wwspam.fu) or by visiting our customer feedback page
(http://wwspam.fu/feedback).
```

To test your implementation, just use this document as input and view the results in a web browser, or perhaps examine the added tags directly.

Note It is usually better to have an automated test suite than to check your test results manually. (Do you see any way of automating this test?)

First Implementation

One of the first things you need to do is split the text into paragraphs. It's obvious from Listing 20-1 that the paragraphs are separated by one or more empty lines. A better word than *paragraph* might be *block*, because this name can apply to headlines and list items as well.

Finding Blocks of Text

A simple way to find these blocks is to collect all the lines you encounter until you find an empty line, and then return the lines you have collected so far. That would be one block. Then, you could start all over again. You don't need to bother collecting empty lines, and you won't return empty blocks (where you have encountered more than one empty line). Also, you should make sure that the last line of the file is empty; otherwise, you won't know when the last block is finished. (There are other ways of finding out, of course.)

Listing 20-2 shows an implementation of this approach.

Listing 20-2. *A Text Block Generator (util.py)*

```
def lines(file):
    for line in file: yield line
    yield '\n'

def blocks(file):
    block = []
    for line in lines(file):
        if line.strip():
            block.append(line)
        elif block:
            yield ''.join(block).strip()
            block = []
```

The `lines` generator is just a little utility that tacks an empty line onto the end of the file. The `blocks` generator implements the approach described. When a block is yielded, its lines are joined, and the resulting string is stripped, giving you a single string representing the block, with excessive whitespace at either end (such as list indentations or newlines) removed. (If you don't like this way of finding paragraphs, I'm sure you can figure out several other approaches. It might even be fun to see how many you can invent.)

■**Note** In older versions of Python (prior to 2.3), you needed to add `from __future__ import generators` as the first line of this module. See also the section "Simulating Generators" in Chapter 9.

I've put the code in the file `util.py`, which means that you can import the utility generators in your program later.

Adding Some Markup

With the basic functionality from Listing 20-2, you can create a simple markup script. The basic steps of this program are as follows:

1. Print some beginning markup.
2. For each block, print the block enclosed in paragraph tags.
3. Print some ending markup.

This isn't very difficult, but it's not extremely useful either. Let's say that instead of enclosing the first block in paragraph tags, you enclose it in top heading tags (`h1`). Also, you replace any text enclosed in asterisks with emphasized text (using `em` tags). At least that's a *bit* more useful. Given the `blocks` function, and using `re.sub`, the code is very simple. See Listing 20-3.

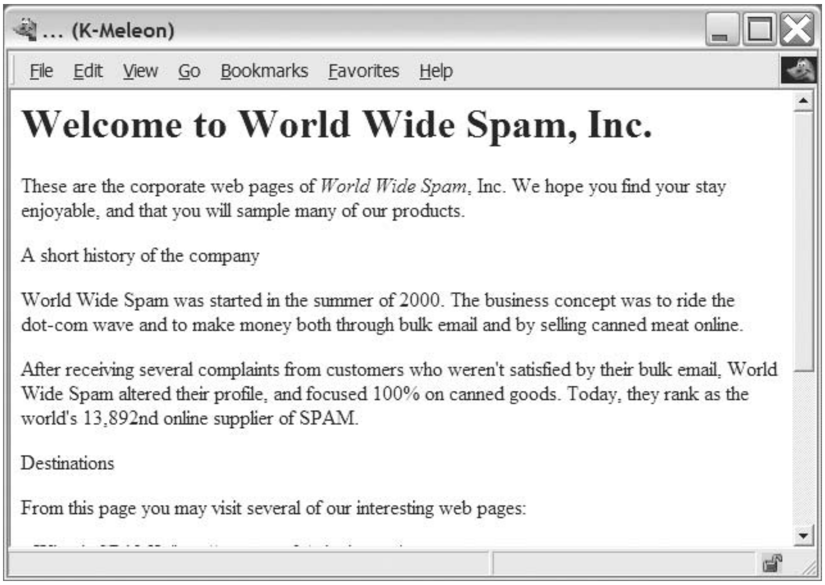
Listing 20-3. *A Simple Markup Program (simple_markup.py)*

```
import sys, re
from util import *

print '<html><head><title>...</title><body>'

title = True
for block in blocks(sys.stdin):
    block = re.sub(r'\*(.+?)\*', r'<em>\1</em>', block)
    if title:
        print '<h1>'
        print block
        print '</h1>'
        title = False
    else:
        print '<p>'
        print block
        print '</p>'

print '</body></html>'
```

This program can be executed on the sample input as follows:

```
$ python simple_markup.py < test_input.txt > test_output.html
```

The file `test_output.html` will then contain the generated HTML code. Figure 20-1 shows how this HTML code looks in a web browser.



Welcome to World Wide Spam, Inc.

These are the corporate web pages of *World Wide Spam*, Inc. We hope you find your stay enjoyable, and that you will sample many of our products.

A short history of the company

World Wide Spam was started in the summer of 2000. The business concept was to ride the dot-com wave and to make money both through bulk email and by selling canned meat online.

After receiving several complaints from customers who weren't satisfied by their bulk email, World Wide Spam altered their profile, and focused 100% on canned goods. Today, they rank as the world's 13,892nd online supplier of SPAM.

Destinations

From this page you may visit several of our interesting web pages:

Figure 20-1. *The first attempt at generating a web page*

Although not very impressive, this prototype does perform some important tasks. It divides the text into blocks that can be handled separately, and it applies a filter (consisting of a call to `re.sub`) to each block in turn. This seems like a good approach to use in your final program.

Now what would happen if you tried to extend this prototype? You would probably add checks inside the `for` loop to see whether the block was a heading, a list item, or something else. You would add more regular expressions. It could quickly grow into a mess. Even more important, it would be very difficult to make it output anything other than HTML; and one of the goals of this project is to make it easy to add other output formats. Let's assume you want to refactor your program and structure it a bit differently.

Second Implementation

So, what did you learn from this first implementation? To make it more extensible, you need to make your program more *modular* (divide the functionality into independent components). One way of achieving modularity is through object-oriented design (see Chapter 7). You need to find some abstractions to make your program more manageable as its complexity grows. Let's begin by listing some possible components:

- **A parser**: Add an object that reads the text and manages the other classes.
- **Rules**: You can make one rule for each type of block. The rule should be able to detect the applicable block type and to format it appropriately.
- **Filters**: Use filters to wrap up some regular expressions to deal with in-line elements.
- **Handlers**: The parser uses handlers to generate output. Each handler can produce a different kind of markup.

Although this isn't a very detailed design, at least it gives you some ideas about how to divide your code into smaller parts and make each part manageable.

Handlers

Let's begin with the handlers. A handler is responsible for generating the resulting marked-up text, but it receives detailed instructions from the parser. Let's say it has a pair of methods for each block type: one for starting the block and one for ending it. For example, it might have the methods `start_paragraph` and `end_paragraph` to deal with paragraph blocks. For HTML, these could be implemented as follows:

```
class HTMLRenderer:
    def start_paragraph(self):
        print '<p>'
    def end_paragraph(self):
        print '</p>'
```

Of course, you'll need similar methods for other block types. (For the full code of the `HTMLRenderer` class, see Listing 20-4 later in this chapter.) This seems flexible enough. If you wanted some other type of markup, you would just make another handler (or renderer) with other implementations of the start and end methods.

Note The term *handler* (as opposed to *renderer*, for example) was chosen to indicate that it handles the method calls generated by the parser (see also the following section, "A Handler Superclass"). It doesn't *have* to render the text in some markup language, as `HTMLRenderer` does. A similar handler mechanism is used in the XML parsing scheme called SAX, which is explained in Chapter 22.

How do you deal with regular expressions? As you may recall, the `re.sub` function can take a function as its second argument (the replacement). This function is called with the match object, and its return value is inserted into the text. This fits nicely with the handler philosophy discussed previously—you just let the handlers implement the replacement methods. For example, emphasis can be handled like this:

```
    def sub_emphasis(self, match):
        return '<em>%s</em>' % match.group(1)
```

If you don't understand what the group method does, perhaps you should take another look at the re module, described in Chapter 10.

In addition to the start, end, and sub methods, you'll have a method called feed, which you use to feed actual text to the handler. In your simple HTML renderer, you'll just implement it like this:

```
    def feed(self, data):
        print data
```

A Handler Superclass

In the interest of flexibility, let's add a Handler class, which will be the superclass of your handlers and will take care of some administrative details. Instead of needing to call the methods by their full name (for example, start_paragraph), it may at times be useful to handle the block types as strings (for example, 'paragraph') and supply the handler with those. You can do this by adding some generic methods called start(type), end(type), and sub(type). In addition, you can make start, end, and sub check whether the corresponding methods (such as start_paragraph for start('paragraph')) are really implemented, and do nothing if no such method is found. An implementation of this Handler class follows. (This code is taken from the module handlers shown later in Listing 20-4.)

```
class Handler:
    def callback(self, prefix, name, *args):
        method = getattr(self, prefix+name, None)
        if callable(method): return method(*args)
    def start(self, name):
        self.callback('start_', name)
    def end(self, name):
        self.callback('end_', name)
    def sub(self, name):
        def substitution(match):
            result = self.callback('sub_', name, match)
            if result is None: match.group(0)
            return result
        return substitution
```

Note This code requires nested scopes, which are not available prior to Python 2.1. If, for some reason, you're using Python 2.1, you need to add the line `from __future__ import nested_scopes` at the top of the handlers module. (To some degree, nested scopes can be simulated with default arguments. See the sidebar "Nested Scopes" in Chapter 6.) Also, callable is not available in Python 3.0. To get around that, you could simply use a try/except statement to see if you're able to call it.

Several things in this code warrant some explanation:

- The callback method is responsible for finding the correct method (such as start_paragraph), given a prefix (such as 'start_') and a name (such as 'paragraph'). It performs its task by using getattr with None as the default value. If the object returned from getattr is callable, it is called with any additional arguments supplied. So, for example, calling handler.callback ('start_', 'paragraph') calls the method handler.start_paragraph with no arguments, given that it exists.
- The start and end methods are just helper methods that call callback with the respective prefixes start_ and end_.
- The sub method is a bit different. It doesn't call callback directly, but returns a new function, which is used as the replacement function in re.sub (which is why it takes a match object as its only argument).

Let's consider an example. Say HTMLRenderer is a subclass of Handler and it implements the method sub_emphasis as described in the previous section (see Listing 20-4 for the actual code of handlers.py). Let's say you have an HTMLRenderer instance in the variable handler:

```
>>> from handlers import HTMLRenderer
>>> handler = HTMLRenderer()
```

What then will handler.sub('emphasis') do?

```
>>> handler.sub('emphasis')
<function substitution at 0x168cf8>
```

It returns a function (substitution) that basically calls the handler.sub_emphasis method when you call it. That means that you can use this function in a re.sub statement:

```
>>> import re
>>> re.sub(r'\*(.+?)\*', handler.sub('emphasis'), 'This *is* a test')
'This <em>is</em> a test'
```

Magic! (The regular expression matches occurrences of text bracketed by asterisks, which I'll discuss shortly.) But why go to such lengths? Why not just use r'<em>\1</em>', as in the simple version? Because then you would be committed to using the em tag, but you want the handler to be able to decide which markup to use. If your handler were a (hypothetical) LaTeXRenderer, for example, you might get another result altogether:

```
>> re.sub(r'\*(.+?)\*', handler.sub('emphasis'), 'This *is* a test')
'This \emph{is} a test'
```

The markup has changed, but the code has not.

We also have a backup, in case no substitution is implemented. The callback method tries to find a suitable sub_something method, but if it doesn't find one, it returns None. Because your function is a re.sub replacement function, you *don't* want it to return None. Instead, if you do not find a substitution method, you just return the original match without any modifications. If the callback returns None, substitution (inside sub) returns the original matched text (match.group(0)) instead.

Rules

Now that you've made the handlers quite extensible and flexible, it's time to turn to the parsing (interpretation of the original text). Instead of making one big `if` statement with various conditions and actions, such as in the simple markup program, let's make the rules a separate kind of object.

The rules are used by the main program (the parser), which must determine which rules are applicable for a given block, and then make each rule do what is needed to transform the block. In other words, a rule must be able to do the following:

- Recognize blocks where it applies (the *condition*).
- Transform blocks (the *action*).

So each rule object must have two methods: `condition` and `action`.

The `condition` method needs only one argument: the block in question. It should return a Boolean value indicating whether the rule is applicable to the given block.

■ **Tip** For complex rule parsing, you might want to give the rule object access to some state variables as well, so it knows more about what has happened so far, or which other rules have or have not been applied.

The `action` method also needs the block as an argument, but to be able to affect the output, it must also have access to the handler object.

In many circumstances, only one rule may be applicable; that is, if you find that a headline rule is used (indicating that the block is a headline), you should *not* attempt to use the paragraph rule. A simple implementation of this would be to have the parser try the rules one by one, and stop the processing of the block once one of the rules is triggered. This would be fine in general, but as you'll see, sometimes a rule may not preclude the execution of other rules. Therefore, you add another piece of functionality to your action method: it returns a Boolean value indicating whether the rule processing for the current block should stop. (You could also use an exception for this, similarly to the `StopIteration` mechanism of iterators.)

Pseudocode for the headline rule might be as follows:

```
class HeadlineRule:
    def condition(self, block):
        if the block fits the definition of a headline, return True;
        otherwise, return False.
    def action(self, block, handler):
        call methods such as handler.start('headline'), handler.feed(block) and
        handler.end('headline').
        because we don't want to attempt to use any other rules,
        return True, which will end the rule processing for this block.
```

A Rule Superclass

Although you don't strictly need a common superclass for your rules, several of them may share the same general action—calling the start, feed, and end methods of the handler with the appropriate type string argument, and then returning True (to stop the rule processing). Assuming that all the subclasses have an attribute called type containing this type name as a string, you can implement your superclass as shown in the code that follows. (The Rule class is found in the rules module; the full code is shown later in Listing 20-5.)

```
class Rule:
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block)
        handler.end(self.type)
        return True
```

The condition method is the responsibility of each subclass. The Rule class and its subclasses are put in the rules module.

Filters

You won't need a separate class for your filters. Given the sub method of your Handler class, each filter can be represented by a regular expression and a name (such as emphasis or url). You see how in the next section, when I show you how to deal with the parser.

The Parser

We've come to the heart of the application: the Parser class. It uses a handler and a set of rules and filters to transform a plain-text file into a marked-up file—in this specific case, an HTML file. Which methods does it need? It needs a constructor to set things up, a method to add rules, a method to add filters, and a method to parse a given file.

The following is the code for the Parser class (from Listing 20-6, later in this chapter, which details markup.py).

```
class Parser:
    """
    A Parser reads a text file, applying rules and controlling a
    handler.
    """
    def __init__(self, handler):
        self.handler = handler
        self.rules = []
        self.filters = []
    def addRule(self, rule):
        self.rules.append(rule)
    def addFilter(self, pattern, name):
        def filter(block, handler):
            return re.sub(pattern, handler.sub(name), block)
        self.filters.append(filter)
    def parse(self, file):
        self.handler.start('document')
        for block in blocks(file):
            for filter in self.filters:
                block = filter(block, self.handler)
            for rule in self.rules:
                if rule.condition(block):
                    last = rule.action(block, self.handler)
                    if last: break
        self.handler.end('document')
```

Although there is quite a lot to digest in this class, most of it isn't very complicated. The constructor simply assigns the supplied handler to an instance variable (attribute) and then initializes two lists: one of rules and one of filters. The `addRule` method adds a rule to the rule list. The `addFilter` method, however, does a bit more work. Like `addRule`, it adds a filter to the filter list, but before doing so, it creates that filter. The filter is simply a function that applies `re.sub` with the appropriate regular expression (pattern) and uses a replacement from the handler, accessed with `handler.sub(name)`.

The `parse` method, although it might look a bit complicated, is perhaps the easiest method to implement because it merely does what you've been planning to do all along. It begins by calling `start('document')` on the handler, and ends by calling `end('document')`. Between these calls, it iterates over all the blocks in the text file. For each block, it applies both the filters and the rules. Applying a filter is simply a matter of calling the `filter` function with the block and handler as arguments, and rebinding the block variable to the result, as follows:

```
block = filter(block, self.handler)
```

This enables each of the filters to do its work, which is replacing parts of the text with marked-up text (such as replacing `*this*` with `<em>this</em>`).

There is a bit more logic in the rule loop. For each rule, there is an `if` statement, checking whether the rule applies by calling `rule.condition(block)`. If the rule applies, `rule.action` is called with the block and handler as arguments. Remember that the `action` method returns a Boolean value indicating whether to finish the rule application for this block. Finishing the rule application is done by setting the variable `last` to the return value of action, and then conditionally breaking out of the `for` loop:

```
if last: break
```

Note You can collapse these two statements into one, eliminating the `last` variable:

```
if rule.action(block, self.handler): break
```

Whether or not to do so is largely a matter of taste. Removing the temporary variable makes the code simpler, but leaving it in clearly labels the return value.

Constructing the Rules and Filters

Now you have all the tools you need, but you haven't created any specific rules or filters yet. The motivation behind much of the code you've written so far is to make the rules and filters as flexible as the handlers. You can write several independent rules and filters and add them to your parser through the `addRule` and `addFilter` methods, making sure to implement the appropriate methods in your handlers.

A complicated rule set makes it possible to deal with complicated documents. However, let's keep it simple for now. Let's create one rule for the title, one rule for other headings, and one for list items. Because list items should be treated collectively as a list, you'll create a separate list rule, which deals with the entire list. Lastly, you can create a default rule for paragraphs, which covers all blocks not dealt with by the previous rules.

We can specify the rules in informal terms as follows:

- A heading is a block that consists of only one line, which has a length of at most 70 characters. If the block ends with a colon, it is not a heading.
- The title is the first block in the document, provided that it is a heading.
- A list item is a block that begins with a hyphen (`-`).
- A list begins between a block that is not a list item and a following list item and ends between a list item and a following block that is not a list item.

These rules follow some of my intuitions about how a text document is structured. Your opinions on this (and your text documents) may differ. Also, the rules have weaknesses (for example, what happens if the document ends with a list item?). Feel free to improve on them.

The complete source code for the rules is shown later in Listing 20-5 (`rules.py`, which also contains the basic `Rule` class).

Let's begin with the heading rule:

```
class HeadingRule(Rule):
    """
    A heading is a single line that is at most 70 characters and
    that doesn't end with a colon.
    """
    type = 'heading'
    def condition(self, block):
        return not '\n' in block and len(block) <= 70 and not block[-1] == ':'
```

The attribute type has been set to the string `'heading'`, which is used by the `action` method inherited from `Rule`. The condition simply checks that the block does not contain a newline (`\n`) character, that its length is at most 70, and that the last character is not a colon.

The title rule is similar, but only works once, for the first block. After that, it ignores all blocks because its attribute `first` has been set to a *false* value.

```
class TitleRule(HeadingRule):
    """
    The title is the first block in the document, provided that it is
    a heading.
    """
    type = 'title'
    first = True

    def condition(self, block):
        if not self.first: return False
        self.first = False
        return HeadingRule.condition(self, block)
```

The list item rule condition is a direct implementation of the preceding specification.

```
class ListItemRule(Rule):
    """
    A list item is a paragraph that begins with a hyphen. As part of
    the formatting, the hyphen is removed.
    """
    type = 'listitem'
    def condition(self, block):
        return block[0] == '-'
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block[1:].strip())
        handler.end(self.type)
        return True
```

Its action is a reimplementation of that found in `Rule`. The only difference is that it removes the first character from the block (the hyphen) and strips away excessive whitespace from the remaining text. The markup provides its own "list bullet," so you won't need the hyphen anymore.

All the rule actions so far have returned `True`. The list rule does not, because it is triggered when you encounter a list item after a nonlist item or when you encounter a nonlist item after a list item. Because it doesn't actually mark up these blocks but merely indicates the beginning and end of a list (a group of list items) you don't want to halt the rule processing—so it returns `False`.

```
class ListRule(ListItemRule):
    """
    A list begins between a block that is not a list item and a
    subsequent list item. It ends after the last consecutive list
    item.
    """
    type = 'list'
    inside = False
    def condition(self, block):
        return True
    def action(self, block, handler):
        if not self.inside and ListItemRule.condition(self, block):
            handler.start(self.type)
            self.inside = True
        elif self.inside and not ListItemRule.condition(self, block):
            handler.end(self.type)
            self.inside = False
        return False
```

The list rule might require some further explanation. Its condition is always true because you want to examine all blocks. In the action method, you have two alternatives that may lead to action:

- If the attribute `inside` (indicating whether the parser is currently inside the list) is false (as it is initially), and the condition from the list item rule is true, you have just entered a list. Call the appropriate `start` method of the handler, and set the `inside` attribute to `True`.
- Conversely, if `inside` is true, and the list item rule condition is false, you have just left a list. Call the appropriate end method of the handler, and set the `inside` attribute to `False`.

After this processing, the function returns `False` to let the rule handling continue. (This means, of course, that the order of the rules is critical.)

The final rule is `ParagraphRule`. Its condition is always true because it is the "default" rule. It is added as the last element of the rule list, and handles all blocks that aren't dealt with by any other rule.

```
class ParagraphRule(Rule):
    """
    A paragraph is simply a block that isn't covered by any of the
    other rules.
    """
    type = 'paragraph'
    def condition(self, block):
        return True
```

The filters are simply regular expressions. Let's add three filters: one for emphasis, one for URLs, and one for email addresses. Let's use the following three regular expressions:

```
r'\*(.+?)\*'
r'(http://[\.a-zA-Z/]+)'
r'([\.a-zA-Z]+@[\.a-zA-Z]+[a-zA-Z]+)'
```

The first pattern (emphasis) matches an asterisk followed by one or more arbitrary characters (matching as few as possible, hence the question mark), followed by another asterisk. The second pattern (URLs) matches the string `'http://'` (here, you could add more protocols) followed by one or more characters that are dots, letters, or slashes. (This pattern will not match all legal URLs—feel free to improve it.) Finally, the email pattern matches a sequence of letters and dots followed by an at sign (`@`), followed by more letters and dots, finally followed by a sequence of letters, ensuring that you don't end with a dot. (Again—feel free to improve this.)

Putting It All Together

You now just need to create a `Parser` object and add the relevant rules and filters. Let's do that by creating a subclass of `Parser` that does the initialization in its constructor. Then let's use that to parse `sys.stdin`. The final program is shown in Listings 20-4 through 20-6. (These listings depend on the utility code in Listing 20-2.) The final program may be run just like the prototype:

```
$ python markup.py < test_input.txt > test_output.html
```

Listing 20-4. *The Handlers (handlers.py)*

```
class Handler:
    """
    An object that handles method calls from the Parser.

    The Parser will call the start() and end() methods at the
    beginning of each block, with the proper block name as a
    parameter. The sub() method will be used in regular expression
    substitution. When called with a name such as 'emphasis', it will
    return a proper substitution function.
    """
    def callback(self, prefix, name, *args):
        method = getattr(self, prefix+name, None)
        if callable(method): return method(*args)
    def start(self, name):
        self.callback('start_', name)
    def end(self, name):
        self.callback('end_', name)
    def sub(self, name):
        def substitution(match):
            result = self.callback('sub_', name, match)
            if result is None: match.group(0)
            return result
        return substitution
```

```
class HTMLRenderer(Handler):
    """
    A specific handler used for rendering HTML.

    The methods in HTMLRenderer are accessed from the superclass
    Handler's start(), end(), and sub() methods. They implement basic
    markup as used in HTML documents.
    """
    def start_document(self):
        print '<html><head><title>...</title></head><body>'
    def end_document(self):
        print '</body></html>'
    def start_paragraph(self):
        print '<p>'
    def end_paragraph(self):
        print '</p>'
    def start_heading(self):
        print '<h2>'
    def end_heading(self):
        print '</h2>'
    def start_list(self):
        print '<ul>'
    def end_list(self):
        print '</ul>'
    def start_listitem(self):
        print '<li>'
    def end_listitem(self):
        print '</li>'
    def start_title(self):
        print '<h1>'
    def end_title(self):
        print '</h1>'
    def sub_emphasis(self, match):
        return '<em>%s</em>' % match.group(1)
    def sub_url(self, match):
        return '<a href="%s">%s</a>' % (match.group(1), match.group(1))
    def sub_mail(self, match):
        return '<a href="mailto:%s">%s</a>' % (match.group(1), match.group(1))
    def feed(self, data):
        print data
```

Listing 20-5. *The Rules (rules.py)*

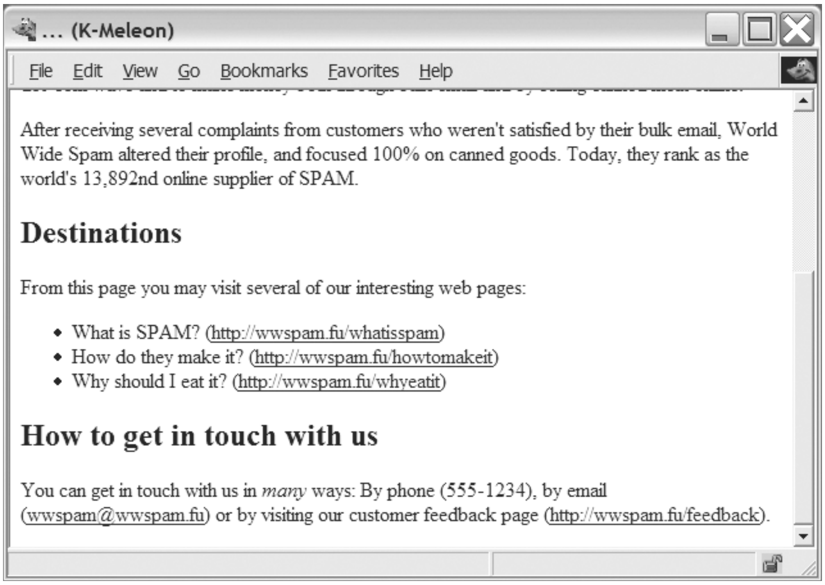
```
class Rule:
    """
    Base class for all rules.
    """
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block)
        handler.end(self.type)
        return True

class HeadingRule(Rule):
    """
    A heading is a single line that is at most 70 characters and
    that doesn't end with a colon.
    """
    type = 'heading'
    def condition(self, block):
        return not '\n' in block and len(block) <= 70 and not block[-1] == ':'

class TitleRule(HeadingRule):
    """
    The title is the first block in the document, provided that it is
    a heading.
    """
    type = 'title'
    first = True

    def condition(self, block):
        if not self.first: return False
        self.first = False
        return HeadingRule.condition(self, block)

class ListItemRule(Rule):
    """
    A list item is a paragraph that begins with a hyphen. As part of
    the formatting, the hyphen is removed.
    """
    type = 'listitem'
    def condition(self, block):
        return block[0] == '-'
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block[1:].strip())
        handler.end(self.type)
        return True

class ListRule(ListItemRule):
    """
    A list begins between a block that is not a list item and a
    subsequent list item. It ends after the last consecutive list
    item.
    """
    type = 'list'
    inside = False
    def condition(self, block):
        return True
    def action(self, block, handler):
        if not self.inside and ListItemRule.condition(self, block):
            handler.start(self.type)
            self.inside = True
        elif self.inside and not ListItemRule.condition(self, block):
            handler.end(self.type)
            self.inside = False
        return False

class ParagraphRule(Rule):
    """
    A paragraph is simply a block that isn't covered by any of the
    other rules.
    """
    type = 'paragraph'
    def condition(self, block):
        return True
```

Listing 20-6. *The Main Program (markup.py)*

```
import sys, re
from handlers import *
from util import *
from rules import *

class Parser:
    """
    A Parser reads a text file, applying rules and controlling a
    handler.
    """
    def __init__(self, handler):
        self.handler = handler
        self.rules = []
        self.filters = []
    def addRule(self, rule):
        self.rules.append(rule)
    def addFilter(self, pattern, name):
        def filter(block, handler):
            return re.sub(pattern, handler.sub(name), block)
        self.filters.append(filter)
    def parse(self, file):
        self.handler.start('document')
        for block in blocks(file):
            for filter in self.filters:
                block = filter(block, self.handler)
            for rule in self.rules:
                if rule.condition(block):
                    last = rule.action(block, self.handler)
                    if last: break
        self.handler.end('document')

class BasicTextParser(Parser):
    """
    A specific Parser that adds rules and filters in its
    constructor.
    """
    def __init__(self, handler):
        Parser.__init__(self, handler)
        self.addRule(ListRule())
        self.addRule(ListItemRule())
        self.addRule(TitleRule())
        self.addRule(HeadingRule())
        self.addRule(ParagraphRule())

        self.addFilter(r'\*(.+?)\*', 'emphasis')
        self.addFilter(r'(http://[\.a-zA-Z/]+)', 'url')
        self.addFilter(r'([\.a-zA-Z]+@[\.a-zA-Z]+[a-zA-Z]+)', 'mail')

handler = HTMLRenderer()
parser = BasicTextParser(handler)

parser.parse(sys.stdin)
```

You can see the result of running the program on the sample text in Figure 20-2.

The second implementation is clearly more complicated and extensive than the first version. The added complexity is well worth the effort because the resulting program is much more flexible and extensible. Adapting it to new input and output formats is merely a matter of subclassing and initializing the existing classes, rather than rewriting everything from scratch, as you would have had to do in the first prototype.



After receiving several complaints from customers who weren't satisfied by their bulk email, World Wide Spam altered their profile, and focused 100% on canned goods. Today, they rank as the world's 13,892nd online supplier of SPAM.

Destinations

From this page you may visit several of our interesting web pages:

- What is SPAM? (http://wwspam.fu/whatisspam)
- How do they make it? (http://wwspam.fu/howtomakeit)
- Why should I eat it? (http://wwspam.fu/whyeatit)

How to get in touch with us

You can get in touch with us in *many* ways: By phone (555-1234), by email (wwspam@wwspam.fu) or by visiting our customer feedback page (http://wwspam.fu/feedback).

Figure 20-2. *The second attempt at generating a web page*

Further Exploration

Several expansions are possible for this program. Here are some possibilities:

- Add support for tables. Find all aligning left word borders and split the block into columns.
- Add support for interpreting all uppercase words as emphasis. (To do this properly, you will need to take into account acronyms, punctuations, names, and other capitalized words.)
- Add support for $\LaTeX$ output.
- Write a handler that does something other than markup. Perhaps write a handler that analyzes the document in some way.
- Create a script that automatically converts all text files in a directory to HTML files.
- Check out some existing plain-text formats (such as various forms of wiki markup). See Table 20-1 for some ideas. A web search (or a look at some wiki or blog systems) will probably turn up more results.

Table 20-1. *Some Plain-Text and Wiki-Style Markup Systems*

Markup System	Web Site
Atox	`http://atox.sf.net`
atx	`http://www.aaronsw.com/2002/atx`
BBCode	`http://www.bbcode.org`
Epytext	`http://epydoc.sourceforge.net/epytext.html`
EtText	`http://ettext.taint.org`
Grutatxt	`http://www.triptico.com/software/grutatxt.html`
Markdown	`http://daringfireball.net/projects/markdown`
reStructuredText	`http://docutils.sourceforge.net/rst.html`
Setext	`http://www.valdemar.net/~erik/setext`
SmartASCII	`http://www.gnosis.cx/TPiP`
Textile	`http://www.textism.com/tools/textile`
txt2html	`http://txt2html.sourceforge.net`
WikiCreole	`http://www.wikicreole.org`
WikiMarkupStandard	`http://www.usemod.com/cgi-bin/mb.pl?WikiMarkupStandard`
Wikitext	`http://en.wikipedia.org/wiki/Wikitext`
YAML	`http://www.yaml.org`

What Now?

Phew! After this strenuous (but useful, I hope) project, it's time for some lighter material. In the next chapter, you create some graphics based on data that is automatically downloaded from the Internet. Piece of cake.