Chapter 4

PARSERS AND STATE MACHINES

All the techniques presented in the prior chapters of this book have something in common, but something that is easy to overlook. In a sense, every basic string and regular expression operation treats strings as *homogeneous*. Put another way: String and regex techniques operate on *flat* texts. While said techniques are largely in keeping with the "Zen of Python" maxim that "Flat is better than nested," sometimes the maxim (and homogeneous operations) cannot solve a problem. Sometimes the data in a text has a deeper *structure* than the linear sequence of bytes that make up strings.

It is not entirely true that the prior chapters have eschewed data structures. From time to time, the examples presented broke flat texts into lists of lines, or of fields, or of segments matched by patterns. But the structures used have been quite simple and quite regular. Perhaps a text was treated as a list of substrings, with each substring manipulated in some manner—or maybe even a list of lists of such substrings, or a list of tuples of data fields. But overall, the data structures have had limited (and mostly fixed) nesting depth and have consisted of sequences of items that are themselves treated similarly. What this chapter introduces is the notion of thinking about texts as *trees* of nodes, or even still more generally as graphs.

Before jumping too far into the world of nonflat texts, I should repeat a warning this book has issued from time to time. If you do not *need* to use the techniques in this chapter, you are better off sticking with the simpler and more maintainable techniques discussed in the prior chapters. Solving too general a problem too soon is a pitfall for application development—it is almost always better to do less than to do more. Full-scale parsers and state machines fall to the "more" side of such a choice. As we have seen already, the class of problems you can solve using regular expressions—or even only string operations—is quite broad.

There is another warning that can be mentioned at this point. This book does not attempt to explain parsing theory or the design of parseable languages. There are a lot of intricacies to these matters, about which a reader can consult a specialized text like the so-called "Dragon Book"—Aho, Sethi, and Ullman's *Compilers: Principle, Techniques and Tools* (Addison-Wesley, 1986; ISBN: 0201100886)—or Levine, Mason, and

Brown's *Lex & Yacc* (Second Edition, O'Reilly, 1992; ISBN: 1-56592-000-7). When Extended Backus-Naur Form (EBNF) grammars or other parsing descriptions are discussed below, it is in a general fashion that does not delve into algorithmic resolution of ambiguities or big-O efficiencies (at least not in much detail). In practice, everyday Python programmers who are processing texts—but who are not designing new programming languages—need not worry about those parsing subtleties omitted from this book.

4.1 An Introduction to Parsers

4.1.1 When Data Becomes Deep and Texts Become Stateful

Regular expressions can match quite complicated patterns, but they fall short when it comes to matching arbitrarily nested subpatterns. Such nested subpatterns occur quite often in programming languages and textual markup languages (and other places sometimes). For example, in HTML documents, you can find lists or tables nested inside each other. For that matter, character-level markup is also allowed to nest arbitrarily—the following defines a valid HTML fragment:

```
>>> s = '''<p>Plain text, <i>italicized phrase,
           <i>italicized subphrase</i>, <b>bold
           subphrase</b></i>, <i>other italic
           phrase</i></p>'''
```

The problem with this fragment is that most any regular expression will match either less or more than a desired `<i>` element body. For example:

```
>>> ital = r'''(?sx)<i>.+</i>'''
>>> for phrs in re.findall(ital, s):
...     print phrs, '\n-----'
...
<i>italicized phrase,
       <i>italicized subphrase</i>, <b>bold
       subphrase</b></i>, <i>other italic
       phrase</i>
-----
>>> ital2 = r'''(?sx)<i>.+?</i>'''
>>> for phrs in re.findall(ital2, s):
...     print phrs, '\n-----'
...
<i>italicized phrase,
       <i>italicized subphrase</i>
-----
<i>other italic
       phrase</i>
-----
```

What is missing in the proposed regular expressions is a concept of *state*. If you imagine reading through a string character-by-character (which a regular expression match must do within the underlying regex engine), it would be useful to keep track of "How many layers of italics tags am I in?" With such a count of nesting depth, it would be possible to figure out which opening tag `<i>` a given closing tag `</i>` was meant to match. But regular expressions are not stateful in the right way to do this.

You encounter a similar nesting in most programming languages. For example, suppose we have a hypothetical (somewhat BASIC-like) language with an IF/THEN/END structure. To simplify, suppose that every condition is spelled to match the regex `cond\d+`, and every action matches `act\d+`. But the wrinkle is that IF/THEN/END structures can nest within each other also. So for example, let us define the following three top-level structures:

```
>>> s = '''
IF cond1 THEN act1 END
-----
IF cond2 THEN
  IF cond3 THEN act3 END
END
-----
IF cond4 THEN
  act4
END
'''
```

As with the markup example, you might first try to identify the three structures using a regular expression like:

```
>>> pat = r'''(?sx)
IF \s+
cond\d+ \s+
THEN \s+
act\d+ \s+
END'''
>>> for stmt in re.findall(pat, s):
...     print stmt, '\n-----'
...
IF cond1 THEN act1 END
-----
IF cond3 THEN act3 END
-----
IF cond4 THEN
  act4
END
-----
```

This indeed finds three structures, but the wrong three. The second top-level structure should be the compound statement that used `cond2`, not its child using `cond3`. It is not too difficult to allow a nested IF/THEN/END structure to optionally substitute for a simple action; for example:

```
>>> pat2 = '''(?sx)(
IF \s+
cond\d+ \s+
THEN \s+
(  (IF \s+ cond\d+ \s+ THEN \s+ act\d+ \s+ END)
 | (act\d+)
) \s+
END
)'''
>>> for stmt in re.findall(pat2, s):
...     print stmt[0], '\n-----'
...
IF cond1 THEN act1 END
-----
IF cond2 THEN
  IF cond3 THEN act3 END
END
-----
IF cond4 THEN
  act4
END
-----
```

By manually nesting a "first order" IF/THEN/END structure as an alternative to a simple action, we can indeed match the example in the desired fashion. But we have assumed that nesting of IF/THEN/END structures goes only one level deep. What if a "second order" structure is nested inside a "third order" structure—and so on, ad infinitum? What we would like is a means of describing arbitrarily nested structures in a text, in a manner similar to, but more general than, what regular expressions can describe.

4.1.2 What Is a Grammar?

In order to parse nested structures in a text, you usually use something called a "grammar." A grammar is a specification of a set of "nodes" (also called "productions") arranged into a strictly hierarchical "tree" data structure. A node can have a name—and perhaps some other properties—and it can also have an ordered collection of child nodes. When a document is parsed under a grammar, no resultant node can ever be a descendent of itself; this is another way of saying that a grammar produces a tree rather than a graph.

In many actual implementations, such as the famous C-based tools `lex` and `yacc`, a grammar is expressed at two layers. At the first layer, a "lexer" (or "tokenizer") produces a stream of "tokens" for a "parser" to operate on. Such tokens are frequently what you might think of as words or fields, but in principle they can split the text differently than does our normal idea of a "word." In any case tokens are nonoverlapping subsequences of the original text. Depending on the specific tool and specification used, some subsequences may be dropped from the token stream. A "zero-case" lexer is one that simply treats the actual input bytes as the tokens a parser operates on (some modules discussed do this, without losing generality).

The second layer of a grammar is the actual parser. A parser reads a stream or sequence of tokens and generates a "parse tree" out of it. Or rather, a tree is generated under the assumption that the underlying input text is "well-formed" according to the grammar—that is, there is a way to consume the tokens within the grammar specification. With most parser tools, a grammar is specified using a variant on EBNF.

An EBNF grammar consists of a set of rule declarations, where each rule allows similar quantification and alternation as that in regular expressions. Different tools use slightly different syntax for specifying grammars, and different tools also differ in expressivity and available quantifiers. But almost all tools have a fairly similar feel in their grammar specifications. Even the DTDs used in XML dialect specifications (see Chapter 5) have a very similar syntax to other grammar languages—which makes sense since an XML dialect is a particular grammar. A DTD entry looks like:

```
<!ELEMENT body  ((example-column | image-column)?, text-column) >
```

In brief, under the sample DTD, a `<body>` element may contain either one or zero occurrences of a "first thing"—that first thing being *either* an `<example-column>` or an `<image-column>`. Following the optional first component, exactly one `<text-column>` must occur. Of course, we would need to see the rest of the DTD to see what can go in a `<text-column>`, or to see what other element(s) a `<body>` might be contained in. But each such rule is similar in form.

A familiar EBNF grammar to Python programmers is the grammar for Python itself. On many Python installations, this grammar as a single file can be found at a disk location like `[...]/Python22/Doc/ref/grammar.txt`. The online and downloadable *Python Language Reference* excerpts from the grammar at various points. As an example, a floating point number in Python is identified by the specification:

EBNF-style description of Python floating point

```
floatnumber   ::= pointfloat | exponentfloat
pointfloat    ::= [intpart] fraction | intpart "."
exponentfloat ::= (intpart | pointfloat) exponent
intpart       ::= digit+
fraction      ::= "." digit+
exponent      ::= ("e" | "E") ["+" | "-"] digit+
digit         ::= "0"..."9"
```

The Python grammar is given in an EBNF variant that allows considerable expressivity. Most of the tools this chapter discusses are comparatively limited (but are still ultimately capable of expressing just as general grammars, albeit more verbosely). Both literal strings and character ranges may be specified as part of a production. Alternation is expressed with "|". Quantifications with both "+" and "*" are used. These features are very similar to those in regular expression syntax. Additionally, optional groups are indicated with square brackets ("[" and "]"), and mandatory groups with parentheses. Conceptually the former is the same as the regex "?" quantifier.

Where an EBNF grammar goes beyond a regular expression pattern is in its use of named terms as parts of patterns. At first glance, it might appear possible simply to substitute regular expression patterns for named subexpressions. In fact, in the floating point pattern presented, we could simply do this as:

Regular expression to identify a floating point

```
pat = r'''(?x)
      (                    # exponentfloat
        (                  # intpart or pointfloat
          (                # pointfloat
            (\d+)?[.]\d+   # optional intpart with fraction
            |
            \d+[.]         # intpart with period
          )                # end pointfloat
          |
          \d+              # intpart
        )                  # end intpart or pointfloat
        [eE][+-]?\d+       # exponent
      )                    # end exponentfloat
      |
      (                    # pointfloat
        (\d+)?[.]\d+       # optional intpart with fraction
        |
        \d+[.]             # intpart with period
      )                    # end pointfloat
      '''
```

As a regular expression, the description is harder to read, even with the documentation added to a verbose regex. The EBNF grammar is more or less self-documenting. Moreover, some care had to be taken about the order of the regular expression—the `exponentfloat` alternative is required to be listed before the `pointfloat` alternative since the latter can form a subsequence of the latter. But aside from the need for a little tweaking and documentation, the regular expression above is exactly as general—and exactly equivalent—to the Python grammar for a floating point number.

You might wonder, therefore, what the point of a grammar is. It turns out that a floating point number is an unusually simple structure in one very specific respect. A `floatnumber` requires no recursion or self-reference in its definition. Everything that makes up a `floatnumber` is something simpler, and everything that makes up one of those simpler components is itself made up of still simpler ones. You reach a bottom in defining a Python floating point number.

In the general case, structures can recursively contain themselves, either directly or by containing other structures that in turn contain the first structures. It is not even entirely absurd to imagine floating point numbers with such a grammar (whatever language had them would not be Python, however). For example, the famous number a "googol" was defined in 1938 by Edward Kasner as 10 to the 100th power (otherwise called "10 dotrigintillion"). As a Python floating point, you could write this as `1e100`. Kasner also defined a "googolplex" as 10 to the googol power (a number much larger than anyone needs for any practical reason). While you can create a Python expression to name a googolplex—for example, `10**1e100`—it is not difficult to conceive a programming language that allowed the term `1e1e100` as a name for a googolplex. By the way: If you try to actually *compute* a googolplex in Python (or any other programming language), you will be in for disappointment; expect a frozen computer and/or some sort of crash or overflow. The numbers you can express in most language grammars are quite a bit more numerous than those your computer can actually do anything with.

Suppose that you wanted to allow these new "extended" floating point terms in a language. In terms of the grammar, you could just change a line of the EBNF description:

```
exponent ::= ("e" | "E") ["+" | "-"] floatnumber
```

In the regular expression, the change is a problem. A portion of the regular expression identifies the (optional) exponent:

```
[eE][+-]?\d+       # exponent
```

In this case, an exponent is just a series of digit characters. But for "extended" floating point terms, the regular expression would need to substitute the entire `pat` regular expression in place of `\d+`. Unfortunately, this is impossible, since each replacement would still contain the insufficient `\d+` description, which would again require substitution. The sequence of substitutions continues ad infinitum, until the regular expression is infinitely long.

4.1.3 An EBNF Grammar for IF/THEN/END Structures

The IF/THEN/END language structure presented above is a more typical and realistic example of nestable grammatical structures than are our "extended" floating point numbers. In fact, Python—along with almost every other programming language—allows precisely such `if` statements inside other `if` statements. It is worthwhile to look at how we might describe our hypothetical simplified IF/THEN/END structure in the same EBNF variant used for Python's grammar.

Recall first our simplified rules for allowable structures: The keywords are `IF`, `THEN`, and `END`, and they always occur in that order within a completed structure. Keywords in this language are always in all capitals. Any whitespace in a source text is insignificant, except that each term is separated from others by at least some whitespace.

Every condition is spelled to match the regular expression `cond\d+`. Every IF "body" either contains an action that matches the regular expression `act\d+`, *or* it contains another IF/THEN/END structure. In our example, we created three IF/THEN/END structures, one of which contained a nested structure:

```
IF cond1 THEN act1 END
-----
IF cond2 THEN
  IF cond3 THEN act3 END
END
-----
IF cond4 THEN
  act4
END
```

Let us try a grammar:

EBNF grammar for IF/THEN/END structures

```
if_expr   ::= "IF" ws cond ws "THEN" ws action ws "END"
whitechar ::= " " | "\t" | "\n" | "\r" | "\f" | "\v"
ws        ::= whitechar+
digit     ::= "0"..."9"
number    ::= digit+
cond      ::= "cond" number
action    ::= simpleact | if_expr
simpleact ::= "act" number
```

This grammar is fairly easy to follow. It defines a few "convenience" productions like `ws` and `number` that consist of repetitions of simpler productions. `whitechar` is defined as an explicit alternation of individual characters, as is `digit` for a continuous range. Taken to the extreme, every production could actually be included in a much more verbose `if_expr` production—you would just substitute all the right-hand sides of nested productions for the names in the `if_expr` production. But as given, the grammar is much easier to read. The most notable aspect of this grammar is the `action` production, since an `action` can itself recursively contain an `if_expr`.

For this problem, the reader is encouraged to develop grammars for some more robust variations on the very simple IF/THEN/END language we have looked at. As is evident, it is difficult to actually do much with this language by itself, even if its actions and conditions are given semantic meaning outside the structure. Readers can invent their own variations, but a few are proposed below.

4.1.4 Pencil-and-Paper Parsing

To test a grammar at this point, just try to expand each successive character into some production that is allowed at that point in the parent production, using pencil and paper. Think of the text of test cases as a tape: Each symbol either completes a production (if so, write the satisfied production down next to the subsequence), or the symbol is added to the "unsatisfied register." There is one more rule to follow with pencil and paper, however: It is better to satisfy a production with a longer subsequence than a shorter one. If a parent production consists of child productions, the children must be satisfied in the specified order (and in the quantity required). For now, assume only one character of lookahead in trying to follow this rule. For example, suppose you find the following sequence in a test case:

```
"IF   cond1..."
```

Your steps with the pencil would be something like this:

1. Read the "I"—no production is satisfied.
2. Read the "F", unsatisfied becomes "I"-"F". Note that "I"-"F" matches the literal term in `if_expr` (a literal is considered a production). Since the literal term contains no quantifiers or alternates, write down the "IF" production. Unsatisfied becomes empty.
3. Read the space, Unsatisfied becomes simply a space. Space satisfies the production `ws`, but hold off for a character since `ws` contains a quantifier that allows a longer substring to satisfy it.
4. Read the second space, unsatisfied becomes space-space. Space-space satisfies the production `ws`. But again hold off for a character.
5. Read the third space, unsatisfied becomes space-space-space. This again satisfies the production `ws`. But keep holding off for the next character.
6. Read the "c", unsatisfied becomes "space-space-space-c". This does not satisfy any production, so revert to the production in 5. Unsatisfied becomes "c".
7. Et cetera.

If you get to the last character, and everything fits into some production, the test case is valid under the grammar. Otherwise, the test case is nongrammatical. Try a few IF/THEN/END structures that you think are and are not valid against the provided grammar.

4.1.5 Exercise: Some variations on the language

1. Create and test an IF/THEN/END grammar that allows multiple actions to occur between the THEN and the END. For example, the following structures are valid under this variation:

   ```
   IF cond1 THEN act1 act2 act3 END
   -----
   ```

```
IF cond2 THEN
  IF cond3 THEN act3 END
  IF cond4 THEN act4 END
END
-----
IF cond5 THEN IF cond6 THEN act6 act7 END act8 END
```

2. Create and test an IF/THEN/END grammar that allows for arithmetic comparisons of numbers as conditions (as an enhancement of variation 1, if you wish). Specifically, a comparison consists of two numbers with one of “<”, “>”, or “=” between them. There might or might not be any whitespace between a comparison symbol and surrounding numbers. Use your judgment about what a number consists of (the Python floating point grammar might provide an example, but yours could be simpler).

3. Create and test an IF/THEN/END grammar that includes a loop expression as a valid action. A loop consists of the keyword `LOOP`, followed by a positive integer, followed by action(s), and terminated by the `END` keyword. Loops should be considered actions, and therefore ifs and loops can be contained inside one another; for example:

```
IF cond1 THEN
  LOOP 100
    IF cond2 THEN
      act2
    END
  END
END
```

   You can make this LOOP-enhanced grammar an enhancement of whichever variant you wish.

4. Create and test an IF/THEN/END grammar that includes an optional `ELSE` keyword. If an `ELSE` occurs, it is within an `IF` body, but `ELSE` might not occur. An `ELSE` has its own body that can contain action(s). For example (assuming variant 1):

```
IF cond1 THEN
  act1
  act2
ELSE
  act3
  act4
END
```

5. Create and test an IF/THEN/END grammar that may include *zero* actions inside an IF, ELSE, or LOOP body. For example, the following structures are valid under this variant:

```
IF cond1 THEN
ELSE act2
END
-*-
IF cond1 THEN
  LOOP 100 END
ELSE
END
```

4.2 An Introduction to State Machines

State machines, in a theoretical sense, underlay almost everything computer- and programming-related. But a Python programmer does not necessarily need to consider highly theoretical matters in writing programs. Nonetheless, there is a large class of ordinary programming problems where the best and most natural approach is to explicitly code a state machine as the solution. At heart, a state machine is just a way of thinking about the flow control in an application.

A parser is a specialized type of state machine that analyzes the components and meaning of structured texts. Generally a parser is accompanied by its own high-level description language that describes the states and transitions used by the implied state machine. The state machine is in turn applied to text obeying a "grammar."

In some text processing problems, the processing must be *stateful*: How we handle the next bit of text depends upon what we have done so far with the prior text. In some cases, statefulness can be naturally expressed using a parser grammar, but in other cases the state has more to do with the semantics of the prior text than with its syntax. That is, the issue of what grammatical properties a portion of a text has is generally orthogonal to the issue of what predicates it fulfills. Concretely, we might calculate some arithmetic result on numeric fields, or we might look up a name encountered in a text file in a database, before deciding how to proceed with the text processing. Where the parsing of a text depends on semantic features, a state machine is often a useful approach.

Implementing an elementary and generic state machine in Python is simple to do, and may be used for a variety of purposes. The third-party C-extension module *mx.TextTools*, which is discussed later in this chapter, can also be used to create far faster state machine text processors.

4.2.1 Understanding State Machines

A much too accurate description of a state machine is that it is a directed graph, consisting of a set of nodes and a set of transition functions. Such a machine "runs" by responding to a series of events; each event is in the domain of the transition function of the "current" node, where the range is a subset of the nodes. The function return is a "next" (maybe self-identical) node. A subset of the nodes are end-states; if an end-state is reached, the machine stops.

An abstract mathematical description—like the one above—is of little use for most practical programming problems. Equally picayune is the observation that every program in an imperative programming language like Python is a state machine whose nodes are its source lines (but not really in a declarative—functional or constraint-based—language such as Haskell, Scheme, or Prolog). Furthermore, every regular expression is logically equivalent to a state machine, and every parser implements an abstract state machine. Most programmers write lots of state machines without really thinking about it, but that fact provides little guidance to specific programming techniques.

An informal, heuristic definition is more useful than an abstract one. Often we encounter a program requirement that includes a handful of distinct ways of treating clusters of events. Furthermore, it is sometimes the case that individual events need to be put in a context to determine which type of treatment is appropriate (as opposed to each event being "self-identifying"). The state machines discussed in this introduction are high-level machines that are intended to express clearly the programming requirements of a class of problems. If it makes sense to talk about your programming problem in terms of categories of behavior in response to events, it is likely to be a good idea to program the solution in terms of explicit state machines.

4.2.2 Text Processing State Machines

One of the programming problems most likely to call for an explicit state machine is processing text files. Processing a text file very often consists of sequential reading of each chunk of a text file (typically either a character or a line), and doing something in response to each chunk read. In some cases, this processing is "stateless"—that is, each chunk has enough information internally to determine exactly what to do in response to that chunk of text. And in other cases, even though the text file is not 100 percent stateless, there is a very limited context to each chunk (for example, the line number might matter for the action taken, but not much else besides the line number). But in other common text processing problems, the text files we deal with are highly "stateful"—the meaning of a chunk depends on what types of chunks preceded it (and maybe even on what chunks come next). Files like report files, mainframe data-feeds, human-readable texts, programming source files, and other sorts of text files are stateful. A very simple example of a stateful chunk is a line that might occur in a Python source file:*

```
myObject = SomeClass(this, that, other)
```

That line means something very different if it happens to be surrounded by these lines:

```
"""How to use SomeClass:
myObject = SomeClass(this, that, other)
"""
```

That is, we needed to know that we were in a "blockquote" *state* to determine that the line was a comment rather than an action. Of course, a program that deals with Python programs in a more general way will usually use a parser and grammar.

4.2.3 When Not to Use a State Machine

When we begin the task of writing a processor for any stateful text file, the first question we should ask ourselves is "What types of things do we expect to find in the file?" Each type of thing is a candidate for a state. These types should be several in number, but if the number is huge or indefinite, a state machine is probably not the right approach—maybe some sort of database solution is appropriate. Or maybe the problem has not been formulated right if there appear to be that many types of things.

Moreover, we are not quite ready for a state machine yet; there may yet be a simpler approach. It might turn out that even though our text file is stateful there is an easy way to read in chunks where each chunk is a single type of thing. A state machine is really only worth implementing if the transitions between types of text require some calculation based on the content within a single state-block.

An example of a somewhat stateful text file that is nonetheless probably not best handled with a state machine is a Windows-style `.ini` file (generally replaced nowadays by use of the binary-data-with-API Windows registry). Those files consist of some section headers, some comments, and a number of value assignments. For example:

File: hypothetical.ini

```
; set the colorscheme and userlevel
[colorscheme]
background=red
foreground=blue
title=green

[userlevel]
login=2
; admin=0
title=1
```

This example has no real-life meaning, but it was constructed to indicate some features of the `.ini` format. (1) In one sense, the type of each line is determined by its first character (either semicolon, left brace, or alphabetic). (2) In another sense, the format is "stateful" insofar as the keyword "title" presumably means something independent when it occurs in each section. You could program a text processor that had a COLORSCHEME state and a USERLEVEL state, and processed the value assignments of each state. But that does not seem like the *right* way to handle this problem.

On the one hand, we could simply create the natural chunks in this text file with some Python code like:

Chunking Python code to process .ini file

```
txt = open('hypothetical.ini').read()
from string import strip, split
nocomm = lambda s: s[0] != ';'        # "no comment" util
eq2pair = lambda s: split(s,'=')      # assignmet -> pair
def assignments(sect):
    name, body = split(sect,']')      # identify name, body
    assigns = split(body,'\n')        # find assign lines
    assigns = filter(strip, assigns)  # remove outside space
    assigns = filter(None, assigns)   # remove empty lines
    assigns = filter(nocomm, assigns) # remove comment lines
    assigns = map(eq2pair, assigns)   # make name/val pairs
    assigns = map(tuple, assigns)     # prefer tuple pairs
    return (name, assigns)
sects = split(txt,'[')                # divide named sects
sects = map(strip, sects)             # remove outside newlines
sects = filter(nocomm, sects)         # remove comment sects
config = map(assignments, sects)      # find assigns by sect
pprint.pprint(config)
```

Applied to the hypothetical.ini file above, this code produces output similar to:

```
[('colorscheme',
  [('background', 'red'),
   ('foreground', 'blue'),
   ('title', 'green')]),
 ('userlevel',
  [('login', '2'),
   ('title', '1')])]
```

This particular list-oriented data structure may or may not be what you want, but it is simple enough to transform this into dictionary entries, instance attributes, or whatever is desired. Or slightly modified code could generate other data representations in the first place.

An alternative approach is to use a single current_section variable to keep track of relevant state and process lines accordingly:

```
for line in open('hypothetical.ini').readlines():
    if line[0] == '[':
        current_section = line[1:-2]
    elif line[0] == ';':
        pass    # ignore comments
    else:
        apply_value(current_section, line)
```

Sidebar: A digression on functional programming

Readers will have noticed that the `.ini` chunking code given in the example above has more of a functional programming (FP) style to it than does most Python code (in this book or elsewhere). I wrote the presented code this way for two reasons. The more superficial reason is just to emphasize the contrast with a state machine approach. Much of the special quality of FP lies in its eschewal of state (see the discussion of functional programming in Chapter 1); so the example is, in a sense, even farther from a state machine technique than would be a coding style that used a few nested loops in place of the `map()` and `filter()` calls.

The more substantial reason I adopted a functional programming style is because I feel that this type of problem is precisely the sort that can often be expressed more compactly and more *clearly* using FP constructs. Basically, our source text document expresses a data structure that is homogeneous at each level. Each section is similar to other sections; and within a section, each assignment is similar to others. A clear—and stateless—way to manipulate these sorts of implicit structures is applying an operation uniformly to each thing at a given level. In the example, we do a given set of operations to find the assignments contained within a section, so we might as well just `map()` that set of operations to the collection of (massaged, noncomment) sections. This approach is more terse than a bunch of nested `for` loops, while simultaneously (in my opinion) better expressing the underlying intention of the textual analysis.

Use of a functional programming style, however, can easily be taken too far. Deeply nested calls to `map()`, `reduce()`, and `filter()` can quickly become difficult to read, especially if whitespace and function/variable names are not chosen carefully. Inasmuch as it is possible to write "obfuscated Python" code (a popular competition for other languages), it is almost always done using FP constructs. Warnings in mind, it is possible to create an even terser and more functional variant of the `.ini` chunking code (that produces identical results). I believe that the following falls considerably short of obfuscated, but will still be somewhat more difficult to read for most programmers. On the plus side, it is half the length of the prior code and is entirely free of accidental side effects:

Strongly functional code to process .ini file

```
from string import strip, split
eq2tup = lambda s: tuple(split(s,'='))
splitnames = lambda s: split(s,']')
parts = lambda s, delim: map(strip, split(s, delim))
useful = lambda ss: filter(lambda s: s and s[0]!=';', ss)
config = map(lambda _:(_[0], map(eq2tup, useful(parts(_[1],'\n')))),
             map(splitnames, useful(parts(txt,'['))) )
pprint.pprint(config)
```

In brief, this functional code says that a configuration consists of a list of pairs of (1) names plus (2) a list of key/value pairs. Using list comprehensions might make this expression clearer, but the example code is compatible back to Python 1.5. Moreover, the utility function names `useful()` and `parts()` go a long way towards keeping the example readable. Utility functions of this sort are, furthermore, potentially worth saving in a separate module for other use (which, in a sense, makes the relevant .ini chunking code even shorter).

A reader exercise is to consider how the higher-order functions proposed in Chapter 1's section on functional programming could further improve the sort of "stateless" text processing presented in this subsection.

4.2.4 When to Use a State Machine

Now that we have established not to use a state machine if the text file is "too simple," we should look at a case where a state machine is worthwhile. The utility Txt2Html is listed in Appendix D. Txt2Html converts "smart ASCII" files to HTML.

In very brief recap, smart ASCII format is a text format that uses a few spacing conventions to distinguish different types of text blocks, such as headers, regular text, quotations, and code samples. While it is easy for a human reader or writer to visually parse the transitions between these text block types, there is no simple way to chunk a whole text file into its text blocks. Unlike in the .ini file example, text block types can occur in any pattern of alternation. There is no single delimiter that separates blocks in all cases (a blank line *usually* separates blocks, but a blank line within a code sample does not necessarily end the code sample, and blocks need not be separated by blank lines). But we do need to perform somewhat different formatting behavior on each text block type for the correct final XML output. A state machine suggests itself as a natural solution here.

The general behavior of the Txt2Html reader is as follows: (1) Start in a particular state. (2) Read a line of the text file and go to current state context. (3) Decide if conditions have been met to leave the current state and enter another. (4) Failing (3), process the line in a manner appropriate for the current state. This example is about the simplest case you would encounter, but it expresses the pattern described:

A simple state machine input loop in Python

```
global state, blocks, newblock
for line in fpin.readlines():
    if state == "HEADER":         # blank line means new block of ?
        if blankln.match(line):   newblock = 1
        elif textln.match(line):  startText(line)
        elif codeln.match(line):  startCode(line)
        else:
            if newblock: startHead(line)
            else: blocks[-1] += line
    elif state == "TEXT":         # blank line means new block of ?
        if blankln.match(line):   newblock = 1
        elif headln.match(line):  startHead(line)
        elif codeln.match(line):  startCode(line)
        else:
            if newblock: startText(line)
            else: blocks[-1] += line
    elif state == "CODE":          # blank line does not change state
        if blankln.match(line):    blocks[-1] += line
        elif headln.match(line):   startHead(line)
        elif textln.match(line):   startText(line)
        else: blocks[-1] += line
    else:
        raise ValueError, "unexpected input block state: "+state
```

The only real thing to notice is that the variable `state` is declared `global`, and its value is changed in functions like `startText()`. The transition conditions—such as `textln.match()`—are regular expression patterns, but they could just as well be custom functions. The formatting itself is actually done later in the program; the state machine just parses the text file into labeled blocks in the `blocks` list. In a sense, the state machine here is acting as a tokenizer for the later block processor.

4.2.5 An Abstract State Machine Class

It is easy in Python to abstract the form of a state machine. Coding in this manner makes the state machine model of the program stand out more clearly than does the simple conditional block in the previous example (which doesn't right away look all that much different from any other conditional). Furthermore, the class presented—and the associated handlers—does a very good job of isolating in-state behavior. This improves both encapsulation and readability in many cases.

File: statemachine.py

```
class InitializationError(Exception): pass

class StateMachine:
    def __init__(self):
        self.handlers = []
        self.startState = None
        self.endStates = []

    def add_state(self, handler, end_state=0):
        self.handlers.append(handler)
        if end_state:
            self.endStates.append(name)

    def set_start(self, handler):
        self.startState = handler

    def run(self, cargo=None):
        if not self.startState:
            raise InitializationError,\
                  "must call .set_start() before .run()"
        if not self.endStates:
            raise InitializationError, \
                  "at least one state must be an end_state"
        handler = self.startState
        while 1:
            (newState, cargo) = handler(cargo)
            if newState in self.endStates:
                newState(cargo)
                break
            elif newState not in self.handlers:
                raise RuntimeError, "Invalid target %s" % newState
            else:
                handler = newState
```

The `StateMachine` class is really all you need for the form of a state machine. It is a whole lot fewer lines than something similar would require in most languages—mostly because of the ease of passing function objects in Python. You could even save a few lines by removing the target state check and the `self.handlers` list, but the extra formality helps enforce and document programmer intention.

To actually *use* the `StateMachine` class, you need to create some handlers for each state you want to use. A handler must follow a particular pattern. Generally, it should loop indefinitely; but in any case it must have some breakout condition(s). Each pass through the state handler's loop should process another event of the state's type. But probably even before handling events, the handler should check for breakout conditions and determine what state is appropriate to transition to. At the end, a handler should pass back a tuple consisting of the target state's name and any cargo the new state handler will need.

An encapsulation device is the use of `cargo` as a variable in the `StateMachine` class (not necessarily called `cargo` by the handlers). This is used to pass around "whatever is needed" by one state handler to take over where the last state handler left off. Most typically, `cargo` will consist of a file handle, which would allow the next handler to read some more data after the point where the last state handler stopped. But a database connection might get passed, or a complex class instance, or a tuple with several things in it.

4.2.6 Processing a Report with a Concrete State Machine

A moderately complicated report format provides a good example of some processing amenable to a state machine programming style—and specifically, to use of the `StateMachine` class above. The hypothetical report below has a number of state-sensitive features. Sometimes lines belong to buyer orders, but at other times the identical lines could be part of comments or the heading. Blank lines, for example, are processed differently from different states. Buyers, who are each processed according to different rules, each get their own machine state. Moreover, within each order, a degree of stateful processing is performed, dependent on locally accumulated calculations:

Sample Buyer/Order Report

```
MONTHLY REPORT -- April 2002
===================================================================

Rules:
 - Each buyer has price schedule for each item (func of quantity).
 - Each buyer has a discount schedule based on dollar totals.
 - Discounts are per-order (i.e., contiguous block)
 - Buyer listing starts with line containing ">>", then buyer name.
 - Item quantities have name-whitespace-number, one per line.
 - Comment sections begin with line starting with an asterisk,
   and ends with first line that ends with an asterisk.

>> Acme Purchasing

  widgets      100
  whatzits    1000
  doodads     5000
  dingdongs   20

* Note to Donald: The best contact for Acme is Debbie Franlin, at
* 413-555-0001.  Fallback is Sue Fong (call switchboard). *

>> Megamart

doodads   10k
whatzits  5k

>> Fly-by-Night Sellers
   widgets        500
   whatzits      4
   flazs         1000

* Note to Harry: Have Sales contact FbN for negotiations *

*
Known buyers:
>>  Acme
>>  Megamart
>>  Standard (default discounts)
*

*** LATE ADDITIONS ***

>> Acme Purchasing
widgets      500     (rush shipment)**
```

The code to processes this report below is a bit simplistic. Within each state, almost all the code is devoted merely to deciding when to leave the state and where to go next. In the sample, each of the "buyer states" is sufficiently similar that they could well be generalized to one parameterized state; but in a real-world application, each state is likely to contain much more detailed custom programming for both in-state behavior and out-from-state transition conditions. For example, a report might allow different formatting and fields within different buyer blocks.

buyer_invoices.py

```
from statemachine import StateMachine
from buyers import STANDARD, ACME, MEGAMART
from pricing import discount_schedules, item_prices
import sys, string

#-- Machine States
def error(cargo):
    # Don't want to get here! Unidentifiable line
    sys.stderr.write('Unidentifiable line:\n'+ line)

def eof(cargo):
    # Normal termination -- Cleanup code might go here.
    sys.stdout.write('Processing Successful\n')

def read_through(cargo):
    # Skip through headers until buyer records are found
    fp, last = cargo
    while 1:
        line = fp.readline()
        if not line:            return eof, (fp, line)
        elif line[:2] == '>>':  return whichbuyer(line), (fp, line)
        elif line[0] == '*':    return comment, (fp, line)
        else:                   continue

def comment(cargo):
    # Skip comments
    fp, last = cargo
    if len(last) > 2 and string.rstrip(last)[-1:] == '*':
        return read_through, (fp, '')
    while 1:
        # could save or process comments here, if desired
        line = fp.readline()
        lastchar = string.rstrip(line)[-1:]
        if not line:            return eof, (fp, line)
        elif lastchar == '*':   return read_through, (fp, line)
```

```
def STANDARD(cargo, discounts=discount_schedules[STANDARD],
                    prices=item_prices[STANDARD]):
    fp, company = cargo
    invoice = 0
    while 1:
        line = fp.readline()
        nextstate = buyerbranch(line)
        if nextstate == 0: continue         # blank line
        elif nextstate == 1:                # order item
            invoice = invoice + calc_price(line, prices)
        else:                               # create invoice
            pr_invoice(company, 'standard', discount(invoice,discounts))
            return nextstate, (fp, line)

def ACME(cargo, discounts=discount_schedules[ACME],
                prices=item_prices[ACME]):
    fp, company = cargo
    invoice = 0
    while 1:
        line = fp.readline()
        nextstate = buyerbranch(line)
        if nextstate == 0: continue         # blank line
        elif nextstate == 1:                # order item
            invoice = invoice + calc_price(line, prices)
        else:                               # create invoice
            pr_invoice(company, 'negotiated', discount(invoice,discounts))
            return nextstate, (fp, line)

def MEGAMART(cargo, discounts=discount_schedules[MEGAMART],
                    prices=item_prices[MEGAMART]):
    fp, company = cargo
    invoice = 0
    while 1:
        line = fp.readline()
        nextstate = buyerbranch(line)
        if nextstate == 0: continue         # blank line
        elif nextstate == 1:                # order item
            invoice = invoice + calc_price(line, prices)
        else:                               # create invoice
            pr_invoice(company, 'negotiated', discount(invoice,discounts))
            return nextstate, (fp, line)

#-- Support function for buyer/state switch
def whichbuyer(line):
    # What state/buyer does this line identify?
    line = string.upper(string.replace(line, '-', ''))
    find = string.find
    if find(line,'ACME') >= 0:      return ACME
    elif find(line,'MEGAMART')>= 0: return MEGAMART
    else:                           return STANDARD

def buyerbranch(line):
    if not line:                    return eof
    elif not string.strip(line):    return 0
    elif line[0] == '*':            return comment
    elif line[:2] == '>>':          return whichbuyer(line)
    else:                           return 1

#-- General support functions
def calc_price(line, prices):
    product, quant = string.split(line)[:2]
    quant = string.replace(string.upper(quant),'K','000')
    quant = int(quant)
    return quant*prices[product]

def discount(invoice, discounts):
    multiplier = 1.0
    for threshhold, percent in discounts:
        if invoice >= threshhold: multiplier = 1 - float(percent)/100
    return invoice*multiplier

def pr_invoice(company, disctype, amount):
    print "Company name:", company[3:-1], "(%s discounts)" % disctype
    print "Invoice total: $", amount, '\n'

if __name__== "__main__":
    m = StateMachine()
    m.add_state(read_through)
    m.add_state(comment)
    m.add_state(STANDARD)
    m.add_state(ACME)
    m.add_state(MEGAMART)
    m.add_state(error, end_state=1)
    m.add_state(eof, end_state=1)
    m.set_start(read_through)
    m.run((sys.stdin, ''))
```

The body of each state function consists mostly of a `while 1:` loop that sometimes breaks out by returning a new target state, along with a cargo tuple. In our particular machine, `cargo` consists of a file handle and the last line read. In some cases, the line that signals a state transition is also needed for use by the subsequent state. The cargo could contain whatever we wanted. A flow diagram lets you see the set of transitions easily:

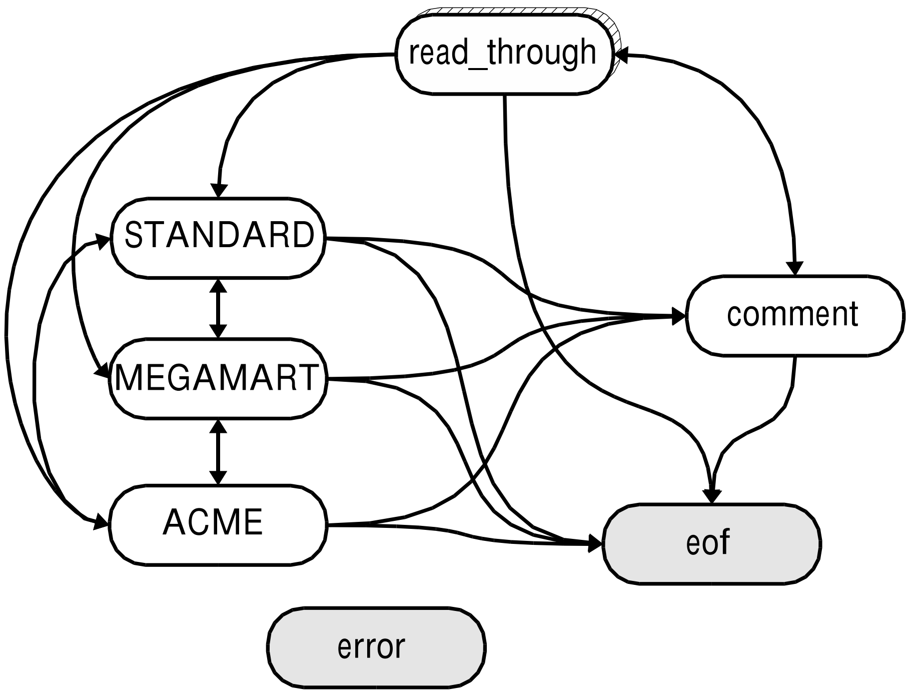

Figure 4.1: Buyer state machine diagram

All of the buyer states are "initialized" using default argument values that are never changed during calls by a normal state machine `.run()` cycle. You could also perhaps design state handlers as classes instead of as functions, but that feels like extra conceptual overhead to me. The specific initializer values are contained in a support module that looks like:

pricing.py support data

```
from buyers import STANDARD, ACME, MEGAMART, BAGOBOLTS

# Discount consists of dollar requirement and a percentage reduction
# Each buyer can have an ascending series of discounts, the highest
# one applicable to a month is used.
discount_schedules = {
    STANDARD  : [(5000,10),(10000,20),(15000,30),(20000,40)],
    ACME      : [(1000,10),(5000,15),(10000,30),(20000,40)],
    MEGAMART  : [(2000,10),(5000,20),(10000,25),(30000,50)],
    BAGOBOLTS : [(2500,10),(5000,15),(10000,25),(30000,50)],
  }
item_prices = {
    STANDARD  : {'widgets':1.0, 'whatzits':0.9, 'doodads':1.1,
                 'dingdongs':1.3, 'flazs':0.7},
    ACME      : {'widgets':0.9, 'whatzits':0.9, 'doodads':1.0,
                 'dingdongs':0.9, 'flazs':0.6},
    MEGAMART  : {'widgets':1.0, 'whatzits':0.8, 'doodads':1.0,
                 'dingdongs':1.2, 'flazs':0.7},
    BAGOBOLTS : {'widgets':0.8, 'whatzits':0.9, 'doodads':1.1,
                 'dingdongs':1.3, 'flazs':0.5},
  }
```

In place of reading in such a data structure, a full application might calculate some values or read them from a database of some sort. Nonetheless, the division of data, state logic, and abstract flow into separate modules makes for a good design.

4.2.7 Subgraphs and State Reuse

Another benefit of the state machine design approach is that you can use different start and end states without touching the state handlers at all. Obviously, you do not have complete freedom in doing so—if a state branches to another state, the branch target needs to be included in the list of "registered" states. You can, however, add homonymic handlers in place of target processing states. For example:

Creating end states for subgraphs

```
from statemachine import StateMachine
from BigGraph import *

def subgraph_end(cargo): print "Leaving subgraph..."
foo = subgraph_end
bar = subgraph_end

def spam_return(cargo): return spam, None
baz = spam_return
```

```
if __name__=='__main__':
    m = StateMachine()
    m.add_state(foo, end_state=1)
    m.add_state(bar, end_state=1)
    m.add_state(baz)
    map(m.add_state, [spam, eggs, bacon])
    m.set_start(spam)
    m.run(None)
```

In a complex state machine graph, you often encounter relatively isolated subgraphs. That is, a particular collection of states—i.e., nodes—might have many connections between them, but only a few connections out to the rest of the graph. Usually this occurs because a subgraph concerns a related set of functionality.

For processing the buyer report discussed earlier, only seven states were involved, so no meaningful subgraphs really exist. But in the subgraph example above, you can imagine that the *BigGraph* module contains hundreds or thousands of state handlers, whose targets define a very complex complete graph. Supposing that the states `spam`, `eggs`, and `bacon` define a useful subgraph, and all branches out of the subgraph lead to `foo`, `bar`, or `baz`, the code above could be an entire new application.

The example redefined `foo` and `bar` as end states, so processing (at least in that particular `StateMachine` object) ends when they are reached. However, `baz` is redefined to transition back into the spam-eggs-bacon subgraph. A subgraph exit need not represent a termination of the state machine. It is actually the `end_state` flag that controls termination—but if `foo` was not marked as an end state, it would raise a `RuntimeError` when it failed to return a valid state target.

If you create large graphs—especially with the intention of utilizing subgraphs as state machines—it is often useful to create a state diagram. Pencil and paper are perfectly adequate tools for doing this; a variety of flow-chart software also exists to do it on a computer. The goal of a diagram is to allow you to identify clustered subgraphs and most especially to help identify paths in and out of a functional subgraph. A state diagram from our buyer report example is given as illustration. A quick look at Figure 4.1, for example, allows the discovery that the `error` end state is isolated, which might not have been evident in the code itself. This is not a problem, necessarily; a future enhancement to the diagram and handlers might utilize this state, and whatever logic was written into it.

4.2.8 Exercise: Finding other solutions

1. On the face of it, a lot of "machinery" went into processing what is not really that complicated a report above. The goal of the state machine formality was both to be robust and to allow for expansion to larger problems. Putting aside the state machine approach in your mind, how else might you go about processing reports of the presented type (assume that "reasonable" variations occur between reports of the same type).

   Try writing a fresh report processing application that produces the same results as the presented application (or at least something close to it). Test your application against the sample report and against a few variants you create yourself.

What errors did you encounter running your application? Why? Is your application more concise than the presented one? Which modules do you count as part of the presented application? Is your application's code clearer or less clear to follow for another programmer? Which approach would be easier to expand to encompass other report formats? In what respect is your application better/worse than the state machine example?

2. The `error` state is never actually reached in the `buyer_invoices.py` application. What other transition conditions into the `error` state would be reasonable to add to the application? What types of corruption or mistakes in reports do you expect most typically to encounter? Sometimes reports, or other documents, are flawed, but it is still desirable to utilize as much of them as possible. What are good approaches to recover from error conditions? How could you express those approaches in state machine terms, using the presented `StateMachine` class and framework?

4.3 Parser Libraries for Python

4.3.1 Specialized Parsers in the Standard Library

Python comes standard with a number of modules that perform specialized parsing tasks. A variety of custom formats are in sufficiently widespread use that it is convenient to have standard library support for them. Aside from those listed in this chapter, Chapter 5 discusses the *email* and *xml* packages, and the modules *mailbox*, *HTMLParser*, and *urlparse*, each of which performs parsing of sorts. A number of additional modules listed in Chapter 1, which handle and process audio and image formats, in a broad sense could be considered parsing tools. However, these media formats are better considered as byte streams and structures than as token streams of the sort parsers handle (the distinction is fine, though).

The specialized tools discussed under this section are presented only in summary. Consult the *Python Library Reference* for detailed documentation of their various APIs and features. It is worth knowing what is available, but for space reasons, this book does not document usage specifics of these few modules.

ConfigParser

Parse and modify Windows-style configuration files.

```
>>> import ConfigParser
>>> config = ConfigParser.ConfigParser()
>>> config.read(['test.ini','nonesuch.ini'])
>>> config.sections()
['userlevel', 'colorscheme']
>>> config.get('userlevel','login')
```

```
'2'
>>> config.set('userlevel','login',5)
>>> config.write(sys.stdout)
[userlevel]
login = 5
title = 1

[colorscheme]
background = red
foreground = blue
```

difflib
.../Tools/scripts/ndiff.py

The module *difflib*, introduced in Python 2.1, contains a variety of functions and classes to help you determine the difference and similarity of pairs of sequences. The API of *difflib* is flexible enough to work with sequences of all kinds, but the typical usage is in comparing sequences of lines or sequences of characters.

Word similarity is useful for determining likely misspellings and typos and/or edit changes required between strings. The function *difflib.get_close_matches()* is a useful way to perform "fuzzy matching" of a string against patterns. The required similarity is configurable.

```
>>> users = ['j.smith', 't.smith', 'p.smyth', 'a.simpson']
>>> maxhits = 10
>>> login = 'a.smith'
>>> difflib.get_close_matches(login, users, maxhits)
['t.smith', 'j.smith', 'p.smyth']
>>> difflib.get_close_matches(login, users, maxhits, cutoff=.75)
['t.smith', 'j.smith']
>>> difflib.get_close_matches(login, users, maxhits, cutoff=.4)
['t.smith', 'j.smith', 'p.smyth', 'a.simpson']
```

Line matching is similar to the behavior of the Unix `diff` (or `ndiff`) and `patch` utilities. The latter utility is able to take a source and a difference, and produce the second compared line-list (file). The functions *difflib.ndiff()* and *difflib.restore()* implement these capabilities. Much of the time, however, the bundled `ndiff.py` tool performs the comparisons you are interested in (and the "patches" with an `-r#` option).

```
% ./ndiff.py chap4.txt chap4.txt~ | grep '^[+-]'
-: chap4.txt
+: chap4.txt~
+     against patterns.
```

```
-     against patterns.  The required similarity is configurable.
-
-     >>> users = ['j.smith', 't.smith', 'p.smyth', 'a.simpson']
-     >>> maxhits = 10
-     >>> login = 'a.smith'
```

There are a few more capabilities in the *difflib* module, and considerable customization is possible.

formatter

Transform an abstract sequence of formatting events into a sequence of callbacks to "writer" objects. Writer objects, in turn, produce concrete outputs based on these callbacks. Several parent formatter and writer classes are contained in the module.

In a way, *formatter* is an "anti-parser"—that is, while a parser transforms a series of tokens into program events, *formatter* transforms a series of program events into output tokens.

The purpose of the *formatter* module is to structure creation of streams such as word processor file formats. The module *htmllib* utilizes the *formatter* module. The particular API details provide calls related to features like fonts, margins, and so on.

For highly structured output of prose-oriented documents, the *formatter* module is useful, albeit requiring learning a fairly complicated API. At the minimal level, you may use the classes included to create simple tools. For example, the following utility is approximately equivalent to `lynx -dump`:

urldump.py

```
#!/usr/bin/env python
import sys
from urllib import urlopen
from htmllib import HTMLParser
from formatter import AbstractFormatter, DumbWriter
if len(sys.argv) > 1:
    fpin = urlopen(sys.argv[1])
    parser = HTMLParser(AbstractFormatter(DumbWriter()))
    parser.feed(fpin.read())
    print '----------------------------------------------------'
    print fpin.geturl()
    print fpin.info()
else:
    print "No specified URL"
```

SEE ALSO: htmllib *285*; urllib *388*;

htmllib

Parse and process HTML files, using the services of *sgmllib*. In contrast to the *HTMLParser* module, *htmllib* relies on the user constructing a suitable "formatter" object to accept callbacks from HTML events, usually utilizing the *formatter* module. A formatter, in turn, uses a "writer" (also usually based on the *formatter* module). In my opinion, there are enough layers of indirection in the *htmllib* API to make *HTMLParser* preferable for almost all tasks.

SEE ALSO: HTMLParser *384*; formatter *284*; sgmllib *285*;

multifile

The class *`multifile.MultiFile`* allows you to treat a text file composed of multiple delimited parts as if it were several files, each with their own FILE methods: `.read()`, `.readline()`, `.readlines()`, `.seek()`, and `.tell()` methods. In iterator fashion, advancing to the next virtual file is performed with the method *`multifile.MultiFile.next()`*.

SEE ALSO: fileinput *61*; mailbox *372*; email.Parser *363*; string.split() *142*; file *15*;

parser
symbol
token
tokenize

Interface to Python's internal parser and tokenizer. Although parsing Python source code is arguably a text processing task, the complexities of parsing Python are too specialized for this book.

robotparser

Examine a `robots.txt` access control file. This file is used by Web servers to indicate the desired behavior of automatic indexers and Web crawlers—all the popular search engines honor these requests.

sgmllib

A partial parser for SGML. Standard Generalized Markup Language (SGML) is an enormously complex document standard; in its full generality, SGML cannot be considered a *format*, but rather a grammar for describing concrete formats. HTML is one particular SGML dialect, and XML is (almost) a simplified subset of SGML.

Although it might be nice to have a Python library that handled generic SGML, *sgmllib* is not such a thing. Instead, *sgmllib* implements just enough SGML parsing to support HTML parsing with *htmllib*. You might be able to coax parsing an XML library out of *sgmllib*, with some work, but Python's standard XML tools are far more refined for this purpose.

SEE ALSO: htmllib *285*; xml.sax *405*;

shlex

A lexical analyzer class for simple Unix shell-like syntaxes. This capability is primarily useful to implement small command language within Python applications.

tabnanny

This module is generally used as a command-line script rather than imported into other applications. The module/script *tabnanny* checks Python source code files for mixed use of tabs and spaces within the same block. Behind the scenes, the Python source is fully tokenized, but normal usage consists of something like:

```
% /sw/lib/python2.2/tabnanny.py SCRIPTS/
SCRIPTS/cmdline.py 165 '\treturn 1\r\n'
'SCRIPTS/HTMLParser_stack.py': Token Error: ('EOF in
                                 multi-line string', (3, 7))
SCRIPTS/outputters.py 18 '\tself.writer=writer\r\n'
SCRIPTS/txt2bookU.py 148 '\ttry:\n'
```

The tool is single purpose, but that purpose addresses a common pitfall in Python programming.

SEE ALSO: tokenize *285*;

4.3.2 Low-Level State Machine Parsing

mx.TextTools ◇ Fast Text Manipulation Tools

Marc-Andre Lemburg's *mx.TextTools* is a remarkable tool that is a bit difficult to grasp the gestalt of. *mx.TextTools* can be blazingly fast and extremely powerful. But at the same time, as difficult as it might be to "get" the mindset of *mx.TextTools*, it is still more difficult to get an application written with it working just right. Once it is working, an application that utilizes *mx.TextTools* can process a larger class of text structures than can regular expressions, while simultaneously operating much faster. But debugging an *mx.TextTools* "tag table" can make you wish you were merely debugging a cryptic regular expression!

In recent versions, *mx.TextTools* has come in a larger package with eGenix.com's several other "mx Extensions for Python." Most of the other subpackages add highly efficient C implementations of datatypes not found in a base Python system.

mx.TextTools stands somewhere between a state machine and a full-fledged parser. In fact, the module *SimpleParse*, discussed below, is an EBNF parser library that is built on top of *mx.TextTools*. As a state machine, *mx.TextTools* feels like a lower-level tool than the *statemachine* module presented in the prior section. And yet, *mx.TextTools* is simultaneously very close to a high-level parser. This is how Lemburg characterizes it in the documentation accompanying *mx.TextTools*:

> mxTextTools is an extension package for Python that provides several useful functions and types that implement high-performance text manipulation and searching algorithms in addition to a very flexible and extendable state machine, the Tagging Engine, that allows scanning and processing text based on low-level byte-code "programs" written using Python tuples. It gives you access to the speed of C without the need to do any compile and link steps every time you change the parsing description.
>
> Applications include parsing structured text, finding and extracting text (either exact or using translation tables) and recombining strings to form new text.

The Python standard library has a good set of text processing tools. The basic tools are powerful, flexible, and easy to work with. But Python's basic text processing is *not* particularly fast. Mind you, for most problems, Python by itself is as fast as you need. But for a certain class of problems, being able to choose *mx.TextTools* is invaluable.

The unusual structure of *mx.TextTools* applications warrants some discussion of concrete usage. After a few sample applications are presented, a listing of *mx.TextTools* constants, commands, modifiers, and functions is given.

BENCHMARKS

A familiar computer-industry paraphrase of Mark Twain (who repeats Benjamin Disraeli) dictates that there are "Lies, Damn Lies, and Benchmarks." I will not argue with that and certainly do not want readers to put too great an import on the timings suggested. Nonetheless, in exploring *mx.TextTools*, I wanted to get some sense of just how fast it is. So here is a rough idea.

The second example below presents part of a reworked version of the state machine-based `Txt2Html` application reproduced in Appendix D. The most time-consuming aspect of `Txt2Html` is the regular expression replacements performed in the function `Typography()` for smart ASCII inline markup of words and phrases.

In order to get a timeable test case, I concatenated 110 copies of an article I wrote to get a file a bit over 2MB, and about 41k lines and 300k words. My test processes an entire input as one text block, first using an *mx.TextTools* version of `Typography()`, then using the *re* version.

Processing time of the same test file went from about 34 seconds to about 12 seconds on one slowish Linux test machine (running Python 1.5.2). In other words, *mx.TextTools* gave me about a 3x speedup over what I get with the *re* module. This speedup is probably typical, but particular applications might gain significantly more or less from use of *mx.TextTools*. Moreover, 34 seconds is a long time in an interactive application, but is not very long at all for a batch process done once a day, or once a week.

Example: Buyer/Order Report Parsing

Recall (or refer to) the sample report presented in the previous section "An Introduction to State Machines." A report contained a mixture of header material, buyer orders, and comments. The state machine we used looked at each successive line of the file and decided based on context whether the new line indicated a new state should start. It would be possible to write almost the same algorithm utilizing *mx.TextTools* only to speed up the decisions, but that is not what we will do.

A more representative use of *mx.TextTools* is to produce a concrete parse tree of the interesting components of the report document. In principle, you should be able to create a "grammar" that describes every valid "buyer report" document, but in practice using a mixed procedural/grammar approach is much easier, and more maintainable—at least for the test report.

An *mx.TextTools* tag table is a miniature state machine that either matches or fails to match a portion of a string. Matching, in this context, means that a "success" end state is reached, while nonmatching means that a "failure" end state is reached. Falling off the end of the tag table is a success state. Each individual state in a tag table tries to match some smaller construct by reading from the "read-head" and moving the read-head correspondingly. On either success or failure, program flow jumps to an indicated target state (which might be a success or failure state for the tag table as a whole). Of course, the jump target for success is often different from the jump target for failure—but there are only these two possible choices for jump targets, unlike the *statemachine* module's indefinite number.

Notably, one of the types of states you can include in a tag table is another tag table. That one state can "externally" look like a simple match attempt, but internally it might involve complex subpatterns and machine flow in order to determine if the state is a match or nonmatch. Much as in an EBNF grammar, you can build nested constructs for recognition of complex patterns. States can also have special behavior, such as function callbacks—but in general, an *mx.TextTools* tag table state is simply a binary match/nonmatch switch.

Let us look at an *mx.TextTools* parsing application for "buyer reports" and then examine how it works:

buyer_report.py

```
from mx.TextTools import *

word_set = set(alphanumeric+white+'-')
quant_set = set(number+'kKmM')

item   = ( (None, AllInSet, newline_set, +1),             # 1
           (None, AllInSet, white_set, +1),               # 2
           ('Prod', AllInSet, a2z_set, Fail),             # 3
           (None, AllInSet, white_set, Fail),             # 4
           ('Quant', AllInSet, quant_set, Fail),          # 5
           (None, WordEnd, '\n', -5) )                    # 6

buyers = ( ('Order', Table,                               # 1
                 ( (None, WordEnd, '\n>> ', Fail),        # 1.1
                   ('Buyer', AllInSet, word_set, Fail),    # 1.2
                 ('Item', Table, item, MatchOk, +0) ),    # 1.3
               Fail, +0), )

comments = ( ('Comment', Table,                           # 1
               ( (None, Word, '\n*', Fail),               # 1.1
                 (None, WordEnd, '*\n', Fail),            # 1.2
                 (None, Skip, -1) ),                      # 1.3
               +1, +2),
             (None, Skip, +1),                            # 2
             (None, EOF, Here, -2) )                      # 3

def unclaimed_ranges(tagtuple):
    starts = [0] + [tup[2] for tup in tagtuple[1]]
    stops = [tup[1] for tup in tagtuple[1]] + [tagtuple[2]]
    return zip(starts, stops)

def report2data(s):
    comtuple = tag(s, comments)
    taglist = comtuple[1]
    for beg,end in unclaimed_ranges(comtuple):
        taglist.extend(tag(s, buyers, beg, end)[1])
    taglist.sort(cmp)
    return taglist

if __name__=='__main__':
    import sys, pprint
    pprint.pprint(report2data(sys.stdin.read()))
```

Several tag tables are defined in *buyer_report*: `item`, `buyers`, and `comments`. State machines such as those in each tag table are general matching engines that can be used to identify patterns; after working with *mx.TextTools* for a while, you might accumulate a library of useful tag tables. As mentioned above, states in tag tables can reference other tag tables, either by name or inline. For example, `buyers` contains an inline tag table, while this inline tag table utilizes the tag table named `item`.

Let us take a look, step by step, at what the `buyers` tag table does. In order to *do* anything, a tag table needs to be passed as an argument to the `mx.TextTools.tag()` function, along with a string to match against. That is done in the `report2data()` function in the example. But in general, `buyers`—or any tag table—contains a list of states, each containing branch offsets. In the example, all such states are numbered in comments. `buyers` in particular contains just one state, which contains a subtable with three states.

Tag table state in `buyers`

1. Try to match the subtable. If the match succeeds, add the name `Order` to the taglist of matches. If the match fails, do not add anything. If the match succeeds, jump back into the one state (i.e., +0). In effect, `buyers` loops as long as it succeeds, advancing the read-head on each such match.

Subtable states in `buyers`

1. Try to find the end of the "word" `\n>>` in the string. That is, look for two greater-than symbols at the beginning of a line. If successful, move the read-head just past the point that first matched. If this state match fails, jump to `Fail`—that is, the (sub)table as a whole fails to match. No jump target is given for a successful match, so the default jump of +1 is taken. Since `None` is the tag object, do not add anything to the taglist upon a state match.

2. Try to find some `word_set` characters. This set of characters is defined in *buyer_report*; various other sets are defined in *mx.TextTools* itself. If the match succeeds, add the name `Buyer` to the taglist of matches. As many contiguous characters in the set as possible are matched. The match is considered a failure if there is not at least one such character. If this state match fails, jump to `Fail`, as in state (1).

3. Try to match the `item` tag table. If the match succeeds, add the name `Item` to the taglist of matches. What gets added, moreover, includes anything added within the `item` tag table. If the match fails, jump to `MatchOk`—that is, the (sub)table as a whole matches. If the match succeeds, jump +0—that is, keep looking for another `Item` to add to the taglist.

What *buyer_report* actually does is to first identify any comments, then to scan what is left in between comments for buyer orders. This approach proved easier to understand. Moreover, the design of *mx.TextTools* allows us to do this with no real inefficiency. Tagging a string does not involve actually pulling out the slices that match patterns, but simply identifying numerically the offset ranges where they occur. This approach is much "cheaper" than performing repeated slices, or otherwise creating new strings.

The following is important to notice: As of version 2.1.0, the documentation of the `mx.TextTools.tag()` function that accompanies *mx.TextTools* does not match its behavior! If the optional third and fourth arguments are passed to `tag()` they must indicate the start and end offsets within a larger string to scan, *not* the starting offset and length. Hopefully, later versions will fix the discrepancy (either approach would be fine, but could cause some breakage in existing code).

What *buyer_report* produces is a data structure, not final output. This data structure looks something like:

buyer_report.py data structure

```
$ python ex_mx.py < recs.tmp
[('Order', 0,  638,
  [('Buyer', 547, 562, None),
   ('Item', 562, 583,
    [('Prod', 566, 573, None), ('Quant', 579, 582, None)]),
   ('Item', 583, 602,
     [('Prod', 585, 593, None), ('Quant', 597, 601, None)]),
   ('Item', 602, 621,
     [('Prod', 604, 611, None), ('Quant', 616, 620, None)]),
   ('Item', 621, 638,
     [('Prod', 623, 632, None), ('Quant', 635, 637, None)])]),
 ('Comment', 638, 763, []),
 ('Order', 763, 805,
  [('Buyer', 768, 776, None),
   ('Item', 776, 792,
    [('Prod', 778, 785, None), ('Quant', 788, 791, None)]),
   ('Item', 792, 805,
    [('Prod', 792, 800, None), ('Quant', 802, 804, None)])]),
 ('Order', 805, 893,
  [('Buyer', 809, 829, None),
   ('Item', 829, 852,
    [('Prod', 833, 840, None), ('Quant', 848, 851, None)]),
   ('Item', 852, 871,
    [('Prod', 855, 863, None), ('Quant', 869, 870, None)]),
   ('Item', 871, 893,
    [('Prod', 874, 879, None), ('Quant', 888, 892, None)])]),
 ('Comment', 893, 952, []),
 ('Comment', 952, 1025, []),
 ('Comment', 1026, 1049, []),
 ('Order', 1049, 1109,
  [('Buyer', 1054, 1069, None),
   ('Item',1069, 1109,
    [('Prod', 1070, 1077, None), ('Quant', 1083, 1086, None)])])]
```

While this is "just" a new data structure, it is quite easy to deal with compared to raw textual reports. For example, here is a brief function that will create well-formed XML out of any taglist. You could even arrange for it to be valid XML by designing tag tables to match DTDs (see Chapter 5 for details about XML, DTDs, etc.):

```
def taglist2xml(s, taglist, root):
    print '<%s>' % root
    for tt in taglist:
        if tt[3]:
            taglist2xml(s, tt[3], tt[0])
        else:
            print '<%s>%s</%s>' % (tt[0], s[tt[1]:tt[2]], tt[0])
    print '</%s>' % root
```

Example: Marking up smart ASCII

The "smart ASCII" format uses email-like conventions to lightly mark features like word emphasis, source code, and URL links. This format—with LaTeX as an intermediate format—was used to produce the book you hold (which was written using a variety of plaintext editors). By obeying just a few conventions (that are almost the same as you would use on Usenet or in email), a writer can write without much clutter, but still convert to production-ready markup.

The `Txt2Html` utility uses a block-level state machine, combined with a collection of inline-level regular expressions, to identify and modify markup patterns in smart ASCII texts. Even though Python's regular expression engine is moderately slow, converting a five-page article takes only a couple seconds. In practice, `Txt2Html` is more than adequate for my own 20 kilobyte documents. However, it is easy to imagine a not-so-different situation where you were converting multimegabyte documents and/or delivering such dynamically converted content on a high-volume Web site. In such a case, Python's string operations, and especially regular expressions, would simply be too slow.

mx.TextTools can do everything regular expressions can, plus some things regular expressions cannot. In particular, a taglist can contain recursive references to matched patterns, which regular expressions cannot. The utility `mxTypography.py` utilizes several *mx.TextTools* capabilities the prior example did not use. Rather than create a nested data structure, `mxTypography.py` utilizes a number of callback functions, each responding to a particular match event. As well, `mxTypography.py` adds some important debugging techniques. Something similar to these techniques is almost required for tag tables that are likely to be updated over time (or simply to aid the initial development). Overall, this looks like a robust application should.

mx.TextTools version of Typography()

```
from mx.TextTools import *
import string, sys

#-- List of all words with  markup, head position, loop count
ws, head_pos, loops = [], None, 0

#-- Define "emitter" callbacks for each output format
def emit_misc(tl,txt,l,r,s):
    ws.append(txt[l:r])
def emit_func(tl,txt,l,r,s):
    ws.append(’<code>’+txt[l+1:r-1]+’</code>’)
def emit_modl(tl,txt,l,r,s):
    ws.append(’<em><code>’+txt[l+1:r-1]+’</code></em>’)
def emit_emph(tl,txt,l,r,s):
    ws.append(’<em>’+txt[l+1:r-1]+’</em>’)
def emit_strg(tl,txt,l,r,s):
    ws.append(’<strong>’+txt[l+1:r-1]+’</strong>’)
def emit_titl(tl,txt,l,r,s):
    ws.append(’<cite>’+txt[l+1:r-1]+’</cite>’)
def jump_count(tl,txt,l,r,s):
    global head_pos, loops
    loops = loops+1
    if head_pos is None: head_pos = r
    elif head_pos == r:
        raise "InfiniteLoopError", \
              txt[l-20:l]+’{’+txt[l]+’}’+txt[l+1:r+15]
    else: head_pos = r

#-- What can appear inside, and what can be, markups?
punct_set = set("‘!@#$%^&*()_-+=|\{}[]:;’<>,.?/"+’"’)
markable = alphanumeric+whitespace+"‘!@#$%^&()+=|\{}:;<>,.?/"+’"’
markable_func = set(markable+"*-_[]")
markable_modl = set(markable+"*-_’")
markable_emph = set(markable+"*_’[]")
markable_strg = set(markable+"-_’[]")
markable_titl = set(markable+"*-’[]")
markup_set    = set("-*’[]_")

#-- What can precede and follow markup phrases?
darkins = ’(/"’
leadins = whitespace+darkins      # might add from "-*’[]_"
darkouts = ’/.),:;?!"’
darkout_set = set(darkouts)
leadouts = whitespace+darkouts    # for non-conflicting markup
leadout_set = set(leadouts)

#-- What can appear inside plain words?
word_set = set(alphanumeric+’{}/@#$%^&-_+=|\><’+darkouts)
wordinit_set = set(alphanumeric+"$#+\<.&{"+darkins)

#-- Define the word patterns (global so as to do it only at import)
# Special markup
def markup_struct(lmark, rmark, callback, markables, x_post="-"):
    struct = \
      ( callback, Table+CallTag,
        ( (None, Is, lmark),                 # Starts with left marker
          (None, AllInSet, markables),       # Stuff marked
          (None, Is, rmark),                 # Ends with right marker
          (None, IsInSet, leadout_set,+2,+1),# EITHR: postfix w/ leadout
          (None, Skip, -1,+1, MatchOk),      # ..give back trailng ldout
          (None, IsIn, x_post, MatchFail),   # OR: special case postfix
          (None, Skip, -1,+1, MatchOk)       # ..give back trailing char
        )
      )
    return struct
funcs   = markup_struct("'", "'", emit_func, markable_func)
modules = markup_struct("[", "]", emit_modl, markable_modl)
emphs   = markup_struct("-", "-", emit_emph, markable_emph, x_post="")
strongs = markup_struct("*", "*", emit_strg, markable_strg)
titles  = markup_struct("_", "_", emit_titl, markable_titl)

# All the stuff not specially marked
plain_words = \
 ( ws, Table+AppendMatch,           # AppendMatch only -slightly-
   ( (None, IsInSet,                #  faster than emit_misc callback
        wordinit_set, MatchFail),   # Must start with word-initial
     (None, Is, "'",+1),            # May have apostrophe next
     (None, AllInSet, word_set,+1), # May have more word-internal
     (None, Is, "'", +2),           # May have trailing apostrophe
     (None, IsIn, "st",+1),         # May have [ts] after apostrophe
     (None, IsInSet,
        darkout_set,+1, MatchOk),   # Postfixed with dark lead-out
     (None, IsInSet,
        whitespace_set, MatchFail), # Give back trailing whitespace
     (None, Skip, -1)
   ) )
# Catch some special cases
bullet_point = \
 ( ws, Table+AppendMatch,
   ( (None, Word+CallTag, "* "),       # Asterisk bullet is a word
   ) )
horiz_rule = \
 ( None, Table,
   ( (None, Word, "-"*50),             # 50 dashes in a row
     (None, AllIn, "-"),               # More dashes
   ) )
into_mark = \
 ( ws, Table+AppendMatch,             # Special case where dark leadin
   ( (None, IsInSet, set(darkins)),   #   is followed by markup char
     (None, IsInSet, markup_set),
     (None, Skip, -1)                 # Give back the markup char
   ) )
stray_punct = \
 ( ws, Table+AppendMatch,              # Pickup any cases where multiple
   ( (None, IsInSet, punct_set),       # punctuation character occur
     (None, AllInSet, punct_set),      # alone (followed by whitespace)
     (None, IsInSet, whitespace_set),
     (None, Skip, -1)                  # Give back the whitespace
   ) )
leadout_eater = (ws, AllInSet+AppendMatch, leadout_set)

#-- Tag all the (possibly marked-up) words
tag_words = \
 ( bullet_point+(+1,),
   horiz_rule + (+1,),
   into_mark  + (+1,),
   stray_punct+ (+1,),
   emphs   + (+1,),
   funcs   + (+1,),
   strongs + (+1,),
   modules + (+1,),
   titles  + (+1,),
   into_mark+(+1,),
   plain_words +(+1,),             # Since file is mstly plain wrds, can
   leadout_eater+(+1,-1),          # shortcut by tight looping (w/ esc)
   (jump_count, Skip+CallTag, 0),  # Check for infinite loop
   (None, EOF, Here, -13)          # Check for EOF
 )
def Typography(txt):
    global ws
    ws = []    # clear the list before we proceed
    tag(txt, tag_words, 0, len(txt), ws)
    return string.join(ws, '')

if __name__ == '__main__':
    print Typography(open(sys.argv[1]).read())
```

`mxTypographify.py` reads through a string and determines if the next bit of text matches one of the markup patterns in `tag_words`. Or rather, it better match some pattern or the application just will not know what action to take for the next bit of text. Whenever a named subtable matches, a callback function is called, which leads to a properly annotated string being appended to the global list `ws`. In the end, all such appended strings are concatenated.

Several of the patterns given are mostly fallback conditions. For example, the `stray_punct` tag table detects the condition where the next bit of text is some punctuation symbols standing alone without abutting any words. In most cases, you don't *want* smart ASCII to contain such a pattern, but *mxTypographify* has to do *something* with them if they are encountered.

Making sure that every subsequence is matched by some subtable or another is tricky. Here are a few examples of matches and failures for the `stray_punct` subtable. Everything that does not match this subtable needs to match some other subtable instead:

```
-- spam      # matches "--"
& spam       # fails at "AllInSet" since '&' advanced head
#@$ %% spam  # matches "#@$"
**spam       # fails (whitespace isn't encountered before 's')
```

After each success, the read-head is at the space right before the next word "spam" or "%%". After a failure, the read-head remains where it started out (at the beginning of the line).

Like `stray_punct`, `emphs`, `funcs`, `strongs`, `plain_words`, et cetera contain tag tables. Each entry in `tag_words` has its appropriate callback functions (all "emitters" of various names, because they "emit" the match, along with surrounding markup if needed). Most lines each have a "+1" appended to their tuple; what this does is specify where to jump in case of a match failure. That is, even if these patterns fail to match, we continue on—with the read-head in the same position—to try matching against the other patterns.

After the basic word patterns each attempt a match, we get to the "leadout eater" line. For `mxTypography.py`, a "leadout" is the opposite of a "leadin." That is, the latter are things that might precede a word pattern, and the former are things that might follow a word pattern. The `leadout_set` includes whitespace characters, but it also includes things like a comma, period, and question mark, which might end a word. The "leadout eater" uses a callback function, too. As designed, it preserves exactly the whitespace the input has. However, it would be easy to normalize whitespace here by emitting something other than the actual match (e.g., a single space always).

The `jump_count` is extremely important; we will come back to it momentarily. For now, it is enough to say that we *hope* the line never does anything.

The `EOF` line is our flow control, in a way. The call made by this line is to `None`, which is to say that nothing is actually *done* with any match. The command `EOF` is the important thing (`Here` is just a filler value that occupies the tuple position). It succeeds if the read-head is past the end of the read buffer. On success, the whole tag table `tag_words` succeeds, and having succeeded, processing stops. `EOF` failure is more interesting. Assuming we haven't reached the end of our string, we jump -13 states (to `bullet_point`). From there, the whole process starts over, hopefully with the read-head advanced to the next word. By looping back to the start of the list of tuples, we continue eating successive word patterns until the read buffer is exhausted (calling callbacks along the way).

The `tag()` call simply launches processing of the tag table we pass to it (against the read buffer contained in `txt`). In our case, we do not care about the return value of `tag()` since everything is handled in callbacks. However, in cases where the tag table does not loop itself, the returned tuple can be used to determine if there is reason to call `tag()` again with a tail of the read buffer.

DEBUGGING A TAG TABLE

Describing it is easy, but I spent a large number of hours finding the exact collection of tag tables that would match every pattern I was interested in without mismatching any pattern as something it wasn't. While smart ASCII markup seems pretty simple, there are actually quite a few complications (e.g., markup characters being used in nonmarkup contexts, or markup characters and other punctuation appearing in various sequences). Any structured document format that is complicated enough to warrant using *mx.TextTools* instead of *string* is likely to have similar complications.

Without question, the worst thing that can go wrong in a looping state pattern like the one above is that *none* of the listed states match from the current read-head position. If that happens, your program winds up in a tight infinite loop (entirely inside the extension module, so you cannot get at it with Python code directly). I wound up forcing a manual kill of the process *countless* times during my first brush at *mx.TextTools* development.

Fortunately, there is a solution to the infinite loop problem. This is to use a callback like `jump_count`.

mxTypography.py infinite loop catcher

```
def jump_count(taglist,txt,l,r,subtag):
    global head_pos
    if head_pos is None: head_pos = r
    elif head_pos == r:
        raise "InfiniteLoopError", \
              txt[l-20:l]+'{'+txt[l]+'}'+txt[l+1:r+15]
    else: head_pos = r
```

The basic purpose of `jump_count` is simple: We want to catch the situation where our tag table has been run through multiple times without matching anything. The simplest way to do this is to check whether the last read-head position is the same as the current. If it is, more loops cannot get anywhere, since we have reached the exact same state twice, and the same thing is fated to happen forever. `mxTypography.py` simply raises an error to stop the program (and reports a little bit of buffer context to see what is going on).

It is also possible to move the read-head manually and try again from a different starting position. To manipulate the read head in this fashion, you could use the `Call` command in tag table items. But a better approach is to create a nonlooping tag table that is called repeatedly from a Python loop. This Python loop can look at a returned tuple and use adjusted offsets in the next call if no match occurred. Either way, since much more time is spent in Python this way than with the loop tag table approach, less speed would be gained from *mx.TextTools*.

Not as bad as an infinite loop, but still undesirable, is having patterns within a tag table match when they are not supposed to or not match when they are suppose to (but something else has to match, or we would have an infinite loop issue). Using callbacks everywhere makes examining this situation much easier. During development, I frequently create temporary changes to my `emit_*` callbacks to print or log when certain emitters get called. By looking at output from these temporary `print` statements, most times you can tell where the problem lies.

CONSTANTS

The *mx.TextTools* module contains constants for a number of frequently used collections of characters. Many of these character classes are the same as ones in the *string* module. Each of these constants also has a `set` version predefined; a set is an efficient representation of a character class that may be used in tag tables and other *mx.TextTools* functions. You may also obtain a character set from a (custom) character class using the *`mx.TextTools.set()`* function:

```
>>> from mx.TextTools import a2z, set
>>> varname_chars = a2z + '_'
>>> varname_set = set(varname_chars)
```

mx.TextTools.a2z
mx.TextTools.a2z_set

English lowercase letters ("abcdefghijklmnopqrstuvwxyz").

mx.TextTools.A2Z
mx.TextTools.A2Z_set

English uppercase letters ("ABCDEFGHIJKLMNOPQRSTUVWXYZ").

mx.TextTools.umlaute
mx.TextTools.umlaute_set

Extra German lowercase hi-bit characters.

mx.TextTools.Umlaute
mx.TextTools.Umlaute_set

Extra German uppercase hi-bit characters.

mx.TextTools.alpha
mx.TextTools.alpha_set

English letters (A2Z + a2z).

mx.TextTools.german_alpha
mx.TextTools.german_alpha_set

German letters (A2Z + a2z + umlaute + Umlaute).

mx.TextTools.number
mx.TextTools.number_set

The decimal numerals ("0123456789").

mx.TextTools.alphanumeric
mx.TextTools.alphanumeric_set

English numbers and letters (alpha + number).

mx.TextTools.white
mx.TextTools.white_set

Spaces and tabs (" \t\v"). This is more restricted than *string.whitespace*.

mx.TextTools.newline
mx.TextTools.newline_set

Line break characters for various platforms ("\n\r").

mx.TextTools.formfeed
mx.TextTools.formfeed_set

Formfeed character ("\f").

mx.TextTools.whitespace
mx.TextTools.whitespace_set

Same as *string.whitespace* (`white+newline+formfeed`).

mx.TextTools.any
mx.TextTools.any_set

All characters (0x00-0xFF).

SEE ALSO: string.digits *130*; string.hexdigits *130*; string.octdigits *130*; string.lowercase *131*; string.uppercase *131*; string.letters *131*; string.punctuation *131*; string.whitespace *131*; string.printable *132*;

COMMANDS

Programming in *mx.TextTools* amounts mostly to correctly configuring tag tables. Utilizing a tag table requires just one call to the *mx.TextTools.tag()*, but inside a tag table is a kind of mini-language—something close to a specialized Assembly language, in many ways.

Each tuple within a tag table contains several elements, of the form:

```
(tagobj, command[+modifiers], argument
         [,jump_no_match=MatchFail [,jump_match=+1]])
```

The "tag object" may be `None`, a callable object, or a string. If `tagobj` is `None`, the indicated pattern may match, but nothing is added to a taglist data structure if so, nor is a callback invoked. If a callable object (usually a function) is given, it acts as a callback for a match. If a string is used, it is used to name a part of the taglist data structure returned by a call to *mx.TextTools.tag()*.

A command indicates a type of pattern to match, and a modifier can change the behavior that occurs in case of such a match. Some commands succeed or fail unconditionally, but allow you to specify behaviors to take if they are reached. An argument is required, but the specific values that are allowed and how they are interpreted depends on the command used.

Two jump conditions may optionally be specified. If no values are given, `jump_no_match` defaults to `MatchFail`—that is, unless otherwise specified, failing to match a tuple in a tag table causes the tag table as a whole to fail. If a value *is* given, `jump_no_match` branches to a tuple the specified number of states forward or backward. For clarity, an explicit leading "+" is used in forward branches. Branches backward will begin with a minus sign. For example:

```
# Branch forward one state if next character -is not- an X
# ... branch backward three states if it is an X
tupX = (None, Is, 'X', +1, -3)
# assume all the tups are defined somewhere...
tagtable = (tupA, tupB, tupV, tupW, tupX, tupY, tupZ)
```

If no value is given for `jump_match`, branching is one state forward in the case of a match.

Version 2.1.0 of *mx.TextTools* adds named jump targets, which are often easier to read (and maintain) than numeric offsets. An example is given in the *mx.TextTools* documentation:

```
tag_table = ('start',
             ('lowercase',AllIn,a2z,+1,'skip'),
             ('upper',AllIn,A2Z,'skip'),
             'skip',
             (None,AllIn,white+newline,+1),
             (None,AllNotIn,alpha+white+newline,+1),
             (None,EOF,Here,'start') )
```

It is easy to see that if you were to add or remove a tuple, it is less error prone to retain a jump to, for example, `skip` than to change every necessary `+2` to a `+3` or the like.

UNCONDITIONAL COMMANDS

mx.TextTools.Fail
mx.TextTools.Jump

Nonmatch at this tuple. Used mostly for documentary purposes in a tag table, usually with the `Here` or `To` placeholder. The tag tables below are equivalent:

```
table1 = ( ('foo', Is, 'X', MatchFail, MatchOk), )
table2 = ( ('foo', Is, 'X', +1, +2),
           ('Not_X', Fail, Here) )
```

The `Fail` command may be preferred if several other states branch to the same failure, or if the condition needs to be documented explicitly.

`Jump` is equivalent to `Fail`, but it is often better self-documenting to use one rather than the other; for example:

```
tup1 = (None, Fail, Here, +3)
tup2 = (None, Jump, To, +3)
```

mx.TextTools.Skip
mx.TextTools.Move

Match at this tuple, and change the read-head position. `Skip` moves the read-head by a relative amount, `Move` to an absolute offset (within the slice the tag table is operating on). For example:

```
# read-head forward 20 chars, jump to next state
tup1 = (None, Skip, 20)
# read-head to position 10, and jump back 4 states
tup2 = (None, Move, 10, 0, -4)
```

Negative offsets are allowed, as in Python list indexing.

MATCHING PARTICULAR CHARACTERS

mx.TextTools.AllIn
mx.TextTools.AllInSet
mx.TextTools.AllInCharSet

Match all characters up to the first that is not included in `argument`. `AllIn` uses a character string while `AllInSet` uses a set as `argument`. For version 2.1.0, you may also use `AllInCharSet` to match `CharSet` objects. In general, the set or CharSet form will be faster and is preferable. The following are functionally the same:

```
tup1 = ('xyz', AllIn, 'XYZxyz')
tup2 = ('xyz', AllInSet, set('XYZxyz')
tup3 = ('xyz', AllInSet, CharSet('XYZxyz'))
```

At least one character must match for the tuple to match.

mx.TextTools.AllNotIn

Match all characters up to the first that *is* included in `argument`. As of version 2.1.0, *mx.TextTools* does not include an `AllNotInSet` command. However, the following tuples are functionally the same (the second usually faster):

```
from mx.TextTools import AllNotIn, AllInSet, invset
tup1 = ('xyz', AllNotIn, 'XYZxyz')
tup2 = ('xyz', AllInSet, invset('xyzXYZ'))
```

At least one character must match for the tuple to match.

mx.TextTools.Is

Match specified character. For example:

```
tup = ('X', Is, 'X')
```

mx.TextTools.IsNot

Match any one character except the specified character.

```
tup = ('X', IsNot, 'X')
```

mx.TextTools.IsIn
mx.TextTools.IsInSet
mx.TextTools.IsInCharSet

Match exactly one character if it is in `argument`. `IsIn` uses a character string while `IsInSet` use a set as `argument`. For version 2.1.0, you may also use `IsInCharSet` to match `CharSet` objects. In general, the set or CharSet form will be faster and is preferable. The following are functionally the same:

```
tup1 = ('xyz', IsIn, 'XYZxyz')
tup2 = ('xyz', IsInSet, set('XYZxyz')
tup3 = ('xyz', IsInSet, CharSet('XYZxyz')
```

mx.TextTools.IsNotIn

Match exactly one character if it is *not* in `argument`. As of version 2.1.0, *mx.TextTools* does not include an `'AllNotInSet` command. However, the following tuples are functionally the same (the second usually faster):

```
from mx.TextTools import IsNotIn, IsInSet, invset
tup1 = ('xyz', IsNotIn, 'XYZxyz')
tup2 = ('xyz', IsInSet, invset('xyzXYZ'))
```

MATCHING SEQUENCES

mx.TextTools.Word

Match a word at the current read-head position. For example:

```
tup = ('spam', Word, 'spam')
```

mx.TextTools.WordStart
mx.TextTools.sWordStart
mx.TextTools.WordEnd
mx.TextTools.sWordEnd

Search for a word, and match up to the point of the match. Searches performed in this manner are extremely fast, and this is one of the most powerful elements of tag tables. The commands `sWordStart` and `sWordEnd` use “search objects” rather than plaintexts (and are significantly faster).

WordStart and sWordStart leave the read-head immediately prior to the matched word, if a match succeeds. WordEnd and sWordEnd leave the read-head immediately after the matched word. On failure, the read-head is not moved for any of these commands.

```
>>> from mx.TextTools import *
>>> s = 'spam and eggs taste good'
>>> tab1 = ( ('toeggs', WordStart, 'eggs'), )
>>> tag(s, tab1)
(1, [('toeggs', 0, 9, None)], 9)
>>> s[0:9]
'spam and '
>>> tab2 = ( ('pasteggs', sWordEnd, BMS('eggs')), )
>>> tag(s, tab2)
(1, [('pasteggs', 0, 13, None)], 13)
>>> s[0:13]
'spam and eggs'
```

SEE ALSO: mx.TextTools.BMS() *307*; mx.TextTools.sFindWord *303*;

mx.TextTools.sFindWord

Search for a word, and match only that word. Any characters leading up to the match are ignored. This command accepts a search object as an argument. In case of a match, the read-head is positioned immediately after the matched word.

```
>>> from mx.TextTools import *
>>> s = 'spam and eggs taste good'
>>> tab3 = ( ('justeggs', sFindWord, BMS('eggs')), )
>>> tag(s, tab3)
(1, [('justeggs', 9, 13, None)], 13)
>>> s[9:13]
'eggs'
```

SEE ALSO: mx.TextTools.sWordEnd *302*;

mx.TextTools.EOF

Match if the read-head is past the end of the string slice. Normally used with placeholder argument Here, for example:

```
tup = (None, EOF, Here)
```

COMPOUND MATCHES

mx.TextTools.Table
mx.TextTools.SubTable

Match if the table given as `argument` matches at the current read-head position. The difference between the `Table` and the `SubTable` commands is in where matches get inserted. When the `Table` command is used, any matches in the indicated table are nested in the data structure associated with the tuple. When `SubTable` is used, matches are written into the current level taglist. For example:

```
>>> from mx.TextTools import *
>>> from pprint import pprint
>>> caps = ('Caps', AllIn, A2Z)
>>> lower = ('Lower', AllIn, a2z)
>>> words = ( ('Word', Table, (caps, lower)),
...           (None, AllIn, whitespace, MatchFail, -1) )
>>> from pprint import pprint
>>> pprint(tag(s, words))
(0,
 [('Word', 0, 4, [('Caps', 0, 1, None), ('Lower', 1, 4, None)]),
  ('Word', 5, 19, [('Caps', 5, 6, None), ('Lower', 6, 19, None)]),
  ('Word', 20, 29, [('Caps', 20, 24, None), ('Lower', 24, 29, None)]),
  ('Word', 30, 35, [('Caps', 30, 32, None), ('Lower', 32, 35, None)])
 ],
 35)
>>> flatwords = ( (None, SubTable, (caps, lower)),
...               (None, AllIn, whitespace, MatchFail, -1) )
>>> pprint(tag(s, flatwords))
(0,
 [('Caps', 0, 1, None),
  ('Lower', 1, 4, None),
  ('Caps', 5, 6, None),
  ('Lower', 6, 19, None),
  ('Caps', 20, 24, None),
  ('Lower', 24, 29, None),
  ('Caps', 30, 32, None),
  ('Lower', 32, 35, None)],
 35)
```

For either command, if a match occurs, the read-head is moved to immediately after the match.

The special constant `ThisTable` can be used instead of a tag table to call the current table recursively.

mx.TextTools.TableInList
mx.TextTools.SubTableInList

Similar to `Table` and `SubTable` except that the `argument` is a tuple of the form `(list_of_tables,index)`. The advantage (and the danger) of this is that a list is mutable and may have tables added after the tuple defined—in particular, the containing tag table may be added to `list_of_tables` to allow recursion. Note, however, that the special value `ThisTable` can be used with the `Table` or `SubTable` commands and is usually more clear.

SEE ALSO: mx.TextTools.Table *304*; mx.TextTools.SubTable *304*;

mx.TextTools.Call

Match on any computable basis. Essentially, when the `Call` command is used, control over parsing/matching is turned over to Python rather than staying in the *mx.TextTools* engine. The function that is called must accept arguments `s`, `pos`, and `end`—where `s` is the underlying string, `pos` is the current read-head position, and `end` is ending of the slice being processed. The called function must return an integer for the new read-head position; if the return is different from `pos`, the match is a success.

As an example, suppose you want to match at a certain point only if the next N characters make up a dictionary word. Perhaps an efficient stemmed data structure is used to represent the dictionary word list. You might check dictionary membership with a tuple like:

```
tup = ('DictWord', Call, inDict)
```

Since the function `inDict` is written in Python, it will generally not operate as quickly as does an *mx.TextTools* pattern tuple.

mx.TextTools.CallArg

Same as `Call`, except `CallArg` allows passing additional arguments. For example, suppose the dictionary example given in the discussion of `Call` also allows you to specify language and maximum word length for a match:

```
tup = ('DictWord', Call, (inDict,['English',10]))
```

SEE ALSO: mx.TextTools.Call *305*;

MODIFIERS

mx.TextTools.CallTag

Instead of appending `(tagobj,l,r,subtags)` to the taglist upon a successful match, call the function indicated as the tag object (which must be a function rather than

`None` or a string). The function called must accept the arguments `taglist`, `s`, `start`, `end`, and `subtags`—where `taglist` is the present taglist, `s` is the underlying string, `start` and `end` are the slice indices of the match, and `subtags` is the nested taglist. The function called *may*, but need not, append to or modify `taglist` or `subtags` as part of its action. For example, a code parsing application might include:

```
>>> def todo_flag(taglist, s, start, end, subtags):
...     sys.stderr.write("Fix issue at offset %d\n" % start)
...
>>> tup = (todo_flag, Word+CallTag, 'XXX')
>>> tag('XXX more stuff', (tup,))
Fix issue at offset 0
(1, [], 3)
```

mx.TextTools.AppendMatch

Instead of appending `(tagobj,start,end,subtags)` to the taglist upon successful matching, append the match found as string. The produced taglist is "flattened" and cannot be used in the same manner as "normal" taglist data structures. The flat data structure is often useful for joining or for list processing styles.

```
>>> from mx.TextTools import *
>>> words = (('Word', AllIn+AppendMatch, alpha),
...          (None, AllIn, whitespace, MatchFail, -1))
>>> tag('this and that', words)
(0, ['this', 'and', 'that'], 13)
>>> join(tag('this and that', words)[1], '-')
'this-and-that'
```

SEE ALSO: string.split() *142*;

mx.TextTools.AppendToTagobj

Instead of appending `(tagobj,start,end,subtags)` to the taglist upon successful matching, call the `.append()` method of the tag object. The tag object must be a list (or a descendent of `list` in Python 2.2+).

```
>>> from mx.TextTools import *
>>> ws = []
>>> words = ((ws, AllIn+AppendToTagobj, alpha),
...          (None, AllIn, whitespace, MatchFail, -1))
>>> tag('this and that', words)
(0, [], 13)
>>> ws
[(None, 0, 4, None), (None, 5, 8, None), (None, 9, 13, None)]
```

SEE ALSO: mx.TextTools.CallTag *305*;

mx.TextTools.AppendTagobj

Instead of appending (tagobj,start,end,subtags) to the taglist upon successful matching, append the tag object. The produced taglist is usually nonstandard and cannot be used in the same manner as "normal" taglist data structures. A flat data structure is often useful for joining or for list processing styles.

```
>>> from mx.TextTools import *
>>> words = (('word', AllIn+AppendTagobj, alpha),
...          (None, AllIn, whitespace, MatchFail, -1))
>>> tag('this and that', words)
(0, ['word', 'word', 'word'], 13)
```

mx.TextTools.LookAhead

If this modifier is used, the read-head position is not changed when a match occurs. As the name suggests, this modifier allows you to create patterns similar to regular expression lookaheads.

```
>>> from mx.TextTools import *
>>> from pprint import pprint
>>> xwords = ((None, IsIn+LookAhead, 'Xx', +2),
...           ('xword', AllIn, alpha, MatchFail, +2),
...           ('other', AllIn, alpha),
...           (None, AllIn, whitespace, MatchFail, -3))
>>> pprint(tag('Xylophone trumpet xray camera', xwords))
(0,
 [('xword', 0, 9, None),
  ('other', 10, 17, None),
  ('xword', 18, 22, None),
  ('other', 23, 29, None)],
 29)
```

CLASSES

mx.TextTools.BMS(word [,translate])
mx.TextTools.FS(word [,translate])
mx.TextTools.TextSearch(word [,translate [,algorithm=BOYERMOORE]])

Create a search object for the string word. This is similar in concept to a compiled regular expression. A search object has several methods to locate its encoded string within another string. The BMS name is short for "Boyer-Moore," which is a particular search algorithm. The name FS is reserved for accessing the "Fast Search" algorithm in future versions, but currently both classes use Boyer-Moore. For *mx.TextTools* 2.1.0+, you are urged to use the .TextSearch() constructor.

If a `translate` argument is given, the searched string is translated during the search. This is equivalent to transforming the string with *`string.translate()`* prior to searching it.

SEE ALSO: string.translate() *145*;

mx.TextTools.CharSet(definition)

Version 2.1.0 of *mx.TextTools* adds the Unicode-compatible `CharSet` object. `CharSet` objects may be initialized to support character ranges, as in regular expressions; for example, `definition="a-mXYZ"`. In most respects, `CharSet` objects are similar to older sets.

METHODS AND ATTRIBUTES

mx.TextTools.BMS.search(s [,start [,end]])
mx.TextTools.FS.search(s [,start [,end]])
mx.TextTools.TextSearch.search(s [,start [,end]])

Locate as a slice the first match of the search object against `s`. If optional arguments `start` and `end` are used, only the slice `s[start:end]` is considered. Note: As of version 2.1.0, the documentation that accompanies *mx.TextTools* inaccurately describes the `end` parameter of search object methods as indicating the length of the slice rather than its ending offset.

mx.TextTools.BMS.find(s, [,start [,end]])
mx.TextTools.FS.find(s, [,start [,end]])
mx.TextTools.TextSearch.search(s [,start [,end]])

Similar to *`mx.TextTools.BMS.search()`*, except return only the starting position of the match. The behavior is similar to that of *`string.find()`*.

SEE ALSO: string.find() *135*; mx.TextTools.find() *312*;

mx.TextTools.BMS.findall(s [,start [,end]])
mx.TextTools.FS.findall(s [,start [,end]])
mx.TextTools.TextSearch.search(s [,start [,end]])

Locate as slices *every* match of the search object against `s`. If the optional arguments `start` and `end` are used, only the slice `s[start:end]` is considered.

```
>>> from mx.TextTools import BMS, any, upper
>>> foosrch = BMS('FOO', upper(any))
>>> foosrch.search('foo and bar and FOO and BAR')
(0, 3)
>>> foosrch.find('foo and bar and FOO and BAR')
0
>>> foosrch.findall('foo and bar and FOO and BAR')
[(0, 3), (16, 19)]
>>> foosrch.search('foo and bar and FOO and BAR', 10, 20)
(16, 19)
```

SEE ALSO: re.findall *245*; mx.TextTools.findall() *312*;

mx.TextTools.BMS.match
mx.TextTools.FS.match
mx.TextTools.TextSearch.match

The string that the search object will look for in the search text (read-only).

mx.TextTools.BMS.translate
mx.TextTools.FS.translate
mx.TextTools.TextSearch.match

The translation string used by the object, or `None` if no `translate` string was specified.

mx.TextTools.CharSet.contains(c)

Return a true value if character `c` is in the `CharSet`.

mx.TextTools.CharSet.search(s [,direction [,start=0 [,stop=len(s)]]])

Return the position of the first `CharSet` character that occurs in `s[start:end]`. Return `None` if there is no match. You may specify a negative `direction` to search backwards.

SEE ALSO: re.search() *249*;

mx.TextTools.CharSet.match(s [,direction [,start=0 [,stop=len(s)]]])

Return the length of the longest contiguous match of the `CharSet` object against substrings of `s[start:end]`.

mx.TextTools.CharSet.split(s [,start=0 [,stop=len(text)]])

Return a list of substrings of `s[start:end]` divided by occurrences of characters in the `CharSet`.

SEE ALSO: re.search() *249*;

mx.TextTools.CharSet.splitx(s [,start=0 [,stop=len(text)]])

Like *`mx.TextTools.CharSet.split()`* except retain characters from `CharSet` in interspersed list elements.

mx.TextTools.CharSet.strip(s [,where=0 [,start=0 [,stop=len(s)]]])

Strip all characters in `s[start:stop]` appearing in the character set.

FUNCTIONS

Many of the functions in *mx.TextTools* are used by the tagging engine. A number of others are higher-level utility functions that do not require custom development of tag tables. The latter are listed under a separate heading and generally resemble faster versions of functions in the *string* module.

mx.TextTools.cmp(t1, t2)

Compare two valid taglist tuples on their slice positions. Taglists generated with multiple passes of *`mx.TextTools.tag()`*, or combined by other means, may not have tuples sorted in string order. This custom comparison function is coded in C and is very fast.

```
>>> import mx.TextTools
>>> from pprint import pprint
>>> tl = [('other', 10, 17, None),
...       ('other', 23, 29, None),
...       ('xword', 0, 9, None),
...       ('xword', 18, 22, None)]
>>> tl.sort(mx.TextTools.cmp)
>>> pprint(tl)
[('xword', 0, 9, None),
 ('other', 10, 17, None),
 ('xword', 18, 22, None),
 ('other', 23, 29, None)]
```

mx.TextTools.invset(s)

Identical to `mx.TextTools.set(s, 0)`.

SEE ALSO: mx.TextTools.set() *310*;

mx.TextTools.set(s [,includechars=1])

Return a bit-position encoded character set. Bit-position encoding makes tag table commands like `InSet` and `AllInSet` operate more quickly than their character-string equivalents (e.g, `In`, `AllIn`).

If `includechars` is set to 0, invert the character set.

SEE ALSO: mx.TextTools.invset() *310*;

mx.TextTools.tag(s, table [,start [,end [,taglist]]])

Apply a tag table to a string. The return value is a tuple of the form `(success, taglist, next)`. `success` is a binary value indicating whether the table matched. `next` is the read-head position after the match attempt. Even on a nonmatch of the table, the read-head might have been advanced to some degree by member tuples matching. The `taglist` return value contains the data structure generated by application. Modifiers and commands within the tag table can alter the composition of `taglist`; but in the normal case, `taglist` is composed of zero or more tuples of the form `(tagname, start, end, subtaglist)`.

Assuming a "normal" taglist is created, `tagname` is a string value that was given as a tag object in a tuple within the tag table. `start` and `end` the slice ends of that particular match. `subtaglist` is either `None` or a taglist for a subtable match.

If `start` or `end` are given as arguments to *mx.TextTools.tag()*, application is restricted to the slice `s[start:end]` (or `s[start:]` if only `start` is used). If a `taglist` argument is passed, that list object is used instead of a new list. This allows extending a previously generated taglist, for example. If `None` is passed as `taglist`, no taglist is generated.

See the application examples and command illustrations for a number of concrete uses of *mx.TextTools.tag()*.

UTILITY FUNCTIONS

mx.TextTools.charsplit(s, char, [start [,end]])

Return a list split around each `char`. Similar to *string.split()*, but faster. If the optional arguments `start` and `end` are used, only the slice `s[start:end]` is operated on.

SEE ALSO: string.split() *142*; mx.TextTools.setsplit() *314*;

mx.TextTools.collapse(s, sep=' ')

Return a string with normalized whitespace. This is equivalent to `string.join( string.split(s),sep)`, but faster.

```
>>> from mx.TextTools import collapse
>>> collapse('this and   that','-')
'this-and-that'
```

SEE ALSO: string.join() *137*; string.split() *142*;

mx.TextTools.countlines(s)

Returns the number of lines in `s` in a platform-portable way. Lines may end with CR (Mac-style), LF (Unix-style), or CRLF (DOS-style), including a mixture of these.

SEE ALSO: FILE.readlines() *17*; mx.TextTools.splitlines() *315*;

mx.TextTools.find(s, search_obj, [start, [,end]])

Return the position of the first match of `search_obj` against `s`. If the optional arguments `start` and `end` are used, only the slice `s[start:end]` is considered. This function is identical to the search object method of the same name; the syntax is just slightly different. The following are synonyms:

```
from mx.TextTools import BMS, find
s = 'some string with a pattern in it'
pos1 = find(s, BMS('pat'))
pos2 = BMS('pat').find(s)
```

SEE ALSO: string.find() *135*; mx.TextTools.BMS.find() *308*;

mx.TextTools.findall(s, search_obj [,start [,end]])

Return as slices *every* match of `search_obj` against `s`. If the optional arguments `start` and `end` are used, only the slice `s[start:end]` is considered. This function is identical to the search object method of the same name; the syntax is just slightly different. The following are synonyms:

```
from mx.TextTools import BMS, findall
s = 'some string with a pattern in it'
pos1 = findall(s, BMS('pat'))
pos2 = BMS('pat').findall(s)
```

SEE ALSO: mx.TextTools.find() *312*; mx.TextTools.BMS.findall() *308*;

mx.TextTools.hex2str(hexstr)

Returns a string based on the hex-encoded string `hexstr`.

```
>>> from mx.TextTools import hex2str, str2hex
>>> str2hex('abc')
'616263'
>>> hex2str('616263')
'abc'
```

SEE ALSO: mx.TextTools.str2hex() *315*;

mx.TextTools.is_whitespace(s [,start [,end]])

Returns a Boolean value indicating whether `s[start:end]` contains only whitespace characters. `start` and `end` are optional, and will default to `0` and `len(s)`, respectively.

mx.TextTools.isascii(s)

Returns a Boolean value indicating whether `s` contains only ASCII characters.

mx.TextTools.join(joinlist [,sep="" [,start [,end]]])

Return a string composed of slices from other strings. `joinlist` is a sequence of tuples of the form `(s, start, end, ...)` each indicating the source string and offsets for the utilized slice. Negative offsets do not behave like Python slice offsets and should not be used. If a `joinlist` item tuple contains extra entries, they are ignored, but are permissible.

If the optional argument `sep` is specified, a delimiter between each joined slice is added. If `start` and `end` are specified, only `joinlist[start:end]` is utilized in the joining.

```
>>> from mx.TextTools import join
>>> s = 'Spam and eggs for breakfast'
>>> t = 'This and that for lunch'
>>> jl = [(s, 0, 4), (s, 9, 13), (t, 0, 4), (t, 9, 13)]
>>> join(jl, '/', 1, 4)
'/eggs/This/that'
```

SEE ALSO: string.join() *137*;

mx.TextTools.lower(s)

Return a string with any uppercase letters converted to lowercase. Functionally identical to *string.lower()*, but much faster.

SEE ALSO: string.lower() *138*; mx.TextTools.upper() *316*;

mx.TextTools.prefix(s, prefixes [,start [,stop [,translate]]])

Return the first prefix in the tuple `prefixes` that matches the end of `s`. If `start` and `end` are specified, only operate on the slice `s[start:end]`. Return `None` if no prefix matches.

If a `translate` argument is given, the searched string is translated during the search. This is equivalent to transforming the string with *string.translate()* prior to searching it.

```
>>> from mx.TextTools import prefix
>>> prefix('spam and eggs', ('spam','and','eggs'))
'spam'
```

SEE ALSO: mx.TextTools.suffix() *316*;

mx.TextTools.multireplace(s ,replacements [,start [,stop]])

Replace multiple nonoverlapping slices in `s` with string values. `replacements` must be list of tuples of the form `(new, left, right)`. Indexing is always relative to `s`, even if an earlier replacement changes the length of the result. If `start` and `end` are specified, only operate on the slice `s[start:end]`.

```
>>> from mx.TextTools import findall, multireplace
>>> s = 'spam, bacon, sausage, and spam'
>>> repls = [('X',l,r) for l,r in findall(s, 'spam')]
>>> multireplace(s, repls)
'X, bacon, sausage, and X'
>>> repls
[('X', 0, 4), ('X', 26, 30)]
```

mx.TextTools.replace(s, old, new [,start [,stop]])

Return a string where the pattern matched by search object `old` is replaced by string `new`. If `start` and `end` are specified, only operate on the slice `s[start:end]`. This function is much faster than *`string.replace()`*, since a search object is used in the search aspect.

```
>>> from mx.TextTools import replace, BMS
>>> s = 'spam, bacon, sausage, and spam'
>>> spam = BMS('spam')
>>> replace(s, spam, 'eggs')
'eggs, bacon, sausage, and eggs'
>>> replace(s, spam, 'eggs', 5)
' bacon, sausage, and eggs'
```

SEE ALSO: string.replace() *139*; mx.TextTools.BMS *307*;

mx.TextTools.setfind(s, set [,start [,end]])

Find the first occurence of any character in `set`. If `start` is specified, look only in `s[start:]`; if `end` is specified, look only in `s[start:end]`. The argument `set` must be a set.

```
>>> from mx.TextTools import *
>>> s = 'spam and eggs'
>>> vowel = set('aeiou')
>>> setfind(s, vowel)
2
>>> setfind(s, vowel, 7, 10)
9
```

SEE ALSO: mx.TextTools.set() *310*;

mx.TextTools.setsplit(s, set [,start [,stop]])

Split `s` into substrings divided at any characters in `set`. If `start` is specified, create a list of substrings of `s[start:]`; if `end` is specified, use `s[start:end]`. The argument `set` must be a set.

SEE ALSO: string.split() *142*; mx.TextTools.set() *310*; mx.TextTools.setsplitx() *315*;

mx.TextTools.setsplitx(text,set[,start=0,stop=len(text)])

Split `s` into substrings divided at any characters in `set`. Include the split characters in the returned list. Adjacent characters in `set` are returned in the same list element. If `start` is specified, create a list of substrings of `s[start:]`; if `end` is specified, use `s[start:end]`. The argument `set` must be a set.

```
>>> s = 'do you like spam'
>>> setsplit(s, vowel)
['d', ' y', ' l', 'k', ' sp', 'm']
>>> setsplitx(s, vowel)
['d', 'o', ' y', 'ou', ' l', 'i', 'k', 'e', ' sp', 'a', 'm']
```

SEE ALSO: string.split() *142*; mx.TextTools.set() *310*; mx.TextTools.setsplit() *314*;

mx.TextTools.splitat(s, char, [n=1 [,start [end]]])

Return a 2-element tuple that divides `s` around the n'th occurence of `char`. If `start` and `end` are specified, only operate on the slice `s[start:end]`.

```
>>> from mx.TextTools import splitat
>>> s = 'spam, bacon, sausage, and spam'
>>> splitat(s, 'a', 3)
('spam, bacon, s', 'usage, and spam')
>>> splitat(s, 'a', 3, 5, 20)
(' bacon, saus', 'ge')
```

mx.TextTools.splitlines(s)

Return a list of lines in `s`. Line-ending combinations for Mac, PC, and Unix platforms are recognized in any combination, which makes this function more portable than is `string.split(s,"\n")` or *`FILE.readlines()`*.

SEE ALSO: string.split() *142*; FILE.readlines() *17*; mx.TextTools.setsplit() *314*; mx.TextTools.countlines() *311*;

mx.TextTools.splitwords(s)

Return a list of whitespace-separated words in `s`. Equivalent to `string.split(s)`.

SEE ALSO: string.split() *142*;

mx.TextTools.str2hex(s)

Returns a hexadecimal representation of a string. For Python 2.0+, this is equivalent to `s.encode("hex")`.

SEE ALSO: "".encode() *188*; mx.TextTools.hex2str() *312*;

mx.TextTools.suffix(s, suffixes [,start [,stop [,translate]]])

Return the first suffix in the tuple `suffixes` that matches the end of `s`. If `start` and `end` are specified, only operate on the slice `s[start:end]`. Return `None` if no suffix matches.

If a `translate` argument is given, the searched string is translated during the search. This is equivalent to transforming the string with *`string.translate()`* prior to searching it.

```
>>> from mx.TextTools import suffix
>>> suffix('spam and eggs', ('spam','and','eggs'))
'eggs'
```

SEE ALSO: mx.TextTools.prefix() *313*;

mx.TextTools.upper(s)

Return a string with any lowercase letters converted to uppercase. Functionally identical to *`string.upper()`*, but much faster.

SEE ALSO: string.upper() *146*; mx.TextTools.lower() *313*;

4.3.3 High-Level EBNF Parsing

SimpleParse ◇ A Parser Generator for mx.TextTools

SimpleParse is an interesting tool. To use this module, you need to have the *mx.TextTools* module installed. While there is nothing you can do with *SimpleParse* that cannot be done with *mx.TextTools* by itself, *SimpleParse* is often much easier to work with. There exist other modules to provide higher-level APIs for *mx.TextTools*; I find *SimpleParse* to be the most useful of these, and the only one that this book will present. The examples in this section were written against *SimpleParse* version 1.0, but the documentation is updated to include new features of 2.0. Version 2.0 is fully backward compatible with existing *SimpleParse* code.

SimpleParse substitutes an EBNF-style grammar for the low-level state matching language of *mx.TextTools* tag tables. Or more accurately, *SimpleParse* is a tool for generating tag tables based on friendlier and higher-level EBNF grammars. In principle, *SimpleParse* lets you access and modify tag tables before passing them to *`mx.TextTools.tag()`*. But in practice, you usually want to stick wholly with *SimpleParse*'s EBNF variant when your processing is amenable to a grammatical description of the text format.

An application based on *SimpleParse* has two main aspects. The first aspect is the grammar that defines the structure of a processed text. The second aspect is the traversal and use of a generated *mx.TextTools* taglist. *SimpleParse* 2.0 adds facilities for the traversal aspect, but taglists present a data structure that is quite easy to work with in any case. The tree-walking tools in *SimpleParse* 2.0 are not covered here, but the examples given in the discussion of *mx.TextTools* illustrate such traversal.

Example: Marking up smart ASCII (Redux)

Elsewhere in this book, applications to process the smart ASCII format are also presented. Appendix D lists the `Txt2Html` utility, which uses a combination of a state machine for parsing paragraphs and regular expressions for identifying inline markup. A functionally similar example was given in the discussion of *mx.TextTools*, where a complex and compound tag table was developed to recognize inline markup elements. Using *SimpleParse* and an EBNF grammar is yet another way to perform the same sort of processing. Comparing the several styles will highlight a number of advantages that *SimpleParse* has—its grammars are clear and concise, and applications built around it can be extremely fast.

The application `simpleTypography.py` is quite simple; most of the work of programming it lies in creating a grammar to describe smart ASCII. EBNF grammars are almost self-explanatory to read, but designing one *does* require a bit of thought and testing:

typography.def

```
para           := (plain / markup)+
plain          := (word / whitespace / punctuation)+
<whitespace>   := [ \t\r\n]+
<alphanums>    := [a-zA-Z0-9]+
<word>         := alphanums, (wordpunct, alphanums)*, contraction?
<wordpunct>    := [-_]
<contraction>  := "’", (’am’/’clock’/’d’/’ll’/’m’/’re’/’s’/’t’/’ve’)
markup         := emph / strong / module / code / title
emph           := ’-’, plain, ’-’
strong         := ’*’, plain, ’*’
module         := ’[’, plain, ’]’
code           := "’", plain, "’"
title          := ’_’, plain, ’_’
<punctuation>  := (safepunct / mdash)
<mdash>        := ’--’
<safepunct>    := [!@#$%^&()+=|\{}:;<>,.?/"]
```

This grammar is almost exactly the way you would describe the smart ASCII language verbally, which is a nice sort of clarity. A paragraph consist of some plaintext and some marked-up text. Plaintext consists of some collection of words, whitespace, and punctuation. Marked-up text might be emphasized, or strongly emphasized, or module names, and so on. Strongly emphasized text is surrounded by asterisks. And so on. A couple of features like just what a “word” really is, or just what a contraction can end with, take a bit of thought, but the syntax of EBNF doesn’t get in the way.

Notice that some declarations have their left side surrounded in angle brackets. Those productions will not be written to the taglist—this is the same as using `None` as a `tagobj` in an *mx.Texttools* tag table. Of course, if a production is not written to the taglist, then its children cannot be, either. By omitting some productions from the resultant taglist, a simpler data structure is produced (with only those elements that interest us).

In contrast to the grammar above, the same sort of rules can be described even more tersely using regular expressions. This is what the `Txt2Html` version of the smart ASCII markup program does. But this terseness is much harder to write and harder still to tweak later. The *re* code below expresses largely (but not precisely) the same set of rules:

Python regexes for smart ASCII markup

```
# [module] names
re_mods =   r"""([\(\s'/">]|^)\[(.*?)\]([<\s\.\),:;'"?!/-])"""
# *strongly emphasize* words
re_strong = r"""([\(\s'/"]|^)\*(.*?)\*([\s\.\),:;'"?!/-])"""
# -emphasize- words
re_emph =   r"""([\(\s'/"]|^)-(.*?)-([\s\.\),:;'"?!/])"""
# _Book Title_ citations
re_title =  r"""([\(\s'/"]|^)_(.*?)_([\s\.\),:;'"?!/-])"""
# 'Function()' names
re_funcs =  r"""([\(\s/"]|^)'(.*?)'([\s\.\),:;"?!/-])"""
```

If you discover or invent some slightly new variant of the language, it is *a lot* easier to play with the EBNF grammar than with those regular expressions. Moreover, using *SimpleParse*—and therefore *mx.TextTools*—will generally be even faster in performing the manipulations of the patterns.

GENERATING AND USING A TAGLIST

For `simpleTypography.py`, I put the actual grammar in a separate file. For most purposes, this is a good organization to use. Changing the grammar is usually a different sort of task than changing the application logic, and the files reflect this. But the grammar is just read as a string, so in principle you could include it in the main application (or even dynamically generate it in some way).

Let us look at the entire—compact—tagging application:

simpleTypography.py

```
from sys import stdin, stdout, stderr
from simpleparse import generator
from mx.TextTools import TextTools
from typo_html import codes
from pprint import pprint

src = stdin.read()
decl = open('typography.def').read()
parser = generator.buildParser(decl).parserbyname('para')
taglist = TextTools.tag(src, parser)
pprint(taglist, stderr)

for tag, beg, end, parts in taglist[1]:
    if tag == 'plain':
        stdout.write(src[beg:end])
    elif tag == 'markup':
        markup = parts[0]
        mtag, mbeg, mend = markup[:3]
        start, stop = codes.get(mtag, ('<!-- unknown -->',
                                       '<!-- /unknown -->'))
        stdout.write(start + src[mbeg+1:mend-1] + stop)
    else:
        raise TypeError, "Top level tagging should be plain/markup"
```

With version 2.0 of *SimpleParse*, you may use a somewhat more convenient API to create a taglist:

```
from simpleparse.parser import Parser
parser = Parser(open('typography.def').read(), 'para')
taglist = parser.parse(src)
```

Here is what it does. First read in the grammar and create an *mx.TextTools* parser from the grammar. The generated parser is similar to the tag table that is found in the hand-written `mxTypography.py` module discussed earlier (but without the human-friendly comments and structure). Next, apply the tag table/parser to the input source to create a taglist. Finally, loop through the taglist, and emit some new marked-up text. The loop could, of course, do anything else desired with each production encountered.

For the particular grammar used for smart ASCII, everything in the source text is expected to fall into either a "plain" production or a "markup" production. Therefore, it suffices to loop across a single level in the taglist (except when we look exactly one level lower for the specific markup production, such as "title"). But a more free-form grammar—such as occurs for most programming languages—could easily recursively descend into the taglist and look for production names at every level. For example, if the grammar were to allow nested markup codes, this recursive style would probably be used. Readers might enjoy the exercise of figuring out how to adjust the grammar (hint: Remember that productions are allowed to be mutually recursive).

The particular markup codes that go to the output live in yet another file for organizational, not essential, reasons. A little trick of using a dictionary as a `switch` statement is used here (although the `otherwise` case remains too narrow in the example). The idea behind this organization is that we might in the future want to create multiple "output format" files for, say, HTML, DocBook, LaTeX, or others. The particular markup file used for the example just looks like:

typo_html.py

```
codes = \
{ 'emph'    : ('<em>', '</em>'),
  'strong'  : ('<strong>', '</strong>'),
  'module'  : ('<em><code>', '</code></em>'),
  'code'    : ('<code>', '</code>'),
  'title'   : ('<cite>', '</cite>'),
}
```

Extending this to other output formats is straightforward.

THE TAGLIST AND THE OUTPUT

The *tag table* generated from the grammar in `typography.def` is surprisingly complicated and includes numerous recursions. Only the exceptionally brave of heart will want to attempt manual—let alone automated—modification of tag tables created by *SimpleParse*. Fortunately, an average user need not even look at these tags, but simply *use* them, as is done with `parser` in `simpleTypography.py`.

The *taglist* produced by applying a grammar, in contrast, can be remarkably simple. Here is a run of `simpleTypography.py` against a small input file:

```
% python simpleTypography.py < p.txt > p.html
(1,
 [('plain', 0, 15, []),
  ('markup', 15, 27, [('emph', 15, 27, [('plain', 16, 26, [])])]),
  ('plain', 27, 42, []),
  ('markup', 42, 51, [('module', 42, 51, [('plain', 43, 50, [])])]),
  ('plain', 51, 55, []),
  ('markup', 55, 70, [('code', 55, 70, [('plain', 56, 69, [])])]),
  ('plain', 70, 90, []),
  ('markup', 90, 96, [('strong', 90, 96, [('plain', 91, 95, [])])]),
  ('plain', 96, 132, []),
  ('markup', 132, 145, [('title', 132, 145, [('plain',133,144,[])])]),
  ('plain', 145, 174, [])],
 174)
```

Most productions that were satisfied are not written into the taglist, because they are not needed for the application. You can control this aspect simply by defining productions with or without angle braces on the left side of their declaration. The output looks like you would expect:

```
% cat p.txt
Some words are -in italics-, others
name [modules] or 'command lines'.
Still others are *bold* -- that's how
it goes. Maybe some _book titles_.
And some in-fixed dashes.
% cat p.html
Some words are <em>in italics</em>, others
name <em><code>modules</code></em> or <code>command lines</code>.
Still others are <strong>bold</strong> -- that's how
it goes. Maybe some <cite>book titles</cite>.
And some in-fixed dashes.
```

○ · · ○ · · ○ · · ○ · · ○ · · ○ · · ○ · · ○ · · ○ · · ○ · · ○ · · ○ · · ○ · · ○

GRAMMAR

The language of *SimpleParse* grammars is itself defined using a *SimpleParse* EBNF-style grammar. In principle, you could refine the language *SimpleParse* uses by changing the variable `declaration` in `bootstrap.py`, or `simpleparsegrammar.py` in recent versions. For example, extended regular expressions, W3C XML Schemas, and some EBNF variants allow integer occurrence quantification. To specify that three to seven `foo` tokens occur, you could use the following declaration in *SimpleParse*:

```
foos := foo, foo, foo, foo?, foo?, foo?, foo?
```

Hypothetically, it might be more elegant to write something like:

```
foos := foo{3,7}
```

In practice, only someone developing a custom/enhanced parsing module would have any reason to fiddle quite so deeply; "normal" programmers should use the particular EBNF variant defined by default. Nonetheless, taking a look at `simpleparse/bootstrap.py` can be illustrative in understanding the module.

DECLARATION PATTERNS

A *SimpleParse* grammar consists of a set of one or more declarations. Each declaration generally occurs on a line by itself; within a line, horizontal whitespace may be used as desired to improve readability. A common strategy is to align the right sides of declarations, but any other use of internal whitespace is acceptable. A declaration contains a term, followed by the assignment symbol ":=", followed by a definition. An end-of-line comment may be added to a declaration, following an unquoted "#" (just as in Python).

In contrast to most imperative-style programming, the declarations within a grammar may occur in any order. When a parser generator's `.parserbyname()` method is called, the "top level" of the grammar is given as an argument. The documented API for *SimpleParse* uses a call of the form:

```
from simpleparse import generator
parser = generator.buildParser(decl).parserbyname('toplevel')
from mx.TextTools import TextTools
taglist = TextTools.tag(src, parser)
```

Under *SimpleParse* 2.0, you may simplify this to:

```
from simpleparse.parser import Parser
parser = Parser(decl,'toplevel')
taglist = parser.parse(src)
```

A left side term may be surrounded by angle brackets ("<", ">") to prevent that production from being written into a taglist produced by *`mx.TextTools.tag()`*. This is called an "unreported" production. Other than in relation to the final taglist, an unreported production acts just like a reported one. Either type of term may be used on the right sides of other productions in the same manner (without angle brackets when occurring on the right side).

In *SimpleParse* 2.0 you may also use reversed angle brackets to report the children of a production, but not the production itself. As with the standard angle brackets, the production functions normally in matching inputs; it differs only in produced taglist. For example:

```
PRODUCTIONS               TAGLIST
--------------------------------------------------------
a   := (b,c)              ('a', l, r, [
b   := (d,e)                  ('b', l, r, [...]),
c   := (f,g)                  ('c', l, r, [...]) ] )
--------------------------------------------------------
a   := (b,c)              ('a', l, r, [
<b> := (d,e)                  # no b, and no children
c   := (f,g)                  ('c', l, r, [...]) ] )
--------------------------------------------------------
# Only in 2.0+            ('a', l, r, [
a   := (b,c)                  # no b, but raise children
>b< := (d,e)                  ('d', l, r, [...]),
c   := (f,g)                  ('e', l, r, [...]),
                              ('c', l, r, [...]) ] )
--------------------------------------------------------
```

The remainder of the documentation of the *SimpleParse* module covers elements that may occur on the right sides of declarations. In addition to the elements listed, a term from another production may occur anywhere any element may. Terms may thus stand in mutually recursive relations to one another.

LITERALS

Literal string

A string enclosed in single quotes matches the exact string quoted. Python escaping may be used for the characters `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`, and octal escapes of one to three digits may used. To include a literal backslash, it should be escaped as \\.

```
foo := "bar"
```

Character class: "[", "]"

Specify a set of characters that may occur at a position. The list of allowable characters may be enumerated with no delimiter. A range of characters may be indicated with a dash ("-"). Multiple ranges are allowed within a class.

To include a "]" character in a character class, make it the first character. Similarly, a literal "-" character must be either the first (after the optional "]" character) or the last character.

```
varchar := [a-zA-Z_0-9]
```

QUANTIFIERS

Universal quantifier: "*"

Match zero or more occurrences of the preceding expression. Quantification has a higher precedence than alternation or sequencing; grouping may be used to clarify quantification scope as well.

```
any_Xs     := "X"*
any_digits := [0-9]*
```

Existential quantifier: "+"

Match one or more occurrences of the preceding expression. Quantification has a higher precedence than alternation or sequencing; grouping may be used to clarify quantification scope as well.

```
some_Xs     := "X"+
some_digits := [0-9]+
```

Potentiality quantifier: "?"

Match at most one occurrence of the preceding expression. Quantification has a higher precedence than alternation or sequencing; grouping may be used to clarify quantification scope as well.

```
maybe_Xs     := "X"?
maybe_digits := [0-9]?
```

Lookahead quantifier: "?"

In *SimpleParse* 2.0+, you may place a question mark *before* a pattern to assert that it occurs, but should not actually claim the pattern. As with regular expressions, you can create either positive or negative lookahead assertions.

```
next_is_Xs         := ?"X"
next_is_not_digits := ?-[0-9]
```

Error on Failure: "!"

In *SimpleParse* 2.0+, you may cause a descriptive exception to be raised when a production does not match, rather than merely stopping parsing at that point.

```
require_Xs   := "X"!
require_code := ([A-Z]+, [0-9])!
contraction := "'", ('clock'/'d'/'ll'/'m'/'re'/'s'/'t'/'ve')!
```

For example, modifying the `contraction` production from the prior discussion could require that every apostrophe is followed by an ending. Since this doesn't hold, you might see an exception like:

```
% python typo2.py < p.txt
Traceback (most recent call last):
[...]
simpleparse.error.ParserSyntaxError:  ParserSyntaxError:
Failed parsing production "contraction" @pos 84 (~line 1:29).
Expected syntax: ('clock'/'d'/'ll'/'m'/'re'/'s'/'t'/'ve')
Got text: 'command lines'.  Still others are *bold*
```

STRUCTURES

Alternation operator: "/"

Match the first pattern possible from several alternatives. This operator allows any of a list of patterns to match. Some EBNF-style parsers will match the *longest* possible pattern, but *SimpleParse* more simply matches the *first* possible pattern. For example:

```
>>> from mx.TextTools import tag
>>> from simpleparse import generator
>>> decl = '''
... short := "foo", " "*
... long  := "foobar", " "*
... sl    := (short / long)*
... ls    := (long / short)*
... '''
>>> parser = generator.buildParser(decl).parserbyname('sl')
>>> tag('foo foobar foo bar', parser)[1]
[('short', 0, 4, []), ('short', 4, 7, [])]
>>> parser = generator.buildParser(decl).parserbyname('ls')
>>> tag('foo foobar foo bar', parser)[1]
[('short', 0, 4, []), ('long', 4, 11, []), ('short', 11, 15, [])]
```

Sequence operator: ","

Match the first pattern followed by the second pattern (followed by the third pattern, if present, ...). Whenever a definition needs several elements in a specific order, the comma sequence operator is used.

```
term := someterm, [0-9]*, "X"+, (otherterm, stillother)?
```

Negation operator: "-"

Match anything that the next pattern *does not* match. The pattern negated can be either a simple term or a compound expression.

```
nonletters   := -[a-zA-Z]
nonfoo       := -foo
notfoobarbaz := -(foo, bar, baz)
```

An expression modified by the negation operator is very similar conceptually to a regular expression with a negative lookahead assertion. For example:

```
>>> from mx.TextTools import tag
>>> from simpleparse import generator
>>> decl = '''not_initfoo := [ \t]*, -"foo", [a-zA-Z ]+'''
>>> p = generator.buildParser(decl).parserbyname('not_initfoo')
>>> tag('  foobar and baz', p)     # no match
(0, [], 0)
>>> tag('  bar, foo and baz', p)   # match on part
(1, [], 5)
>>> tag('  bar foo and baz', p)    # match on all
(1, [], 17)
```

Grouping operators: "(", ")"

Parentheses surrounding any pattern turn that pattern into an expression (possibly within a larger expression). Quantifiers and operators refer to the immediately adjacent expression, if one is defined, otherwise to the adjacent literal string, character class, or term.

```
>>> from mx.TextTools import tag
>>> from simpleparse import generator
>>> decl = '''
... foo      := "foo"
... bar      := "bar"
... foo_bars := foo, bar+
... foobars  := (foo, bar)+
... '''
>>> p1 = generator.buildParser(decl).parserbyname('foobars')
>>> p2 = generator.buildParser(decl).parserbyname('foo_bars')
>>> tag('foobarfoobar', p1)
(1, [('foo', 0, 3, []), ('bar', 3, 6, []),
     ('foo', 6, 9, []), ('bar', 9, 12, [])], 12)
>>> tag('foobarfoobar', p2)
(1, [('foo', 0, 3, []), ('bar', 3, 6, [])], 6)
>>> tag('foobarbarbar', p1)
(1, [('foo', 0, 3, []), ('bar', 3, 6, [])], 6)
>>> tag('foobarbarbar', p2)
(1, [('foo', 0, 3, []), ('bar', 3, 6, []),
     ('bar', 6, 9, []), ('bar', 9, 12, [])], 12)
```

USEFUL PRODUCTIONS

In version 2.0+, *SimpleParse* includes a number of useful productions that may be included in your grammars. See the examples and documentation that accompany *SimpleParse* for details on the many included productions and their usage.

The included productions, at the time of this writing, fall into the categories below:

simpleparse.common.calendar_names

Locale-specific names of months, and days of the week, including abbreviated forms.

simpleparse.common.chartypes

Locale-specific categories of characters, such as digits, uppercase, octdigits, punctuation, locale_decimal_point, and so on.

simpleparse.common.comments

Productions to match comments in a variety of programming languages, such as hash (`#`) end-of-line comments (Python, Bash, Perl, etc.); C paired comments (`/* comment */`); and others.

simpleparse.common.iso_date

Productions for strictly conformant ISO date and time formats.

simpleparse.common.iso_date_loose

Productions for ISO date and time formats with some leeway as to common variants in formatting.

simpleparse.common.numbers

Productions for common numeric formats, such as integers, floats, hex numbers, binary numbers, and so on.

simpleparse.common.phonetics

Productions to match phonetically spelled words. Currently, the US military style of "alpha, bravo, charlie, ..." spelling is the only style supported (with some leeway in word spellings).

simpleparse.common.strings

Productions to match quoted strings as used in various programming languages.

simpleparse.common.timezone_names

Productions to match descriptions of timezones, as you might find in email headers or other data/time fields.

GOTCHAS

There are a couple of problems that can easily arise in constructed *SimpleParse* grammars. If you are having problems in your application, keep a careful eye out for these issues:

1. Bad recursion. You might fairly naturally construct a pattern of the form:

   ```
   a := b, a?
   ```

   Unfortunately, if a long string of `b` rules are matched, the repeated recognition can either exceed the C-stack's recursion limit, or consume inordinate amounts of memory to construct nested tuples. Use an alternate pattern like:

   ```
   a := b+
   ```

   This will grab all the `b` productions in one tuple instead (you could separately parse out each `b` if necessary).

2. Quantified potentiality. That is a mouthful; consider patterns like:

   ```
   a := (b? / c)*
   x := (y?, z?)+
   ```

The first alternate `b?` in the first—and both `y?` and `z?` in the second—are happy to match zero characters (if a `b` or `y` or `z` do not occur at the current position). When you match "as many as possible" zero-width patterns, you get into an infinite loop. Unfortunately, the pattern is not always simple; it might not be `b` that is qualified as potential, but rather `b` productions (or the productions *in* `b` productions, etc.).

3. No backtracking. Based on working with regular expression, you might expect *SimpleParse* productions to use backtracking. They do not. For example:

```
a := ((b/c)*, b)
```

If this were a regular expression, it would match a string of `b` productions, then back up one to match the final `b`. As a *SimpleParse* production, this definition can never match. If any `b` productions occur, they will be claimed by `(b/c)*`, leaving nothing for the final `b` to grab.

4.3.4 High-Level Programmatic Parsing

PLY ◇ Python Lex-Yacc

One module that I considered covering to round out this chapter is John Aycock's *Spark* module. This module is both widely used in the Python community and extremely powerful. However, I believe that the audience of this book is better served by working with David Beazley's *PLY* module than with the older *Spark* module.

In the documentation accompanying *PLY*, Beazley consciously acknowledges the influence of *Spark* on his design and development. While the *PLY* module is far from being a clone of *Spark*—the APIs are significantly different—there is a very similar *feeling* to working with each module. Both modules require a very different style of programming and style of thinking than do *mx.TextTools*, *SimpleParse*, or the state machines discussed earlier in this chapter. In particular, both *PLY* and *Spark* make heavy use of Python introspection to create underlying state machines out of specially named variables and functions.

Within an overall similarity, *PLY* has two main advantages over *Spark* in a text processing context. The first, and probably greatest, advantage *PLY* has is its far greater speed. Although *PLY* has implemented some rather clever optimizations—such as preconstruction of state tables for repeated runs—the main speed difference lies in the fact that *PLY* uses a far faster, albeit slightly less powerful, parsing algorithm. For text processing applications (as opposed to compiler development), *PLY*'s LR parsing is plenty powerful for almost any requirement.

A second advantage *PLY* has over every other Python parsing library that I am aware of is a flexible and fine-grained error reporting and error correction facility. Again, in a text processing context, this is particularly important.

For compiling a programming language, it is generally reasonable to allow compilation to fail in the case of even small errors. But for processing a text file full of data fields and structures, you usually want to be somewhat tolerant of minor formatting errors; getting as much data as possible from a text automatically is frequently the preferred approach. *PLY* does an excellent job of handling "allowable" error conditions gracefully.

PLY consists of two modules: a lexer/tokenizer named `lex.py`, and a parser named `yacc.py`. The choice of names is taken from the popular C-oriented tools `lex` and `yacc`, and the behavior is correspondingly similar. Parsing with *PLY* usually consists of the two steps that were discussed at the beginning of this chapter: (1) Divide the input string into a set of nonoverlapping tokens using `lex.py`. (2) Generate a parse tree from the series of tokens using `yacc.py`.

When processing text with *PLY*, it is possible to attach "action code" to any lexing or parsing event. Depending on application requirements, this is potentially much more powerful than *SimpleParse*. For example, each time a specific token is encountered during lexing, you can modify the stored token according to whatever rule you wish, or even trigger an entirely different application action. Likewise, during parsing, each time a node of a parse tree is constructed, the node can be modified and/or other actions can be taken. In contrast, *SimpleParse* simply delivers a completed parse tree (called a "taglist") that must be traversed separately. However, while *SimpleParse* does not provide the fine-tunable event control that *PLY* does, *SimpleParse* offers a higher-level and cleaner grammar language—the choice between the two modules is full of pros and cons.

Example: Marking up smart ASCII (yet again)

This chapter has returned several times to applications for processing smart ASCII: a state machine in Appendix D; a functionally similar example using *mx.TextTools*; an EBNF grammar with *SimpleParse*. This email-like markup format is not in itself all that important, but it presents just enough complications to make for a good comparison between programming techniques and libraries. In many ways, an application using *PLY* is similar to the *SimpleParse* version above—both use grammars and parsing strategies.

GENERATING A TOKEN LIST

The first step in most *PLY* applications is the creation of a token stream. Tokens are identified by a series of regular expressions attached to special pattern names of the form `t_RULENAME`. By convention, the *PLY* token types are in all caps. In the simple case, a regular expression string is merely assigned to a variable. If action code is desired when a token is recognized, the rule name is defined as a function, with the regular expression string as its docstring; passed to the function is a `LexToken` object (with attributes `.value`, `.type`, and `.lineno`), which may be modified and returned. The pattern is clear in practice:

wordscanner.py

```
# List of token names.  This is always required.
tokens = [ 'ALPHANUMS','SAFEPUNCT','BRACKET','ASTERISK',
          'UNDERSCORE','APOSTROPHE','DASH' ]

# Regular expression rules for simple tokens
t_ALPHANUMS     = r"[a-zA-Z0-9]+"
t_SAFEPUNCT     = r'[!@#$%^&()+=|\{}:;<>,.?/"]+'
t_BRACKET       = r'[][]'
t_ASTERISK      = r'[*]'
t_UNDERSCORE    = r'_'
t_APOSTROPHE    = r"'"
t_DASH          = r'-'

# Regular expression rules with action code
def t_newline(t):
    r"\n+"
    t.lineno += len(t.value)

# Special case (faster) ignored characters
t_ignore = " \t\r"

# Error handling rule
def t_error(t):
    sys.stderr.write("Illegal character '%s' (%s)\n"
                     % (t.value[0], t.lineno))
    t.skip(1)

import lex, sys
def stdin2tokens():
    lex.input(sys.stdin.read())     # Give the lexer some input
    toklst = []                     # Tokenize
    while 1:
        t = lex.token()
        if not t: break   # No more input
        toklst.append(t)
    return toklst

if __name__=='__main__':
    lex.lex()                       # Build the lexer
    for t in stdin2tokens():
        print '%s<%s>' % (t.value.ljust(15), t.type)
```

You are required to list the token types you wish to recognize, using the `tokens` variable. Each such token, and any special patterns that are not returned as tokens, is defined either as a variable or as a function. After that, you just initialize the lexer, read a string, and pull tokens off sequentially. Let us look at some results:

```
% cat p.txt
-Itals-, [modname]--let's add ~ underscored var_name.
% python wordscanner.py < p.txt
Illegal character '~' (1)
-               <DASH>
Itals           <ALPHANUMS>
-               <DASH>
,               <SAFEPUNCT>
[               <BRACKET>
modname         <ALPHANUMS>
]               <BRACKET>
-               <DASH>
-               <DASH>
let             <ALPHANUMS>
'               <APOSTROPHE>
s               <ALPHANUMS>
add             <ALPHANUMS>
underscored     <ALPHANUMS>
var             <ALPHANUMS>
_               <UNDERSCORE>
name            <ALPHANUMS>
.               <SAFEPUNCT>
```

The output illustrates several features. For one thing, we have successfully tagged each nondiscarded substring as constituting some token type. Notice also that the unrecognized tilde character is handled gracefully by being omitted from the token list—you could do something different if desired, of course. Whitespace is discarded as insignificant by this tokenizer—the special `t_ignore` variable quickly ignores a set of characters, and the `t_newline()` function contains some extra code to maintain the line number during processing.

The simple tokenizer above has some problems, however. Dashes can be used either in an m-dash or to mark italicized phrases; apostrophes can be either part of a contraction or a marker for a function name; underscores can occur both to mark titles and within variable names. Readers who have used *Spark* will know of its capability to enhance a lexer or parser by inheritance; *PLY* cannot do that, but it can utilize Python namespaces to achieve almost exactly the same effect:

wordplusscanner.py

```
"Enhanced word/markup tokenization"
from wordscanner import *
tokens.extend(['CONTRACTION','MDASH','WORDPUNCT'])
t_CONTRACTION   = r"(?<=[a-zA-Z])'(am|clock|d|ll|m|re|s|t|ve)"
t_WORDPUNCT     = r'(?<=[a-zA-Z0-9])[-_](?=[a-zA-Z0-9])'
def t_MDASH(t): # Use HTML style mdash
    r'--'
    t.value = '&mdash;'
    return t

if __name__=='__main__':
    lex.lex()                        # Build the lexer
    for t in stdin2tokens():
        print '%s<%s>' % (t.value.ljust(15), t.type)
```

Although the tokenization produced by `wordscanner.py` would work with the right choice of grammar rules, producing more specific tokens allows us to simplify the grammar accordingly. In the case of `t_MDASH()`, `wordplusscanner.py` also modifies the token itself as part of recognition:

```
% python wordplusscanner.py < p.txt
Illegal character '~' (1)
-              <DASH>
Itals          <ALPHANUMS>
-              <DASH>
,              <SAFEPUNCT>
[              <BRACKET>
modname        <ALPHANUMS>
]              <BRACKET>
&mdash;        <MDASH>
let            <ALPHANUMS>
's             <CONTRACTION>
add            <ALPHANUMS>
underscored    <ALPHANUMS>
var            <ALPHANUMS>
_              <WORDPUNCT>
name           <ALPHANUMS>
.              <SAFEPUNCT>
```

Parsing a token list

A parser in *PLY* is defined in almost the same manner as a tokenizer. A collection of specially named functions of the form `p_rulename()` are defined, each containing an EBNF-style pattern to match (or a disjunction of several such patterns). These functions receive as argument a `YaccSlice` object, which is list-like in assigning each component of the EBNF declaration to an indexed position.

The code within each function should assign a useful value to `t[0]`, derived in some way from `t[1:]`. If you would like to create a parse tree out of the input source, you can define a `Node` class of some sort and assign each right-hand rule or token as a subnode/leaf of that node; for example:

```
def p_rulename(t):
    'rulename : somerule SOMETOKEN otherrule'
    #   ^          ^          ^          ^
    #  t[0]       t[1]       t[2]       t[3]
    t[0] = Node('rulename', t[1:])
```

Defining an appropriate Node class is left as an exercise. With this approach, the final result would be a traversable tree structure.

It is fairly simple to create a set of rules to combine the fairly smart token stream produced by wordplusscanner.py. In the sample application, a simpler structure than a parse tree is built. markupbuilder.py simply creates a list of matched patterns, interspersed with added markup codes. Other data structures are possible too, and/or you could simply take some action each time a rule is matched (e.g., write to STDOUT).

markupbuilder.py

```
import yacc
from wordplusscanner import *

def p_para(t):
    '''para : para plain
            | para emph
            | para strong
            | para module
            | para code
            | para title
            | plain
            | emph
            | strong
            | module
            | code
            | title '''
    try:    t[0] = t[1] + t[2]
    except: t[0] = t[1]

def p_plain(t):
    '''plain : ALPHANUMS
             | CONTRACTION
             | SAFEPUNCT
             | MDASH
             | WORDPUNCT
             | plain plain '''
    try:    t[0] = t[1] + t[2]
    except: t[0] = [t[1]]

def p_emph(t):
    '''emph : DASH plain DASH'''
    t[0] = ['<i>'] + t[2] + ['</i>']

def p_strong(t):
    '''strong : ASTERISK plain ASTERISK'''
    t[0] = ['<b>'] + t[2] + ['</b>']
```

```
def p_module(t):
    '''module : BRACKET plain BRACKET'''
    t[0] = ['<em><tt>'] + t[2] + ['</tt></em>']

def p_code(t):
    '''code : APOSTROPHE plain APOSTROPHE'''
    t[0] = ['<code>'] + t[2] + ['</code>']

def p_title(t):
    '''title : UNDERSCORE plain UNDERSCORE'''
    t[0] = ['<cite>'] + t[2] + ['</cite>']

def p_error(t):
    sys.stderr.write('Syntax error at "%s" (%s)\n'
                     % (t.value,t.lineno))

if __name__=='__main__':
    lex.lex()               # Build the lexer
    yacc.yacc()             # Build the parser
    result = yacc.parse(sys.stdin.read())
    print result
```

The output of this script, using the same input as above, is:

```
% python markupbuilder.py < p.txt
Illegal character '~' (1)
['<i>', 'Itals', '</i>', ',', '<em><tt>', 'modname',
'</tt></em>', '&mdash;', 'let', "'s", 'add', 'underscored',
'var', '_', 'name', '.']
```

One thing that is less than ideal in the *PLY* grammar is that it has no quantifiers. In *SimpleParse* or another EBNF library, we might give, for example, a `plain` declaration as:

```
plain := (ALPHANUMS | CONTRACTION | SAFEPUNCT | MDASH | WORDPUNCT)+
```

Quantification can make declarations more direct. But you can achieve the same effect by using self-referential rules whose left-hand terms also occur on the right-hand side. This style is similar to recursive definitions, for example:

```
plain : plain plain
      | OTHERSTUFF
```

For example, `markupbuilder.py`, above, uses this technique.

If a tree structure were generated in this parser, a `plain` node might wind up being a subtree containing lower `plain` nodes (and terminal leaves of `ALPHANUMS`, `CONTRACTION`, etc.). Traversal would need to account for this possibility. The flat list structure used simplifies the issue, in this case. A particular `plain` object might result from the concatenation of several smaller lists, but either way it is a list by the time another rule includes the object.

LEX

A *PLY* lexing module that is intended as support for a parsing application must do four things. A lexing module that constitutes a stand-alone application must do two additional things:

1. Import the *lex* module:

   ```
   import lex
   ```

2. Define a list or tuple variable `tokens` that contains the name of every token type the lexer is allowed to produce. A list may be modified in-place should you wish to specialize the lexer in an importing module; for example:

   ```
   tokens = ['FOO', 'BAR', 'BAZ', 'FLAM']
   ```

3. Define one or more regular expression patterns matching tokens. Each token type listed in `tokens` should have a corresponding pattern; other patterns may be defined also, but the corresponding substrings will not be included in the token stream.

   Token patterns may be defined in one of two ways: (1) By assigning a regular expression string to a specially named variable. (2) By defining a specially named function whose docstring is a regular expression string. In the latter case, "action code" is run when the token is matched. In both styles, the token name is preceded by the prefix `t_`. If a function is used, it should return the `LexToken` object passed to it, possibly after some modification, unless you do not wish to include the token in the token stream. For example:

   ```
   t_FOO = r"[Ff][Oo]{1,2}"
   t_BAR = r"[Bb][Aa][Rr]"
   def t_BAZ(t):
       r"([Bb][Aa][Zz])+"
       t.value = 'BAZ'   # canonical caps BAZ
       return t
   def t_FLAM(t):
       r"(FLAM|flam)*"
       # flam's are discarded (no return)
   ```

   Tokens passed into a pattern function have three attributes: `.type`, `.value`, and `.lineno`. `.lineno` contains the current line number within the string being processed and may be modified to change the reported position, even if the token is not returned. The attribute `.value` is normally the string matched by the regular expression, but a new string, or a compound value like a tuple or instance, may be assigned instead. The `.type` of a `LexToken`, by default, is a string naming the token (the same as the part of the function name after the `t_` prefix).

There is a special order in which various token patterns will be considered. Depending on the patterns used, several patterns could grab the same substring—so it is important to allow the desired pattern first claim on a substring. Each pattern defined with a function is considered in the order it is defined in the lexer file; all patterns defined by assignment to a variable are considered *after* every function-defined pattern. Patterns defined by variable assignment, however, are not considered in the order they are defined, but rather by decreasing length. The purpose of this ordering is to let longer patterns match before their subsequences (e.g., "==" would be claimed before "=", allowing the former comparison operator to match correctly, rather than as sequential assignments).

The special variable `t_ignore` may contain a string of characters to skip during pattern matching. These characters are skipped more efficiently than is a token function that has no return value. The token name `ignore` is, therefore, reserved and may not be used as a regular token (if the all-cap token name convention is followed, it assures no such conflict).

The special function `t_error()` may be used to process illegal characters. The `.value` attribute of the passed-in `LexToken` will contain the remainder of the string being processed (after the last match). If you want to skip past a problem area (perhaps after taking some corrective action in the body of `t_error()`), use the `.skip()` method of the passed-in `LexToken`.

4. Build the lexer. The *lex* module performs a bit of namespace magic so that you normally do not need to name the built lexer. Most applications can use just one default lexer. However, if you wish to—or if you need multiple lexers in the same application—you may bind a built lexer to a name. For example:

```
mylexer = lex.lex()   # named lexer
lex.lex()             # default lexer
mylexer.input(mytext) # set input for named lexer
lex.input(othertext)  # set input for default lexer
```

5. Give the lexer a string to process. This step is handled by the parser when *yacc* is used in conjunction with *lex*, and nothing need be done explicitly. For stand-alone tokenizers, set the input string using `lex.input()` (or similarly with the `.input()` method of named lexers).

6. Read the token stream (for stand-alone tokenizers) using repeated invocation of the default `lex.token()` function or the `.token()` method of a named lexer. Unfortunately, as of version 1.1, *PLY* does not treat the token stream as a Python 2.2 iterator/generator. You can create an iterator wrapper with:

```
from __future__ import generators
# ...define the lexer rules, etc...
def tokeniterator(lexer=lex):
    while 1:
        t = lexer.token()
        if t is None:
            raise StopIteration
        yield t
# Loop through the tokens
for t in tokeniterator():
    # ...do something with each token...
```

Without this wrapper, or generally in earlier versions of Python, you should use a `while 1` loop with a break condition:

```
# ...define the lexer rules, etc...
while 1:
    t = lex.token()
    if t is None:   # No more input
        break
    # ...do something with each token...
```

YACC

A *PLY* parsing module must do five things:

1. Import the `yacc` module:

   ```
   import yacc
   ```

2. Get a token map from a lexer. Suppose a lexer module named `mylexer.py` includes requirements 1 through 4 in the above LEX description. You would get the token map with:

   ```
   from mylexer import *
   ```

   Given the special naming convention `t_*` used for token patterns, the risk of namespace pollution from `import *` is minimal.

   You could also, of course, simply include the necessary lexer setup code in the parsing module itself.

3. Define a collection of grammar rules. Grammar rules are defined in a similar fashion to token functions. Specially named functions having a `p_` prefix contain one or more productions and corresponding action code. Whenever a production contained in the docstring of a `p_*()` function matches, the body of that function runs.

   Productions in *PLY* are described with a simplified EBNF notation. In particular, no quantifiers are available in rules; only sequencing and alternation is used (the rest must be simulated with recursion and component productions).

The left side of each rule contains a single rule name. Following the rule name is one or more spaces, a colon, and an additional one or more spaces. The right side of a rule is everything following this. The right side of a rule can occupy one or more lines; if alternative patterns are allowed to fulfill a rule name, each such pattern occurs on a new line, following a pipe symbol ("|"). Within each right side line, a production is defined by a space-separated sequence of terms—which may be either tokens generated by the lexer or parser productions. More than one production may be included in the same `p_*()` function, but it is generally more clear to limit each function to one production (you are free to create more functions). For example:

```
def p_rulename(t):
    '''rulename   : foo SPACE bar
                  | foo bar baz
                  | bar SPACE baz
       otherrule  : this that other
                  | this SPACE that '''
#...action code...
```

The argument to each `p_*()` function is a `YaccSlice` object, which assigns each component of the rule to an indexed position. The left side rule name is index position 0, and each term/token on the right side is listed thereafter, left to right. The list-like `YaccSlice` is sized just large enough to contain every term needed; this might vary depending on which alternative production is fulfilled on a particular call.

Empty productions are allowed by *yacc* (matching zero-width); you never need more than one empty production in a grammar, but this empty production might be a component of multiple higher-level productions. An empty production is basically a way of getting around the absence of (potentiality) quantification in *PLY*; for example:

```
def p_empty(t):
    '''empty : '''
    pass
def p_maybefoo(t):
    '''foo  : FOOTOKEN
            | empty '''
    t[0] = t[1]
def p_maybebar(t):
    '''bar  : BARTOKEN
            | empty '''
    t[0] = t[1]
```

If a fulfilled production is used in other productions (including itself recursively), the action code should assign a meaningful value to index position 0. This position *is* the value of the production. Moreover what is returned by the actual parsing is this position 0 of the top-level production. For example:

```
# Sum N different numbers: "1.0 + 3 + 3.14 + 17"
def p_sum(t):
    '''sum : number PLUS number'''
    #   ^      ^      ^    ^
    #  t[0]   t[1]  t[2]  t[3]
    t[0] = t[1] + t[3]
def p_number(t):
    '''number : BASICNUMBER
              | sum         '''
    #    ^        ^
    #   t[0]     t[1]
    t[0] = float(t[1])
# Create the parser and parse some strings
yacc.yacc()
print yacc.parse('1.0')
```

The example simply assigns a numeric value with every production, but it could also assign to position 0 of the `YaccSlice` a list, `Node` object, or some other data structure that was useful to higher-level productions.

4. To build the parser the *yacc* module performs a bit of namespace magic so that you normally do not need to name the built parser. Most applications can use just one default parser. However, if you wish to—or if you need multiple parsers in the same application—you may bind a built parser to a name. For example:

```
myparser = yacc.yacc()      # named parser
yacc.yacc()                 # default parser
r1 = myparser.parse(mytext) # set input for named parser
r0 = yacc.parse(othertext)  # set input for default parser
```

When parsers are built, *yacc* will produce diagnostic messages if any errors are encountered in the grammar.

5. Parse an input string. The lexer is implicitly called to get tokens as needed by the grammar rules. The return value of a parsing action can be whatever thing invocation of matched rules builds. It might be an abstract syntax tree, if a `Node` object is used with each parse rule; it might be a simple list as in the smart ASCII example; it might be a modified string based on concatenations and modifications during parsing; or the return value could simply be `None` if parsing was done wholly to trigger side effects in parse rules. In any case, what is returned is index position 0 of the root rule's `LexToken`.

MORE ON PLY PARSERS

Some of the finer points of *PLY* parsers will not be covered in this book. The documentation accompanying *PLY* contains some additional implementational discussion, and a book devoted more systematically to parsing theory will address theoretical issues. But a few aspects can at least be touched on.

Error Recovery

A *PLY* grammar may contain a special `p_error()` function to catch tokens that cannot be matched (at the current position) by any other rule. The first time `p_error()` is invoked, *PLY* enters an "error-recovery" mode. If the parser cannot process the next three tokens successfully, a traceback is generated. You may include the production `error` in other rules to catch errors that occur at specific points in the input.

To implement recovery within the `p_error()` function, you may use the functions/methods `yacc.token()`, `yacc.restart()`, and `yacc.errok()`. The first grabs the next token from the lexer; if this token—or some sequence of tokens—meets some recovery criteria, you may call `yacc.restart()` or `yacc.errok()`. The first of these, `yacc.restart()`, returns the parser to its initial state—basically, only the final substring of the input is used in this case (however, a separate data structure you have built will remain as it was). Calling `yacc.errok()` tells the parser to stay in its last state and just ignore any bad tokens pulled from the lexer (either via the call to `p_error()` itself, or via calls to `yacc.token()` in the body).

The Parser State Machine

When a parser is first compiled, the files `parsetab.py` and `parser.out` are generated. The first, `parsetab.py`, contains more or less unreadable compact data structures that are used by subsequent parser invocations. These structures are used even during later invocation of the applications; timestamps and signatures are compared to determine if the grammar has been changed. Pregenerating state tables speeds up later operations.

The file `parser.out` contains a fairly readable description of the actual state machine generated by *yacc*. Although you cannot manually modify this state machine, examination of `parser.out` can help you in understanding error messages and undesirable behavior you encounter in your grammars.

Precedence and Associativity

To resolve ambiguous grammars, you may set the variable `precedence` to indicate both the precedence and the associativity of tokens. Absent an explicit indication, *PLY* always shifts a new symbol rather than reduce a rule where both are allowable by some grammar rule.

The *PLY* documentation gives an example of an ambiguous arithmetic expression, such as `3 * 4 + 5`. After the tokens `3`, `*`, and `4` have been read from the token list, a `p_mul()` rule might allow reduction of the product. But at the same time, a `p_add()` rule might contain `NUMBER PLUS NUMBER`, which would allow a lookahead to the `PLUS` token (since `4` is a `NUMBER` token). Moreover, the same token can have different meanings in different contexts, such as the unary-minus and minus operators in `3 - 4 * -5`.

To solve both the precedence ambiguity and the ambiguous meaning of the token `MINUS`, you can declare an explicit precedence and associativity such as:

Declaring precedence and associativity

```
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES, 'DIVIDE'),
    ('right', 'UMINUS'),
)
def p_expr_uminus(t):
    'expr : MINUS expr % prec UMINUS'
    t[0] = -1 * t[2]
def p_expr_minus(t):
    'expr : expr MINUS expr'
    t[0] = t[1] - t[3]
def p_expr_plus(t):
    'expr : expr PLUS expr'
    t[0] = t[1] + t[3]
```