

Chapter 9. XML Processing

9.1. Diving in

These next two chapters are about XML processing in Python. It would be helpful if you already knew what an XML document looks like, that it's made up of structured tags to form a hierarchy of elements, and so on. If this doesn't make sense to you, there are many XML tutorials (http://directory.google.com/Top/Computers/Data_Formats/Markup_Languages/XML/Resources/FAQs,_Help,_and_Tutorials/) that can explain the basics.

If you're not particularly interested in XML, you should still read these chapters, which cover important topics like Python packages, Unicode, command line arguments, and how to use `getattr` for method dispatching.

Being a philosophy major is not required, although if you have ever had the misfortune of being subjected to the writings of Immanuel Kant, you will appreciate the example program a lot more than if you majored in something useful, like computer science.

There are two basic ways to work with XML. One is called SAX ("Simple API for XML"), and it works by reading the XML a little bit at a time and calling a method for each element it finds. (If you read Chapter 8, *HTML Processing*, this should sound familiar, because that's how the `sgmlib` module works.) The other is called DOM ("Document Object Model"), and it works by reading in the entire XML document at once and creating an internal representation of it using native Python classes linked in a tree structure. Python has standard modules for both kinds of parsing, but this chapter will only deal with using the DOM.

The following is a complete Python program which generates pseudo-random output based on a context-free grammar defined in an XML format. Don't worry yet if you don't understand what that means; you'll examine both the program's input and its output in more depth throughout these next two chapters.

Example 9.1. `kgp.py`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
"""Kant Generator for Python

Generates mock philosophy based on a context-free grammar

Usage: python kgp.py [options] [source]

Options:
  -g ..., --grammar=...    use specified grammar file or URL
  -h, --help              show this help
  -d                      show debugging information while parsing

Examples:
  kgp.py                  generates several paragraphs of Kantian philosophy
  kgp.py -g husserl.xml   generates several paragraphs of Husserl
  kgp.py "<xref id='paragraph'/>" generates a paragraph of Kant
  kgp.py template.xml     reads from template.xml to decide what to generate
"""

from xml.dom import minidom
import random
import toolbox
import sys
```

```

import getopt

_debug = 0

class NoSourceError(Exception): pass

class KantGenerator:
    """generates mock philosophy based on a context-free grammar"""

    def __init__(self, grammar, source=None):
        self.loadGrammar(grammar)
        self.loadSource(source and source or self.getDefaultSource())
        self.refresh()

    def _load(self, source):
        """load XML input source, return parsed XML document

        - a URL of a remote XML file ("http://diveintopython.org/kant.xml")
        - a filename of a local XML file ("~/diveintopython/common/py/kant.xml")
        - standard input ("-")
        - the actual XML document, as a string
        """
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc

    def loadGrammar(self, grammar):
        """load context-free grammar"""
        self.grammar = self._load(grammar)
        self.refs = {}
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref

    def loadSource(self, source):
        """load source"""
        self.source = self._load(source)

    def getDefaultSource(self):
        """guess default source of the current grammar

        The default source will be one of the <ref>s that is not
        cross-referenced. This sounds complicated but it's not.
        Example: The default source for kant.xml is
        "<xref id='section'/>", because 'section' is the one <ref>
        that is not <xref>'d anywhere in the grammar.
        In most grammars, the default source will produce the
        longest (and most interesting) output.
        """
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        if not standaloneXrefs:
            raise NoSourceError, "can't guess source, and no source specified"
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)

    def reset(self):
        """reset parser"""
        self.pieces = []
        self.capitalizeNextWord = 0

```

```

def refresh(self):
    """reset output buffer, re-parse entire source file, and return output

    Since parsing involves a good deal of randomness, this is an
    easy way to get new output without having to reload a grammar file
    each time.
    """
    self.reset()
    self.parse(self.source)
    return self.output()

def output(self):
    """output generated text"""
    return "".join(self.pieces)

def randomChildElement(self, node):
    """choose a random child element of a node

    This is a utility method used by do_xref and do_choice.
    """
    choices = [e for e in node.childNodes
                if e.nodeType == e.ELEMENT_NODE]
    chosen = random.choice(choices)
    if _debug:
        sys.stderr.write('%s available choices: %s\n' % \
                          (len(choices), [e.toxml() for e in choices]))
        sys.stderr.write('Chosen: %s\n' % chosen.toxml())
    return chosen

def parse(self, node):
    """parse a single XML node

    A parsed XML document (from minidom.parse) is a tree of nodes
    of various types. Each node is represented by an instance of the
    corresponding Python class (Element for a tag, Text for
    text data, Document for the top-level document). The following
    statement constructs the name of a class method based on the type
    of node we're parsing ("parse_Element" for an Element node,
    "parse_Text" for a Text node, etc.) and then calls the method.
    """
    parseMethod = getattr(self, "parse_%s" % node.__class__.__name__)
    parseMethod(node)

def parse_Document(self, node):
    """parse the document node

    The document node by itself isn't interesting (to us), but
    its only child, node.documentElement, is: it's the root node
    of the grammar.
    """
    self.parse(node.documentElement)

def parse_Text(self, node):
    """parse a text node

    The text of a text node is usually added to the output buffer
    verbatim. The one exception is that <p class='sentence'> sets
    a flag to capitalize the first letter of the next word. If
    that flag is set, we capitalize the text and reset the flag.
    """
    text = node.data
    if self.capitalizeNextWord:
        self.pieces.append(text[0].upper())

```

```

        self.pieces.append(text[1:])
        self.capitalizeNextWord = 0
    else:
        self.pieces.append(text)

def parse_Element(self, node):
    """parse an element

    An XML element corresponds to an actual tag in the source:
    <xref id='...'>, <p chance='...'>, <choice>, etc.
    Each element type is handled in its own method. Like we did in
    parse(), we construct a method name based on the name of the
    element ("do_xref" for an <xref> tag, etc.) and
    call the method.
    """
    handlerMethod = getattr(self, "do_%s" % node.tagName)
    handlerMethod(node)

def parse_Comment(self, node):
    """parse a comment

    The grammar can contain XML comments, but we ignore them
    """
    pass

def do_xref(self, node):
    """handle <xref id='...'> tag

    An <xref id='...'> tag is a cross-reference to a <ref id='...'>
    tag. <xref id='sentence'> evaluates to a randomly chosen child of
    <ref id='sentence'>.
    """
    id = node.attributes["id"].value
    self.parse(self.randomChildElement(self.refs[id]))

def do_p(self, node):
    """handle <p> tag

    The <p> tag is the core of the grammar. It can contain almost
    anything: freeform text, <choice> tags, <xref> tags, even other
    <p> tags. If a "class='sentence'" attribute is found, a flag
    is set and the next word will be capitalized. If a "chance='X'"
    attribute is found, there is an X% chance that the tag will be
    evaluated (and therefore a (100-X)% chance that it will be
    completely ignored)
    """
    keys = node.attributes.keys()
    if "class" in keys:
        if node.attributes["class"].value == "sentence":
            self.capitalizeNextWord = 1
    if "chance" in keys:
        chance = int(node.attributes["chance"].value)
        doit = (chance > random.randrange(100))
    else:
        doit = 1
    if doit:
        for child in node.childNodes: self.parse(child)

def do_choice(self, node):
    """handle <choice> tag

    A <choice> tag contains one or more <p> tags. One <p> tag
    is chosen at random and evaluated; the rest are ignored.

```

```

        """
        self.parse(self.randomChildElement(node))

def usage():
    print __doc__

def main(argv):
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage()
            sys.exit()
        elif opt == '-d':
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):
            grammar = arg

    source = "".join(args)

    k = KantGenerator(grammar, source)
    print k.output()

if __name__ == "__main__":
    main(sys.argv[1:])

```

Example 9.2. toolbox.py

```

"""Miscellaneous utility functions"""

def openAnything(source):
    """URI, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner. Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.

    Examples:
    >>> from xml.dom import minidom
    >>> sock = openAnything("http://localhost/kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("c:\\inetpub\\wwwroot\\kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("<ref id='conjunction'><text>and</text><text>or</text></ref>")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    """
    if hasattr(source, "read"):
        return source

    if source == '-':
        import sys

```

```

    return sys.stdin

# try to open with urllib (if source is http, ftp, or file URL)
import urllib
try:
    return urllib.urlopen(source)
except (IOError, OSError):
    pass

# try to open with native open function (if source is pathname)
try:
    return open(source)
except (IOError, OSError):
    pass

# treat source as string
import StringIO
return StringIO.StringIO(str(source))

```

Run the program `kgp.py` by itself, and it will parse the default XML-based grammar, in `kant.xml`, and print several paragraphs worth of philosophy in the style of Immanuel Kant.

Example 9.3. Sample output of `kgp.py`

```
[you@localhost kgp]$ python kgp.py
```

```

    As is shown in the writings of Hume, our a priori concepts, in
reference to ends, abstract from all content of knowledge; in the study
of space, the discipline of human reason, in accordance with the
principles of philosophy, is the clue to the discovery of the
Transcendental Deduction. The transcendental aesthetic, in all
theoretical sciences, occupies part of the sphere of human reason
concerning the existence of our ideas in general; still, the
never-ending regress in the series of empirical conditions constitutes
the whole content for the transcendental unity of apperception. What
we have alone been able to show is that, even as this relates to the
architectonic of human reason, the Ideal may not contradict itself, but
it is still possible that it may be in contradictions with the
employment of the pure employment of our hypothetical judgements, but
natural causes (and I assert that this is the case) prove the validity
of the discipline of pure reason. As we have already seen, time (and
it is obvious that this is true) proves the validity of time, and the
architectonic of human reason, in the full sense of these terms,
abstracts from all content of knowledge. I assert, in the case of the
discipline of practical reason, that the Antinomies are just as
necessary as natural causes, since knowledge of the phenomena is a
posteriori.

```

```

    The discipline of human reason, as I have elsewhere shown, is by
its very nature contradictory, but our ideas exclude the possibility of
the Antinomies. We can deduce that, on the contrary, the pure
employment of philosophy, on the contrary, is by its very nature
contradictory, but our sense perceptions are a representation of, in
the case of space, metaphysics. The thing in itself is a
representation of philosophy. Applied logic is the clue to the
discovery of natural causes. However, what we have alone been able to
show is that our ideas, in other words, should only be used as a canon
for the Ideal, because of our necessary ignorance of the conditions.

```

```
[...snip...]
```

This is, of course, complete gibberish. Well, not complete gibberish. It is syntactically and grammatically correct (although very verbose — Kant wasn't what you would call a get-to-the-point kind of guy). Some of it may actually be true (or at least the sort of thing that Kant would have agreed with), some of it is blatantly false, and most of it is simply incoherent. But all of it is in the style of Immanuel Kant.

Let me repeat that this is much, much funnier if you are now or have ever been a philosophy major.

The interesting thing about this program is that there is nothing Kant-specific about it. All the content in the previous example was derived from the grammar file, `kant.xml`. If you tell the program to use a different grammar file (which you can specify on the command line), the output will be completely different.

Example 9.4. Simpler output from `kgp.py`

```
[you@localhost kgp]$ python kgp.py -g binary.xml
00101001
[you@localhost kgp]$ python kgp.py -g binary.xml
10110100
```

You will take a closer look at the structure of the grammar file later in this chapter. For now, all you need to know is that the grammar file defines the structure of the output, and the `kgp.py` program reads through the grammar and makes random decisions about which words to plug in where.

9.2. Packages

Actually parsing an XML document is very simple: one line of code. However, before you get to that line of code, you need to take a short detour to talk about packages.

Example 9.5. Loading an XML document (a sneak peek)

```
>>> from xml.dom import minidom ❶
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')
```

- ❶ This is a syntax you haven't seen before. It looks almost like the `from module import you know and love`, but the `" . "` gives it away as something above and beyond a simple import. In fact, `xml` is what is known as a package, `dom` is a nested package within `xml`, and `minidom` is a module within `xml.dom`.

That sounds complicated, but it's really not. Looking at the actual implementation may help. Packages are little more than directories of modules; nested packages are subdirectories. The modules within a package (or a nested package) are still just `.py` files, like always, except that they're in a subdirectory instead of the main `lib/` directory of your Python installation.

Example 9.6. File layout of a package

Python21/	root Python installation (home of the executable)
+--lib/	library directory (home of the standard library modules)
+-- xml/	xml package (really just a directory with other stuff in it)
+--sax/	xml.sax package (again, just a directory)
+--dom/	xml.dom package (contains minidom.py)

```
+--parsers/  xml.parsers package (used internally)
```

So when you say `from xml.dom import minidom`, Python figures out that that means "look in the `xml` directory for a `dom` directory, and look in *that* for the `minidom` module, and import it as `minidom`". But Python is even smarter than that; not only can you import entire modules contained within a package, you can selectively import specific classes or functions from a module contained within a package. You can also import the package itself as a module. The syntax is all the same; Python figures out what you mean based on the file layout of the package, and automatically does the right thing.

Example 9.7. Packages are modules, too

```
>>> from xml.dom import minidom ❶
>>> minidom
<module 'xml.dom.minidom' from 'C:\Python21\lib\xml\dom\minidom.pyc'>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml.dom.minidom import Element ❷
>>> Element
<class xml.dom.minidom.Element at 01095744>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml import dom ❸
>>> dom
<module 'xml.dom' from 'C:\Python21\lib\xml\dom\__init__.pyc'>
>>> import xml ❹
>>> xml
<module 'xml' from 'C:\Python21\lib\xml\__init__.pyc'>
```

- ❶ Here you're importing a module (`minidom`) from a nested package (`xml.dom`). The result is that `minidom` is imported into your namespace, and in order to reference classes within the `minidom` module (like `Element`), you need to preface them with the module name.
- ❷ Here you are importing a class (`Element`) from a module (`minidom`) from a nested package (`xml.dom`). The result is that `Element` is imported directly into your namespace. Note that this does not interfere with the previous import; the `Element` class can now be referenced in two ways (but it's all still the same class).
- ❸ Here you are importing the `dom` package (a nested package of `xml`) as a module in and of itself. Any level of a package can be treated as a module, as you'll see in a moment. It can even have its own attributes and methods, just the modules you've seen before.
- ❹ Here you are importing the root level `xml` package as a module.

So how can a package (which is just a directory on disk) be imported and treated as a module (which is always a file on disk)? The answer is the magical `__init__.py` file. You see, packages are not simply directories; they are directories with a specific file, `__init__.py`, inside. This file defines the attributes and methods of the package. For instance, `xml.dom` contains a `Node` class, which is defined in `xml/dom/__init__.py`. When you import a package as a module (like `dom` from `xml`), you're really importing its `__init__.py` file.

A package is a directory with the special `__init__.py` file in it. The `__init__.py` file defines the attributes and methods of the package. It doesn't need to define anything; it can just be an empty file, but it has to exist. But if `__init__.py` doesn't exist, the directory is just a directory, not a package, and it can't be imported or contain modules or nested packages.

So why bother with packages? Well, they provide a way to logically group related modules. Instead of having an `xml` package with `sax` and `dom` packages inside, the authors could have chosen to put all the `sax` functionality in `xmlsax.py` and all the `dom` functionality in `xmldom.py`, or even put all of it in a single module. But that would

have been unwieldy (as of this writing, the XML package has over 3000 lines of code) and difficult to manage (separate source files mean multiple people can work on different areas simultaneously).

If you ever find yourself writing a large subsystem in Python (or, more likely, when you realize that your small subsystem has grown into a large one), invest some time designing a good package architecture. It's one of the many things Python is good at, so take advantage of it.

9.3. Parsing XML

As I was saying, actually parsing an XML document is very simple: one line of code. Where you go from there is up to you.

Example 9.8. Loading an XML document (for real this time)

```
>>> from xml.dom import minidom ❶
>>> xmldoc = minidom.parse('~/.diveintopython/common/py/kgp/binary.xml') ❷
>>> xmldoc ❸
<xml.dom.minidom.Document instance at 010BE87C>
>>> print xmldoc.toxml() ❹
<?xml version="1.0" ?>
<grammar>
  <ref id="bit">
    <p>0</p>
    <p>1</p>
  </ref>
  <ref id="byte">
    <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
  </ref>
</grammar>
```

- ❶ As you saw in the previous section, this imports the `minidom` module from the `xml.dom` package.
- ❷ Here is the one line of code that does all the work: `minidom.parse` takes one argument and returns a parsed representation of the XML document. The argument can be many things; in this case, it's simply a filename of an XML document on my local disk. (To follow along, you'll need to change the path to point to your downloaded examples directory.) But you can also pass a file object, or even a file-like object. You'll take advantage of this flexibility later in this chapter.
- ❸ The object returned from `minidom.parse` is a `Document` object, a descendant of the `Node` class. This `Document` object is the root level of a complex tree-like structure of interlocking Python objects that completely represent the XML document you passed to `minidom.parse`.
- ❹ `toxml` is a method of the `Node` class (and is therefore available on the `Document` object you got from `minidom.parse`). `toxml` prints out the XML that this `Node` represents. For the `Document` node, this prints out the entire XML document.

Now that you have an XML document in memory, you can start traversing through it.

Example 9.9. Getting child nodes

```
>>> xmldoc.childNodes ❶
[<DOM Element: grammar at 17538908>]
>>> xmldoc.childNodes[0] ❷
<DOM Element: grammar at 17538908>
>>> xmldoc.firstChild ❸
<DOM Element: grammar at 17538908>
```

- ❶ Every Node has a `childNodes` attribute, which is a list of the Node objects. A Document always has only one child node, the root element of the XML document (in this case, the grammar element).
- ❷ To get the first (and in this case, the only) child node, just use regular list syntax. Remember, there is nothing special going on here; this is just a regular Python list of regular Python objects.
- ❸ Since getting the first child node of a node is a useful and common activity, the Node class has a `firstChild` attribute, which is synonymous with `childNodes[0]`. (There is also a `lastChild` attribute, which is synonymous with `childNodes[-1]`.)

Example 9.10. `toxml` works on any node

```
>>> grammarNode = xmldoc.firstChild
>>> print grammarNode.toxml() ❶
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

- ❶ Since the `toxml` method is defined in the Node class, it is available on any XML node, not just the Document element.

Example 9.11. Child nodes can be text

```
>>> grammarNode.childNodes ❶
[<DOM Text node "\n">, <DOM Element: ref at 17533332>, \
<DOM Text node "\n">, <DOM Element: ref at 17549660>, <DOM Text node "\n">]
>>> print grammarNode.firstChild.toxml() ❷

>>> print grammarNode.childNodes[1].toxml() ❸
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print grammarNode.childNodes[3].toxml() ❹
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
>>> print grammarNode.lastChild.toxml() ❺
```

- ❶ Looking at the XML in `binary.xml`, you might think that the grammar has only two child nodes, the two `ref` elements. But you're missing something: the carriage returns! After the '`<grammar>`' and before the first '`<ref>`' is a carriage return, and this text counts as a child node of the grammar element. Similarly, there is a carriage return after each '`</ref>`'; these also count as child nodes. So `grammar.childNodes` is actually a list of 5 objects: 3 Text objects and 2 Element objects.
- ❷ The first child is a Text object representing the carriage return after the '`<grammar>`' tag and before the first '`<ref>`' tag.

- ③ The second child is an `Element` object representing the first `ref` element.
- ④ The fourth child is an `Element` object representing the second `ref` element.
- ⑤ The last child is a `Text` object representing the carriage return after the `'</ref>'` end tag and before the `'</grammar>'` end tag.

Example 9.12. Drilling down all the way to text

```
>>> grammarNode
<DOM Element: grammar at 19167148>
>>> refNode = grammarNode.childNodes[1] ❶
>>> refNode
<DOM Element: ref at 17987740>
>>> refNode.childNodes ❷
[<DOM Text node "\n">, <DOM Text node " ">, <DOM Element: p at 19315844>, \
<DOM Text node "\n">, <DOM Text node " ">, \
<DOM Element: p at 19462036>, <DOM Text node "\n">]
>>> pNode = refNode.childNodes[2]
>>> pNode
<DOM Element: p at 19315844>
>>> print pNode.toxml() ❸
<p>0</p>
>>> pNode.firstChild ❹
<DOM Text node "0">
>>> pNode.firstChild.data ❺
u'0'
```

- ❶ As you saw in the previous example, the first `ref` element is `grammarNode.childNodes[1]`, since `childNodes[0]` is a `Text` node for the carriage return.
- ❷ The `ref` element has its own set of child nodes, one for the carriage return, a separate one for the spaces, one for the `p` element, and so forth.
- ❸ You can even use the `toxml` method here, deeply nested within the document.
- ❹ The `p` element has only one child node (you can't tell that from this example, but look at `pNode.childNodes` if you don't believe me), and it is a `Text` node for the single character `'0'`.
- ❺ The `.data` attribute of a `Text` node gives you the actual string that the text node represents. But what is that `'u'` in front of the string? The answer to that deserves its own section.

9.4. Unicode

Unicode is a system to represent characters from all the world's different languages. When Python parses an XML document, all data is stored in memory as unicode.

You'll get to all that in a minute, but first, some background.

Historical note. Before unicode, there were separate character encoding systems for each language, each using the same numbers (0–255) to represent that language's characters. Some languages (like Russian) have multiple conflicting standards about how to represent the same characters; other languages (like Japanese) have so many characters that they require multiple-byte character sets. Exchanging documents between systems was difficult because there was no way for a computer to tell for certain which character encoding scheme the document author had used; the computer only saw numbers, and the numbers could mean different things. Then think about trying to store these documents in the same place (like in the same database table); you would need to store the character encoding alongside each piece of text, and make sure to pass it around whenever you passed the text around. Then think about

multilingual documents, with characters from multiple languages in the same document. (They typically used escape codes to switch modes; poof, you're in Russian koi8-r mode, so character 241 means this; poof, now you're in Mac Greek mode, so character 241 means something else. And so on.) These are the problems which unicode was designed to solve.

To solve these problems, unicode represents each character as a 2-byte number, from 0 to 65535.^[5] Each 2-byte number represents a unique character used in at least one of the world's languages. (Characters that are used in multiple languages have the same numeric code.) There is exactly 1 number per character, and exactly 1 character per number. Unicode data is never ambiguous.

Of course, there is still the matter of all these legacy encoding systems. 7-bit ASCII, for instance, which stores English characters as numbers ranging from 0 to 127. (65 is capital "A", 97 is lowercase "a", and so forth.) English has a very simple alphabet, so it can be completely expressed in 7-bit ASCII. Western European languages like French, Spanish, and German all use an encoding system called ISO-8859-1 (also called "latin-1"), which uses the 7-bit ASCII characters for the numbers 0 through 127, but then extends into the 128-255 range for characters like n-with-a-tilde-over-it (241), and u-with-two-dots-over-it (252). And unicode uses the same characters as 7-bit ASCII for 0 through 127, and the same characters as ISO-8859-1 for 128 through 255, and then extends from there into characters for other languages with the remaining numbers, 256 through 65535.

When dealing with unicode data, you may at some point need to convert the data back into one of these other legacy encoding systems. For instance, to integrate with some other computer system which expects its data in a specific 1-byte encoding scheme, or to print it to a non-unicode-aware terminal or printer. Or to store it in an XML document which explicitly specifies the encoding scheme.

And on that note, let's get back to Python.

Python has had unicode support throughout the language since version 2.0. The XML package uses unicode to store all parsed XML data, but you can use unicode anywhere.

Example 9.13. Introducing unicode

```
>>> s = u'Dive in' ❶
>>> s
u'Dive in'
>>> print s ❷
Dive in
```

- ❶ To create a unicode string instead of a regular ASCII string, add the letter "u" before the string. Note that this particular string doesn't have any non-ASCII characters. That's fine; unicode is a superset of ASCII (a very large superset at that), so any regular ASCII string can also be stored as unicode.
- ❷ When printing a string, Python will attempt to convert it to your default encoding, which is usually ASCII. (More on this in a minute.) Since this unicode string is made up of characters that are also ASCII characters, printing it has the same result as printing a normal ASCII string; the conversion is seamless, and if you didn't know that `s` was a unicode string, you'd never notice the difference.

Example 9.14. Storing non-ASCII characters

```
>>> s = u'La Pe\xfla' ❶
>>> print s ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> print s.encode('latin-1') ❸
```

- ❶ The real advantage of unicode, of course, is its ability to store non-ASCII characters, like the Spanish "ñ" (n with a tilde over it). The unicode character code for the tilde-n is `0xf1` in hexadecimal (241 in decimal), which you can type like this: `\xf1`.
- ❷ Remember I said that the `print` function attempts to convert a unicode string to ASCII so it can print it? Well, that's not going to work here, because your unicode string contains non-ASCII characters, so Python raises a `UnicodeError` error.
- ❸ Here's where the conversion-from-unicode-to-other-encoding-schemes comes in. `s` is a unicode string, but `print` can only print a regular string. To solve this problem, you call the `encode` method, available on every unicode string, to convert the unicode string to a regular string in the given encoding scheme, which you pass as a parameter. In this case, you're using `latin-1` (also known as `iso-8859-1`), which includes the tilde-n (whereas the default ASCII encoding scheme did not, since it only includes characters numbered 0 through 127).

Remember I said Python usually converted unicode to ASCII whenever it needed to make a regular string out of a unicode string? Well, this default encoding scheme is an option which you can customize.

Example 9.15. `sitecustomize.py`

```
# sitecustomize.py
# this file can be anywhere in your Python path,
# but it usually goes in ${pythondir}/lib/site-packages/
import sys
sys.setdefaultencoding('iso-8859-1')
```

- ❶ `sitecustomize.py` is a special script; Python will try to import it on startup, so any code in it will be run automatically. As the comment mentions, it can go anywhere (as long as `import` can find it), but it usually goes in the `site-packages` directory within your Python `lib` directory.
- ❷ `setdefaultencoding` function sets, well, the default encoding. This is the encoding scheme that Python will try to use whenever it needs to auto-coerce a unicode string into a regular string.

Example 9.16. Effects of setting the default encoding

```
>>> import sys
>>> sys.getdefaultencoding()
'iso-8859-1'
>>> s = u'La Pe\xf1a'
>>> print s
La Peña
```

- ❶ This example assumes that you have made the changes listed in the previous example to your `sitecustomize.py` file, and restarted Python. If your default encoding still says `'ascii'`, you didn't set up your `sitecustomize.py` properly, or you didn't restart Python. The default encoding can only be changed during Python startup; you can't change it later. (Due to some wacky programming tricks that I won't get into right now, you can't even call `sys.setdefaultencoding` after Python has started up. Dig into `site.py` and search for `"setdefaultencoding"` to find out how.)
- ❷ Now that the default encoding scheme includes all the characters you use in your string, Python has no problem auto-coercing the string and printing it.

Example 9.17. Specifying encoding in `.py` files

If you are going to be storing non-ASCII strings within your Python code, you'll need to specify the encoding of each individual .py file by putting an encoding declaration at the top of each file. This declaration defines the .py file to be UTF-8:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
```

Now, what about XML? Well, every XML document is in a specific encoding. Again, ISO-8859-1 is a popular encoding for data in Western European languages. KOI8-R is popular for Russian texts. The encoding, if specified, is in the header of the XML document.

Example 9.18. russiansample.xml

```
<?xml version="1.0" encoding="koi8-r"?> ❶
<preface>
<title> @548A;>285</title> ❷
</preface>
```

- ❶ This is a sample extract from a real Russian XML document; it's part of a Russian translation of this very book. Note the encoding, `koi8-r`, specified in the header.
- ❷ These are Cyrillic characters which, as far as I know, spell the Russian word for "Preface". If you open this file in a regular text editor, the characters will most likely look like gibberish, because they're encoded using the `koi8-r` encoding scheme, but they're being displayed in `iso-8859-1`.

Example 9.19. Parsing russiansample.xml

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('russiansample.xml') ❶
>>> title = xmldoc.getElementsByTagName('title')[0].firstChild.data
>>> title ❷
u'\u041f\u0440\u0435\u0434\u0438\u0441\u043b\u043e\u0432\u0438\u0435\u0438'
>>> print title ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> convertedtitle = title.encode('koi8-r') ❹
>>> convertedtitle
'\xf0\xd2\xc5\xc9\xd3\xcc\xcf\xd7\xc9\xc5'
>>> print convertedtitle ❺
@548A;>285
```

- ❶ I'm assuming here that you saved the previous example as `russiansample.xml` in the current directory. I am also, for the sake of completeness, assuming that you've changed your default encoding back to `'ascii'` by removing your `sitcustomize.py` file, or at least commenting out the `setdefaultencoding` line.
- ❷ Note that the text data of the `title` tag (now in the `title` variable, thanks to that long concatenation of Python functions which I hastily skipped over and, annoyingly, won't explain until the next section) — the text data inside the XML document's `title` element is stored in unicode.
- ❸ Printing the title is not possible, because this unicode string contains non-ASCII characters, so Python can't convert it to ASCII because that doesn't make sense.
- ❹ You can, however, explicitly convert it to `koi8-r`, in which case you get a (regular, not unicode) string of single-byte characters (`f0`, `d2`, `c5`, and so forth) that are the `koi8-r`-encoded versions

of the characters in the original unicode string.

- ⑤ Printing the `koi8-r`-encoded string will probably show gibberish on your screen, because your Python IDE is interpreting those characters as `iso-8859-1`, not `koi8-r`. But at least they do print. (And, if you look carefully, it's the same gibberish that you saw when you opened the original XML document in a non-unicode-aware text editor. Python converted it from `koi8-r` into unicode when it parsed the XML document, and you've just converted it back.)

To sum up, unicode itself is a bit intimidating if you've never seen it before, but unicode data is really very easy to handle in Python. If your XML documents are all 7-bit ASCII (like the examples in this chapter), you will literally never think about unicode. Python will convert the ASCII data in the XML documents into unicode while parsing, and auto-coerce it back to ASCII whenever necessary, and you'll never even notice. But if you need to deal with that in other languages, Python is ready.

Further reading

- Unicode.org (<http://www.unicode.org/>) is the home page of the unicode standard, including a brief technical introduction (<http://www.unicode.org/standard/principles.html>).
- Unicode Tutorial (http://www.reportlab.com/i18n/python_unicode_tutorial.html) has some more examples of how to use Python's unicode functions, including how to force Python to coerce unicode into ASCII even when it doesn't really want to.
- PEP 263 (<http://www.python.org/peps/pep-0263.html>) goes into more detail about how and when to define a character encoding in your `.py` files.

9.5. Searching for elements

Traversing XML documents by stepping through each node can be tedious. If you're looking for something in particular, buried deep within your XML document, there is a shortcut you can use to find it quickly: `getElementsByTagName`.

For this section, you'll be using the `binary.xml` grammar file, which looks like this:

Example 9.20. `binary.xml`

```
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
  <ref id="bit">
    <p>0</p>
    <p>1</p>
  </ref>
  <ref id="byte">
    <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
    <xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
  </ref>
</grammar>
```

It has two refs, 'bit' and 'byte'. A bit is either a '0' or '1', and a byte is 8 bits.

Example 9.21. Introducing `getElementsByTagName`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref') ❶
```



```
>>> reflist
[<DOM Element: ref at 136138108>, <DOM Element: ref at 136144292>]
>>> print reflist[0].toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print reflist[1].toxml()
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
```

- ❶ `getElementsByTagName` takes one argument, the name of the element you wish to find. It returns a list of `Element` objects, corresponding to the XML elements that have that name. In this case, you find two `ref` elements.

Example 9.22. Every element is searchable

```
>>> firstref = reflist[0]
>>> print firstref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> plist = firstref.getElementsByTagName("p")
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>]
>>> print plist[0].toxml()
<p>0</p>
>>> print plist[1].toxml()
<p>1</p>
```

- ❶ Continuing from the previous example, the first object in your `reflist` is the 'bit' `ref` element.
- ❷ You can use the same `getElementsByTagName` method on this `Element` to find all the `<p>` elements within the 'bit' `ref` element.
- ❸ Just as before, the `getElementsByTagName` method returns a list of all the elements it found. In this case, you have two, one for each bit.

Example 9.23. Searching is actually recursive

```
>>> plist = xmldoc.getElementsByTagName("p")
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>, <DOM Element: p at 136146124>]
>>> plist[0].toxml()
'<p>0</p>'
>>> plist[1].toxml()
'<p>1</p>'
>>> plist[2].toxml()
'<p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>'
```

- ❶ Note carefully the difference between this and the previous example. Previously, you were searching for `p` elements within `firstref`, but here you are searching for `p` elements within `xmldoc`, the root-level object that represents the entire XML document. This *does* find the `p` elements nested within the `ref` elements within the root grammar element.
- ❷ The first two `p` elements are within the first `ref` (the 'bit' `ref`).

③ The last p element is the one within the second ref (the 'byte' ref).

9.6. Accessing element attributes

XML elements can have one or more attributes, and it is incredibly simple to access them once you have parsed an XML document.

For this section, you'll be using the `binary.xml` grammar file that you saw in the previous section.

This section may be a little confusing, because of some overlapping terminology. Elements in an XML document have attributes, and Python objects also have attributes. When you parse an XML document, you get a bunch of Python objects that represent all the pieces of the XML document, and some of these Python objects represent attributes of the XML elements. But the (Python) objects that represent the (XML) attributes also have (Python) attributes, which are used to access various parts of the (XML) attribute that the object represents. I told you it was confusing. I am open to suggestions on how to distinguish these more clearly.

Example 9.24. Accessing element attributes

```
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref')
>>> bitref = reflist[0]
>>> print bitref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> bitref.attributes ①
<xml.dom.minidom.NamedNodeMap instance at 0x81e0c9c>
>>> bitref.attributes.keys() ② ③
[u'id']
>>> bitref.attributes.values() ④
[<xml.dom.minidom.Attr instance at 0x81d5044>]
>>> bitref.attributes["id"] ⑤
<xml.dom.minidom.Attr instance at 0x81d5044>
```

- ① Each Element object has an attribute called `attributes`, which is a `NamedNodeMap` object. This sounds scary, but it's not, because a `NamedNodeMap` is an object that acts like a dictionary, so you already know how to use it.
- ② Treating the `NamedNodeMap` as a dictionary, you can get a list of the names of the attributes of this element by using `attributes.keys()`. This element has only one attribute, 'id'.
- ③ Attribute names, like all other text in an XML document, are stored in unicode.
- ④ Again treating the `NamedNodeMap` as a dictionary, you can get a list of the values of the attributes by using `attributes.values()`. The values are themselves objects, of type `Attr`. You'll see how to get useful information out of this object in the next example.
- ⑤ Still treating the `NamedNodeMap` as a dictionary, you can access an individual attribute by name, using normal dictionary syntax. (Readers who have been paying extra-close attention will already know how the `NamedNodeMap` class accomplishes this neat trick: by defining a `__getitem__` special method. Other readers can take comfort in the fact that they don't need to understand how it works in order to use it effectively.)

Example 9.25. Accessing individual attributes

```
>>> a = bitref.attributes["id"]
```

```
>>> a
<xml.dom.minidom.Attr instance at 0x81d5044>
>>> a.name ❶
u'id'
>>> a.value ❷
u'bit'
```

- ❶ The `Attr` object completely represents a single XML attribute of a single XML element. The name of the attribute (the same name as you used to find this object in the `bitref.attributes` `NamedNodeMap` pseudo-dictionary) is stored in `a.name`.
- ❷ The actual text value of this XML attribute is stored in `a.value`.

Like a dictionary, attributes of an XML element have no ordering. Attributes may *happen to be* listed in a certain order in the original XML document, and the `Attr` objects may *happen to be* listed in a certain order when the XML document is parsed into Python objects, but these orders are arbitrary and should carry no special meaning. You should always access individual attributes by name, like the keys of a dictionary.

9.7. Segue

OK, that's it for the hard-core XML stuff. The next chapter will continue to use these same example programs, but focus on other aspects that make the program more flexible: using streams for input processing, using `getattr` for method dispatching, and using command-line flags to allow users to reconfigure the program without changing the code.

Before moving on to the next chapter, you should be comfortable doing all of these things:

- Parsing XML documents using `minidom`, searching through the parsed document, and accessing arbitrary element attributes and element children
- Organizing complex libraries into packages
- Converting unicode strings to different character encodings

^[5] This, sadly, is *still* an oversimplification. Unicode now has been extended to handle ancient Chinese, Korean, and Japanese texts, which had so many different characters that the 2-byte unicode system could not represent them all. But Python doesn't currently support that out of the box, and I don't know if there is a project afoot to add it. You've reached the limits of my expertise, sorry.

Chapter 10. Scripts and Streams

10.1. Abstracting input sources

One of Python's greatest strengths is its dynamic binding, and one powerful use of dynamic binding is the *file-like object*.

Many functions which require an input source could simply take a filename, go open the file for reading, read it, and close it when they're done. But they don't. Instead, they take a *file-like object*.

In the simplest case, a *file-like object* is any object with a `read` method with an optional `size` parameter, which returns a string. When called with no `size` parameter, it reads everything there is to read from the input source and returns all the data as a single string. When called with a `size` parameter, it reads that much from the input source and returns that much data; when called again, it picks up where it left off and returns the next chunk of data.

This is how reading from real files works; the difference is that you're not limiting yourself to real files. The input source could be anything: a file on disk, a web page, even a hard-coded string. As long as you pass a file-like object to the function, and the function simply calls the object's `read` method, the function can handle any kind of input source without specific code to handle each kind.

In case you were wondering how this relates to XML processing, `minidom.parse` is one such function which can take a file-like object.

Example 10.1. Parsing XML from a file

```
>>> from xml.dom import minidom
>>> fsock = open('binary.xml')
>>> xmldoc = minidom.parse(fsock)
>>> fsock.close()
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar>
  <ref id="bit">
    <p>0</p>
    <p>1</p>
  </ref>
  <ref id="byte">
    <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
  </ref>
</grammar>
```

- ❶ First, you open the file on disk. This gives you a file object.
- ❷ You pass the file object to `minidom.parse`, which calls the `read` method of `fsock` and reads the XML document from the file on disk.
- ❸ Be sure to call the `close` method of the file object after you're done with it. `minidom.parse` will not do this for you.
- ❹ Calling the `toxml()` method on the returned XML document prints out the entire thing.

Well, that all seems like a colossal waste of time. After all, you've already seen that `minidom.parse` can simply take the filename and do all the opening and closing nonsense automatically. And it's true that if you know you're just going to be parsing a local file, you can pass the filename and `minidom.parse` is smart enough to Do The Right Thing(tm). But notice how similar — and easy — it is to parse an XML document straight from the Internet.

Example 10.2. Parsing XML from a URL

```
>>> import urllib
>>> usock = urllib.urlopen('http://slashdot.org/slashdot.rdf') ❶
>>> xmldoc = minidom.parse(usock) ❷
>>> usock.close() ❸
>>> print xmldoc.toxml() ❹
<?xml version="1.0" ?>
<rdf:RDF xmlns="http://my.netscape.com/rdf/simple/0.9/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<channel>
<title>Slashdot</title>
<link>http://slashdot.org/</link>
<description>News for nerds, stuff that matters</description>
</channel>

<image>
<title>Slashdot</title>
<url>http://images.slashdot.org/topics/topicslashdot.gif</url>
<link>http://slashdot.org/</link>
</image>

<item>
<title>To HDTV or Not to HDTV?</title>
<link>http://slashdot.org/article.pl?sid=01/12/28/0421241</link>
</item>

[...snip...]
```

- ❶ As you saw in a previous chapter, `urlopen` takes a web page URL and returns a file-like object. Most importantly, this object has a `read` method which returns the HTML source of the web page.
- ❷ Now you pass the file-like object to `minidom.parse`, which obediently calls the `read` method of the object and parses the XML data that the `read` method returns. The fact that this XML data is now coming straight from a web page is completely irrelevant. `minidom.parse` doesn't know about web pages, and it doesn't care about web pages; it just knows about file-like objects.
- ❸ As soon as you're done with it, be sure to close the file-like object that `urlopen` gives you.
- ❹ By the way, this URL is real, and it really is XML. It's an XML representation of the current headlines on Slashdot (<http://slashdot.org/>), a technical news and gossip site.

Example 10.3. Parsing XML from a string (the easy but inflexible way)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> xmldoc = minidom.parseString(contents) ❶
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

❶ `minidom` has a method, `parseString`, which takes an entire XML document as a string and parses it. You can use this instead of `minidom.parse` if you know you already have your entire XML document in a string. OK, so you can use the `minidom.parse` function for parsing both local files and remote URLs, but for parsing strings, you use... a different function. That means that if you want to be able to take input from a file, a URL, or a string, you'll need special logic to check whether it's a string, and call the `parseString` function instead. How unsatisfying.

If there were a way to turn a string into a file-like object, then you could simply pass this object to `minidom.parse`. And in fact, there is a module specifically designed for doing just that: `StringIO`.

Example 10.4. Introducing StringIO

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> import StringIO
>>> ssock = StringIO.StringIO(contents) ❶
>>> ssock.read() ❷
"<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock.read() ❸
''
>>> ssock.seek(0) ❹
>>> ssock.read(15) ❺
'<grammar><ref i'
>>> ssock.read(15)
"<d='bit'><p>0</p>"
>>> ssock.read()
'><p>1</p></ref></grammar>'
>>> ssock.close() ❻
```

- ❶ The `StringIO` module contains a single class, also called `StringIO`, which allows you to turn a string into a file-like object. The `StringIO` class takes the string as a parameter when creating an instance.
- ❷ Now you have a file-like object, and you can do all sorts of file-like things with it. Like `read`, which returns the original string.
- ❸ Calling `read` again returns an empty string. This is how real file objects work too; once you read the entire file, you can't read any more without explicitly seeking to the beginning of the file. The `StringIO` object works the same way.
- ❹ You can explicitly seek to the beginning of the string, just like seeking through a file, by using the `seek` method of the `StringIO` object.
- ❺ You can also read the string in chunks, by passing a `size` parameter to the `read` method.
- ❻ At any time, `read` will return the rest of the string that you haven't read yet. All of this is exactly how file objects work; hence the term *file-like object*.

Example 10.5. Parsing XML from a string (the file-like object way)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock = StringIO.StringIO(contents)
>>> xmldoc = minidom.parse(ssock) ❶
>>> ssock.close()
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

- ❶ Now you can pass the file-like object (really a `StringIO`) to `minidom.parse`, which will call the object's `read` method and happily parse away, never knowing that its input came from a hard-coded string.

So now you know how to use a single function, `minidom.parse`, to parse an XML document stored on a web page, in a local file, or in a hard-coded string. For a web page, you use `urlopen` to get a file-like object; for a local file, you use `open`; and for a string, you use `StringIO`. Now let's take it one step further and generalize *these* differences as well.

Example 10.6. openAnything

```
def openAnything(source): ❶
    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
```

```

    return urllib.urlopen(source) ❷
except (IOError, OSError):
    pass

# try to open with native open function (if source is pathname)
try:
    return open(source) ❸
except (IOError, OSError):
    pass

# treat source as string
import StringIO
return StringIO.StringIO(str(source)) ❹

```

- ❶ The `openAnything` function takes a single parameter, `source`, and returns a file-like object. `source` is a string of some sort; it can either be a URL (like `'http://slashdot.org/slashdot.rdf'`), a full or partial pathname to a local file (like `'binary.xml'`), or a string that contains actual XML data to be parsed.
- ❷ First, you see if `source` is a URL. You do this through brute force: you try to open it as a URL and silently ignore errors caused by trying to open something which is not a URL. This is actually elegant in the sense that, if `urllib` ever supports new types of URLs in the future, you will also support them without recoding. If `urllib` is able to open `source`, then the `return` kicks you out of the function immediately and the following `try` statements never execute.
- ❸ On the other hand, if `urllib` yelled at you and told you that `source` wasn't a valid URL, you assume it's a path to a file on disk and try to open it. Again, you don't do anything fancy to check whether `source` is a valid filename or not (the rules for valid filenames vary wildly between different platforms anyway, so you'd probably get them wrong anyway). Instead, you just blindly open the file, and silently trap any errors.
- ❹ By this point, you need to assume that `source` is a string that has hard-coded data in it (since nothing else worked), so you use `StringIO` to create a file-like object out of it and return that. (In fact, since you're using the `str` function, `source` doesn't even need to be a string; it could be any object, and you'll use its string representation, as defined by its `__str__` special method.)

Now you can use this `openAnything` function in conjunction with `minidom.parse` to make a function that takes a `source` that refers to an XML document somehow (either as a URL, or a local filename, or a hard-coded XML document in a string) and parses it.

Example 10.7. Using `openAnything` in `kgp.py`

```

class KantGenerator:
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc

```

10.2. Standard input, output, and error

UNIX users are already familiar with the concept of standard input, standard output, and standard error. This section is for the rest of you.

Standard output and standard error (commonly abbreviated `stdout` and `stderr`) are pipes that are built into every UNIX system. When you `print` something, it goes to the `stdout` pipe; when your program crashes and prints out debugging information (like a traceback in Python), it goes to the `stderr` pipe. Both of these pipes are ordinarily just connected to the terminal window where you are working, so when a program prints, you see the output, and when a program crashes, you see the debugging information. (If you're working on a system with a window-based Python

IDE, `stdout` and `stderr` default to your "Interactive Window".)

Example 10.8. Introducing `stdout` and `stderr`

```
>>> for i in range(3):
...     print 'Dive in'
Dive in
Dive in
Dive in
>>> import sys
>>> for i in range(3):
...     sys.stdout.write('Dive in')
Dive inDive inDive in
>>> for i in range(3):
...     sys.stderr.write('Dive in')
Dive inDive inDive in
```

- ❶ As you saw in Example 6.9, Simple Counters , you can use Python's built-in `range` function to build simple counter loops that repeat something a set number of times.
- ❷ `stdout` is a file-like object; calling its `write` function will print out whatever string you give it. In fact, this is what the `print` function really does; it adds a carriage return to the end of the string you're printing, and calls `sys.stdout.write`.
- ❸ In the simplest case, `stdout` and `stderr` send their output to the same place: the Python IDE (if you're in one), or the terminal (if you're running Python from the command line). Like `stdout`, `stderr` does not add carriage returns for you; if you want them, add them yourself.

`stdout` and `stderr` are both file-like objects, like the ones you discussed in Section 10.1, Abstracting input sources , but they are both write-only. They have no `read` method, only `write`. Still, they are file-like objects, and you can assign any other file- or file-like object to them to redirect their output.

Example 10.9. Redirecting output

```
[you@localhost kgp]$ python stdout.py
Dive in
[you@localhost kgp]$ cat out.log
This message will be logged instead of displayed
```

(On Windows, you can use `type` instead of `cat` to display the contents of a file.)

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
#stdout.py
import sys

print 'Dive in'
saveout = sys.stdout
fsock = open('out.log', 'w')
sys.stdout = fsock
print 'This message will be logged instead of displayed'
sys.stdout = saveout
fsock.close()
```

- ❶ This will print to the IDE "Interactive Window" (or the terminal, if running the script from the command line).

- ❷ Always save `stdout` before redirecting it, so you can set it back to normal later.
- ❸ Open a file for writing. If the file doesn't exist, it will be created. If the file does exist, it will be overwritten.
- ❹ Redirect all further output to the new file you just opened.
- ❺ This will be "printed" to the log file only; it will not be visible in the IDE window or on the screen.
- ❻ Set `stdout` back to the way it was before you mucked with it.
- ❼ Close the log file.

Redirecting `stderr` works exactly the same way, using `sys.stderr` instead of `sys.stdout`.

Example 10.10. Redirecting error information

```
[you@localhost kgp]$ python stderr.py
[you@localhost kgp]$ cat error.log
Traceback (most recent line last):
  File "stderr.py", line 5, in ?
    raise Exception, 'this error will be logged'
Exception: this error will be logged
```

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
#stderr.py
import sys

fsock = open('error.log', 'w')
sys.stderr = fsock
raise Exception, 'this error will be logged'
```

- ❶ Open the log file where you want to store debugging information.
- ❷ Redirect standard error by assigning the file object of the newly-opened log file to `stderr`.
- ❸ Raise an exception. Note from the screen output that this does *not* print anything on screen. All the normal traceback information has been written to `error.log`.
- ❹ Also note that you're not explicitly closing your log file, nor are you setting `stderr` back to its original value. This is fine, since once the program crashes (because of the exception), Python will clean up and close the file for us, and it doesn't make any difference that `stderr` is never restored, since, as I mentioned, the program crashes and Python ends. Restoring the original is more important for `stdout`, if you expect to go do other stuff within the same script afterwards.

Since it is so common to write error messages to standard error, there is a shorthand syntax that can be used instead of going through the hassle of redirecting it outright.

Example 10.11. Printing to `stderr`

```
>>> print 'entering function'
entering function
>>> import sys
>>> print >> sys.stderr, 'entering function'
entering function
```

- ❶ This shorthand syntax of the `print` statement can be used to write to any open file, or file-like object. In this case, you can redirect a single `print` statement to `stderr` without affecting subsequent `print` statements.

Standard input, on the other hand, is a read-only file object, and it represents the data flowing into the program from some previous program. This will likely not make much sense to classic Mac OS users, or even Windows users unless you were ever fluent on the MS-DOS command line. The way it works is that you can construct a chain of commands in a single line, so that one program's output becomes the input for the next program in the chain. The first program simply outputs to standard output (without doing any special redirecting itself, just doing normal `print` statements or whatever), and the next program reads from standard input, and the operating system takes care of connecting one program's output to the next program's input.

Example 10.12. Chaining commands

```
[you@localhost kgp]$ python kgp.py -g binary.xml ❶
01100111
[you@localhost kgp]$ cat binary.xml ❷
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
[you@localhost kgp]$ cat binary.xml | python kgp.py -g - ❸ ❹
10110001
```

- ❶ As you saw in Section 9.1, `Diving in`, this will print a string of eight random bits, 0 or 1.
- ❷ This simply prints out the entire contents of `binary.xml`. (Windows users should use `type` instead of `cat`.)
- ❸ This prints the contents of `binary.xml`, but the `|` character, called the "pipe" character, means that the contents will not be printed to the screen. Instead, they will become the standard input of the next command, which in this case calls your Python script.
- ❹ Instead of specifying a module (like `binary.xml`), you specify `-`, which causes your script to load the grammar from standard input instead of from a specific file on disk. (More on how this happens in the next example.) So the effect is the same as the first syntax, where you specified the grammar filename directly, but think of the expansion possibilities here. Instead of simply doing `cat binary.xml`, you could run a script that dynamically generates the grammar, then you can pipe it into your script. It could come from anywhere: a database, or some grammar-generating meta-script, or whatever. The point is that you don't need to change your `kgp.py` script at all to incorporate any of this functionality. All you need to do is be able to take grammar files from standard input, and you can separate all the other logic into another program.

So how does the script "know" to read from standard input when the grammar file is `-`? It's not magic; it's just code.

Example 10.13. Reading from standard input in `kgp.py`

```
def openAnything(source):
    if source == "-": ❶
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
```

```
[... snip ...]
```

- 1 This is the `openAnything` function from `toolbox.py`, which you previously examined in Section 10.1, *Abstracting input sources*. All you've done is add three lines of code at the beginning of the function to check if the source is `"-"`; if so, you return `sys.stdin`. Really, that's it! Remember, `stdin` is a file-like object with a `read` method, so the rest of the code (in `kgp.py`, where you call `openAnything`) doesn't change a bit.

10.3. Caching node lookups

`kgp.py` employs several tricks which may or may not be useful to you in your XML processing. The first one takes advantage of the consistent structure of the input documents to build a cache of nodes.

A grammar file defines a series of `ref` elements. Each `ref` contains one or more `p` elements, which can contain a lot of different things, including `xrefs`. Whenever you encounter an `xref`, you look for a corresponding `ref` element with the same `id` attribute, and choose one of the `ref` element's children and parse it. (You'll see how this random choice is made in the next section.)

This is how you build up the grammar: define `ref` elements for the smallest pieces, then define `ref` elements which "include" the first `ref` elements by using `xref`, and so forth. Then you parse the "largest" reference and follow each `xref`, and eventually output real text. The text you output depends on the (random) decisions you make each time you fill in an `xref`, so the output is different each time.

This is all very flexible, but there is one downside: performance. When you find an `xref` and need to find the corresponding `ref` element, you have a problem. The `xref` has an `id` attribute, and you want to find the `ref` element that has that same `id` attribute, but there is no easy way to do that. The slow way to do it would be to get the entire list of `ref` elements each time, then manually loop through and look at each `id` attribute. The fast way is to do that once and build a cache, in the form of a dictionary.

Example 10.14. `loadGrammar`

```
def loadGrammar(self, grammar):
    self.grammar = self._load(grammar)
    self.refs = {}
    for ref in self.grammar.getElementsByTagName("ref"):
        self.refs[ref.attributes["id"].value] = ref
```

- 1 Start by creating an empty dictionary, `self.refs`.
- 2 As you saw in Section 9.5, *Searching for elements*, `getElementsByTagName` returns a list of all the elements of a particular name. You easily can get a list of all the `ref` elements, then simply loop through that list.
- 3 As you saw in Section 9.6, *Accessing element attributes*, you can access individual attributes of an element by name, using standard dictionary syntax. So the keys of the `self.refs` dictionary will be the values of the `id` attribute of each `ref` element.
- 4 The values of the `self.refs` dictionary will be the `ref` elements themselves. As you saw in Section 9.3, *Parsing XML*, each element, each node, each comment, each piece of text in a parsed XML document is an object.

Once you build this cache, whenever you come across an `xref` and need to find the `ref` element with the same `id` attribute, you can simply look it up in `self.refs`.

Example 10.15. Using the `ref` element cache

```
def do_xref(self, node):
    id = node.attributes["id"].value
    self.parse(self.randomChildElement(self.refs[id]))
```

You'll explore the `randomChildElement` function in the next section.

10.4. Finding direct children of a node

Another useful technique when parsing XML documents is finding all the direct child elements of a particular element. For instance, in the grammar files, a `ref` element can have several `p` elements, each of which can contain many things, including other `p` elements. You want to find just the `p` elements that are children of the `ref`, not `p` elements that are children of other `p` elements.

You might think you could simply use `getElementsByTagName` for this, but you can't. `getElementsByTagName` searches recursively and returns a single list for all the elements it finds. Since `p` elements can contain other `p` elements, you can't use `getElementsByTagName`, because it would return nested `p` elements that you don't want. To find only direct child elements, you'll need to do it yourself.

Example 10.16. Finding direct child elements

```
def randomChildElement(self, node):
    choices = [e for e in node.childNodes
               if e.nodeType == e.ELEMENT_NODE] ❶ ❷ ❸
    chosen = random.choice(choices)               ❹
    return chosen
```

- ❶ As you saw in Example 9.9, Getting child nodes, the `childNodes` attribute returns a list of all the child nodes of an element.
- ❷ However, as you saw in Example 9.11, Child nodes can be text, the list returned by `childNodes` contains all different types of nodes, including text nodes. That's not what you're looking for here. You only want the children that are elements.
- ❸ Each node has a `nodeType` attribute, which can be `ELEMENT_NODE`, `TEXT_NODE`, `COMMENT_NODE`, or any number of other values. The complete list of possible values is in the `__init__.py` file in the `xml.dom` package. (See Section 9.2, Packages for more on packages.) But you're just interested in nodes that are elements, so you can filter the list to only include those nodes whose `nodeType` is `ELEMENT_NODE`.
- ❹ Once you have a list of actual elements, choosing a random one is easy. Python comes with a module called `random` which includes several useful functions. The `random.choice` function takes a list of any number of items and returns a random item. For example, if the `ref` elements contains several `p` elements, then `choices` would be a list of `p` elements, and `chosen` would end up being assigned exactly one of them, selected at random.

10.5. Creating separate handlers by node type

The third useful XML processing tip involves separating your code into logical functions, based on node types and element names. Parsed XML documents are made up of various types of nodes, each represented by a Python object. The root level of the document itself is represented by a `Document` object. The `Document` then contains one or more `Element` objects (for actual XML tags), each of which may contain other `Element` objects, `Text` objects (for bits of text), or `Comment` objects (for embedded comments). Python makes it easy to write a dispatcher to separate the logic for each node type.

Example 10.17. Class names of parsed XML objects

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('kant.xml') ❶
>>> xmldoc
<xml.dom.minidom.Document instance at 0x01359DE8>
>>> xmldoc.__class__ ❷
<class xml.dom.minidom.Document at 0x01105D40>
>>> xmldoc.__class__.__name__ ❸
'Document'
```

- ❶ Assume for a moment that `kant.xml` is in the current directory.
- ❷ As you saw in Section 9.2, `Packages`, the object returned by parsing an XML document is a `Document` object, as defined in the `minidom.py` in the `xml.dom` package. As you saw in Section 5.4, `Instantiating Classes`, `__class__` is built-in attribute of every Python object.
- ❸ Furthermore, `__name__` is a built-in attribute of every Python class, and it is a string. This string is not mysterious; it's the same as the class name you type when you define a class yourself. (See Section 5.3, `Defining Classes`.)

Fine, so now you can get the class name of any particular XML node (since each XML node is represented as a Python object). How can you use this to your advantage to separate the logic of parsing each node type? The answer is `getattr`, which you first saw in Section 4.4, `Getting Object References With getattr`.

Example 10.18. `parse`, a generic XML node dispatcher

```
def parse(self, node):
    parseMethod = getattr(self, "parse_%s" % node.__class__.__name__) ❶ ❷
    parseMethod(node) ❸
```

- ❶ First off, notice that you're constructing a larger string based on the class name of the node you were passed (in the `node` argument). So if you're passed a `Document` node, you're constructing the string `'parse_Document'`, and so forth.
- ❷ Now you can treat that string as a function name, and get a reference to the function itself using `getattr`.
- ❸ Finally, you can call that function and pass the node itself as an argument. The next example shows the definitions of each of these functions.

Example 10.19. Functions called by the `parse` dispatcher

```
def parse_Document(self, node): ❶
    self.parse(node.documentElement)

def parse_Text(self, node): ❷
    text = node.data
    if self.capitalizeNextWord:
        self.pieces.append(text[0].upper())
        self.pieces.append(text[1:])
        self.capitalizeNextWord = 0
    else:
        self.pieces.append(text)

def parse_Comment(self, node): ❸
    pass

def parse_Element(self, node): ❹
    handlerMethod = getattr(self, "do_%s" % node.tagName)
    handlerMethod(node)
```

- ❶ `parse_Document` is only ever called once, since there is only one Document node in an XML document, and only one Document object in the parsed XML representation. It simply turns around and parses the root element of the grammar file.
- ❷ `parse_Text` is called on nodes that represent bits of text. The function itself does some special processing to handle automatic capitalization of the first word of a sentence, but otherwise simply appends the represented text to a list.
- ❸ `parse_Comment` is just a pass, since you don't care about embedded comments in the grammar files. Note, however, that you still need to define the function and explicitly make it do nothing. If the function did not exist, the generic parse function would fail as soon as it stumbled on a comment, because it would try to find the non-existent `parse_Comment` function. Defining a separate function for every node type, even ones you don't use, allows the generic parse function to stay simple and dumb.
- ❹ The `parse_Element` method is actually itself a dispatcher, based on the name of the element's tag. The basic idea is the same: take what distinguishes elements from each other (their tag names) and dispatch to a separate function for each of them. You construct a string like `'do_xref'` (for an `<xref>` tag), find a function of that name, and call it. And so forth for each of the other tag names that might be found in the course of parsing a grammar file (`<p>` tags, `<choice>` tags).

In this example, the dispatch functions `parse` and `parse_Element` simply find other methods in the same class. If your processing is very complex (or you have many different tag names), you could break up your code into separate modules, and use dynamic importing to import each module and call whatever functions you needed. Dynamic importing will be discussed in Chapter 16, *Functional Programming*.

10.6. Handling command-line arguments

Python fully supports creating programs that can be run on the command line, complete with command-line arguments and either short- or long-style flags to specify various options. None of this is XML-specific, but this script makes good use of command-line processing, so it seemed like a good time to mention it.

It's difficult to talk about command-line processing without understanding how command-line arguments are exposed to your Python program, so let's write a simple program to see them.

Example 10.20. Introducing `sys.argv`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
#argecho.py
import sys

for arg in sys.argv: ❶
    print arg
```

- ❶ Each command-line argument passed to the program will be in `sys.argv`, which is just a list. Here you are printing each argument on a separate line.

Example 10.21. The contents of `sys.argv`

```
[you@localhost py]$ python argecho.py ❶
argecho.py
[you@localhost py]$ python argecho.py abc def ❷
argecho.py
abc
def
```

```
[you@localhost py]$ python argecho.py --help ❸
argecho.py
--help
[you@localhost py]$ python argecho.py -m kant.xml ❹
argecho.py
-m
kant.xml
```

- ❶ The first thing to know about `sys.argv` is that it contains the name of the script you're calling. You will actually use this knowledge to your advantage later, in Chapter 16, *Functional Programming*. Don't worry about it for now.
- ❷ Command-line arguments are separated by spaces, and each shows up as a separate element in the `sys.argv` list.
- ❸ Command-line flags, like `--help`, also show up as their own element in the `sys.argv` list.
- ❹ To make things even more interesting, some command-line flags themselves take arguments. For instance, here you have a flag (`-m`) which takes an argument (`kant.xml`). Both the flag itself and the flag's argument are simply sequential elements in the `sys.argv` list. No attempt is made to associate one with the other; all you get is a list.

So as you can see, you certainly have all the information passed on the command line, but then again, it doesn't look like it's going to be all that easy to actually use it. For simple programs that only take a single argument and have no flags, you can simply use `sys.argv[1]` to access the argument. There's no shame in this; I do it all the time. For more complex programs, you need the `getopt` module.

Example 10.22. Introducing `getopt`

```
def main(argv):
    grammar = "kant.xml" ❶
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="]) ❷
    except getopt.GetoptError: ❸
        usage() ❹
        sys.exit(2)

...

if __name__ == "__main__":
    main(sys.argv[1:])
```

- ❶ First off, look at the bottom of the example and notice that you're calling the `main` function with `sys.argv[1:]`. Remember, `sys.argv[0]` is the name of the script that you're running; you don't care about that for command-line processing, so you chop it off and pass the rest of the list.
- ❷ This is where all the interesting processing happens. The `getopt` function of the `getopt` module takes three parameters: the argument list (which you got from `sys.argv[1:]`), a string containing all the possible single-character command-line flags that this program accepts, and a list of longer command-line flags that are equivalent to the single-character versions. This is quite confusing at first glance, and is explained in more detail below.
- ❸ If anything goes wrong trying to parse these command-line flags, `getopt` will raise an exception, which you catch. You told `getopt` all the flags you understand, so this probably means that the end user passed some command-line flag that you don't understand.
- ❹ As is standard practice in the UNIX world, when the script is passed flags it doesn't understand, you print out a summary of proper usage and exit gracefully. Note that I haven't shown the `usage` function here. You would still need to code that somewhere and have it print out the appropriate summary; it's not automatic.

So what are all those parameters you pass to the `getopt` function? Well, the first one is simply the raw list of command-line flags and arguments (not including the first element, the script name, which you already chopped off before calling the `main` function). The second is the list of short command-line flags that the script accepts.

```
"hg:d"
```

```
-h
    print usage summary
-g ...
    use specified grammar file or URL
-d
    show debugging information while parsing
```

The first and third flags are simply standalone flags; you specify them or you don't, and they do things (print help) or change state (turn on debugging). However, the second flag (`-g`) *must* be followed by an argument, which is the name of the grammar file to read from. In fact it can be a filename or a web address, and you don't know which yet (you'll figure it out later), but you know it has to be *something*. So you tell `getopt` this by putting a colon after the `g` in that second parameter to the `getopt` function.

To further complicate things, the script accepts either short flags (like `-h`) or long flags (like `--help`), and you want them to do the same thing. This is what the third parameter to `getopt` is for, to specify a list of the long flags that correspond to the short flags you specified in the second parameter.

```
["help", "grammar="]
```

```
--help
    print usage summary
--grammar ...
    use specified grammar file or URL
```

Three things of note here:

1. All long flags are preceded by two dashes on the command line, but you don't include those dashes when calling `getopt`. They are understood.
2. The `--grammar` flag must always be followed by an additional argument, just like the `-g` flag. This is notated by an equals sign, `"grammar="`.
3. The list of long flags is shorter than the list of short flags, because the `-d` flag does not have a corresponding long version. This is fine; only `-d` will turn on debugging. But the order of short and long flags needs to be the same, so you'll need to specify all the short flags that *do* have corresponding long flags first, then all the rest of the short flags.

Confused yet? Let's look at the actual code and see if it makes sense in context.

Example 10.23. Handling command-line arguments in `kgp.py`

```
def main(argv):
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
```

❶

❷

```

    if opt in ("-h", "--help"): ❸
        usage()
        sys.exit()
    elif opt == '-d': ❹
        global _debug
        _debug = 1
    elif opt in ("-g", "--grammar"): ❺
        grammar = arg

source = "".join(args) ❻

k = KantGenerator(grammar, source)
print k.output()

```

- ❶ The `grammar` variable will keep track of the grammar file you're using. You initialize it here in case it's not specified on the command line (using either the `-g` or the `--grammar` flag).
- ❷ The `opts` variable that you get back from `getopt` contains a list of tuples: `flag` and `argument`. If the flag doesn't take an argument, then `arg` will simply be `None`. This makes it easier to loop through the flags.
- ❸ `getopt` validates that the command-line flags are acceptable, but it doesn't do any sort of conversion between short and long flags. If you specify the `-h` flag, `opt` will contain `"-h"`; if you specify the `--help` flag, `opt` will contain `"--help"`. So you need to check for both.
- ❹ Remember, the `-d` flag didn't have a corresponding long flag, so you only need to check for the short form. If you find it, you set a global variable that you'll refer to later to print out debugging information. (I used this during the development of the script. What, you thought all these examples worked on the first try?)
- ❺ If you find a grammar file, either with a `-g` flag or a `--grammar` flag, you save the argument that followed it (stored in `arg`) into the `grammar` variable, overwriting the default that you initialized at the top of the `main` function.
- ❻ That's it. You've looped through and dealt with all the command-line flags. That means that anything left must be command-line arguments. These come back from the `getopt` function in the `args` variable. In this case, you're treating them as source material for the parser. If there are no command-line arguments specified, `args` will be an empty list, and `source` will end up as the empty string.

10.7. Putting it all together

You've covered a lot of ground. Let's step back and see how all the pieces fit together.

To start with, this is a script that takes its arguments on the command line, using the `getopt` module.

```

def main(argv):
    ...
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
    ...
    for opt, arg in opts:
    ...

```

You create a new instance of the `KantGenerator` class, and pass it the grammar file and source that may or may not have been specified on the command line.

```

k = KantGenerator(grammar, source)

```

The `KantGenerator` instance automatically loads the grammar, which is an XML file. You use your custom `openAnything` function to open the file (which could be stored in a local file or a remote web server), then use the built-in `minidom` parsing functions to parse the XML into a tree of Python objects.


```
def _load(self, source):
    sock = toolbox.openAnything(source)
    xmldoc = minidom.parse(sock).documentElement
    sock.close()
```

Oh, and along the way, you take advantage of your knowledge of the structure of the XML document to set up a little cache of references, which are just elements in the XML document.

```
def loadGrammar(self, grammar):
    for ref in self.grammar.getElementsByTagName("ref"):
        self.refs[ref.attributes["id"].value] = ref
```

If you specified some source material on the command line, you use that; otherwise you rip through the grammar looking for the "top-level" reference (that isn't referenced by anything else) and use that as a starting point.

```
def getDefaultSource(self):
    xrefs = {}
    for xref in self.grammar.getElementsByTagName("xref"):
        xrefs[xref.attributes["id"].value] = 1
    xrefs = xrefs.keys()
    standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
    return '<xref id="%s"/>' % random.choice(standaloneXrefs)
```

Now you rip through the source material. The source material is also XML, and you parse it one node at a time. To keep the code separated and more maintainable, you use separate handlers for each node type.

```
def parse_Element(self, node):
    handlerMethod = getattr(self, "do_%s" % node.tagName)
    handlerMethod(node)
```

You bounce through the grammar, parsing all the children of each p element,

```
def do_p(self, node):
...
    if doit:
        for child in node.childNodes: self.parse(child)
```

replacing choice elements with a random child,

```
def do_choice(self, node):
    self.parse(self.randomChildElement(node))
```

and replacing xref elements with a random child of the corresponding ref element, which you previously cached.

```
def do_xref(self, node):
    id = node.attributes["id"].value
    self.parse(self.randomChildElement(self.refs[id]))
```

Eventually, you parse your way down to plain text,

```
def parse_Text(self, node):
    text = node.data
...
    self.pieces.append(text)
```

which you print out.

```
def main(argv):
```

```
...
    k = KantGenerator(grammar, source)
    print k.output()
```

10.8. Summary

Python comes with powerful libraries for parsing and manipulating XML documents. The `minidom` takes an XML file and parses it into Python objects, providing for random access to arbitrary elements. Furthermore, this chapter shows how Python can be used to create a "real" standalone command-line script, complete with command-line flags, command-line arguments, error handling, even the ability to take input from the piped result of a previous program.

Before moving on to the next chapter, you should be comfortable doing all of these things:

- Chaining programs with standard input and output
- Defining dynamic dispatchers with `getattr`.
- Using command-line flags and validating them with `getopt`

Chapter 11. HTTP Web Services

11.1. Diving in

You've learned about HTML processing and XML processing, and along the way you saw how to download a web page and how to parse XML from a URL, but let's dive into the more general topic of HTTP web services.

Simply stated, HTTP web services are programmatic ways of sending and receiving data from remote servers using the operations of HTTP directly. If you want to get data from the server, use a straight HTTP GET; if you want to send new data to the server, use HTTP POST. (Some more advanced HTTP web service APIs also define ways of modifying existing data and deleting data, using HTTP PUT and HTTP DELETE.) In other words, the "verbs" built into the HTTP protocol (GET, POST, PUT, and DELETE) map directly to application-level operations for receiving, sending, modifying, and deleting data.

The main advantage of this approach is simplicity, and its simplicity has proven popular with a lot of different sites. Data — usually XML data — can be built and stored statically, or generated dynamically by a server-side script, and all major languages include an HTTP library for downloading it. Debugging is also easier, because you can load up the web service in any web browser and see the raw data. Modern browsers will even nicely format and pretty-print XML data for you, to allow you to quickly navigate through it.

Examples of pure XML-over-HTTP web services:

- Amazon API (<http://www.amazon.com/webservices>) allows you to retrieve product information from the Amazon.com online store.
- National Weather Service (<http://www.nws.noaa.gov/alerts/>) (United States) and Hong Kong Observatory (<http://demo.xml.weather.gov.hk/>) (Hong Kong) offer weather alerts as a web service.
- Atom API (<http://atomenabled.org/>) for managing web-based content.
- Syndicated feeds (<http://syndic8.com/>) from weblogs and news sites bring you up-to-the-minute news from a variety of sites.

In later chapters, you'll explore APIs which use HTTP as a transport for sending and receiving data, but don't map application semantics to the underlying HTTP semantics. (They tunnel everything over HTTP POST.) But this chapter will concentrate on using HTTP GET to get data from a remote server, and you'll explore several HTTP features you can use to get the maximum benefit out of pure HTTP web services.

Here is a more advanced version of the `openanything` module that you saw in the previous chapter:

Example 11.1. `openanything.py`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
import urllib2, urlparse, gzip
from StringIO import StringIO

USER_AGENT = 'OpenAnything/1.0 +http://diveintopython.org/http_web_services/'

class SmartRedirectHandler(urllib2.HTTPRedirectHandler):
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301(
            self, req, fp, code, msg, headers)
        result.status = code
```

```

        return result

def http_error_302(self, req, fp, code, msg, headers):
    result = urllib2.HTTPRedirectHandler.http_error_302(
        self, req, fp, code, msg, headers)
    result.status = code
    return result

class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):
    def http_error_default(self, req, fp, code, msg, headers):
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code
        return result

def openAnything(source, etag=None, lastmodified=None, agent=USER_AGENT):
    '''URL, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner. Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.

    If the etag argument is supplied, it will be used as the value of an
    If-None-Match request header.

    If the lastmodified argument is supplied, it must be a formatted
    date/time string in GMT (as returned in the Last-Modified header of
    a previous request). The formatted date/time will be used
    as the value of an If-Modified-Since request header.

    If the agent argument is supplied, it will be used as the value of a
    User-Agent request header.
    '''

    if hasattr(source, 'read'):
        return source

    if source == '-':
        return sys.stdin

    if urlparse.urlparse(source)[0] == 'http':
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)
        if etag:
            request.add_header('If-None-Match', etag)
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)
        request.add_header('Accept-encoding', 'gzip')
        opener = urllib2.build_opener(SmartRedirectHandler(), DefaultErrorHandler())
        return opener.open(request)

    # try to open with native open function (if source is a filename)
    try:
        return open(source)
    except (IOError, OSError):
        pass

    # treat source as string
    return StringIO(str(source))

```

```
def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)
    result['data'] = f.read()
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')
        if f.headers.get('content-encoding', '') == 'gzip':
            # data came back gzip-compressed, decompress it
            result['data'] = gzip.GzipFile(fileobj=StringIO(result['data'])).read()
    if hasattr(f, 'url'):
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):
        result['status'] = f.status
    f.close()
    return result
```

Further reading

- Paul Prescod believes that pure HTTP web services are the future of the Internet (<http://webservices.xml.com/pub/a/ws/2002/02/06/rest.html>).

11.2. How not to fetch data over HTTP

Let's say you want to download a resource over HTTP, such as a syndicated Atom feed. But you don't just want to download it once; you want to download it over and over again, every hour, to get the latest news from the site that's offering the news feed. Let's do it the quick-and-dirty way first, and then see how you can do better.

Example 11.2. Downloading a feed the quick-and-dirty way

```
>>> import urllib
>>> data = urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()
>>> print data
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/">
  <-- rest of feed omitted for brevity -->
```

- ❶ Downloading anything over HTTP is incredibly easy in Python; in fact, it's a one-liner. The `urllib` module has a handy `urlopen` function that takes the address of the page you want, and returns a file-like object that you can just `read()` from to get the full contents of the page. It just can't get much easier.

So what's wrong with this? Well, for a quick one-off during testing or development, there's nothing wrong with it. I do it all the time. I wanted the contents of the feed, and I got the contents of the feed. The same technique works for any web page. But once you start thinking in terms of a web service that you want to access on a regular basis — and remember, you said you were planning on retrieving this syndicated feed once an hour — then you're being inefficient, and you're being rude.

Let's talk about some of the basic features of HTTP.

11.3. Features of HTTP

There are five important features of HTTP which you should support.

11.3.1. User-Agent

The User-Agent is simply a way for a client to tell a server who it is when it requests a web page, a syndicated feed, or any sort of web service over HTTP. When the client requests a resource, it should always announce who it is, as specifically as possible. This allows the server-side administrator to get in touch with the client-side developer if anything is going fantastically wrong.

By default, Python sends a generic User-Agent: Python-urllib/1.15. In the next section, you'll see how to change this to something more specific.

11.3.2. Redirects

Sometimes resources move around. Web sites get reorganized, pages move to new addresses. Even web services can reorganize. A syndicated feed at `http://example.com/index.xml` might be moved to `http://example.com/xml/atom.xml`. Or an entire domain might move, as an organization expands and reorganizes; for instance, `http://www.example.com/index.xml` might be redirected to `http://server-farm-1.example.com/index.xml`.

Every time you request any kind of resource from an HTTP server, the server includes a status code in its response. Status code 200 means "everything's normal, here's the page you asked for". Status code 404 means "page not found". (You've probably seen 404 errors while browsing the web.)

HTTP has two different ways of signifying that a resource has moved. Status code 302 is a *temporary redirect*; it means "oops, that got moved over here temporarily" (and then gives the temporary address in a `Location:` header). Status code 301 is a *permanent redirect*; it means "oops, that got moved permanently" (and then gives the new address in a `Location:` header). If you get a 302 status code and a new address, the HTTP specification says you should use the new address to get what you asked for, but the next time you want to access the same resource, you should retry the old address. But if you get a 301 status code and a new address, you're supposed to use the new address from then on.

`urllib.urlopen` will automatically "follow" redirects when it receives the appropriate status code from the HTTP server, but unfortunately, it doesn't tell you when it does so. You'll end up getting data you asked for, but you'll never know that the underlying library "helpfully" followed a redirect for you. So you'll continue pounding away at the old address, and each time you'll get redirected to the new address. That's two round trips instead of one: not very efficient! Later in this chapter, you'll see how to work around this so you can deal with permanent redirects properly and efficiently.

11.3.3. Last-Modified/If-Modified-Since

Some data changes all the time. The home page of CNN.com is constantly updating every few minutes. On the other hand, the home page of Google.com only changes once every few weeks (when they put up a special holiday logo, or advertise a new service). Web services are no different; usually the server knows when the data you requested last changed, and HTTP provides a way for the server to include this last-modified date along with the data you requested.

If you ask for the same data a second time (or third, or fourth), you can tell the server the last-modified date that you got last time: you send an `If-Modified-Since` header with your request, with the date you got back from the server last time. If the data hasn't changed since then, the server sends back a special HTTP status code 304, which

means "this data hasn't changed since the last time you asked for it". Why is this an improvement? Because when the server sends a 304, *it doesn't re-send the data*. All you get is the status code. So you don't need to download the same data over and over again if it hasn't changed; the server assumes you have the data cached locally.

All modern web browsers support last-modified date checking. If you've ever visited a page, re-visited the same page a day later and found that it hadn't changed, and wondered why it loaded so quickly the second time — this could be why. Your web browser cached the contents of the page locally the first time, and when you visited the second time, your browser automatically sent the last-modified date it got from the server the first time. The server simply says 304: Not Modified, so your browser knows to load the page from its cache. Web services can be this smart too.

Python's URL library has no built-in support for last-modified date checking, but since you can add arbitrary headers to each request and read arbitrary headers in each response, you can add support for it yourself.

11.3.4. ETag/If-None-Match

ETags are an alternate way to accomplish the same thing as the last-modified date checking: don't re-download data that hasn't changed. The way it works is, the server sends some sort of hash of the data (in an ETag header) along with the data you requested. Exactly how this hash is determined is entirely up to the server. The second time you request the same data, you include the ETag hash in an If-None-Match: header, and if the data hasn't changed, the server will send you back a 304 status code. As with the last-modified date checking, the server *just* sends the 304; it doesn't send you the same data a second time. By including the ETag hash in your second request, you're telling the server that there's no need to re-send the same data if it still matches this hash, since you still have the data from the last time.

Python's URL library has no built-in support for ETags, but you'll see how to add it later in this chapter.

11.3.5. Compression

The last important HTTP feature is gzip compression. When you talk about HTTP web services, you're almost always talking about moving XML back and forth over the wire. XML is text, and quite verbose text at that, and text generally compresses well. When you request a resource over HTTP, you can ask the server that, if it has any new data to send you, to please send it in compressed format. You include the Accept-encoding: gzip header in your request, and if the server supports compression, it will send you back gzip-compressed data and mark it with a Content-encoding: gzip header.

Python's URL library has no built-in support for gzip compression per se, but you can add arbitrary headers to the request. And Python comes with a separate gzip module, which has functions you can use to decompress the data yourself.

Note that our little one-line script to download a syndicated feed did not support any of these HTTP features. Let's see how you can improve it.

11.4. Debugging HTTP web services

First, let's turn on the debugging features of Python's HTTP library and see what's being sent over the wire. This will be useful throughout the chapter, as you add more and more features.

Example 11.3. Debugging HTTP

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1
```

❶

```

>>> import urllib
>>> feeddata = urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/1.15
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 22:27:30 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284-68e0-4de30f80"
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close

```

- ❶ `urllib` relies on another standard Python library, `httplib`. Normally you don't need to import `httplib` directly (`urllib` does that automatically), but you will here so you can set the debugging flag on the `HTTPConnection` class that `urllib` uses internally to connect to the HTTP server. This is an incredibly useful technique. Some other Python libraries have similar debug flags, but there's no particular standard for naming them or turning them on; you need to read the documentation of each library to see if such a feature is available.
- ❷ Now that the debugging flag is set, information on the the HTTP request and response is printed out in real time. The first thing it tells you is that you're connecting to the server `diveintomark.org` on port 80, which is the standard port for HTTP.
- ❸ When you request the Atom feed, `urllib` sends three lines to the server. The first line specifies the HTTP verb you're using, and the path of the resource (minus the domain name). All the requests in this chapter will use `GET`, but in the next chapter on SOAP, you'll see that it uses `POST` for everything. The basic syntax is the same, regardless of the verb.
- ❹ The second line is the `Host` header, which specifies the domain name of the service you're accessing. This is important, because a single HTTP server can host multiple separate domains. My server currently hosts 12 domains; other servers can host hundreds or even thousands.
- ❺ The third line is the `User-Agent` header. What you see here is the generic `User-Agent` that the `urllib` library adds by default. In the next section, you'll see how to customize this to be more specific.
- ❻ The server replies with a status code and a bunch of headers (and possibly some data, which got stored in the `feeddata` variable). The status code here is 200, meaning "everything's normal, here's the data you requested". The server also tells you the date it responded to your request, some information about the server itself, and the content type of the data it's giving you. Depending on your application, this might be useful, or not. It's certainly reassuring that you thought you were asking for an Atom feed, and lo and behold, you're getting an Atom feed (`application/atom+xml`, which is the registered content type for Atom feeds).
- ❼ The server tells you when this Atom feed was last modified (in this case, about 13 minutes ago). You can send this date back to the server the next time you request the same feed, and the server can do last-modified checking.
- ❽ The server also tells you that this Atom feed has an ETag hash of `"e8284-68e0-4de30f80"`. The hash doesn't mean anything by itself; there's nothing you can do with it, except send it back to the server the next time you request this same feed. Then the server can use it to tell you if the data has changed or not.

11.5. Setting the User-Agent

The first step to improving your HTTP web services client is to identify yourself properly with a User-Agent. To do that, you need to move beyond the basic `urllib` and dive into `urllib2`.

Example 11.4. Introducing `urllib2`

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener()
>>> feeddata = opener.open(request).read()
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:23:12 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284-68e0-4de30f80"
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

- ❶ If you still have your Python IDE open from the previous section's example, you can skip this, but this turns on HTTP debugging so you can see what you're actually sending over the wire, and what gets sent back.
- ❷ Fetching an HTTP resource with `urllib2` is a three-step process, for good reasons that will become clear shortly. The first step is to create a `Request` object, which takes the URL of the resource you'll eventually get around to retrieving. Note that this step doesn't actually retrieve anything yet.
- ❸ The second step is to build a URL opener. This can take any number of handlers, which control how responses are handled. But you can also build an opener without any custom handlers, which is what you're doing here. You'll see how to define and use custom handlers later in this chapter when you explore redirects.
- ❹ The final step is to tell the opener to open the URL, using the `Request` object you created. As you can see from all the debugging information that gets printed, this step actually retrieves the resource and stores the returned data in `feeddata`.

Example 11.5. Adding headers with the `Request`

```
>>> request
<urllib2.Request instance at 0x00250AA8>
>>> request.get_full_url()
http://diveintomark.org/xml/atom.xml
>>> request.add_header('User-Agent',
...                   'OpenAnything/1.0 +http://diveintopython.org/')
>>> feeddata = opener.open(request).read()
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: OpenAnything/1.0 +http://diveintopython.org/
'
```

```

reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:45:17 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284-68e0-4de30f80"
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close

```

- ❶ You're continuing from the previous example; you've already created a Request object with the URL you want to access.
- ❷ Using the `add_header` method on the Request object, you can add arbitrary HTTP headers to the request. The first argument is the header, the second is the value you're providing for that header. Convention dictates that a User-Agent should be in this specific format: an application name, followed by a slash, followed by a version number. The rest is free-form, and you'll see a lot of variations in the wild, but somewhere it should include a URL of your application. The User-Agent is usually logged by the server along with other details of your request, and including a URL of your application allows server administrators looking through their access logs to contact you if something is wrong.
- ❸ The opener object you created before can be reused too, and it will retrieve the same feed again, but with your custom User-Agent header.
- ❹ And here's you sending your custom User-Agent, in place of the generic one that Python sends by default. If you look closely, you'll notice that you defined a User-Agent header, but you actually sent a User-agent header. See the difference? `urllib2` changed the case so that only the first letter was capitalized. It doesn't really matter; HTTP specifies that header field names are completely case-insensitive.

11.6. Handling Last-Modified and ETag

Now that you know how to add custom HTTP headers to your web service requests, let's look at adding support for Last-Modified and ETag headers.

These examples show the output with debugging turned off. If you still have it turned on from the previous section, you can turn it off by setting `httplib.HTTPConnection.debuglevel = 0`. Or you can just leave debugging on, if that helps you.

Example 11.6. Testing Last-Modified

```

>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener()
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.dict
❶
{'date': 'Thu, 15 Apr 2004 20:42:41 GMT',
 'server': 'Apache/2.0.49 (Debian GNU/Linux)',
 'content-type': 'application/atom+xml',
 'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
 'etag': '"e842a-3e53-55d97640"',
 'content-length': '15955',
 'accept-ranges': 'bytes',
 'connection': 'close'}
>>> request.add_header('If-Modified-Since',
...     firstdatastream.headers.get('Last-Modified'))
❷
>>> seconddatastream = opener.open(request)
❸

```

```

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\urllib2.py", line 326, in open
    '_open', req)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 901, in http_open
    return self.do_open(httplib.HTTP, req)
  File "c:\python23\lib\urllib2.py", line 895, in do_open
    return self.parent.error('http', req, fp, code, msg, hdrs)
  File "c:\python23\lib\urllib2.py", line 352, in error
    return self._call_chain(*args)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 412, in http_error_default
    raise HTTPError(req.get_full_url(), code, msg, hdrs, fp)
urllib2.HTTPError: HTTP Error 304: Not Modified

```

- ❶ Remember all those HTTP headers you saw printed out when you turned on debugging? This is how you can get access to them programmatically: `firstdatastream.headers` is an object that acts like a dictionary and allows you to get any of the individual headers returned from the HTTP server.
- ❷ On the second request, you add the `If-Modified-Since` header with the last-modified date from the first request. If the data hasn't changed, the server should return a 304 status code.
- ❸ Sure enough, the data hasn't changed. You can see from the traceback that `urllib2` throws a special exception, `HTTPError`, in response to the 304 status code. This is a little unusual, and not entirely helpful. After all, it's not an error; you specifically asked the server not to send you any data if it hadn't changed, and the data didn't change, so the server told you it wasn't sending you any data. That's not an error; that's exactly what you were hoping for.

`urllib2` also raises an `HTTPError` exception for conditions that you would think of as errors, such as 404 (page not found). In fact, it will raise `HTTPError` for *any* status code other than 200 (OK), 301 (permanent redirect), or 302 (temporary redirect). It would be more helpful for your purposes to capture the status code and simply return it, without throwing an exception. To do that, you'll need to define a custom URL handler.

Example 11.7. Defining URL handlers

This custom URL handler is part of `openanything.py`.

```

class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):
    def http_error_default(self, req, fp, code, msg, headers):
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code
        return result

```

- ❶ `urllib2` is designed around URL handlers. Each handler is just a class that can define any number of methods. When something happens — like an HTTP error, or even a 304 code — `urllib2` introspects into the list of defined handlers for a method that can handle it. You used a similar introspection in Chapter 9, *XML Processing* to define handlers for different node types, but `urllib2` is more flexible, and introspects over as many handlers as are defined for the current request.
- ❷ `urllib2` searches through the defined handlers and calls the `http_error_default` method when it encounters a 304 status code from the server. By defining a custom error handler, you can prevent `urllib2` from raising an exception. Instead, you create the `HTTPError` object, but return it instead of raising it.
- ❸ This is the key part: before returning, you save the status code returned by the HTTP server. This will allow you easy access to it from the calling program.

Example 11.8. Using custom URL handlers

```
>>> request.headers
{'If-modified-since': 'Thu, 15 Apr 2004 19:45:21 GMT'}
>>> import openanything
>>> opener = urllib2.build_opener(
...     openanything.DefaultErrorHandler())
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status
304
>>> seconddatastream.read()
''
```

- ❶ You're continuing the previous example, so the Request object is already set up, and you've already added the If-Modified-Since header.
- ❷ This is the key: now that you've defined your custom URL handler, you need to tell urllib2 to use it. Remember how I said that urllib2 broke up the process of accessing an HTTP resource into three steps, and for good reason? This is why building the URL opener is its own step, because you can build it with your own custom URL handlers that override urllib2's default behavior.
- ❸ Now you can quietly open the resource, and what you get back is an object that, along with the usual headers (use `seconddatastream.headers.dict` to access them), also contains the HTTP status code. In this case, as you expected, the status is 304, meaning this data hasn't changed since the last time you asked for it.
- ❹ Note that when the server sends back a 304 status code, it doesn't re-send the data. That's the whole point: to save bandwidth by not re-downloading data that hasn't changed. So if you actually want that data, you'll need to cache it locally the first time you get it.

Handling ETag works much the same way, but instead of checking for Last-Modified and sending If-Modified-Since, you check for ETag and send If-None-Match. Let's start with a fresh IDE session.

Example 11.9. Supporting ETag/If-None-Match

```
>>> import urllib2, openanything
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener(
...     openanything.DefaultErrorHandler())
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.get('ETag')
'e842a-3e53-55d97640'
>>> firstdata = firstdatastream.read()
>>> print firstdata
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/">
  <-- rest of feed omitted for brevity -->
>>> request.add_header('If-None-Match',
...     firstdatastream.headers.get('ETag'))
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status
304
>>> seconddatastream.read()
''
```

❶

Using the `firstdatastream.headers` pseudo-dictionary, you can get the ETag returned from the server. (What happens if the server didn't send back an ETag? Then this line would return `None`.)

- ❷ OK, you got the data.
- ❸ Now set up the second call by setting the `If-None-Match` header to the ETag you got from the first call.
- ❹ The second call succeeds quietly (without throwing an exception), and once again you see that the server has sent back a 304 status code. Based on the ETag you sent the second time, it knows that the data hasn't changed.
- ❺ Regardless of whether the 304 is triggered by `Last-Modified` date checking or ETag hash matching, you'll never get the data along with the 304. That's the whole point.

In these examples, the HTTP server has supported both `Last-Modified` and ETag headers, but not all servers do. As a web services client, you should be prepared to support both, but you must code defensively in case a server only supports one or the other, or neither.

11.7. Handling redirects

You can support permanent and temporary redirects using a different kind of custom URL handler.

First, let's see why a redirect handler is necessary in the first place.

Example 11.10. Accessing web services without a redirect handler

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> request = urllib2.Request(
...     'http://diveintomark.org/redirect/example301.xml')
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redirect/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
```

```

header: Content-Type: application/atom+xml
>>> f.url
'http://diveintomark.org/xml/atom.xml'
>>> f.headers.dict
{'content-length': '15955',
 'accept-ranges': 'bytes',
 'server': 'Apache/2.0.49 (Debian GNU/Linux)',
 'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
 'connection': 'close',
 'etag': '"e842a-3e53-55d97640"',
 'date': 'Thu, 15 Apr 2004 22:06:25 GMT',
 'content-type': 'application/atom+xml'}
>>> f.status
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: addinfourl instance has no attribute 'status'

```

- ❶ You'll be better able to see what's happening if you turn on debugging.
- ❷ This is a URL which I have set up to permanently redirect to my Atom feed at <http://diveintomark.org/xml/atom.xml>.
- ❸ Sure enough, when you try to download the data at that address, the server sends back a 301 status code, telling you that the resource has moved permanently.
- ❹ The server also sends back a `Location:` header that gives the new address of this data.
- ❺ `urllib2` notices the redirect status code and automatically tries to retrieve the data at the new location specified in the `Location:` header.
- ❻ The object you get back from the `opener` contains the new permanent address and all the headers returned from the second request (retrieved from the new permanent address). But the status code is missing, so you have no way of knowing programmatically whether this redirect was temporary or permanent. And that matters very much: if it was a temporary redirect, then you should continue to ask for the data at the old location. But if it was a permanent redirect (as this was), you should ask for the data at the new location from now on.

This is suboptimal, but easy to fix. `urllib2` doesn't behave exactly as you want it to when it encounters a 301 or 302, so let's override its behavior. How? With a custom URL handler, just like you did to handle 304 codes.

Example 11.11. Defining the redirect handler

This class is defined in `openanything.py`.

```

class SmartRedirectHandler(urllib2.HTTPRedirectHandler):
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301(
            self, req, fp, code, msg, headers)
        result.status = code
        return result

    def http_error_302(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
        return result

```

- ❶ Redirect behavior is defined in `urllib2` in a class called `HTTPRedirectHandler`. You don't want to completely override the behavior, you just want to extend it a little, so you'll subclass `HTTPRedirectHandler` so you can call the ancestor class to do all the hard work.
- ❷

When it encounters a 301 status code from the server, `urllib2` will search through its handlers and call the `http_error_301` method. The first thing ours does is just call the `http_error_301` method in the ancestor, which handles the grunt work of looking for the `Location:` header and following the redirect to the new address.

- ❸ Here's the key: before you return, you store the status code (301), so that the calling program can access it later.
- ❹ Temporary redirects (status code 302) work the same way: override the `http_error_302` method, call the ancestor, and save the status code before returning.

So what has this bought us? You can now build a URL opener with the custom redirect handler, and it will still automatically follow redirects, but now it will also expose the redirect status code.

Example 11.12. Using the redirect handler to detect permanent redirects

```
>>> request = urllib2.Request('http://diveintomark.org/redirect/example301.xml')
>>> import openanything, httpplib
>>> httpplib.HTTPConnection.debuglevel = 1
>>> opener = urllib2.build_opener(
...     openanything.SmartRedirectHandler()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: 'GET /redirect/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml

>>> f.status
301
>>> f.url
'http://diveintomark.org/xml/atom.xml'
```

- ❶ First, build a URL opener with the redirect handler you just defined.
- ❷ You sent off a request, and you got a 301 status code in response. At this point, the `http_error_301` method gets called. You call the ancestor method, which follows the redirect and sends a request at the new location (`http://diveintomark.org/xml/atom.xml`).

- ③ This is the payoff: now, not only do you have access to the new URL, but you have access to the redirect status code, so you can tell that this was a permanent redirect. The next time you request this data, you should request it from the new location (`http://diveintomark.org/xml/atom.xml`, as specified in `f.url`). If you had stored the location in a configuration file or a database, you need to update that so you don't keep pounding the server with requests at the old address. It's time to update your address book.

The same redirect handler can also tell you that you *shouldn't* update your address book.

Example 11.13. Using the redirect handler to detect temporary redirects

```
>>> request = urllib2.Request(
...     'http://diveintomark.org/redirect/example302.xml') ❶
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redirect/example302.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 302 Found\r\n' ❷
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 314
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0 ❸
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.status ❹
302
>>> f.url
http://diveintomark.org/xml/atom.xml
```

- ❶ This is a sample URL I've set up that is configured to tell clients to *temporarily* redirect to `http://diveintomark.org/xml/atom.xml`.
- ❷ The server sends back a 302 status code, indicating a temporary redirect. The temporary new location of the data is given in the `Location:` header.
- ❸ `urllib2` calls your `http_error_302` method, which calls the ancestor method of the same name in `urllib2.HTTPRedirectHandler`, which follows the redirect to the new location. Then your `http_error_302` method stores the status code (302) so the calling application can get it later.
- ❹ And here you are, having successfully followed the redirect to `http://diveintomark.org/xml/atom.xml`. `f.status` tells you that this was a temporary redirect, which means that you should continue to request data from the original address (`http://diveintomark.org/redirect/example302.xml`). Maybe it will redirect next time too, but

maybe not. Maybe it will redirect to a different address. It's not for you to say. The server said this redirect was only temporary, so you should respect that. And now you're exposing enough information that the calling application can respect that.

11.8. Handling compressed data

The last important HTTP feature you want to support is compression. Many web services have the ability to send data compressed, which can cut down the amount of data sent over the wire by 60% or more. This is especially true of XML web services, since XML data compresses very well.

Servers won't give you compressed data unless you tell them you can handle it.

Example 11.14. Telling the server you would like compressed data

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> request.add_header('Accept-encoding', 'gzip') ❶
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
Accept-encoding: gzip
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:24:39 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Vary: Accept-Encoding
header: Content-Encoding: gzip ❸
header: Content-Length: 6289 ❹
header: Connection: close
header: Content-Type: application/atom+xml
```

- ❶ This is the key: once you've created your `Request` object, add an `Accept-encoding` header to tell the server you can accept gzip-encoded data. `gzip` is the name of the compression algorithm you're using. In theory there could be other compression algorithms, but `gzip` is the compression algorithm used by 99% of web servers.
- ❷ There's your header going across the wire.
- ❸ And here's what the server sends back: the `Content-Encoding: gzip` header means that the data you're about to receive has been gzip-compressed.
- ❹ The `Content-Length` header is the length of the compressed data, not the uncompressed data. As you'll see in a minute, the actual length of the uncompressed data was 15955, so gzip compression cut your bandwidth by over 60%!

Example 11.15. Decompressing the data

```
>>> compresseddata = f.read() ❶
>>> len(compresseddata)
6289
>>> import StringIO
```

```

>>> compressedstream = StringIO.StringIO(compresseddata)
>>> import gzip
>>> zipper = gzip.GzipFile(fileobj=compressedstream)
>>> data = zipper.read()
>>> print data
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/">
  <-- rest of feed omitted for brevity -->
>>> len(data)
15955

```

- ❶ Continuing from the previous example, `f` is the file-like object returned from the URL opener. Using its `read()` method would ordinarily get you the uncompressed data, but since this data has been gzip-compressed, this is just the first step towards getting the data you really want.
- ❷ OK, this step is a little bit of messy workaround. Python has a `gzip` module, which reads (and actually writes) gzip-compressed files on disk. But you don't have a file on disk, you have a gzip-compressed buffer in memory, and you don't want to write out a temporary file just so you can uncompress it. So what you're going to do is create a file-like object out of the in-memory data (`compresseddata`), using the `StringIO` module. You first saw the `StringIO` module in the previous chapter, but now you've found another use for it.
- ❸ Now you can create an instance of `GzipFile`, and tell it that its "file" is the file-like object `compressedstream`.
- ❹ This is the line that does all the actual work: "reading" from `GzipFile` will decompress the data. Strange? Yes, but it makes sense in a twisted kind of way. `zipper` is a file-like object which represents a gzip-compressed file. That "file" is not a real file on disk, though; `zipper` is really just "reading" from the file-like object you created with `StringIO` to wrap the compressed data, which is only in memory in the variable `compresseddata`. And where did that compressed data come from? You originally downloaded it from a remote HTTP server by "reading" from the file-like object you built with `urllib2.build_opener`. And amazingly, this all just works. Every step in the chain has no idea that the previous step is faking it.
- ❺ Look ma, real data. (15955 bytes of it, in fact.)

"But wait!" I hear you cry. "This could be even easier!" I know what you're thinking. You're thinking that `opener.open` returns a file-like object, so why not cut out the `StringIO` middleman and just pass `f` directly to `GzipFile`? OK, maybe you weren't thinking that, but don't worry about it, because it doesn't work.

Example 11.16. Decompressing the data directly from the server

```

>>> f = opener.open(request)
>>> f.headers.get('Content-Encoding')
'gzip'
>>> data = gzip.GzipFile(fileobj=f).read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\gzip.py", line 217, in read
    self._read(readsize)
  File "c:\python23\lib\gzip.py", line 252, in _read
    pos = self.fileobj.tell() # Save current position
AttributeError: addinfourl instance has no attribute 'tell'

```

- ❶ Continuing from the previous example, you already have a `Request` object set up with an `Accept-encoding: gzip` header.
- ❷ Simply opening the request will get you the headers (though not download any data yet). As you can see from the returned `Content-Encoding` header, this data has been sent gzip-compressed.
- ❸ Since `opener.open` returns a file-like object, and you know from the headers that when you read it, you're going to get gzip-compressed data, why not simply pass that file-like object directly to `GzipFile`? As you "read" from the `GzipFile` instance, it will "read" compressed data from the remote HTTP server and decompress it on the fly. It's a good idea, but unfortunately it doesn't work. Because of the way gzip compression works, `GzipFile` needs to save its position and move forwards and backwards through the compressed file. This doesn't work when the "file" is a stream of bytes coming from a remote server; all you can do with it is retrieve bytes one at a time, not move back and forth through the data stream. So the inelegant hack of using `StringIO` is the best solution: download the compressed data, create a file-like object out of it with `StringIO`, and then decompress the data from that.

11.9. Putting it all together

You've seen all the pieces for building an intelligent HTTP web services client. Now let's see how they all fit together.

Example 11.17. The `openanything` function

This function is defined in `openanything.py`.

```
def openAnything(source, etag=None, lastmodified=None, agent=USER_AGENT):
    # non-HTTP code omitted for brevity
    if urlparse.urlparse(source)[0] == 'http':
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)
        if etag:
            request.add_header('If-None-Match', etag)
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)
        request.add_header('Accept-encoding', 'gzip')
        opener = urllib2.build_opener(SmartRedirectHandler(), DefaultErrorHandler())
        return opener.open(request)
```

- ❶ `urlparse` is a handy utility module for, you guessed it, parsing URLs. Its primary function, also called `urlparse`, takes a URL and splits it into a tuple of (scheme, domain, path, params, query string parameters, and fragment identifier). Of these, the only thing you care about is the scheme, to make sure that you're dealing with an HTTP URL (which `urllib2` can handle).
- ❷ You identify yourself to the HTTP server with the `User-Agent` passed in by the calling function. If no `User-Agent` was specified, you use a default one defined earlier in the `openanything.py` module. You never use the default one defined by `urllib2`.
- ❸ If an ETag hash was given, send it in the `If-None-Match` header.
- ❹ If a last-modified date was given, send it in the `If-Modified-Since` header.
- ❺ Tell the server you would like compressed data if possible.
- ❻ Build a URL opener that uses *both* of the custom URL handlers: `SmartRedirectHandler` for handling 301 and 302 redirects, and `DefaultErrorHandler` for handling 304, 404, and other error conditions gracefully.
- ❼ That's it! Open the URL and return a file-like object to the caller.

Example 11.18. The `fetch` function

This function is defined in `openanything.py`.

```
def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)
    result['data'] = f.read()
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')
        if f.headers.get('content-encoding', '') == 'gzip':
            # data came back gzip-compressed, decompress it
            result['data'] = gzip.GzipFile(fileobj=StringIO(result['data'])).read()
    if hasattr(f, 'url'):
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):
        result['status'] = f.status
    f.close()
    return result
```

- ❶ First, you call the `openAnything` function with a URL, ETag hash, Last-Modified date, and User-Agent.
- ❷ Read the actual data returned from the server. This may be compressed; if so, you'll decompress it later.
- ❸ Save the ETag hash returned from the server, so the calling application can pass it back to you next time, and you can pass it on to `openAnything`, which can stick it in the `If-None-Match` header and send it to the remote server.
- ❹ Save the Last-Modified date too.
- ❺ If the server says that it sent compressed data, decompress it.
- ❻ If you got a URL back from the server, save it, and assume that the status code is 200 until you find out otherwise.
- ❼ If one of the custom URL handlers captured a status code, then save that too.

Example 11.19. Using `openanything.py`

```
>>> import openanything
>>> useragent = 'MyHTTPWebServicesApp/1.0'
>>> url = 'http://diveintopython.org/redir/example301.xml'
>>> params = openanything.fetch(url, agent=useragent)
>>> params
{'url': 'http://diveintomark.org/xml/atom.xml',
 'lastmodified': 'Thu, 15 Apr 2004 19:45:21 GMT',
 'etag': '"e842a-3e53-55d97640"',
 'status': 301,
 'data': '<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
<-- rest of data omitted for brevity -->'}
>>> if params['status'] == 301:
...     url = params['url']
>>> newparams = openanything.fetch(
...     url, params['etag'], params['lastmodified'], useragent)
>>> newparams
{'url': 'http://diveintomark.org/xml/atom.xml',
 'lastmodified': None,
 'etag': '"e842a-3e53-55d97640"',
 'status': 304,
```

```
'data': ''}
```

5

- ❶ The very first time you fetch a resource, you don't have an `ETag` hash or `Last-Modified` date, so you'll leave those out. (They're optional parameters.)
- ❷ What you get back is a dictionary of several useful headers, the HTTP status code, and the actual data returned from the server. `openanything` handles the `gzip` compression internally; you don't care about that at this level.
- ❸ If you ever get a `301` status code, that's a permanent redirect, and you need to update your URL to the new address.
- ❹ The second time you fetch the same resource, you have all sorts of information to pass back: a (possibly updated) URL, the `ETag` from the last time, the `Last-Modified` date from the last time, and of course your `User-Agent`.
- ❺ What you get back is again a dictionary, but the data hasn't changed, so all you got was a `304` status code and no data.

11.10. Summary

The `openanything.py` and its functions should now make perfect sense.

There are 5 important features of HTTP web services that every client should support:

- Identifying your application by setting a proper `User-Agent`.
- Handling permanent redirects properly.
- Supporting `Last-Modified` date checking to avoid re-downloading data that hasn't changed.
- Supporting `ETag` hashes to avoid re-downloading data that hasn't changed.
- Supporting `gzip` compression to reduce bandwidth even when data *has* changed.

Chapter 12. SOAP Web Services

Chapter 11 focused on document-oriented web services over HTTP. The "input parameter" was the URL, and the "return value" was an actual XML document which it was your responsibility to parse.

This chapter will focus on SOAP web services, which take a more structured approach. Rather than dealing with HTTP requests and XML documents directly, SOAP allows you to simulate calling functions that return native data types. As you will see, the illusion is almost perfect; you can "call" a function through a SOAP library, with the standard Python calling syntax, and the function appears to return Python objects and values. But under the covers, the SOAP library has actually performed a complex transaction involving multiple XML documents and a remote server.

SOAP is a complex specification, and it is somewhat misleading to say that SOAP is all about calling remote functions. Some people would pipe up to add that SOAP allows for one-way asynchronous message passing, and document-oriented web services. And those people would be correct; SOAP can be used that way, and in many different ways. But this chapter will focus on so-called "RPC-style" SOAP — calling a remote function and getting results back.

12.1. Diving In

You use Google, right? It's a popular search engine. Have you ever wished you could programmatically access Google search results? Now you can. Here is a program to search Google from Python.

Example 12.1. `search.py`

```
from SOAPpy import WSDL

# you'll need to configure these two values;
# see http://www.google.com/apis/
WSDLFILE = '/path/to/copy/of/GoogleSearch.wsdl'
APIKEY = 'YOUR_GOOGLE_API_KEY'

_server = WSDL.Proxy(WSDLFILE)
def search(q):
    """Search Google and return list of {title, link, description}"""
    results = _server.doGoogleSearch(
        APIKEY, q, 0, 10, False, "", False, "", "utf-8", "utf-8")
    return [{"title": r.title.encode("utf-8"),
            "link": r.URL.encode("utf-8"),
            "description": r.snippet.encode("utf-8")}
            for r in results.resultElements]

if __name__ == '__main__':
    import sys
    for r in search(sys.argv[1])[:5]:
        print r['title']
        print r['link']
        print r['description']
        print
```

You can import this as a module and use it from a larger program, or you can run the script from the command line. On the command line, you give the search query as a command-line argument, and it prints out the URL, title, and description of the top five Google search results.

Here is the sample output for a search for the word "python".

Example 12.2. Sample Usage of `search.py`

```
C:\diveintopython\common\py> python search.py "python"
<b>Python</b> Programming Language
http://www.python.org/
Home page for <b>Python</b>, an interpreted, interactive, object-oriented,
extensible<br> programming language. <b>...</b> <b>Python</b>
is OSI Certified Open Source: OSI Certified.

<b>Python</b> Documentation Index
http://www.python.org/doc/
<b>...</b> New-style classes (aka descriptro). Regular expressions. Database
API. Email Us.<br> docs@<b>python</b>.org. (c) 2004. <b>Python</b>
Software Foundation. <b>Python</b> Documentation. <b>...</b>

Download <b>Python</b> Software
http://www.python.org/download/
Download Standard <b>Python</b> Software. <b>Python</b> 2.3.3 is the
current production<br> version of <b>Python</b>. <b>...</b>
<b>Python</b> is OSI Certified Open Source:

Pythonline
http://www.pythonline.com/

Dive Into <b>Python</b>
http://diveintopython.org/
Dive Into <b>Python</b>. <b>Python</b> from novice to pro. Find:
<b>...</b> It is also available in multiple<br> languages. Read
Dive Into <b>Python</b>. This book is still being written. <b>...</b>
```

Further Reading on SOAP

- <http://www.xmethods.net/> is a repository of public access SOAP web services.
- The SOAP specification (<http://www.w3.org/TR/soap/>) is surprisingly readable, if you like that sort of thing.

12.2. Installing the SOAP Libraries

Unlike the other code in this book, this chapter relies on libraries that do not come pre-installed with Python.

Before you can dive into SOAP web services, you'll need to install three libraries: PyXML, fpconst, and SOAPpy.

12.2.1. Installing PyXML

The first library you need is PyXML, an advanced set of XML libraries that provide more functionality than the built-in XML libraries we studied in Chapter 9.

Procedure 12.1.

Here is the procedure for installing PyXML:

1. Go to <http://pyxml.sourceforge.net/>, click Downloads, and download the latest version for your operating system.
2. If you are using Windows, there are several choices. Make sure to download the version of PyXML that matches the version of Python you are using.
3. Double-click the installer. If you download PyXML 0.8.3 for Windows and Python 2.3, the installer program

will be `PyXML-0.8.3.win32-py2.3.exe`.

4. Step through the installer program.
5. After the installation is complete, close the installer. There will not be any visible indication of success (no programs installed on the Start Menu or shortcuts installed on the desktop). PyXML is simply a collection of XML libraries used by other programs.

To verify that you installed PyXML correctly, run your Python IDE and check the version of the XML libraries you have installed, as shown here.

Example 12.3. Verifying PyXML Installation

```
>>> import xml
>>> xml.__version__
'0.8.3'
```

This version number should match the version number of the PyXML installer program you downloaded and ran.

12.2.2. Installing fpconst

The second library you need is `fpconst`, a set of constants and functions for working with IEEE754 double-precision special values. This provides support for the special values Not-a-Number (NaN), Positive Infinity (Inf), and Negative Infinity (-Inf), which are part of the SOAP datatype specification.

Procedure 12.2.

Here is the procedure for installing `fpconst`:

1. Download the latest version of `fpconst` from <http://www.analytics.washington.edu/statcomp/projects/rzope/fpconst/>.
2. There are two downloads available, one in `.tar.gz` format, the other in `.zip` format. If you are using Windows, download the `.zip` file; otherwise, download the `.tar.gz` file.
3. Decompress the downloaded file. On Windows XP, you can right-click on the file and choose Extract All; on earlier versions of Windows, you will need a third-party program such as WinZip. On Mac OS X, you can double-click the compressed file to decompress it with Stuffit Expander.
4. Open a command prompt and navigate to the directory where you decompressed the `fpconst` files.
5. Type **`python setup.py install`** to run the installation program.

To verify that you installed `fpconst` correctly, run your Python IDE and check the version number.

Example 12.4. Verifying fpconst Installation

```
>>> import fpconst
>>> fpconst.__version__
'0.6.0'
```

This version number should match the version number of the `fpconst` archive you downloaded and installed.

12.2.3. Installing SOAPpy

The third and final requirement is the SOAP library itself: `SOAPpy`.

Procedure 12.3.

Here is the procedure for installing SOAPpy:

1. Go to <http://pywebsvcs.sourceforge.net/> and select Latest Official Release under the SOAPpy section.
2. There are two downloads available. If you are using Windows, download the .zip file; otherwise, download the .tar.gz file.
3. Decompress the downloaded file, just as you did with fpconst.
4. Open a command prompt and navigate to the directory where you decompressed the SOAPpy files.
5. Type **python setup.py install** to run the installation program.

To verify that you installed SOAPpy correctly, run your Python IDE and check the version number.

Example 12.5. Verifying SOAPpy Installation

```
>>> import SOAPpy
>>> SOAPpy.__version__
'0.11.4'
```

This version number should match the version number of the SOAPpy archive you downloaded and installed.

12.3. First Steps with SOAP

The heart of SOAP is the ability to call remote functions. There are a number of public access SOAP servers that provide simple functions for demonstration purposes.

The most popular public access SOAP server is <http://www.xmethods.net/>. This example uses a demonstration function that takes a United States zip code and returns the current temperature in that region.

Example 12.6. Getting the Current Temperature

```
>>> from SOAPpy import SOAPProxy          ❶
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> namespace = 'urn:xmethods-Temperature' ❷
>>> server = SOAPProxy(url, namespace)      ❸
>>> server.getTemp('27502')                ❹
80.0
```

- ❶ You access the remote SOAP server through a proxy class, `SOAPProxy`. The proxy handles all the internals of SOAP for you, including creating the XML request document out of the function name and argument list, sending the request over HTTP to the remote SOAP server, parsing the XML response document, and creating native Python values to return. You'll see what these XML documents look like in the next section.
- ❷ Every SOAP service has a URL which handles all the requests. The same URL is used for all function calls. This particular service only has a single function, but later in this chapter you'll see examples of the Google API, which has several functions. The service URL is shared by all functions. Each SOAP service also has a namespace, which is defined by the server and is completely arbitrary. It's simply part of the configuration required to call SOAP methods. It allows the server to share a single service URL and route requests between several unrelated services. It's like dividing Python modules into packages.
- ❸ You're creating the `SOAPProxy` with the service URL and the service namespace. This doesn't make any connection to the SOAP server; it simply creates a local Python object.

- ④ Now with everything configured properly, you can actually call remote SOAP methods as if they were local functions. You pass arguments just like a normal function, and you get a return value just like a normal function. But under the covers, there's a heck of a lot going on.

Let's peek under those covers.

12.4. Debugging SOAP Web Services

The SOAP libraries provide an easy way to see what's going on behind the scenes.

Turning on debugging is a simple matter of setting two flags in the SOAPProxy's configuration.

Example 12.7. Debugging SOAP Web Services

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> n = 'urn:xmethods-Temperature'
>>> server = SOAPProxy(url, namespace=n) ❶
>>> server.config.dumpSOAPOut = 1 ❷
>>> server.config.dumpSOAPIn = 1
>>> temperature = server.getTemp('27502') ❸

*** Outgoing SOAP *****
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">27502</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*****
*** Incoming SOAP *****
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">80.0</return>
</ns1:getTempResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*****

>>> temperature
80.0
```

- ❶ First, create the SOAPProxy like normal, with the service URL and the namespace.
- ❷ Second, turn on debugging by setting `server.config.dumpSOAPIn` and `server.config.dumpSOAPOut`.
- ❸ Third, call the remote SOAP method as usual. The SOAP library will print out both the outgoing XML request

document, and the incoming XML response document. This is all the hard work that SOAPProxy is doing for you. Intimidating, isn't it? Let's break it down.

Most of the XML request document that gets sent to the server is just boilerplate. Ignore all the namespace declarations; they're going to be the same (or similar) for all SOAP calls. The heart of the "function call" is this fragment within the `<Body>` element:

```
<ns1:getTemp                                ❶  
  xmlns:ns1="urn:xmethods-Temperature"      ❷  
  SOAP-ENC:root="1">  
<v1 xsi:type="xsd:string">27502</v1>        ❸  
</ns1:getTemp>
```

- ❶ The element name is the function name, `getTemp`. SOAPProxy uses `getattr` as a dispatcher. Instead of calling separate local methods based on the method name, it actually uses the method name to construct the XML request document.
- ❷ The function's XML element is contained in a specific namespace, which is the namespace you specified when you created the SOAPProxy object. Don't worry about the `SOAP-ENC:root`; that's boilerplate too.
- ❸ The arguments of the function also got translated into XML. SOAPProxy introspects each argument to determine its datatype (in this case it's a string). The argument datatype goes into the `xsi:type` attribute, followed by the actual string value.

The XML return document is equally easy to understand, once you know what to ignore. Focus on this fragment within the `<Body>`:

```
<ns1:getTempResponse                          ❶  
  xmlns:ns1="urn:xmethods-Temperature"      ❷  
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">  
<return xsi:type="xsd:float">80.0</return>    ❸  
</ns1:getTempResponse>
```

- ❶ The server wraps the function return value within a `<getTempResponse>` element. By convention, this wrapper element is the name of the function, plus `Response`. But it could really be almost anything; the important thing that SOAPProxy notices is not the element name, but the namespace.
- ❷ The server returns the response in the same namespace we used in the request, the same namespace we specified when we first create the SOAPProxy. Later in this chapter we'll see what happens if you forget to specify the namespace when creating the SOAPProxy.
- ❸ The return value is specified, along with its datatype (it's a float). SOAPProxy uses this explicit datatype to create a Python object of the correct native datatype and return it.

12.5. Introducing WSDL

The SOAPProxy class proxies local method calls and transparently turns them into invocations of remote SOAP methods. As you've seen, this is a lot of work, and SOAPProxy does it quickly and transparently. What it doesn't do is provide any means of method introspection.

Consider this: the previous two sections showed an example of calling a simple remote SOAP method with one argument and one return value, both of simple data types. This required knowing, and keeping track of, the service URL, the service namespace, the function name, the number of arguments, and the datatype of each argument. If any of these is missing or wrong, the whole thing falls apart.

That shouldn't come as a big surprise. If I wanted to call a local function, I would need to know what package or module it was in (the equivalent of service URL and namespace). I would need to know the correct function name and the correct number of arguments. Python deftly handles datatyping without explicit types, but I would still need to know how many arguments to pass, and how many return values to expect.

The big difference is introspection. As you saw in Chapter 4, Python excels at letting you discover things about modules and functions at runtime. You can list the available functions within a module, and with a little work, drill down to individual function declarations and arguments.

WSDL lets you do that with SOAP web services. WSDL stands for "Web Services Description Language". Although designed to be flexible enough to describe many types of web services, it is most often used to describe SOAP web services.

A WSDL file is just that: a file. More specifically, it's an XML file. It usually lives on the same server you use to access the SOAP web services it describes, although there's nothing special about it. Later in this chapter, we'll download the WSDL file for the Google API and use it locally. That doesn't mean we're calling Google locally; the WSDL file still describes the remote functions sitting on Google's server.

A WSDL file contains a description of everything involved in calling a SOAP web service:

- The service URL and namespace
- The type of web service (probably function calls using SOAP, although as I mentioned, WSDL is flexible enough to describe a wide variety of web services)
- The list of available functions
- The arguments for each function
- The datatype of each argument
- The return values of each function, and the datatype of each return value

In other words, a WSDL file tells you everything you need to know to be able to call a SOAP web service.

12.6. Introspecting SOAP Web Services with WSDL

Like many things in the web services arena, WSDL has a long and checkered history, full of political strife and intrigue. I will skip over this history entirely, since it bores me to tears. There were other standards that tried to do similar things, but WSDL won, so let's learn how to use it.

The most fundamental thing that WSDL allows you to do is discover the available methods offered by a SOAP server.

Example 12.8. Discovering The Available Methods

```
>>> from SOAPpy import WSDL           ❶
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl' )
>>> server = WSDL.Proxy(wsdlFile)      ❷
>>> server.methods.keys()               ❸
[u'getTemp']
```

- ❶ SOAPpy includes a WSDL parser. At the time of this writing, it was labeled as being in the early stages of development, but I had no problem parsing any of the WSDL files I tried.
- ❷ To use a WSDL file, you again use a proxy class, `WSDL.Proxy`, which takes a single argument: the WSDL file. Note that in this case you are passing in the URL of a WSDL file stored on the remote server, but the proxy class works just as well with a local copy of the WSDL file. The act of creating the WSDL proxy will download the WSDL file and parse it, so if there are any errors in the WSDL file (or it can't be fetched due to networking problems), you'll know about it immediately.
- ❸ The WSDL proxy class exposes the available functions as a Python dictionary, `server.methods`. So getting the list of available methods is as simple as calling the dictionary method `keys()`.

Okay, so you know that this SOAP server offers a single method: `getTemp`. But how do you call it? The WSDL proxy object can tell you that too.

Example 12.9. Discovering A Method's Arguments

```
>>> callInfo = server.methods['getTemp'] ❶
>>> callInfo.inparams ❷
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AD0>]
>>> callInfo.inparams[0].name ❸
u'zipcode'
>>> callInfo.inparams[0].type ❹
(u'http://www.w3.org/2001/XMLSchema', u'string')
```

- ❶ The `server.methods` dictionary is filled with a SOAPpy-specific structure called `CallInfo`. A `CallInfo` object contains information about one specific function, including the function arguments.
- ❷ The function arguments are stored in `callInfo.inparams`, which is a Python list of `ParameterInfo` objects that hold information about each parameter.
- ❸ Each `ParameterInfo` object contains a `name` attribute, which is the argument name. You are not required to know the argument name to call the function through SOAP, but SOAP does support calling functions with named arguments (just like Python), and `WSDL.Proxy` will correctly handle mapping named arguments to the remote function if you choose to use them.
- ❹ Each parameter is also explicitly typed, using datatypes defined in XML Schema. You saw this in the wire trace in the previous section; the XML Schema namespace was part of the "boilerplate" I told you to ignore. For our purposes here, you may continue to ignore it. The `zipcode` parameter is a string, and if you pass in a Python string to the `WSDL.Proxy` object, it will map it correctly and send it to the server.

WSDL also lets you introspect into a function's return values.

Example 12.10. Discovering A Method's Return Values

```
>>> callInfo.outparams ❶
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AF8>]
>>> callInfo.outparams[0].name ❷
u'return'
>>> callInfo.outparams[0].type
(u'http://www.w3.org/2001/XMLSchema', u'float')
```

- ❶ The adjunct to `callInfo.inparams` for function arguments is `callInfo.outparams` for return value. It is also a list, because functions called through SOAP can return multiple values, just like Python functions.
- ❷ Each `ParameterInfo` object contains `name` and `type`. This function returns a single value, named `return`, which is a float.

Let's put it all together, and call a SOAP web service through a WSDL proxy.

Example 12.11. Calling A Web Service Through A WSDL Proxy

```
>>> from SOAPpy import WSDL
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile) ❶
>>> server.getTemp('90210') ❷
66.0
>>> server.soaproxy.config.dumpSOAPOut = 1 ❸
>>> server.soaproxy.config.dumpSOAPIn = 1
>>> temperature = server.getTemp('90210')
```

```

*** Outgoing SOAP *****
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">90210</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*****
*** Incoming SOAP *****
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">66.0</return>
</ns1:getTempResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*****

>>> temperature
66.0

```

- ❶ The configuration is simpler than calling the SOAP service directly, since the WSDL file contains the both service URL and namespace you need to call the service. Creating the `WSDL.Proxy` object downloads the WSDL file, parses it, and configures a `SOAPProxy` object that it uses to call the actual SOAP web service.
- ❷ Once the `WSDL.Proxy` object is created, you can call a function as easily as you did with the `SOAPProxy` object. This is not surprising; the `WSDL.Proxy` is just a wrapper around the `SOAPProxy` with some introspection methods added, so the syntax for calling functions is the same.
- ❸ You can access the `WSDL.Proxy`'s `SOAPProxy` with `server.soapproxy`. This is useful for turning on debugging, so that when you can call functions through the WSDL proxy, its `SOAPProxy` will dump the outgoing and incoming XML documents that are going over the wire.

12.7. Searching Google

Let's finally turn to the sample code that you saw at the beginning of this chapter, which does something more useful and exciting than get the current temperature.

Google provides a SOAP API for programmatically accessing Google search results. To use it, you will need to sign up for Google Web Services.

Procedure 12.4. Signing Up for Google Web Services

1. Go to <http://www.google.com/apis/> and create a Google account. This requires only an email address. After you sign up you will receive your Google API license key by email. You will need this key to pass as a parameter whenever you call Google's search functions.
2. Also on <http://www.google.com/apis/>, download the Google Web APIs developer kit. This includes some sample code in several programming languages (but not Python), and more importantly, it includes the WSDL

file.

3. Decompress the developer kit file and find `GoogleSearch.wsdl`. Copy this file to some permanent location on your local drive. You will need it later in this chapter.

Once you have your developer key and your Google WSDL file in a known place, you can start poking around with Google Web Services.

Example 12.12. Introspecting Google Web Services

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl') ❶
>>> server.methods.keys() ❷
[u'doGoogleSearch', u'doGetCachedPage', u'doSpellingSuggestion']
>>> callInfo = server.methods['doGoogleSearch']
>>> for arg in callInfo.inparams: ❸
...     print arg.name.ljust(15), arg.type
key                (u'http://www.w3.org/2001/XMLSchema', u'string')
q                  (u'http://www.w3.org/2001/XMLSchema', u'string')
start              (u'http://www.w3.org/2001/XMLSchema', u'int')
maxResults         (u'http://www.w3.org/2001/XMLSchema', u'int')
filter             (u'http://www.w3.org/2001/XMLSchema', u'boolean')
restrict           (u'http://www.w3.org/2001/XMLSchema', u'string')
safeSearch         (u'http://www.w3.org/2001/XMLSchema', u'boolean')
lr                (u'http://www.w3.org/2001/XMLSchema', u'string')
ie                 (u'http://www.w3.org/2001/XMLSchema', u'string')
oe                 (u'http://www.w3.org/2001/XMLSchema', u'string')
```

- ❶ Getting started with Google web services is easy: just create a `WSDL.Proxy` object and point it at your local copy of Google's WSDL file.
- ❷ According to the WSDL file, Google offers three functions: `doGoogleSearch`, `doGetCachedPage`, and `doSpellingSuggestion`. These do exactly what they sound like: perform a Google search and return the results programmatically, get access to the cached version of a page from the last time Google saw it, and offer spelling suggestions for commonly misspelled search words.
- ❸ The `doGoogleSearch` function takes a number of parameters of various types. Note that while the WSDL file can tell you what the arguments are called and what datatype they are, it can't tell you what they mean or how to use them. It could theoretically tell you the acceptable range of values for each parameter, if only specific values were allowed, but Google's WSDL file is not that detailed. `WSDL.Proxy` can't work magic; it can only give you the information provided in the WSDL file.

Here is a brief synopsis of all the parameters to the `doGoogleSearch` function:

- `key` – Your Google API key, which you received when you signed up for Google web services.
- `q` – The search word or phrase you're looking for. The syntax is exactly the same as Google's web form, so if you know any advanced search syntax or tricks, they all work here as well.
- `start` – The index of the result to start on. Like the interactive web version of Google, this function returns 10 results at a time. If you wanted to get the second "page" of results, you would set `start` to 10.
- `maxResults` – The number of results to return. Currently capped at 10, although you can specify fewer if you are only interested in a few results and want to save a little bandwidth.
- `filter` – If `True`, Google will filter out duplicate pages from the results.
- `restrict` – Set this to `country` plus a country code to get results only from a particular country. Example: `countryUK` to search pages in the United Kingdom. You can also specify `linux`, `mac`, or `bsd` to search a Google-defined set of technical sites, or `unclesam` to search sites about the United States

government.

- `safeSearch` – If True, Google will filter out porn sites.
- `lr` ("language restrict") – Set this to a language code to get results only in a particular language.
- `ie` and `oe` ("input encoding" and "output encoding") – Deprecated, both must be `utf-8`.

Example 12.13. Searching Google

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl')
>>> key = 'YOUR_GOOGLE_API_KEY'
>>> results = server.doGoogleSearch(key, 'mark', 0, 10, False, "",
...     False, "", "utf-8", "utf-8")
>>> len(results.resultElements)
10
>>> results.resultElements[0].URL
'http://diveintomark.org/'
>>> results.resultElements[0].title
'dive into <b>mark</b>'
```

- ❶ After setting up the `WSDL.Proxy` object, you can call `server.doGoogleSearch` with all ten parameters. Remember to use your own Google API key that you received when you signed up for Google web services.
- ❷ There's a lot of information returned, but let's look at the actual search results first. They're stored in `results.resultElements`, and you can access them just like a normal Python list.
- ❸ Each element in the `resultElements` is an object that has a URL, title, snippet, and other useful attributes. At this point you can use normal Python introspection techniques like `dir(results.resultElements[0])` to see the available attributes. Or you can introspect through the WSDL proxy object and look through the function's `outparams`. Each technique will give you the same information.

The `results` object contains more than the actual search results. It also contains information about the search itself, such as how long it took and how many results were found (even though only 10 were returned). The Google web interface shows this information, and you can access it programmatically too.

Example 12.14. Accessing Secondary Information From Google

```
>>> results.searchTime
0.224919
>>> results.estimatedTotalResultsCount
29800000
>>> results.directoryCategories
[<SOAPpy.Types.structType item at 14367400>:
 {'fullViewableName':
  'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mark',
  'specialEncoding': ''}]
>>> results.directoryCategories[0].fullViewableName
'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mark'
```

- ❶ This search took 0.224919 seconds. That does not include the time spent sending and receiving the actual SOAP XML documents. It's just the time that Google spent processing your request once it received it.
- ❷ In total, there were approximately 30 million results. You can access them 10 at a time by changing the `start` parameter and calling `server.doGoogleSearch` again.
- ❸ For some queries, Google also returns a list of related categories in the Google Directory (<http://directory.google.com/>). You can append these URLs to <http://directory.google.com/> to

construct the link to the directory category page.

12.8. Troubleshooting SOAP Web Services

Of course, the world of SOAP web services is not all happiness and light. Sometimes things go wrong.

As you've seen throughout this chapter, SOAP involves several layers. There's the HTTP layer, since SOAP is sending XML documents to, and receiving XML documents from, an HTTP server. So all the debugging techniques you learned in Chapter 11, *HTTP Web Services* come into play here. You can **import** `httplib` and then set `httplib.HTTPConnection.debuglevel = 1` to see the underlying HTTP traffic.

Beyond the underlying HTTP layer, there are a number of things that can go wrong. SOAPpy does an admirable job hiding the SOAP syntax from you, but that also means it can be difficult to determine where the problem is when things don't work.

Here are a few examples of common mistakes that I've made in using SOAP web services, and the errors they generated.

Example 12.15. Calling a Method With an Incorrectly Configured Proxy

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> server = SOAPProxy(url)
>>> server.getTemp('27502')
```

1
2

```
<Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element namespaced?>
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element namespaced?>
```

- 1** Did you spot the mistake? You're creating a `SOAPProxy` manually, and you've correctly specified the service URL, but you haven't specified the namespace. Since multiple services may be routed through the same service URL, the namespace is essential to determine which service you're trying to talk to, and therefore which method you're really calling.
- 2** The server responds by sending a SOAP Fault, which SOAPpy turns into a Python exception of type `SOAPpy.Types.faultType`. All errors returned from any SOAP server will always be SOAP Faults, so you can easily catch this exception. In this case, the human-readable part of the SOAP Fault gives a clue to the problem: the method element is not namespaced, because the original `SOAPProxy` object was not configured with a service namespace.

Misconfiguring the basic elements of the SOAP service is one of the problems that WSDL aims to solve. The WSDL file contains the service URL and namespace, so you can't get it wrong. Of course, there are still other things you can get wrong.

Example 12.16. Calling a Method With the Wrong Arguments

```
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
```

```
>>> server = WSDL.Proxy(wsdlFile)
>>> temperature = server.getTemp(27502) ❶
<Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match> ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>
```

- ❶ Did you spot the mistake? It's a subtle one: you're calling `server.getTemp` with an integer instead of a string. As you saw from introspecting the WSDL file, the `getTemp()` SOAP function takes a single argument, `zipcode`, which must be a string. `WSDL.Proxy` will *not* coerce datatypes for you; you need to pass the exact datatypes that the server expects.
- ❷ Again, the server returns a SOAP Fault, and the human-readable part of the error gives a clue as to the problem: you're calling a `getTemp` function with an integer value, but there is no function defined with that name that takes an integer. In theory, SOAP allows you to *overload* functions, so you could have two functions in the same SOAP service with the same name and the same number of arguments, but the arguments were of different datatypes. This is why it's important to match the datatypes exactly, and why `WSDL.Proxy` doesn't coerce datatypes for you. If it did, you could end up calling a completely different function! Good luck debugging that one. It's much easier to be picky about datatypes and fail as quickly as possible if you get them wrong.

It's also possible to write Python code that expects a different number of return values than the remote function actually returns.

Example 12.17. Calling a Method and Expecting the Wrong Number of Return Values

```
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> (city, temperature) = server.getTemp(27502) ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unpack non-sequence
```

- ❶ Did you spot the mistake? `server.getTemp` only returns one value, a float, but you've written code that assumes you're getting two values and trying to assign them to two different variables. Note that this does not fail with a SOAP fault. As far as the remote server is concerned, nothing went wrong at all. The error only occurred *after* the SOAP transaction was complete, `WSDL.Proxy` returned a float, and your local Python interpreter tried to accommodate your request to split it into two different variables. Since the function only returned one value, you get a Python exception trying to split it, not a SOAP Fault.

What about Google's web service? The most common problem I've had with it is that I forget to set the application key properly.

Example 12.18. Calling a Method With An Application-Specific Error

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy(r'path/to/local/GoogleSearch.wsdl')
>>> results = server.doGoogleSearch('foo', 'mark', 0, 10, False, "", ❶
...     False, "", "utf-8", "utf-8")
```

```

<Fault SOAP-ENV:Server:
Exception from service object: Invalid authorization key: foo:
<SOAPPy.Types.structType detail at 14164616>:
{'stackTrace':
'com.google.soap.search.GoogleSearchFault: Invalid authorization key: foo
at com.google.soap.search.QueryLimits.lookupAndLoadFromINSIfNeedBe(
QueryLimits.java:220)
at com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
GoogleSearchService.java:825)
at com.google.soap.search.GoogleSearchService.doGoogleSearch(
GoogleSearchService.java:121)
at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at java.lang.reflect.Method.invoke(Unknown Source)
at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
at org.apache.soap.providers.RPCJavaProvider.invoke(
RPCJavaProvider.java:129)
at org.apache.soap.server.http.RPCRouterServlet.doPost(
RPCRouterServlet.java:288)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
at com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
at com.google.gse.HttpConnection.run(HttpConnection.java:195)
at com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of wrong size.
at com.google.soap.search.UserKey.<init>(UserKey.java:59)
at com.google.soap.search.QueryLimits.lookupAndLoadFromINSIfNeedBe(
QueryLimits.java:217)
... 14 more
'}>
Traceback (most recent call last):
File "<stdin>", line 1, in ?
File "c:\python23\Lib\site-packages\SOAPPy\Client.py", line 453, in __call__
return self.__r_call(*args, **kw)
File "c:\python23\Lib\site-packages\SOAPPy\Client.py", line 475, in __r_call
self.__hd, self.__ma)
File "c:\python23\Lib\site-packages\SOAPPy\Client.py", line 389, in __call
raise p
SOAPPy.Types.faultType: <Fault SOAP-ENV:Server: Exception from service object:
Invalid authorization key: foo:
<SOAPPy.Types.structType detail at 14164616>:
{'stackTrace':
'com.google.soap.search.GoogleSearchFault: Invalid authorization key: foo
at com.google.soap.search.QueryLimits.lookupAndLoadFromINSIfNeedBe(
QueryLimits.java:220)
at com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
GoogleSearchService.java:825)
at com.google.soap.search.GoogleSearchService.doGoogleSearch(
GoogleSearchService.java:121)
at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at java.lang.reflect.Method.invoke(Unknown Source)
at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
at org.apache.soap.providers.RPCJavaProvider.invoke(
RPCJavaProvider.java:129)
at org.apache.soap.server.http.RPCRouterServlet.doPost(
RPCRouterServlet.java:288)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
at com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
at com.google.gse.HttpConnection.run(HttpConnection.java:195)

```

```
at com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of wrong size.
at com.google.soap.search.UserKey.<init>(UserKey.java:59)
at com.google.soap.search.QueryLimits.lookupAndLoadFromINSIfNeedBe(
    QueryLimits.java:217)
... 14 more
'}>
```

- ❶ Can you spot the mistake? There's nothing wrong with the calling syntax, or the number of arguments, or the datatypes. The problem is application-specific: the first argument is supposed to be my application key, but `foo` is not a valid Google key.
- ❷ The Google server responds with a SOAP Fault and an incredibly long error message, which includes a complete Java stack trace. Remember that *all* SOAP errors are signified by SOAP Faults: errors in configuration, errors in function arguments, and application-specific errors like this. Buried in there somewhere is the crucial piece of information: `Invalid authorization key: foo`.

Further Reading on Troubleshooting SOAP

- New developments for SOAPpy (<http://www-106.ibm.com/developerworks/webservices/library/ws-pyth17.html>) steps through trying to connect to another SOAP service that doesn't quite work as advertised.

12.9. Summary

SOAP web services are very complicated. The specification is very ambitious and tries to cover many different use cases for web services. This chapter has touched on some of the simpler use cases.

Before diving into the next chapter, make sure you're comfortable doing all of these things:

- Connecting to a SOAP server and calling remote methods
- Loading a WSDL file and introspecting remote methods
- Debugging SOAP calls with wire traces
- Troubleshooting common SOAP-related errors

Chapter 13. Unit Testing

13.1. Introduction to Roman numerals

In previous chapters, you "dived in" by immediately looking at code and trying to understand it as quickly as possible. Now that you have some Python under your belt, you're going to step back and look at the steps that happen *before* the code gets written.

In the next few chapters, you're going to write, debug, and optimize a set of utility functions to convert to and from Roman numerals. You saw the mechanics of constructing and validating Roman numerals in Section 7.3, Case Study: Roman Numerals, but now let's step back and consider what it would take to expand that into a two-way utility.

The rules for Roman numerals lead to a number of interesting observations:

1. There is only one correct way to represent a particular number as Roman numerals.
2. The converse is also true: if a string of characters is a valid Roman numeral, it represents only one number (*i.e.* it can only be read one way).
3. There is a limited range of numbers that can be expressed as Roman numerals, specifically 1 through 3999. (The Romans did have several ways of expressing larger numbers, for instance by having a bar over a numeral to represent that its normal value should be multiplied by 1000, but you're not going to deal with that. For the purposes of this chapter, let's stipulate that Roman numerals go from 1 to 3999.)
4. There is no way to represent 0 in Roman numerals. (Amazingly, the ancient Romans had no concept of 0 as a number. Numbers were for counting things you had; how can you count what you don't have?)
5. There is no way to represent negative numbers in Roman numerals.
6. There is no way to represent fractions or non-integer numbers in Roman numerals.

Given all of this, what would you expect out of a set of functions to convert to and from Roman numerals?

roman.py requirements

1. `toRoman` should return the Roman numeral representation for all integers 1 to 3999.
2. `toRoman` should fail when given an integer outside the range 1 to 3999.
3. `toRoman` should fail when given a non-integer number.
4. `fromRoman` should take a valid Roman numeral and return the number that it represents.
5. `fromRoman` should fail when given an invalid Roman numeral.
6. If you take a number, convert it to Roman numerals, then convert that back to a number, you should end up with the number you started with. So `fromRoman(toRoman(n)) == n` for all `n` in `1..3999`.
7. `toRoman` should always return a Roman numeral using uppercase letters.
8. `fromRoman` should only accept uppercase Roman numerals (*i.e.* it should fail when given lowercase input).

Further reading

- This site (<http://www.wilkiecollins.demon.co.uk/roman/front.htm>) has more on Roman numerals, including a fascinating history (<http://www.wilkiecollins.demon.co.uk/roman/intro.htm>) of how Romans and other civilizations really used them (short answer: haphazardly and inconsistently).

13.2. Diving in

Now that you've completely defined the behavior you expect from your conversion functions, you're going to do something a little unexpected: you're going to write a test suite that puts these functions through their paces and makes sure that they behave the way you want them to. You read that right: you're going to write code that tests code that you haven't written yet.

This is called unit testing, since the set of two conversion functions can be written and tested as a unit, separate from any larger program they may become part of later. Python has a framework for unit testing, the appropriately-named `unittest` module.

`unittest` is included with Python 2.1 and later. Python 2.0 users can download it from pyunit.sourceforge.net (<http://pyunit.sourceforge.net/>).

Unit testing is an important part of an overall testing-centric development strategy. If you write unit tests, it is important to write them early (preferably before writing the code that they test), and to keep them updated as code and requirements change. Unit testing is not a replacement for higher-level functional or system testing, but it is important in all phases of development:

- Before writing code, it forces you to detail your requirements in a useful fashion.
- While writing code, it keeps you from over-coding. When all the test cases pass, the function is complete.
- When refactoring code, it assures you that the new version behaves the same way as the old version.
- When maintaining code, it helps you cover your ass when someone comes screaming that your latest change broke their old code. ("But *sir*, all the unit tests passed when I checked it in...")
- When writing code in a team, it increases confidence that the code you're about to commit isn't going to break other peoples' code, because you can run their unittests first. (I've seen this sort of thing in code sprints. A team breaks up the assignment, everybody takes the specs for their task, writes unit tests for it, then shares their unit tests with the rest of the team. That way, nobody goes off too far into developing code that won't play well with others.)

13.3. Introducing `romantest.py`

This is the complete test suite for your Roman numeral conversion functions, which are yet to be written but will eventually be in `roman.py`. It is not immediately obvious how it all fits together; none of these classes or methods reference any of the others. There are good reasons for this, as you'll see shortly.

Example 13.1. `romantest.py`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
"""Unit test for roman.py"""

import roman
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
```

```

(6, 'VI'),
(7, 'VII'),
(8, 'VIII'),
(9, 'IX'),
(10, 'X'),
(50, 'L'),
(100, 'C'),
(500, 'D'),
(1000, 'M'),
(31, 'XXXI'),
(148, 'CXLVIII'),
(294, 'CCXCIV'),
(312, 'CCCXII'),
(421, 'CDXXI'),
(528, 'DXXVIII'),
(621, 'DCXXI'),
(782, 'DCCLXXXII'),
(870, 'DCCCLXX'),
(941, 'CMXLI'),
(1043, 'MXLIII'),
(1110, 'MCX'),
(1226, 'MCCXXVI'),
(1301, 'MCCCI'),
(1485, 'MCDLXXXV'),
(1509, 'MDIX'),
(1607, 'MDCVII'),
(1754, 'MDCCCLIV'),
(1832, 'MDCCCXXXII'),
(1993, 'MCMXCIII'),
(2074, 'MMLXXIV'),
(2152, 'MMCLII'),
(2212, 'MMCCXII'),
(2343, 'MMCCCXLIII'),
(2499, 'MMCDXCIX'),
(2574, 'MMDLXXIV'),
(2646, 'MMDCXLVI'),
(2723, 'MMDCCXIII'),
(2892, 'MMDCCCXCII'),
(2975, 'MMCMLXXV'),
(3051, 'MMMLI'),
(3185, 'MMMCLXXXV'),
(3250, 'MMMCCCL'),
(3313, 'MMMCCCXIII'),
(3408, 'MMMCDVIII'),
(3501, 'MMMDI'),
(3610, 'MMMDCX'),
(3743, 'MMMDCCXLIII'),
(3844, 'MMMDCCCXLIV'),
(3888, 'MMMDCCCLXXXVIII'),
(3940, 'MMMCMXL'),
(3999, 'MMMCMXCIX'))

```

```

def testToRomanKnownValues(self):
    """toRoman should give known result with known input"""
    for integer, numeral in self.knownValues:
        result = roman.toRoman(integer)
        self.assertEqual(numeral, result)

def testFromRomanKnownValues(self):
    """fromRoman should give known result with known input"""
    for integer, numeral in self.knownValues:
        result = roman.fromRoman(numeral)
        self.assertEqual(integer, result)

```

```

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()

```

Further reading

- The PyUnit home page (<http://pyunit.sourceforge.net/>) has an in-depth discussion of using the `unittest` framework (<http://pyunit.sourceforge.net/pyunit.html>), including advanced features not covered in this chapter.
- The PyUnit FAQ (<http://pyunit.sourceforge.net/pyunit.html>) explains why test cases are stored separately (<http://pyunit.sourceforge.net/pyunit.html#WHERE>) from the code they test.
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes the `unittest` (<http://www.python.org/doc/current/lib/module-unittest.html>) module.
- ExtremeProgramming.org (<http://www.extremeprogramming.org/>) discusses why you should write unit tests (<http://www.extremeprogramming.org/rules/unittests.html>).
- The Portland Pattern Repository (<http://www.c2.com/cgi/wiki>) has an ongoing discussion of unit tests (<http://www.c2.com/cgi/wiki?UnitTests>), including a standard definition (<http://www.c2.com/cgi/wiki?StandardDefinitionOfUnitTest>), why you should code unit tests first (<http://www.c2.com/cgi/wiki?CodeUnitTestFirst>), and several in-depth case studies (<http://www.c2.com/cgi/wiki?UnitTestTrial>).

13.4. Testing for success

The most fundamental part of unit testing is constructing individual test cases. A test case answers a single question about the code it is testing.

A test case should be able to...

- ...run completely by itself, without any human input. Unit testing is about automation.
- ...determine by itself whether the function it is testing has passed or failed, without a human interpreting the results.
- ...run in isolation, separate from any other test cases (even if they test the same functions). Each test case is an island.

Given that, let's build the first test case. You have the following requirement:

1. `toRoman` should return the Roman numeral representation for all integers 1 to 3999.

Example 13.2. `testToRomanKnownValues`

```
class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
```

❶

```

        (621, 'DCXXI'),
        (782, 'DCCLXXXII'),
        (870, 'DCCCLXX'),
        (941, 'CMXLI'),
        (1043, 'MXLIII'),
        (1110, 'MCX'),
        (1226, 'MCCXXVI'),
        (1301, 'MCCCI'),
        (1485, 'MCDLXXXV'),
        (1509, 'MDIX'),
        (1607, 'MDCVII'),
        (1754, 'MDCCLIV'),
        (1832, 'MDCCCXXXII'),
        (1993, 'MCMXCIII'),
        (2074, 'MMLXXIV'),
        (2152, 'MMCLII'),
        (2212, 'MMCCXII'),
        (2343, 'MMCCCXLIII'),
        (2499, 'MMCDXCIX'),
        (2574, 'MMDLXXIV'),
        (2646, 'MMDCLVI'),
        (2723, 'MMDCCXIII'),
        (2892, 'MMDCCCXCII'),
        (2975, 'MMCMLXXV'),
        (3051, 'MMMLI'),
        (3185, 'MMMCLXXXV'),
        (3250, 'MMMCCCL'),
        (3313, 'MMMCCCXIII'),
        (3408, 'MMMCDVIII'),
        (3501, 'MMM DI'),
        (3610, 'MMMDCX'),
        (3743, 'MMMDCCXLIII'),
        (3844, 'MMMDCCCXLIV'),
        (3888, 'MMMDCCCLXXXVIII'),
        (3940, 'MMMCMXL'),
        (3999, 'MMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)
            self.assertEqual(numeral, result)

```

- ❶ To write a test case, first subclass the `TestCase` class of the `unittest` module. This class provides many useful methods which you can use in your test case to test specific conditions.
- ❷ This is a list of integer/numeral pairs that I verified manually. It includes the lowest ten numbers, the highest number, every number that translates to a single-character Roman numeral, and a random sampling of other valid numbers. The point of a unit test is not to test every possible input, but to test a representative sample.
- ❸ Every individual test is its own method, which must take no parameters and return no value. If the method exits normally without raising an exception, the test is considered passed; if the method raises an exception, the test is considered failed.
- ❹ Here you call the actual `toRoman` function. (Well, the function hasn't be written yet, but once it is, this is the line that will call it.) Notice that you have now defined the API for the `toRoman` function: it must take an integer (the number to convert) and return a string (the Roman numeral representation). If the API is different than that, this test is considered failed.
- ❺ Also notice that you are not trapping any exceptions when you call `toRoman`. This is intentional. `toRoman` shouldn't raise an exception when you call it with valid input, and these input values are all valid. If `toRoman` raises an exception, this test is considered failed.

- ⑥ Assuming the `toRoman` function was defined correctly, called correctly, completed successfully, and returned a value, the last step is to check whether it returned the *right* value. This is a common question, and the `TestCase` class provides a method, `assertEqual`, to check whether two values are equal. If the result returned from `toRoman` (`result`) does not match the known value you were expecting (`numeral`), `assertEqual` will raise an exception and the test will fail. If the two values are equal, `assertEqual` will do nothing. If every value returned from `toRoman` matches the known value you expect, `assertEqual` never raises an exception, so `testToRomanKnownValues` eventually exits normally, which means `toRoman` has passed this test.

13.5. Testing for failure

It is not enough to test that functions succeed when given good input; you must also test that they fail when given bad input. And not just any sort of failure; they must fail in the way you expect.

Remember the other requirements for `toRoman`:

2. `toRoman` should fail when given an integer outside the range 1 to 3999.
3. `toRoman` should fail when given a non-integer number.

In Python, functions indicate failure by raising exceptions, and the `unittest` module provides methods for testing whether a function raises a particular exception when given bad input.

Example 13.3. Testing bad input to `toRoman`

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000) ❶

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0) ❷

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5) ❸
```

- ❶ The `TestCase` class of the `unittest` provides the `assertRaises` method, which takes the following arguments: the exception you're expecting, the function you're testing, and the arguments you're passing that function. (If the function you're testing takes more than one argument, pass them all to `assertRaises`, in order, and it will pass them right along to the function you're testing.) Pay close attention to what you're doing here: instead of calling `toRoman` directly and manually checking that it raises a particular exception (by wrapping it in a `try...except` block), `assertRaises` has encapsulated all of that for us. All you do is give it the exception (`roman.OutOfRangeError`), the function (`toRoman`), and `toRoman`'s arguments (`4000`), and `assertRaises` takes care of calling `toRoman` and checking to make sure that it raises `roman.OutOfRangeError`. (Also note that you're passing the `toRoman` function itself as an argument; you're not calling it, and you're not passing the name of it as a string. Have I mentioned recently how handy it is that everything in Python is an object, including functions and exceptions?)

- ② Along with testing numbers that are too large, you need to test numbers that are too small. Remember, Roman numerals cannot express 0 or negative numbers, so you have a test case for each of those (`testZero` and `testNegative`). In `testZero`, you are testing that `toRoman` raises a `roman.OutOfRangeError` exception when called with 0; if it does *not* raise a `roman.OutOfRangeError` (either because it returns an actual value, or because it raises some other exception), this test is considered failed.
- ③ Requirement #3 specifies that `toRoman` cannot accept a non-integer number, so here you test to make sure that `toRoman` raises a `roman.NotIntegerError` exception when called with 0.5. If `toRoman` does not raise a `roman.NotIntegerError`, this test is considered failed.

The next two requirements are similar to the first three, except they apply to `fromRoman` instead of `toRoman`:

4. `fromRoman` should take a valid Roman numeral and return the number that it represents.
5. `fromRoman` should fail when given an invalid Roman numeral.

Requirement #4 is handled in the same way as requirement #1, iterating through a sampling of known values and testing each in turn. Requirement #5 is handled in the same way as requirements #2 and #3, by testing a series of bad inputs and making sure `fromRoman` raises the appropriate exception.

Example 13.4. Testing bad input to `fromRoman`

```
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s) ❶

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
```

- ❶ Not much new to say about these; the pattern is exactly the same as the one you used to test bad input to `toRoman`. I will briefly note that you have another exception: `roman.InvalidRomanNumeralError`. That makes a total of three custom exceptions that will need to be defined in `roman.py` (along with `roman.OutOfRangeError` and `roman.NotIntegerError`). You'll see how to define these custom exceptions when you actually start writing `roman.py`, later in this chapter.

13.6. Testing for sanity

Often, you will find that a unit of code contains a set of reciprocal functions, usually in the form of conversion functions where one converts A to B and the other converts B to A. In these cases, it is useful to create a "sanity check" to make sure that you can convert A to B and back to A without losing precision, incurring rounding errors, or triggering any other sort of bug.

Consider this requirement:

6. If you take a number, convert it to Roman numerals, then convert that back to a number, you should end up with the number you started with. So `fromRoman(toRoman(n)) == n` for all `n` in `1..3999`.

Example 13.5. Testing `toRoman` against `fromRoman`

```
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000): ❶ ❷
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result) ❸
```

- ❶ You've seen the `range` function before, but here it is called with two arguments, which returns a list of integers starting at the first argument (1) and counting consecutively up to *but not including* the second argument (4000). Thus, `1..3999`, which is the valid range for converting to Roman numerals.
- ❷ I just wanted to mention in passing that `integer` is not a keyword in Python; here it's just a variable name like any other.
- ❸ The actual testing logic here is straightforward: take a number (`integer`), convert it to a Roman numeral (`numeral`), then convert it back to a number (`result`) and make sure you end up with the same number you started with. If not, `assertEqual` will raise an exception and the test will immediately be considered failed. If all the numbers match, `assertEqual` will always return silently, the entire `testSanity` method will eventually return silently, and the test will be considered passed.

The last two requirements are different from the others because they seem both arbitrary and trivial:

- 7. `toRoman` should always return a Roman numeral using uppercase letters.
- 8. `fromRoman` should only accept uppercase Roman numerals (*i.e.* it should fail when given lowercase input).

In fact, they are somewhat arbitrary. You could, for instance, have stipulated that `fromRoman` accept lowercase and mixed case input. But they are not completely arbitrary; if `toRoman` is always returning uppercase output, then `fromRoman` must at least accept uppercase input, or the "sanity check" (requirement #6) would fail. The fact that it *only* accepts uppercase input is arbitrary, but as any systems integrator will tell you, case always matters, so it's worth specifying the behavior up front. And if it's worth specifying, it's worth testing.

Example 13.6. Testing for case

```
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper()) ❶

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper()) ❷ ❸
            self.assertRaises(roman.InvalidRomanNumeralError, ❹
                              roman.fromRoman, numeral.lower())
```

- ❶ The most interesting thing about this test case is all the things it doesn't test. It doesn't test that the value returned from `toRoman` is right or even consistent; those questions are answered by separate test cases. You have a whole test case just to test for uppercase-ness. You might be tempted to combine this with the sanity check, since both run through the entire range of values and call `toRoman`.^[6] But that would violate one of the fundamental rules: each test case should answer only a single question. Imagine that you combined this case check with the sanity check, and then that test case failed. You would need to do further analysis to figure out which part of the test case failed to determine what the problem was. If you need to analyze the results of your unit testing just to figure out what they mean, it's a sure sign that you've mis-designed your test cases.
- ❷ There's a similar lesson to be learned here: even though "you know" that `toRoman` always returns uppercase, you are explicitly converting its return value to uppercase here to test that `fromRoman` accepts uppercase input. Why? Because the fact that `toRoman` always returns uppercase is an independent requirement. If you changed that requirement so that, for instance, it always returned lowercase, the `testToRomanCase` test case would need to change, but this test case would still work. This was another of the fundamental rules: each test case must be able to work in isolation from any of the others. Every test case is an island.
- ❸ Note that you're not assigning the return value of `fromRoman` to anything. This is legal syntax in Python; if a function returns a value but nobody's listening, Python just throws away the return value. In this case, that's what you want. This test case doesn't test anything about the return value; it just tests that `fromRoman` accepts the uppercase input without raising an exception.
- ❹ This is a complicated line, but it's very similar to what you did in the `ToRomanBadInput` and `FromRomanBadInput` tests. You are testing to make sure that calling a particular function (`roman.fromRoman`) with a particular value (`numeral.lower()`, the lowercase version of the current Roman numeral in the loop) raises a particular exception (`roman.InvalidRomanNumeralError`). If it does (each time through the loop), the test passes; if even one time it does something else (like raises a different exception, or returning a value without raising an exception at all), the test fails.

In the next chapter, you'll see how to write code that passes these tests.

^[6] "I can resist everything except temptation." —Oscar Wilde

Chapter 14. Test–First Programming

14.1. roman.py, stage 1

Now that the unit tests are complete, it's time to start writing the code that the test cases are attempting to test. You're going to do this in stages, so you can see all the unit tests fail, then watch them pass one by one as you fill in the gaps in `roman.py`.

Example 14.1. roman1.py

This file is available in `py/roman/stage1/` in the examples directory.

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass ❶
class OutOfRangeError(RomanError): pass ❷
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass ❸

def toRoman(n):
    """convert integer to Roman numeral"""
    pass ❹

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

- ❶ This is how you define your own custom exceptions in Python. Exceptions are classes, and you create your own by subclassing existing exceptions. It is strongly recommended (but not required) that you subclass `Exception`, which is the base class that all built-in exceptions inherit from. Here I am defining `RomanError` (inherited from `Exception`) to act as the base class for all my other custom exceptions to follow. This is a matter of style; I could just as easily have inherited each individual exception from the `Exception` class directly.
- ❷ The `OutOfRangeError` and `NotIntegerError` exceptions will eventually be used by `toRoman` to flag various forms of invalid input, as specified in `ToRomanBadInput`.
- ❸ The `InvalidRomanNumeralError` exception will eventually be used by `fromRoman` to flag invalid input, as specified in `FromRomanBadInput`.
- ❹ At this stage, you want to define the API of each of your functions, but you don't want to code them yet, so you stub them out using the Python reserved word `pass`.

Now for the big moment (drum roll please): you're finally going to run the unit test against this stubby little module. At this point, every test case should fail. In fact, if any test case passes in stage 1, you should go back to `romantest.py` and re-evaluate why you coded a test so useless that it passes with do-nothing functions.

Run `romantest1.py` with the `-v` command-line option, which will give more verbose output so you can see exactly what's going on as each test case runs. With any luck, your output should look like this:

Example 14.2. Output of `romantest1.py` against `roman1.py`

```

fromRoman should only accept uppercase input ... ERROR
toRoman should always return uppercase ... ERROR
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... FAIL
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL

```

```

=====
ERROR: fromRoman should only accept uppercase input
-----

```

```

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 154, in testFromRomanCase
    roman1.fromRoman(numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
=====

```

```

ERROR: toRoman should always return uppercase
-----

```

```

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 148, in testToRomanCase
    self.assertEqual(numeral, numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
=====

```

```

FAIL: fromRoman should fail with malformed antecedents
-----

```

```

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====

```

```

FAIL: fromRoman should fail with repeated pairs of numerals
-----

```

```

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 127, in testRepeatedPairs
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====

```

```

FAIL: fromRoman should fail with too many repeated numerals
-----

```

```

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====

```

```

FAIL: fromRoman should give known result with known input
-----

```

```

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None

```



```

=====
FAIL: toRoman should give known result with known input
-----
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 93, in testToRomanKnownValues
    self.assertEqual(numeral, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: I != None
=====
FAIL: fromRoman(toRoman(n))==n for all n
-----
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
=====
FAIL: toRoman should fail with non-integer input
-----
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 116, in testNonInteger
    self.assertRaises(roman1.NotIntegerError, roman1.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
=====
FAIL: toRoman should fail with negative input
-----
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 112, in testNegative
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
=====
FAIL: toRoman should fail with large input
-----
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 104, in testTooLarge
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
=====
FAIL: toRoman should fail with 0 input
-----
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stagel\romantest1.py", line 108, in testZero
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
-----
Ran 12 tests in 0.040s
-----
FAILED (failures=10, errors=2)

```

❶

❷

❸

❹

- ❶ Running the script runs `unittest.main()`, which runs each test case, which is to say each method defined in each class within `romantest.py`. For each test case, it prints out the doc string of the method and whether that test passed or failed. As expected, none of the test cases passed.

- ❷ For each failed test case, unittest displays the trace information showing exactly what happened. In this case, the call to `assertRaises` (also called `failUnlessRaises`) raised an `AssertionError` because it was expecting `toRoman` to raise an `OutOfRangeError` and it didn't.
- ❸ After the detail, unittest displays a summary of how many tests were performed and how long it took.
- ❹ Overall, the unit test failed because at least one test case did not pass. When a test case doesn't pass, unittest distinguishes between failures and errors. A failure is a call to an `assertXYZ` method, like `assertEqual` or `assertRaises`, that fails because the asserted condition is not true or the expected exception was not raised. An error is any other sort of exception raised in the code you're testing or the unit test case itself. For instance, the `testFromRomanCase` method ("`fromRoman` should only accept uppercase input") was an error, because the call to `numeral.upper()` raised an `AttributeError` exception, because `toRoman` was supposed to return a string but didn't. But `testZero` ("`toRoman` should fail with 0 input") was a failure, because the call to `fromRoman` did not raise the `InvalidRomanNumeral` exception that `assertRaises` was looking for.

14.2. roman.py, stage 2

Now that you have the framework of the `roman` module laid out, it's time to start writing code and passing test cases.

Example 14.3. roman2.py

This file is available in `py/roman/stage2/` in the examples directory.

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M', 1000), ❶
                    ('CM', 900),
                    ('D', 500),
                    ('CD', 400),
                    ('C', 100),
                    ('XC', 90),
                    ('L', 50),
                    ('XL', 40),
                    ('X', 10),
                    ('IX', 9),
                    ('V', 5),
                    ('IV', 4),
                    ('I', 1))

def toRoman(n):
    """convert integer to Roman numeral"""
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer: ❷
            result += numeral
            n -= integer
    return result
```

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

❶ `romanNumeralMap` is a tuple of tuples which defines three things:

1. The character representations of the most basic Roman numerals. Note that this is not just the single-character Roman numerals; you're also defining two-character pairs like CM ("one hundred less than one thousand"); this will make the `toRoman` code simpler later.
2. The order of the Roman numerals. They are listed in descending value order, from M all the way down to I.
3. The value of each Roman numeral. Each inner tuple is a pair of (*numeral*, *value*).

❷ Here's where your rich data structure pays off, because you don't need any special logic to handle the subtraction rule. To convert to Roman numerals, you simply iterate through `romanNumeralMap` looking for the largest integer value less than or equal to the input. Once found, you add the Roman numeral representation to the end of the output, subtract the corresponding integer value from the input, lather, rinse, repeat.

Example 14.4. How `toRoman` works

If you're not clear how `toRoman` works, add a `print` statement to the end of the `while` loop:

```
while n >= integer:
    result += numeral
    n -= integer
    print 'subtracting', integer, 'from input, adding', numeral, 'to output'
```

```
>>> import roman2
>>> roman2.toRoman(1424)
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

So `toRoman` appears to work, at least in this manual spot check. But will it pass the unit testing? Well no, not entirely.

Example 14.5. Output of `romantest2.py` against `roman2.py`

Remember to run `romantest2.py` with the `-v` command-line flag to enable verbose mode.

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL
```

❶

❷

❸

- ❶ toRoman does, in fact, always return uppercase, because romanNumeralMap defines the Roman numeral representations as uppercase. So this test passes already.
- ❷ Here's the big news: this version of the toRoman function passes the known values test. Remember, it's not comprehensive, but it does put the function through its paces with a variety of good inputs, including inputs that produce every single-character Roman numeral, the largest possible input (3999), and the input that produces the longest possible Roman numeral (3888). At this point, you can be reasonably confident that the function works for any good input value you could throw at it.
- ❸ However, the function does not "work" for bad values; it fails every single bad input test. That makes sense, because you didn't include any checks for bad input. Those test cases look for specific exceptions to be raised (via assertRaises), and you're never raising them. You'll do that in the next stage.

Here's the rest of the output of the unit test, listing the details of all the failures. You're down to 10.

```
=====
FAIL: fromRoman should only accept uppercase input
=====
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 156, in testFromRomanCase
    roman2.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
FAIL: fromRoman should fail with malformed antecedents
=====
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
FAIL: fromRoman should fail with repeated pairs of numerals
=====
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 127, in testRepeatedPairs
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
FAIL: fromRoman should fail with too many repeated numerals
=====
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
FAIL: fromRoman should give known result with known input
=====
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
=====
FAIL: fromRoman(toRoman(n))==n for all n
=====
```

```

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
=====
FAIL: toRoman should fail with non-integer input
-----

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 116, in testNonInteger
    self.assertRaises(roman2.NotIntegerError, roman2.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
=====
FAIL: toRoman should fail with negative input
-----

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 112, in testNegative
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
=====
FAIL: toRoman should fail with large input
-----

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 104, in testTooLarge
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
=====
FAIL: toRoman should fail with 0 input
-----

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 108, in testZero
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
-----

Ran 12 tests in 0.320s

FAILED (failures=10)

```

14.3. roman.py, stage 3

Now that `toRoman` behaves correctly with good input (integers from 1 to 3999), it's time to make it behave correctly with bad input (everything else).

Example 14.6. roman3.py

This file is available in `py/roman/stage3/` in the examples directory.

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```

"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M', 1000),
                    ('CM', 900),
                    ('D', 500),
                    ('CD', 400),
                    ('C', 100),
                    ('XC', 90),
                    ('L', 50),
                    ('XL', 40),
                    ('X', 10),
                    ('IX', 9),
                    ('V', 5),
                    ('IV', 4),
                    ('I', 1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError, "number out of range (must be 1..3999)"
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass

```

- ❶ This is a nice Pythonic shortcut: multiple comparisons at once. This is equivalent to `if not ((0 < n) and (n < 4000))`, but it's much easier to read. This is the range check, and it should catch inputs that are too large, negative, or zero.
- ❷ You raise exceptions yourself with the `raise` statement. You can raise any of the built-in exceptions, or you can raise any of your custom exceptions that you've defined. The second parameter, the error message, is optional; if given, it is displayed in the traceback that is printed if the exception is never handled.
- ❸ This is the non-integer check. Non-integers can not be converted to Roman numerals.
- ❹ The rest of the function is unchanged.

Example 14.7. Watching `toRoman` handle bad input

```

>>> import roman3
>>> roman3.toRoman(4000)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 27, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
>>> roman3.toRoman(1.5)

```

```
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 29, in toRoman
    raise NotIntegerError, "non-integers can not be converted"
NotIntegerError: non-integers can not be converted
```

Example 14.8. Output of `romantest3.py` against `roman3.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok ❶
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... ok ❷
toRoman should fail with negative input ... ok ❸
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

- ❶ `toRoman` still passes the known values test, which is comforting. All the tests that passed in stage 2 still pass, so the latest code hasn't broken anything.
- ❷ More exciting is the fact that all of the bad input tests now pass. This test, `testNonInteger`, passes because of the `int(n) <> n` check. When a non-integer is passed to `toRoman`, the `int(n) <> n` check notices it and raises the `NotIntegerError` exception, which is what `testNonInteger` is looking for.
- ❸ This test, `testNegative`, passes because of the `not (0 < n < 4000)` check, which raises an `OutOfRangeError` exception, which is what `testNegative` is looking for.

```
=====
FAIL: fromRoman should only accept uppercase input
-----
```

```
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 156, in testFromRomanCase
    roman3.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
```

```
FAIL: fromRoman should fail with malformed antecedents
-----
```

```
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
```

```
FAIL: fromRoman should fail with repeated pairs of numerals
-----
```

```
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 127, in testRepeatedPairs
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
```

```
FAIL: fromRoman should fail with too many repeated numerals
-----
```

```
Traceback (most recent call last):
```

```

File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
FAIL: fromRoman should give known result with known input
-----

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
=====
FAIL: fromRoman(toRoman(n))==n for all n
-----

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
-----

Ran 12 tests in 0.401s

FAILED (failures=6) ❶

```

- ❶ You're down to 6 failures, and all of them involve `fromRoman`: the known values test, the three separate bad input tests, the case check, and the sanity check. That means that `toRoman` has passed all the tests it can pass by itself. (It's involved in the sanity check, but that also requires that `fromRoman` be written, which it isn't yet.) Which means that you must stop coding `toRoman` now. No tweaking, no twiddling, no extra checks "just in case". Stop. Now. Back away from the keyboard.

The most important thing that comprehensive unit testing can tell you is when to stop coding. When all the unit tests for a function pass, stop coding the function. When all the unit tests for an entire module pass, stop coding the module.

14.4. roman.py, stage 4

Now that `toRoman` is done, it's time to start coding `fromRoman`. Thanks to the rich data structure that maps individual Roman numerals to integer values, this is no more difficult than the `toRoman` function.

Example 14.9. roman4.py

This file is available in `py/roman/stage4/` in the examples directory.

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```

"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

```



```
#Define digit mapping
romanNumeralMap = (('M', 1000),
                   ('CM', 900),
                   ('D', 500),
                   ('CD', 400),
                   ('C', 100),
                   ('XC', 90),
                   ('L', 50),
                   ('XL', 40),
                   ('X', 10),
                   ('IX', 9),
                   ('V', 5),
                   ('IV', 4),
                   ('I', 1))

# toRoman function omitted for clarity (it hasn't changed)

def fromRoman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral: ❶
            result += integer
            index += len(numeral)
    return result
```

- ❶ The pattern here is the same as toRoman. You iterate through your Roman numeral data structure (a tuple of tuples), and instead of matching the highest integer values as often as possible, you match the "highest" Roman numeral character strings as often as possible.

Example 14.10. How fromRoman works

If you're not clear how fromRoman works, add a print statement to the end of the while loop:

```
while s[index:index+len(numeral)] == numeral:
    result += integer
    index += len(numeral)
    print 'found', numeral, 'of length', len(numeral), ', adding', integer

>>> import roman4
>>> roman4.fromRoman('MCMLXXII')
found M , of length 1, adding 1000
found CM , of length 2, adding 900
found L , of length 1, adding 50
found X , of length 1, adding 10
found X , of length 1, adding 10
found I , of length 1, adding 1
found I , of length 1, adding 1
1972
```

Example 14.11. Output of romantest4.py against roman4.py

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... ok ❶
```

```

toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

```

②

- ① Two pieces of exciting news here. The first is that `fromRoman` works for good input, at least for all the known values you test.
- ② The second is that the sanity check also passed. Combined with the known values tests, you can be reasonably sure that both `toRoman` and `fromRoman` work properly for all possible good values. (This is not guaranteed; it is theoretically possible that `toRoman` has a bug that produces the wrong Roman numeral for some particular set of inputs, *and* that `fromRoman` has a reciprocal bug that produces the same wrong integer values for exactly that set of Roman numerals that `toRoman` generated incorrectly. Depending on your application and your requirements, this possibility may bother you; if so, write more comprehensive test cases until it doesn't bother you.)

```

=====
FAIL: fromRoman should only accept uppercase input
-----
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 156, in testFromRomanCase
    roman4.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
FAIL: fromRoman should fail with malformed antecedents
-----
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
FAIL: fromRoman should fail with repeated pairs of numerals
-----
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 127, in testRepeatedPairs
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
=====
FAIL: fromRoman should fail with too many repeated numerals
-----
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
-----
Ran 12 tests in 1.222s

FAILED (failures=4)

```

14.5. roman.py, stage 5

Now that `fromRoman` works properly with good input, it's time to fit in the last piece of the puzzle: making it work properly with bad input. That means finding a way to look at a string and determine if it's a valid Roman numeral. This is inherently more difficult than validating numeric input in `toRoman`, but you have a powerful tool at your disposal: regular expressions.

If you're not familiar with regular expressions and didn't read Chapter 7, *Regular Expressions*, now would be a good time.

As you saw in Section 7.3, Case Study: Roman Numerals, there are several simple rules for constructing a Roman numeral, using the letters M, D, C, L, X, V, and I. Let's review the rules:

1. Characters are additive. I is 1, II is 2, and III is 3. VI is 6 (literally, "5 and 1"), VII is 7, and VIII is 8.
2. The tens characters (I, X, C, and M) can be repeated up to three times. At 4, you need to subtract from the next highest fives character. You can't represent 4 as IIII; instead, it is represented as IV ("1 less than 5"). 40 is written as XL ("10 less than 50"), 41 as XLI, 42 as XLII, 43 as XLIII, and then 44 as XLIV ("10 less than 50, then 1 less than 5").
3. Similarly, at 9, you need to subtract from the next highest tens character: 8 is VIII, but 9 is IX ("1 less than 10"), not VIIII (since the I character can not be repeated four times). 90 is XC, 900 is CM.
4. The fives characters can not be repeated. 10 is always represented as X, never as VV. 100 is always C, never LL.
5. Roman numerals are always written highest to lowest, and read left to right, so order of characters matters very much. DC is 600; CD is a completely different number (400, "100 less than 500"). CI is 101; IC is not even a valid Roman numeral (because you can't subtract 1 directly from 100; you would need to write it as XCIX, "10 less than 100, then 1 less than 10").

Example 14.12. roman5.py

This file is available in `py/roman/stage5/` in the examples directory.

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M', 1000),
                    ('CM', 900),
                    ('D', 500),
                    ('CD', 400),
                    ('C', 100),
                    ('XC', 90),
                    ('L', 50),
                    ('XL', 40),
                    ('X', 10),
                    ('IX', 9),
                    ('V', 5),
```

```

        ('IV', 4),
        ('I', 1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError, "number out of range (must be 1..3999)"
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' ❶

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not re.search(romanNumeralPattern, s): ❷
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result

```

- ❶ This is just a continuation of the pattern you discussed in Section 7.3, Case Study: Roman Numerals . The tens places is either XC (90), XL (40), or an optional L followed by 0 to 3 optional X characters. The ones place is either IX (9), IV (4), or an optional V followed by 0 to 3 optional I characters.
- ❷ Having encoded all that logic into a regular expression, the code to check for invalid Roman numerals becomes trivial. If `re.search` returns an object, then the regular expression matched and the input is valid; otherwise, the input is invalid.

At this point, you are allowed to be skeptical that that big ugly regular expression could possibly catch all the types of invalid Roman numerals. But don't take my word for it, look at the results:

Example 14.13. Output of `romantest5.py` against `roman5.py`

```

fromRoman should only accept uppercase input ... ok ❶
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... ok ❷
fromRoman should fail with repeated pairs of numerals ... ok ❸
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

```

```

Ran 12 tests in 2.864s

```

- ❶ One thing I didn't mention about regular expressions is that, by default, they are case-sensitive. Since the regular expression `romanNumeralPattern` was expressed in uppercase characters, the `re.search` check will reject any input that isn't completely uppercase. So the uppercase input test passes.
- ❷ More importantly, the bad input tests pass. For instance, the malformed antecedents test checks cases like `MCMC`. As you've seen, this does not match the regular expression, so `fromRoman` raises an `InvalidRomanNumeralError` exception, which is what the malformed antecedents test case is looking for, so the test passes.
- ❸ In fact, all the bad input tests pass. This regular expression catches everything you could think of when you made your test cases.
- ❹ And the anticlimax award of the year goes to the word "OK", which is printed by the `unittest` module when all the tests pass.

When all of your tests pass,  coding.

Chapter 15. Refactoring

15.1. Handling bugs

Despite your best efforts to write comprehensive unit tests, bugs happen. What do I mean by "bug"? A bug is a test case you haven't written yet.

Example 15.1. The bug

```
>>> import roman5
>>> roman5.fromRoman("") ❶
0
```

- ❶ Remember in the previous section when you kept seeing that an empty string would match the regular expression you were using to check for valid Roman numerals? Well, it turns out that this is still true for the final version of the regular expression. And that's a bug: you want an empty string to raise an `InvalidRomanNumeralError` exception just like any other sequence of characters that don't represent a valid Roman numeral.

After reproducing the bug, and before fixing it, you should write a test case that fails, thus illustrating the bug.

Example 15.2. Testing for the bug (`romantest61.py`)

```
class FromRomanBadInput(unittest.TestCase):

    # previous test cases omitted for clarity (they haven't changed)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, "") ❶
```

- ❶ Pretty simple stuff here. Call `fromRoman` with an empty string and make sure it raises an `InvalidRomanNumeralError` exception. The hard part was finding the bug; now that you know about it, testing for it is the easy part.

Since your code has a bug, and you now have a test case that tests this bug, the test case will fail:

Example 15.3. Output of `romantest61.py` against `roman61.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... FAIL
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

=====
```

```
FAIL: fromRoman should fail with blank string
```

```
-----  
Traceback (most recent call last):
```

```
File "C:\docbook\dip\py\roman\stage6\romantest61.py", line 137, in testBlank  
    self.assertRaises(roman61.InvalidRomanNumeralError, roman61.fromRoman, "")  
File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises  
    raise self.failureException, excName  
AssertionError: InvalidRomanNumeralError  
-----
```

```
Ran 13 tests in 2.864s
```

```
FAILED (failures=1)
```

Now you can fix the bug.

Example 15.4. Fixing the bug (roman62.py)

This file is available in py/roman/stage6/ in the examples directory.

```
def fromRoman(s):  
    """convert Roman numeral to integer"""  
    if not s: ❶  
        raise InvalidRomanNumeralError, 'Input can not be blank'  
    if not re.search(romanNumeralPattern, s):  
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s  
  
    result = 0  
    index = 0  
    for numeral, integer in romanNumeralMap:  
        while s[index:index+len(numeral)] == numeral:  
            result += integer  
            index += len(numeral)  
    return result
```

- ❶ Only two lines of code are required: an explicit check for an empty string, and a raise statement.

Example 15.5. Output of romantest62.py against roman62.py

```
fromRoman should only accept uppercase input ... ok  
toRoman should always return uppercase ... ok  
fromRoman should fail with blank string ... ok ❶  
fromRoman should fail with malformed antecedents ... ok  
fromRoman should fail with repeated pairs of numerals ... ok  
fromRoman should fail with too many repeated numerals ... ok  
fromRoman should give known result with known input ... ok  
toRoman should give known result with known input ... ok  
fromRoman(toRoman(n))==n for all n ... ok  
toRoman should fail with non-integer input ... ok  
toRoman should fail with negative input ... ok  
toRoman should fail with large input ... ok  
toRoman should fail with 0 input ... ok
```

```
-----  
Ran 13 tests in 2.834s
```

```
OK ❷
```

- ❶ The blank string test case now passes, so the bug is fixed.

② All the other test cases still pass, which means that this bug fix didn't break anything else. Stop coding. Coding this way does not make fixing bugs any easier. Simple bugs (like this one) require simple test cases; complex bugs will require complex test cases. In a testing-centric environment, it may *seem* like it takes longer to fix a bug, since you need to articulate in code exactly what the bug is (to write the test case), then fix the bug itself. Then if the test case doesn't pass right away, you need to figure out whether the fix was wrong, or whether the test case itself has a bug in it. However, in the long run, this back-and-forth between test code and code tested pays for itself, because it makes it more likely that bugs are fixed correctly the first time. Also, since you can easily re-run *all* the test cases along with your new one, you are much less likely to break old code when fixing new code. Today's unit test is tomorrow's regression test.

15.2. Handling changing requirements

Despite your best efforts to pin your customers to the ground and extract exact requirements from them on pain of horrible nasty things involving scissors and hot wax, requirements will change. Most customers don't know what they want until they see it, and even if they do, they aren't that good at articulating what they want precisely enough to be useful. And even if they do, they'll want more in the next release anyway. So be prepared to update your test cases as requirements change.

Suppose, for instance, that you wanted to expand the range of the Roman numeral conversion functions. Remember the rule that said that no character could be repeated more than three times? Well, the Romans were willing to make an exception to that rule by having 4 M characters in a row to represent 4000. If you make this change, you'll be able to expand the range of convertible numbers from 1..3999 to 1..4999. But first, you need to make some changes to the test cases.

Example 15.6. Modifying test cases for new requirements (`romantest71.py`)

This file is available in `py/roman/stage7/` in the examples directory.

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
import roman71
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
```



```
(621, 'DCXXI'),
(782, 'DCCLXXXII'),
(870, 'DCCCLXX'),
(941, 'CMXLI'),
(1043, 'MXLIII'),
(1110, 'MCX'),
(1226, 'MCCXXVI'),
(1301, 'MCCCI'),
(1485, 'MCDLXXXV'),
(1509, 'MDIX'),
(1607, 'MDCVII'),
(1754, 'MDCCLIV'),
(1832, 'MDCCCXXXII'),
(1993, 'MCMXCIII'),
(2074, 'MMLXXIV'),
(2152, 'MMCLII'),
(2212, 'MMCCXII'),
(2343, 'MMCCCXLIII'),
(2499, 'MMCDXCIX'),
(2574, 'MMDLXXIV'),
(2646, 'MMDCLVI'),
(2723, 'MMDCCXIII'),
(2892, 'MMDCCCXCII'),
(2975, 'MMCMLXXV'),
(3051, 'MMMLI'),
(3185, 'MMMCLXXXV'),
(3250, 'MMMCCCL'),
(3313, 'MMMCCCXIII'),
(3408, 'MMMCDVIII'),
(3501, 'MMMCI'),
(3610, 'MMMDCX'),
(3743, 'MMMDCCXLIII'),
(3844, 'MMMDCCCXLIV'),
(3888, 'MMMDCCCLXXXVIII'),
(3940, 'MMMCMXL'),
(3999, 'MMMCMXCIX'),
(4000, 'MMMM'),
(4500, 'MMMMD'),
(4888, 'MMMMDCCCLXXXVIII'),
(4999, 'MMMCMXCIX'))
```

❶

```
def testToRomanKnownValues(self):
    """toRoman should give known result with known input"""
    for integer, numeral in self.knownValues:
        result = roman71.toRoman(integer)
        self.assertEqual(numeral, result)

def testFromRomanKnownValues(self):
    """fromRoman should give known result with known input"""
    for integer, numeral in self.knownValues:
        result = roman71.fromRoman(numeral)
        self.assertEqual(integer, result)
```

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 5000) ❷

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 0)

    def testNegative(self):
```

```

        """toRoman should fail with negative input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman71.NotIntegerError, roman71.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'): ❸
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, "")

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 5000): ❹
            numeral = roman71.toRoman(integer)
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            roman71.fromRoman(numeral.upper())
            self.assertRaises(roman71.InvalidRomanNumeralError,
                            roman71.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()

```

- ❶ The existing known values don't change (they're all still reasonable values to test), but you need to add a few more in the 4000 range. Here I've included 4000 (the shortest), 4500 (the second shortest), 4888 (the longest), and 4999 (the largest).
- ❷ The definition of "large input" has changed. This test used to call `toRoman` with 4000 and expect an error; now that 4000–4999 are good values, you need to bump this up to 5000.
- ❸ The definition of "too many repeated numerals" has also changed. This test used to call `fromRoman` with 'MMMM' and expect an error; now that MMMM is considered a valid Roman numeral, you need to

bump this up to 'MMMM'.

- ④ The sanity check and case checks loop through every number in the range, from 1 to 3999. Since the range has now expanded, these for loops need to be updated as well to go up to 4999.

Now your test cases are up to date with the new requirements, but your code is not, so you expect several of the test cases to fail.

Example 15.7. Output of `romantest71.py` against `roman71.py`

```
fromRoman should only accept uppercase input ... ERROR ❶
toRoman should always return uppercase ... ERROR
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ERROR ❷
toRoman should give known result with known input ... ERROR ❸
fromRoman(toRoman(n))==n for all n ... ERROR ❹
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

- ❶ Our case checks now fail because they loop from 1 to 4999, but `toRoman` only accepts numbers from 1 to 3999, so it will fail as soon the test case hits 4000.
- ❷ The `fromRoman` known values test will fail as soon as it hits 'MMMM', because `fromRoman` still thinks this is an invalid Roman numeral.
- ❸ The `toRoman` known values test will fail as soon as it hits 4000, because `toRoman` still thinks this is out of range.
- ❹ The sanity check will also fail as soon as it hits 4000, because `toRoman` still thinks this is out of range.

```
=====
ERROR: fromRoman should only accept uppercase input
-----

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 161, in testFromRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
=====
ERROR: toRoman should always return uppercase
-----

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 155, in testToRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
=====
ERROR: fromRoman should give known result with known input
-----

Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 102, in testFromRomanKnownValues
    result = roman71.fromRoman(numeral)
  File "roman71.py", line 47, in fromRoman
    raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
InvalidRomanNumeralError: Invalid Roman numeral: MMMM
=====
```

```
ERROR: toRoman should give known result with known input
```

```
-----  
Traceback (most recent call last):
```

```
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 96, in testToRomanKnownValues  
    result = roman71.toRoman(integer)  
  File "roman71.py", line 28, in toRoman  
    raise OutOfRangeError, "number out of range (must be 1..3999)"  
OutOfRangeError: number out of range (must be 1..3999)
```

```
=====
```

```
ERROR: fromRoman(toRoman(n))==n for all n
```

```
-----  
Traceback (most recent call last):
```

```
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 147, in testSanity  
    numeral = roman71.toRoman(integer)  
  File "roman71.py", line 28, in toRoman  
    raise OutOfRangeError, "number out of range (must be 1..3999)"  
OutOfRangeError: number out of range (must be 1..3999)
```

```
-----  
Ran 13 tests in 2.213s
```

```
FAILED (errors=5)
```

Now that you have test cases that fail due to the new requirements, you can think about fixing the code to bring it in line with the test cases. (One thing that takes some getting used to when you first start coding unit tests is that the code being tested is never "ahead" of the test cases. While it's behind, you still have some work to do, and as soon as it catches up to the test cases, you stop coding.)

Example 15.8. Coding the new requirements (`roman72.py`)

This file is available in `py/roman/stage7/` in the examples directory.

```
"""Convert to and from Roman numerals"""  
import re  
  
#Define exceptions  
class RomanError(Exception): pass  
class OutOfRangeError(RomanError): pass  
class NotIntegerError(RomanError): pass  
class InvalidRomanNumeralError(RomanError): pass  
  
#Define digit mapping  
romanNumeralMap = (('M', 1000),  
                    ('CM', 900),  
                    ('D', 500),  
                    ('CD', 400),  
                    ('C', 100),  
                    ('XC', 90),  
                    ('L', 50),  
                    ('XL', 40),  
                    ('X', 10),  
                    ('IX', 9),  
                    ('V', 5),  
                    ('IV', 4),  
                    ('I', 1))  
  
def toRoman(n):  
    """convert integer to Roman numeral"""  
    if not (0 < n < 5000):  
        raise OutOfRangeError, "number out of range (must be 1..4999)"  
    if int(n) <> n:
```

❶

```

        raise NotIntegerError, "non-integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' ❷

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result

```

- ❶ toRoman only needs one small change, in the range check. Where you used to check $0 < n < 4000$, you now check $0 < n < 5000$. And you change the error message that you raise to reflect the new acceptable range (1..4999 instead of 1..3999). You don't need to make any changes to the rest of the function; it handles the new cases already. (It merrily adds 'M' for each thousand that it finds; given 4000, it will spit out 'MMMM'. The only reason it didn't do this before is that you explicitly stopped it with the range check.)
- ❷ You don't need to make any changes to fromRoman at all. The only change is to romanNumeralPattern; if you look closely, you'll notice that you added another optional M in the first section of the regular expression. This will allow up to 4 M characters instead of 3, meaning you will allow the Roman numeral equivalents of 4999 instead of 3999. The actual fromRoman function is completely general; it just looks for repeated Roman numeral characters and adds them up, without caring how many times they repeat. The only reason it didn't handle 'MMMM' before is that you explicitly stopped it with the regular expression pattern matching.

You may be skeptical that these two small changes are all that you need. Hey, don't take my word for it; see for yourself:

Example 15.9. Output of romantest72.py against roman72.py

```

fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

```

Ran 13 tests in 3.685s

OK ①

- ① All the test cases pass. Stop coding.

Comprehensive unit testing means never having to rely on a programmer who says "Trust me."

15.3. Refactoring

The best thing about comprehensive unit testing is not the feeling you get when all your test cases finally pass, or even the feeling you get when someone else blames you for breaking their code and you can actually *prove* that you didn't. The best thing about unit testing is that it gives you the freedom to refactor mercilessly.

Refactoring is the process of taking working code and making it work better. Usually, "better" means "faster", although it can also mean "using less memory", or "using less disk space", or simply "more elegantly". Whatever it means to you, to your project, in your environment, refactoring is important to the long-term health of any program.

Here, "better" means "faster". Specifically, the `fromRoman` function is slower than it needs to be, because of that big nasty regular expression that you use to validate Roman numerals. It's probably not worth trying to do away with the regular expression altogether (it would be difficult, and it might not end up any faster), but you can speed up the function by precompiling the regular expression.

Example 15.10. Compiling regular expressions

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M') ①
<SRE_Match object at 01090490>
>>> compiledPattern = re.compile(pattern) ②
>>> compiledPattern
<SRE_Pattern object at 00F06E28>
>>> dir(compiledPattern) ③
['findall', 'match', 'scanner', 'search', 'split', 'sub', 'subn']
>>> compiledPattern.search('M') ④
<SRE_Match object at 01104928>
```

- ① This is the syntax you've seen before: `re.search` takes a regular expression as a string (`pattern`) and a string to match against it (`'M'`). If the pattern matches, the function returns a match object which can be queried to find out exactly what matched and how.
- ② This is the new syntax: `re.compile` takes a regular expression as a string and returns a pattern object. Note there is no string to match here. Compiling a regular expression has nothing to do with matching it against any specific strings (like `'M'`); it only involves the regular expression itself.
- ③ The compiled pattern object returned from `re.compile` has several useful-looking functions, including several (like `search` and `sub`) that are available directly in the `re` module.
- ④ Calling the compiled pattern object's `search` function with the string `'M'` accomplishes the same thing as calling `re.search` with both the regular expression and the string `'M'`. Only much, much faster. (In fact, the `re.search` function simply compiles the regular expression and calls the resulting pattern object's `search` method for you.)

Whenever you are going to use a regular expression more than once, you should compile it to get a pattern object, then call the methods on the pattern object directly.

Example 15.11. Compiled regular expressions in `roman81.py`

This file is available in `py/roman/stage8/` in the examples directory.

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
# toRoman and rest of module omitted for clarity

romanNumeralPattern = \
    re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$') ❶

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not romanNumeralPattern.search(s): ❷
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

- ❶ This looks very similar, but in fact a lot has changed. `romanNumeralPattern` is no longer a string; it is a pattern object which was returned from `re.compile`.
- ❷ That means that you can call methods on `romanNumeralPattern` directly. This will be much, much faster than calling `re.search` every time. The regular expression is compiled once and stored in `romanNumeralPattern` when the module is first imported; then, every time you call `fromRoman`, you can immediately match the input string against the regular expression, without any intermediate steps occurring under the covers.

So how much faster is it to compile regular expressions? See for yourself:

Example 15.12. Output of `romantest81.py` against `roman81.py`

```
..... ❶
-----
Ran 13 tests in 3.385s ❷
OK ❸
```

- ❶ Just a note in passing here: this time, I ran the unit test *without* the `-v` option, so instead of the full doc string for each test, you only get a dot for each test that passes. (If a test failed, you'd get an F, and if it had an error, you'd get an E. You'd still get complete tracebacks for each failure and error, so you could track down any problems.)
- ❷ You ran 13 tests in 3.385 seconds, compared to 3.685 seconds without precompiling the regular expressions. That's an 8% improvement overall, and remember that most of the time spent during the unit test is spent doing other things. (Separately, I time-tested the regular expressions by themselves, apart from the rest of the unit tests, and found that compiling this regular expression speeds up the search by an average of 54%.) Not bad for such a simple fix.
- ❸ Oh, and in case you were wondering, precompiling the regular expression didn't break anything, and you

just proved it.

There is one other performance optimization that I want to try. Given the complexity of regular expression syntax, it should come as no surprise that there is frequently more than one way to write the same expression. After some discussion about this module on `comp.lang.python` (<http://groups.google.com/groups?group=comp.lang.python>), someone suggested that I try using the `{m,n}` syntax for the optional repeated characters.

Example 15.13. `roman82.py`

This file is available in `py/roman/stage8/` in the examples directory.

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#    re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$')

#new version
romanNumeralPattern = \
    re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$') ❶
```

- ❶ You have replaced `M?M?M?M?` with `M{0,4}`. Both mean the same thing: "match 0 to 4 M characters". Similarly, `C?C?C?` became `C{0,3}` ("match 0 to 3 C characters") and so forth for X and I.

This form of the regular expression is a little shorter (though not any more readable). The big question is, is it any faster?

Example 15.14. Output of `romantest82.py` against `roman82.py`

```
.....
-----
Ran 13 tests in 3.315s ❶
OK ❷
```

- ❶ Overall, the unit tests run 2% faster with this form of regular expression. That doesn't sound exciting, but remember that the `search` function is a small part of the overall unit test; most of the time is spent doing other things. (Separately, I time-tested just the regular expressions, and found that the `search` function is 11% faster with this syntax.) By precompiling the regular expression and rewriting part of it to use this new syntax, you've improved the regular expression performance by over 60%, and improved the overall performance of the entire unit test by over 10%.
- ❷ More important than any performance boost is the fact that the module still works perfectly. This is the freedom I was talking about earlier: the freedom to tweak, change, or rewrite any piece of it and verify that you haven't messed anything up in the process. This is not a license to endlessly tweak your code just for the sake of tweaking it; you had a very specific objective ("make `fromRoman` faster"), and you were able to accomplish that objective without any lingering doubts about whether you introduced new bugs in the process.

One other tweak I would like to make, and then I promise I'll stop refactoring and put this module to bed. As you've seen repeatedly, regular expressions can get pretty hairy and unreadable pretty quickly. I wouldn't like to come back to this module in six months and try to maintain it. Sure, the test cases pass, so I know that it works, but if I can't figure out *how* it works, it's still going to be difficult to add new features, fix new bugs, or otherwise maintain it. As you saw

in Section 7.5, Verbose Regular Expressions, Python provides a way to document your logic line-by-line.

Example 15.15. `roman83.py`

This file is available in `py/roman/stage8/` in the examples directory.

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#    re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')

#new version
romanNumeralPattern = re.compile('''
    ^                # beginning of string
    M{0,4}           # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3}) # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                    # or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3}) # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                    # or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3}) # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                    # or 5-8 (V, followed by 0 to 3 I's)
    $                # end of string
''', re.VERBOSE) ❶
```

- ❶ The `re.compile` function can take an optional second argument, which is a set of one or more flags that control various options about the compiled regular expression. Here you're specifying the `re.VERBOSE` flag, which tells Python that there are in-line comments within the regular expression itself. The comments and all the whitespace around them are *not* considered part of the regular expression; the `re.compile` function simply strips them all out when it compiles the expression. This new, "verbose" version is identical to the old version, but it is infinitely more readable.

Example 15.16. Output of `romantest83.py` against `roman83.py`

```
.....
-----
Ran 13 tests in 3.315s ❶
OK ❷
```

- ❶ This new, "verbose" version runs at exactly the same speed as the old version. In fact, the compiled pattern objects are the same, since the `re.compile` function strips out all the stuff you added.
- ❷ This new, "verbose" version passes all the same tests as the old version. Nothing has changed, except that the programmer who comes back to this module in six months stands a fighting chance of understanding how the function works.

15.4. Postscript

A clever reader read the previous section and took it to the next level. The biggest headache (and performance drain) in the program as it is currently written is the regular expression, which is required because you have no other way of breaking down a Roman numeral. But there's only 5000 of them; why don't you just build a lookup table once, then simply read that? This idea gets even better when you realize that you don't need to use regular expressions at all. As

you build the lookup table for converting integers to Roman numerals, you can build the reverse lookup table to convert Roman numerals to integers.

And best of all, he already had a complete set of unit tests. He changed over half the code in the module, but the unit tests stayed the same, so he could prove that his code worked just as well as the original.

Example 15.17. `roman9.py`

This file is available in `py/roman/stage9/` in the examples directory.

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Roman numerals must be less than 5000
MAX_ROMAN_NUMERAL = 4999

#Define digit mapping
romanNumeralMap = (('M', 1000),
                    ('CM', 900),
                    ('D', 500),
                    ('CD', 400),
                    ('C', 100),
                    ('XC', 90),
                    ('L', 50),
                    ('XL', 40),
                    ('X', 10),
                    ('IX', 9),
                    ('V', 5),
                    ('IV', 4),
                    ('I', 1))

#Create tables for fast conversion of roman numerals.
#See fillLookupTables() below.
toRomanTable = [ None ] # Skip an index since Roman numerals have no zero
fromRomanTable = {}

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n <= MAX_ROMAN_NUMERAL):
        raise OutOfRangeError, "number out of range (must be 1..%s)" % MAX_ROMAN_NUMERAL
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"
    return toRomanTable[n]

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, "Input can not be blank"
    if not fromRomanTable.has_key(s):
        raise InvalidRomanNumeralError, "Invalid Roman numeral: %s" % s
    return fromRomanTable[s]

def toRomanDynamic(n):
```

```

    """convert integer to Roman numeral using dynamic programming"""
    result = ""
    for numeral, integer in romanNumeralMap:
        if n >= integer:
            result = numeral
            n -= integer
            break
    if n > 0:
        result += toRomanTable[n]
    return result

def fillLookupTables():
    """compute all the possible roman numerals"""
    #Save the values in two global tables to convert to and from integers.
    for integer in range(1, MAX_ROMAN_NUMERAL + 1):
        romanNumber = toRomanDynamic(integer)
        toRomanTable.append(romanNumber)
        fromRomanTable[romanNumber] = integer

fillLookupTables()

```

So how fast is it?

Example 15.18. Output of `romantest9.py` against `roman9.py`

```

.....
-----
Ran 13 tests in 0.791s

OK

```

Remember, the best performance you ever got in the original version was 13 tests in 3.315 seconds. Of course, it's not entirely a fair comparison, because this version will take longer to import (when it fills the lookup tables). But since import is only done once, this is negligible in the long run.

The moral of the story?

- Simplicity is a virtue.
- Especially when regular expressions are involved.
- And unit tests can give you the confidence to do large-scale refactoring... even if you didn't write the original code.

15.5. Summary

Unit testing is a powerful concept which, if properly implemented, can both reduce maintenance costs and increase flexibility in any long-term project. It is also important to understand that unit testing is not a panacea, a Magic Problem Solver, or a silver bullet. Writing good test cases is hard, and keeping them up to date takes discipline (especially when customers are screaming for critical bug fixes). Unit testing is not a replacement for other forms of testing, including functional testing, integration testing, and user acceptance testing. But it is feasible, and it does work, and once you've seen it work, you'll wonder how you ever got along without it.

This chapter covered a lot of ground, and much of it wasn't even Python-specific. There are unit testing frameworks for many languages, all of which require you to understand the same basic concepts:

- Designing test cases that are specific, automated, and independent
- Writing test cases *before* the code they are testing
- Writing tests that test good input and check for proper results
- Writing tests that test bad input and check for proper failures
- Writing and updating test cases to illustrate bugs or reflect new requirements
- Refactoring mercilessly to improve performance, scalability, readability, maintainability, or whatever other -ility you're lacking

Additionally, you should be comfortable doing all of the following Python-specific things:

- Subclassing `unittest.TestCase` and writing methods for individual test cases
- Using `assertEqual` to check that a function returns a known value
- Using `assertRaises` to check that a function raises a known exception
- Calling `unittest.main()` in your `if __name__` clause to run all your test cases at once
- Running unit tests in verbose or regular mode

Further reading

- XProgramming.com (<http://www.xprogramming.com/>) has links to download unit testing frameworks (<http://www.xprogramming.com/software.htm>) for many different languages.

Chapter 16. Functional Programming

16.1. Diving in

In Chapter 13, *Unit Testing*, you learned about the philosophy of unit testing. In Chapter 14, *Test–First Programming*, you stepped through the implementation of basic unit tests in Python. In Chapter 15, *Refactoring*, you saw how unit testing makes large–scale refactoring easier. This chapter will build on those sample programs, but here we will focus more on advanced Python–specific techniques, rather than on unit testing itself.

The following is a complete Python program that acts as a cheap and simple regression testing framework. It takes unit tests that you've written for individual modules, collects them all into one big test suite, and runs them all at once. I actually use this script as part of the build process for this book; I have unit tests for several of the example programs (not just the `roman.py` module featured in Chapter 13, *Unit Testing*), and the first thing my automated build script does is run this program to make sure all my examples still work. If this regression test fails, the build immediately stops. I don't want to release non–working examples any more than you want to download them and sit around scratching your head and yelling at your monitor and wondering why they don't work.

Example 16.1. `regression.py`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
"""Regression testing framework

This module will search for scripts in the same directory named
XYZtest.py. Each such script should be a test suite that tests a
module through PyUnit. (As of Python 2.1, PyUnit is included in
the standard library as "unittest".) This script will aggregate all
found test suites into one big test suite and run them all at once.
"""

import sys, os, re, unittest

def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
    load = unittest.defaultTestLoader.loadTestsFromModule
    return unittest.TestSuite(map(load, modules))

if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest")
```

Running this script in the same directory as the rest of the example scripts that come with this book will find all the unit tests, named `moduletest.py`, run them as a single test, and pass or fail them all at once.

Example 16.2. Sample output of `regression.py`

```
[you@localhost py]$ python regression.py -v
help should fail with no object ... ok
```

❶

```

help should return known result for apihelper ... ok
help should honor collapse argument ... ok
help should honor spacing argument ... ok
buildConnectionString should fail with list input ... ok
buildConnectionString should fail with string input ... ok
buildConnectionString should fail with tuple input ... ok
buildConnectionString handles empty dictionary ... ok
buildConnectionString returns known result with known input ... ok
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
kgp a ref test ... ok
kgp b ref test ... ok
kgp c ref test ... ok
kgp d ref test ... ok
kgp e ref test ... ok
kgp f ref test ... ok
kgp g ref test ... ok

```

```
-----
Ran 29 tests in 2.799s
```

```
OK
```

- ❶ The first 5 tests are from `apihelpertest.py`, which tests the example script from Chapter 4, *The Power Of Introspection*.
- ❷ The next 5 tests are from `odbchelpertest.py`, which tests the example script from Chapter 2, *Your First Python Program*.
- ❸ The rest are from `romantest.py`, which you studied in depth in Chapter 13, *Unit Testing*.

16.2. Finding the path

When running Python scripts from the command line, it is sometimes useful to know where the currently running script is located on disk.

This is one of those obscure little tricks that is virtually impossible to figure out on your own, but simple to remember once you see it. The key to it is `sys.argv`. As you saw in Chapter 9, *XML Processing*, this is a list that holds the list of command-line arguments. However, it also holds the name of the running script, exactly as it was called from the command line, and this is enough information to determine its location.

Example 16.3. `fullpath.py`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
import sys, os
```

```

print 'sys.argv[0] =', sys.argv[0]           ❶
pathname = os.path.dirname(sys.argv[0])      ❷
print 'path =', pathname
print 'full path =', os.path.abspath(pathname) ❸

```

- ❶ Regardless of how you run a script, `sys.argv[0]` will always contain the name of the script, exactly as it appears on the command line. This may or may not include any path information, as you'll see shortly.
- ❷ `os.path.dirname` takes a filename as a string and returns the directory path portion. If the given filename does not include any path information, `os.path.dirname` returns an empty string.
- ❸ `os.path.abspath` is the key here. It takes a pathname, which can be partial or even blank, and returns a fully qualified pathname.

`os.path.abspath` deserves further explanation. It is very flexible; it can take any kind of pathname.

Example 16.4. Further explanation of `os.path.abspath`

```

>>> import os
>>> os.getcwd()                               ❶
/home/you
>>> os.path.abspath('')                       ❷
/home/you
>>> os.path.abspath('.ssh')                   ❸
/home/you/.ssh
>>> os.path.abspath('/home/you/.ssh')         ❹
/home/you/.ssh
>>> os.path.abspath('.ssh/../foo/')           ❺
/home/you/foo

```

- ❶ `os.getcwd()` returns the current working directory.
- ❷ Calling `os.path.abspath` with an empty string returns the current working directory, same as `os.getcwd()`.
- ❸ Calling `os.path.abspath` with a partial pathname constructs a fully qualified pathname out of it, based on the current working directory.
- ❹ Calling `os.path.abspath` with a full pathname simply returns it.
- ❺ `os.path.abspath` also *normalizes* the pathname it returns. Note that this example worked even though I don't actually have a 'foo' directory. `os.path.abspath` never checks your actual disk; this is all just string manipulation.

The pathnames and filenames you pass to `os.path.abspath` do not need to exist.

`os.path.abspath` not only constructs full path names, it also normalizes them. That means that if you are in the `/usr/` directory, `os.path.abspath('bin/../local/bin')` will return `/usr/local/bin`. It normalizes the path by making it as simple as possible. If you just want to normalize a pathname like this without turning it into a full pathname, use `os.path.normpath` instead.

Example 16.5. Sample output from `fullpath.py`

```

[you@localhost py]$ python /home/you/diveintopython/common/py/fullpath.py ❶
sys.argv[0] = /home/you/diveintopython/common/py/fullpath.py
path = /home/you/diveintopython/common/py
full path = /home/you/diveintopython/common/py
[you@localhost diveintopython]$ python common/py/fullpath.py           ❷
sys.argv[0] = common/py/fullpath.py
path = common/py
full path = /home/you/diveintopython/common/py
[you@localhost diveintopython]$ cd common/py

```

```
[you@localhost py]$ python fullpath.py
sys.argv[0] = fullpath.py
path =
full_path = /home/you/diveintopython/common/py
```

③

- ① In the first case, `sys.argv[0]` includes the full path of the script. You can then use the `os.path.dirname` function to strip off the script name and return the full directory name, and `os.path.abspath` simply returns what you give it.
- ② If the script is run by using a partial pathname, `sys.argv[0]` will still contain exactly what appears on the command line. `os.path.dirname` will then give you a partial pathname (relative to the current directory), and `os.path.abspath` will construct a full pathname from the partial pathname.
- ③ If the script is run from the current directory without giving any path, `os.path.dirname` will simply return an empty string. Given an empty string, `os.path.abspath` returns the current directory, which is what you want, since the script was run from the current directory.

Like the other functions in the `os` and `os.path` modules, `os.path.abspath` is cross-platform. Your results will look slightly different than my examples if you're running on Windows (which uses backslash as a path separator) or Mac OS (which uses colons), but they'll still work. That's the whole point of the `os` module.

Addendum. One reader was dissatisfied with this solution, and wanted to be able to run all the unit tests in the current directory, not the directory where `regression.py` is located. He suggests this approach instead:

Example 16.6. Running scripts in the current directory

```
import sys, os, re, unittest
```

```
def regressionTest():
    path = os.getcwd()
    sys.path.append(path)
    files = os.listdir(path)
```

①
②
③

- ① Instead of setting `path` to the directory where the currently running script is located, you set it to the current working directory instead. This will be whatever directory you were in before you ran the script, which is not necessarily the same as the directory the script is in. (Read that sentence a few times until you get it.)
- ② Append this directory to the Python library search path, so that when you dynamically import the unit test modules later, Python can find them. You didn't need to do this when `path` was the directory of the currently running script, because Python always looks in that directory.
- ③ The rest of the function is the same.

This technique will allow you to re-use this `regression.py` script on multiple projects. Just put the script in a common directory, then change to the project's directory before running it. All of that project's unit tests will be found and tested, instead of the unit tests in the common directory where `regression.py` is located.

16.3. Filtering lists revisited

You're already familiar with using list comprehensions to filter lists. There is another way to accomplish this same thing, which some people feel is more expressive.

Python has a built-in `filter` function which takes two arguments, a function and a list, and returns a list.^[7] The function passed as the first argument to `filter` must itself take one argument, and the list that `filter` returns will contain all the elements from the list passed to `filter` for which the function passed to `filter` returns true.

Got all that? It's not as difficult as it sounds.

Example 16.7. Introducing `filter`

```
>>> def odd(n):  
...     return n % 2  
...  
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]  
>>> filter(odd, li)  
[1, 3, 5, 9, -3]  
>>> [e for e in li if odd(e)]  
>>> filteredList = []  
>>> for n in li:  
...     if odd(n):  
...         filteredList.append(n)  
...  
>>> filteredList  
[1, 3, 5, 9, -3]
```

- ❶ `odd` uses the built-in mod function `"%"` to return `True` if `n` is odd and `False` if `n` is even.
- ❷ `filter` takes two arguments, a function (`odd`) and a list (`li`). It loops through the list and calls `odd` with each element. If `odd` returns a true value (remember, any non-zero value is true in Python), then the element is included in the returned list, otherwise it is filtered out. The result is a list of only the odd numbers from the original list, in the same order as they appeared in the original.
- ❸ You could accomplish the same thing using list comprehensions, as you saw in Section 4.5, *Filtering Lists*.
- ❹ You could also accomplish the same thing with a `for` loop. Depending on your programming background, this may seem more "straightforward", but functions like `filter` are much more expressive. Not only is it easier to write, it's easier to read, too. Reading the `for` loop is like standing too close to a painting; you see all the details, but it may take a few seconds to be able to step back and see the bigger picture: "Oh, you're just filtering the list!"

Example 16.8. `filter` in `regression.py`

```
files = os.listdir(path)  
test = re.compile("test\\.py$", re.IGNORECASE)  
files = filter(test.search, files)
```

- ❶ As you saw in Section 16.2, *Finding the path*, `path` may contain the full or partial pathname of the directory of the currently running script, or it may contain an empty string if the script is being run from the current directory. Either way, `files` will end up with the names of the files in the same directory as this script you're running.
- ❷ This is a compiled regular expression. As you saw in Section 15.3, *Refactoring*, if you're going to use the same regular expression over and over, you should compile it for faster performance. The compiled object has a `search` method which takes a single argument, the string to search. If the regular expression matches the string, the `search` method returns a `Match` object containing information about the regular expression match; otherwise it returns `None`, the Python null value.
- ❸ For each element in the `files` list, you're going to call the `search` method of the compiled regular expression object, `test`. If the regular expression matches, the method will return a `Match` object, which Python considers to be true, so the element will be included in the list returned by `filter`. If the regular expression does not match, the `search` method will return `None`, which Python considers to be false, so the element will not be included.

Historical note. Versions of Python prior to 2.0 did not have list comprehensions, so you couldn't filter using list comprehensions; the `filter` function was the only game in town. Even with the introduction of list comprehensions in 2.0, some people still prefer the old-style `filter` (and its companion function, `map`, which you'll see later in this chapter). Both techniques work at the moment, so which one you use is a matter of style. There is discussion that `map` and `filter` might be deprecated in a future version of Python, but no decision has been made.

Example 16.9. Filtering using list comprehensions instead

```
files = os.listdir(path)
test = re.compile("test\\.py$", re.IGNORECASE)
files = [f for f in files if test.search(f)] ❶
```

- ❶ This will accomplish exactly the same result as using the `filter` function. Which way is more expressive? That's up to you.

16.4. Mapping lists revisited

You're already familiar with using list comprehensions to map one list into another. There is another way to accomplish the same thing, using the built-in `map` function. It works much the same way as the `filter` function.

Example 16.10. Introducing `map`

```
>>> def double(n):
...     return n*2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> map(double, li) ❶
[2, 4, 6, 10, 18, 20, 512, -6]
>>> [double(n) for n in li] ❷
[2, 4, 6, 10, 18, 20, 512, -6]
>>> newlist = []
>>> for n in li: ❸
...     newlist.append(double(n))
...
>>> newlist
[2, 4, 6, 10, 18, 20, 512, -6]
```

- ❶ `map` takes a function and a list^[8] and returns a new list by calling the function with each element of the list in order. In this case, the function simply multiplies each element by 2.
- ❷ You could accomplish the same thing with a list comprehension. List comprehensions were first introduced in Python 2.0; `map` has been around forever.
- ❸ You could, if you insist on thinking like a Visual Basic programmer, use a `for` loop to accomplish the same thing.

Example 16.11. `map` with lists of mixed datatypes

```
>>> li = [5, 'a', (2, 'b')]
>>> map(double, li) ❶
[10, 'aa', (2, 'b', 2, 'b')]
```

- ❶ As a side note, I'd like to point out that `map` works just as well with lists of mixed datatypes, as long as the function you're using correctly handles each type. In this case, the `double` function simply multiplies the given argument by 2, and Python Does The Right Thing depending on the datatype of the argument. For integers, this means actually multiplying it by 2; for strings, it means concatenating the string with itself; for tuples, it means making a new tuple that has all of the elements of the original, then all of the elements of the original again.

All right, enough play time. Let's look at some real code.

Example 16.12. `map` in `regression.py`

```
filenameToModuleName = lambda f: os.path.splitext(f)[0] ❶  
moduleNames = map(filenameToModuleName, files)           ❷
```

- ❶ As you saw in Section 4.7, *Using lambda Functions*, `lambda` defines an inline function. And as you saw in Example 6.17, *Splitting Pathnames*, `os.path.splitext` takes a filename and returns a tuple (*name*, *extension*). So `filenameToModuleName` is a function which will take a filename and strip off the file extension, and return just the name.
- ❷ Calling `map` takes each filename listed in `files`, passes it to the function `filenameToModuleName`, and returns a list of the return values of each of those function calls. In other words, you strip the file extension off of each filename, and store the list of all those stripped filenames in `moduleNames`.

As you'll see in the rest of the chapter, you can extend this type of data-centric thinking all the way to the final goal, which is to define and execute a single test suite that contains the tests from all of those individual test suites.

16.5. Data-centric programming

By now you're probably scratching your head wondering why this is better than using `for` loops and straight function calls. And that's a perfectly valid question. Mostly, it's a matter of perspective. Using `map` and `filter` forces you to center your thinking around your data.

In this case, you started with no data at all; the first thing you did was get the directory path of the current script, and got a list of files in that directory. That was the bootstrap, and it gave you real data to work with: a list of filenames.

However, you knew you didn't care about all of those files, only the ones that were actually test suites. You had *too much data*, so you needed to `filter` it. How did you know which data to keep? You needed a test to decide, so you defined one and passed it to the `filter` function. In this case you used a regular expression to decide, but the concept would be the same regardless of how you constructed the test.

Now you had the filenames of each of the test suites (and only the test suites, since everything else had been filtered out), but you really wanted module names instead. You had the right amount of data, but it was *in the wrong format*. So you defined a function that would transform a single filename into a module name, and you mapped that function onto the entire list. From one filename, you can get a module name; from a list of filenames, you can get a list of module names.

Instead of `filter`, you could have used a `for` loop with an `if` statement. Instead of `map`, you could have used a `for` loop with a function call. But using `for` loops like that is busywork. At best, it simply wastes time; at worst, it introduces obscure bugs. For instance, you need to figure out how to test for the condition "is this file a test suite?" anyway; that's the application-specific logic, and no language can write that for us. But once you've figured that out, do you really want to go to all the trouble of defining a new empty list and writing a `for` loop and an `if` statement and manually calling `append` to add each element to the new list if it passes the condition and then keeping track of which variable holds the new filtered data and which one holds the old unfiltered data? Why not just define the test condition, then let Python do the rest of that work for us?

Oh sure, you could try to be fancy and delete elements in place without creating a new list. But you've been burned by that before. Trying to modify a data structure that you're looping through can be tricky. You delete an element, then loop to the next element, and suddenly you've skipped one. Is Python one of the languages that works that way? How long would it take you to figure it out? Would you remember for certain whether it was safe the next time you tried? Programmers spend so much time and make so many mistakes dealing with purely technical issues like this, and it's all pointless. It doesn't advance your program at all; it's just busywork.

I resisted list comprehensions when I first learned Python, and I resisted `filter` and `map` even longer. I insisted on making my life more difficult, sticking to the familiar way of `for` loops and `if` statements and step-by-step code-centric programming. And my Python programs looked a lot like Visual Basic programs, detailing every step of

every operation in every function. And they had all the same types of little problems and obscure bugs. And it was all pointless.

Let it all go. Busywork code is not important. Data is important. And data is not difficult. It's only data. If you have too much, filter it. If it's not what you want, map it. Focus on the data; leave the busywork behind.

16.6. Dynamically importing modules

OK, enough philosophizing. Let's talk about dynamically importing modules.

First, let's look at how you normally import modules. The `import module` syntax looks in the search path for the named module and imports it by name. You can even import multiple modules at once this way, with a comma-separated list. You did this on the very first line of this chapter's script.

Example 16.13. Importing multiple modules at once

```
import sys, os, re, unittest ❶
```

- ❶ This imports four modules at once: `sys` (for system functions and access to the command line parameters), `os` (for operating system functions like directory listings), `re` (for regular expressions), and `unittest` (for unit testing).

Now let's do the same thing, but with dynamic imports.

Example 16.14. Importing modules dynamically

```
>>> sys = __import__('sys') ❶
>>> os = __import__('os')
>>> re = __import__('re')
>>> unittest = __import__('unittest')
>>> sys ❷
>>> <module 'sys' (built-in)>
>>> os
>>> <module 'os' from '/usr/local/lib/python2.2/os.pyc'>
```

- ❶ The built-in `__import__` function accomplishes the same goal as using the `import` statement, but it's an actual function, and it takes a string as an argument.
- ❷ The variable `sys` is now the `sys` module, just as if you had said `import sys`. The variable `os` is now the `os` module, and so forth.

So `__import__` imports a module, but takes a string argument to do it. In this case the module you imported was just a hard-coded string, but it could just as easily be a variable, or the result of a function call. And the variable that you assign the module to doesn't need to match the module name, either. You could import a series of modules and assign them to a list.

Example 16.15. Importing a list of modules dynamically

```
>>> moduleNames = ['sys', 'os', 're', 'unittest'] ❶
>>> moduleNames
['sys', 'os', 're', 'unittest']
>>> modules = map(__import__, moduleNames) ❷
>>> modules ❸
[<module 'sys' (built-in)>,
```

```

<module 'os' from 'c:\Python22\lib\os.pyc'>,
<module 're' from 'c:\Python22\lib\re.pyc'>,
<module 'unittest' from 'c:\Python22\lib\unittest.pyc'>]
>>> modules[0].version
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
>>> import sys
>>> sys.version
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'

```

- ❶ moduleNames is just a list of strings. Nothing fancy, except that the strings happen to be names of modules that you could import, if you wanted to.
- ❷ Surprise, you wanted to import them, and you did, by mapping the `__import__` function onto the list. Remember, this takes each element of the list (moduleNames) and calls the function (`__import__`) over and over, once with each element of the list, builds a list of the return values, and returns the result.
- ❸ So now from a list of strings, you've created a list of actual modules. (Your paths may be different, depending on your operating system, where you installed Python, the phase of the moon, etc.)
- ❹ To drive home the point that these are real modules, let's look at some module attributes. Remember, `modules[0]` is the `sys` module, so `modules[0].version` is `sys.version`. All the other attributes and methods of these modules are also available. There's nothing magic about the `import` statement, and there's nothing magic about modules. Modules are objects. Everything is an object.

Now you should be able to put this all together and figure out what most of this chapter's code sample is doing.

16.7. Putting it all together

You've learned enough now to deconstruct the first seven lines of this chapter's code sample: reading a directory and importing selected modules within it.

Example 16.16. The `regressionTest` function

```

def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
    load = unittest.defaultTestLoader.loadTestsFromModule
    return unittest.TestSuite(map(load, modules))

```

Let's look at it line by line, interactively. Assume that the current directory is `c:\diveintopython\py`, which contains the examples that come with this book, including this chapter's script. As you saw in Section 16.2, Finding the path, the script directory will end up in the `path` variable, so let's start hard-code that and go from there.

Example 16.17. Step 1: Get all the files

```

>>> import sys, os, re, unittest
>>> path = r'c:\diveintopython\py'
>>> files = os.listdir(path)
>>> files ❶

```

```
[ 'BaseHTMLProcessor.py', 'LICENSE.txt', 'apihelper.py', 'apihelpertest.py',
'argecho.py', 'autosize.py', 'bulddialectexamples.py', 'dialect.py',
'fileinfo.py', 'fullpath.py', 'kgptest.py', 'makerealworddoc.py',
'odbchelper.py', 'odbchelpertest.py', 'parsephone.py', 'piglatin.py',
'plural.py', 'pluraltest.py', 'pyfontify.py', 'regression.py', 'roman.py', 'romantest.py',
'uncurly.py', 'unicode2koi8r.py', 'urllister.py', 'kgp', 'plural', 'roman',
'colorize.py']
```

- ❶ files is a list of all the files and directories in the script's directory. (If you've been running some of the examples already, you may also see some .pyc files in there as well.)

Example 16.18. Step 2: Filter to find the files you care about

```
>>> test = re.compile("test\\.py$", re.IGNORECASE)
>>> files = filter(test.search, files)
>>> files
['apihelpertest.py', 'kgptest.py', 'odbchelpertest.py', 'pluraltest.py', 'romantest.py']
```

- ❶ This regular expression will match any string that ends with test.py. Note that you need to escape the period, since a period in a regular expression usually means "match any single character", but you actually want to match a literal period instead.
- ❷ The compiled regular expression acts like a function, so you can use it to filter the large list of files and directories, to find the ones that match the regular expression.
- ❸ And you're left with the list of unit testing scripts, because they were the only ones named SOMETHINGtest.py.

Example 16.19. Step 3: Map filenames to module names

```
>>> filenameToModuleName = lambda f: os.path.splitext(f)[0]
>>> filenameToModuleName('romantest.py')
'romantest'
>>> filenameToModuleName('odbchelpertest.py')
'odbchelpertest'
>>> moduleNames = map(filenameToModuleName, files)
>>> moduleNames
['apihelpertest', 'kgptest', 'odbchelpertest', 'pluraltest', 'romantest']
```

- ❶ As you saw in Section 4.7, Using lambda Functions, lambda is a quick-and-dirty way of creating an inline, one-line function. This one takes a filename with an extension and returns just the filename part, using the standard library function os.path.splitext that you saw in Example 6.17, Splitting Pathnames.
- ❷ filenameToModuleName is a function. There's nothing magic about lambda functions as opposed to regular functions that you define with a def statement. You can call the filenameToModuleName function like any other, and it does just what you wanted it to do: strips the file extension off of its argument.
- ❸ Now you can apply this function to each file in the list of unit test files, using map.
- ❹ And the result is just what you wanted: a list of modules, as strings.

Example 16.20. Step 4: Mapping module names to modules

```
>>> modules = map(__import__, moduleNames)
>>> modules
[<module 'apihelpertest' from 'apihelpertest.py'>,
<module 'kgptest' from 'kgptest.py'>,
<module 'odbchelpertest' from 'odbchelpertest.py'>]
```

```
<module 'pluraltest' from 'pluraltest.py'>,
<module 'romantest' from 'romantest.py'>]
>>> modules[-1]
<module 'romantest' from 'romantest.py'>
```

③

- ① As you saw in Section 16.6, Dynamically importing modules, you can use a combination of `map` and `__import__` to map a list of module names (as strings) into actual modules (which you can call or access like any other module).
- ② `modules` is now a list of modules, fully accessible like any other module.
- ③ The last module in the list is the `romantest` module, just as if you had said `import romantest`.

Example 16.21. Step 5: Loading the modules into a test suite

```
>>> load = unittest.defaultTestLoader.loadTestsFromModule
>>> map(load, modules)
[<unittest.TestSuite tests=[
  <unittest.TestSuite tests=[<apihelpertest.BadInput testMethod=testNoObject>]>,
  <unittest.TestSuite tests=[<apihelpertest.KnownValues testMethod=testApiHelper>]>,
  <unittest.TestSuite tests=[
    <apihelpertest.ParamChecks testMethod=testCollapse>,
    <apihelpertest.ParamChecks testMethod=testSpacing>]>,
  ...
]>]
>>> unittest.TestSuite(map(load, modules))
```

①

②

- ① These are real module objects. Not only can you access them like any other module, instantiate classes and call functions, you can also introspect into the module to figure out which classes and functions it has in the first place. That's what the `loadTestsFromModule` method does: it introspects into each module and returns a `unittest.TestSuite` object for each module. Each `TestSuite` object actually contains a list of `TestSuite` objects, one for each `TestCase` class in your module, and each of those `TestSuite` objects contains a list of tests, one for each test method in your module.
- ② Finally, you wrap the list of `TestSuite` objects into one big test suite. The `unittest` module has no problem traversing this tree of nested test suites within test suites; eventually it gets down to an individual test method and executes it, verifies that it passes or fails, and moves on to the next one.

This introspection process is what the `unittest` module usually does for us. Remember that magic-looking `unittest.main()` function that our individual test modules called to kick the whole thing off?

`unittest.main()` actually creates an instance of `unittest.TestProgram`, which in turn creates an instance of a `unittest.defaultTestLoader` and loads it up with the module that called it. (How does it get a reference to the module that called it if you don't give it one? By using the equally-magic `__import__('__main__')` command, which dynamically imports the currently-running module. I could write a book on all the tricks and techniques used in the `unittest` module, but then I'd never finish this one.)

Example 16.22. Step 6: Telling `unittest` to use your test suite

```
if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest")
```

①

- ① Instead of letting the `unittest` module do all its magic for us, you've done most of it yourself. You've created a function (`regressionTest`) that imports the modules yourself, calls `unittest.defaultTestLoader` yourself, and wraps it all up in a test suite. Now all you need to do is tell `unittest` that, instead of looking for tests and building a test suite in the usual way, it should just call the `regressionTest` function,

which returns a ready-to-use `TestSuite`.

16.8. Summary

The `regression.py` program and its output should now make perfect sense.

You should now feel comfortable doing all of these things:

- Manipulating path information from the command line.
- Filtering lists using `filter` instead of list comprehensions.
- Mapping lists using `map` instead of list comprehensions.
- Dynamically importing modules.

^[7] Technically, the second argument to `filter` can be any sequence, including lists, tuples, and custom classes that act like lists by defining the `__getitem__` special method. If possible, `filter` will return the same datatype as you give it, so filtering a list returns a list, but filtering a tuple returns a tuple.

^[8] Again, I should point out that `map` can take a list, a tuple, or any object that acts like a sequence. See previous footnote about `filter`.

Chapter 17. Dynamic functions

17.1. Diving in

I want to talk about plural nouns. Also, functions that return other functions, advanced regular expressions, and generators. Generators are new in Python 2.3. But first, let's talk about how to make plural nouns.

If you haven't read Chapter 7, *Regular Expressions*, now would be a good time. This chapter assumes you understand the basics of regular expressions, and quickly descends into more advanced uses.

English is a schizophrenic language that borrows from a lot of other languages, and the rules for making singular nouns into plural nouns are varied and complex. There are rules, and then there are exceptions to those rules, and then there are exceptions to the exceptions.

If you grew up in an English-speaking country or learned English in a formal school setting, you're probably familiar with the basic rules:

1. If a word ends in S, X, or Z, add ES. "Bass" becomes "basses", "fax" becomes "faxes", and "waltz" becomes "waltzes".
2. If a word ends in a noisy H, add ES; if it ends in a silent H, just add S. What's a noisy H? One that gets combined with other letters to make a sound that you can hear. So "coach" becomes "coaches" and "rash" becomes "rashes", because you can hear the CH and SH sounds when you say them. But "cheetah" becomes "cheetahs", because the H is silent.
3. If a word ends in Y that sounds like I, change the Y to IES; if the Y is combined with a vowel to sound like something else, just add S. So "vacancy" becomes "vacancies", but "day" becomes "days".
4. If all else fails, just add S and hope for the best.

(I know, there are a lot of exceptions. "Man" becomes "men" and "woman" becomes "women", but "human" becomes "humans". "Mouse" becomes "mice" and "louse" becomes "lice", but "house" becomes "houses". "Knife" becomes "knives" and "wife" becomes "wives", but "lowlife" becomes "lowlives". And don't even get me started on words that are their own plural, like "sheep", "deer", and "haiku".)

Other languages are, of course, completely different.

Let's design a module that pluralizes nouns. Start with just English nouns, and just these four rules, but keep in mind that you'll inevitably need to add more rules, and you may eventually need to add more languages.

17.2. plural.py, stage 1

So you're looking at words, which at least in English are strings of characters. And you have rules that say you need to find different combinations of characters, and then do different things to them. This sounds like a job for regular expressions.

Example 17.1. plural1.py

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeioudgkprt]h$', noun):
```

❶
❷

```

    return re.sub('$', 'es', noun)
elif re.search('[^aeiouly]$', noun):
    return re.sub('y$', 'ies', noun)
else:
    return noun + 's'

```

- ❶ OK, this is a regular expression, but it uses a syntax you didn't see in Chapter 7, *Regular Expressions*. The square brackets mean "match exactly one of these characters". So `[sxz]` means "s, or x, or z", but only one of them. The `$` should be familiar; it matches the end of string. So you're checking to see if `noun` ends with s, x, or z.
- ❷ This `re.sub` function performs regular expression–based string substitutions. Let's look at it in more detail.

Example 17.2. Introducing `re.sub`

```

>>> import re
>>> re.search('[abc]', 'Mark') ❶
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.sub('[abc]', 'o', 'Mark') ❷
'Mork'
>>> re.sub('[abc]', 'o', 'rock') ❸
'rook'
>>> re.sub('[abc]', 'o', 'caps') ❹
'oops'

```

- ❶ Does the string `Mark` contain a, b, or c? Yes, it contains a.
- ❷ OK, now find a, b, or c, and replace it with o. `Mark` becomes `Mork`.
- ❸ The same function turns `rock` into `rook`.
- ❹ You might think this would turn `caps` into `oaps`, but it doesn't. `re.sub` replaces *all* of the matches, not just the first one. So this regular expression turns `caps` into `oops`, because both the `c` and the `g` get turned into `o`.

Example 17.3. Back to `plural1.py`

```

import re

def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun) ❶
    elif re.search('[^aeioudgkprt]h$', noun): ❷
        return re.sub('$', 'es', noun) ❸
    elif re.search('[^aeiouly]$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'

```

- ❶ Back to the `plural` function. What are you doing? You're replacing the end of string with `es`. In other words, adding `es` to the string. You could accomplish the same thing with string concatenation, for example `noun + 'es'`, but I'm using regular expressions for everything, for consistency, for reasons that will become clear later in the chapter.
- ❷ Look closely, this is another new variation. The `^` as the first character inside the square brackets means something special: negation. `[^abc]` means "any single character *except* a, b, or c". So `[^aeioudgkprt]` means any character except a, e, i, o, u, d, g, k, p, r, or t. Then that character needs to be followed by h, followed by end of string. You're looking for words that end in H where the H can be heard.
- ❸ Same pattern here: match words that end in Y, where the character before the Y is *not* a, e, i, o, or u. You're looking for words that end in Y that sounds like I.

Example 17.4. More on negation regular expressions

```
>>> import re
>>> re.search('[^aeiouly$', 'vacancy') ❶
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.search('[^aeiouly$', 'boy')      ❷
>>>
>>> re.search('[^aeiouly$', 'day')
>>>
>>> re.search('[^aeiouly$', 'pita')     ❸
>>>
```

- ❶ vacancy matches this regular expression, because it ends in `cy`, and `c` is not `a`, `e`, `i`, `o`, or `u`.
- ❷ boy does not match, because it ends in `oy`, and you specifically said that the character before the `y` could not be `o`. day does not match, because it ends in `ay`.
- ❸ pita does not match, because it does not end in `y`.

Example 17.5. More on `re.sub`

```
>>> re.sub('y$', 'ies', 'vacancy')      ❶
'vacancies'
>>> re.sub('y$', 'ies', 'agency')
'agencies'
>>> re.sub('([^aeiou])y$', r'\1ies', 'vacancy') ❷
'vacancies'
```

- ❶ This regular expression turns `vacancy` into `vacancies` and `agency` into `agencies`, which is what you wanted. Note that it would also turn `boy` into `boies`, but that will never happen in the function because you did that `re.search` first to find out whether you should do this `re.sub`.
- ❷ Just in passing, I want to point out that it is possible to combine these two regular expressions (one to find out if the rule applies, and another to actually apply it) into a single regular expression. Here's what that would look like. Most of it should look familiar: you're using a remembered group, which you learned in Section 7.6, Case study: Parsing Phone Numbers, to remember the character before the `y`. Then in the substitution string, you use a new syntax, `\1`, which means "hey, that first group you remembered? put it here". In this case, you remember the `c` before the `y`, and then when you do the substitution, you substitute `c` in place of `c`, and `ies` in place of `y`. (If you have more than one remembered group, you can use `\2` and `\3` and so on.)

Regular expression substitutions are extremely powerful, and the `\1` syntax makes them even more powerful. But combining the entire operation into one regular expression is also much harder to read, and it doesn't directly map to the way you first described the pluralizing rules. You originally laid out rules like "if the word ends in `S`, `X`, or `Z`, then add `ES`". And if you look at this function, you have two lines of code that say "if the word ends in `S`, `X`, or `Z`, then add `ES`". It doesn't get much more direct than that.

17.3. `plural.py`, stage 2

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

Example 17.6. `plural2.py`

```
import re

def match_sxz(noun):
```

```

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

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

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

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

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

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

def match_default(noun):
    return 1

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

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

def plural(noun):
    for matchesRule, applyRule in rules:
        if matchesRule(noun):
            return applyRule(noun)

```

❶

❷

❸

❹

- ❶ This version looks more complicated (it's certainly longer), but it does exactly the same thing: try to match four different rules, in order, and apply the appropriate regular expression when a match is found. The difference is that each individual match and apply rule is defined in its own function, and the functions are then listed in this `rules` variable, which is a tuple of tuples.
- ❷ Using a `for` loop, you can pull out the match and apply rules two at a time (one match, one apply) from the `rules` tuple. On the first iteration of the `for` loop, `matchesRule` will get `match_sxz`, and `applyRule` will get `apply_sxz`. On the second iteration (assuming you get that far), `matchesRule` will be assigned `match_h`, and `applyRule` will be assigned `apply_h`.
- ❸ Remember that everything in Python is an object, including functions. `rules` contains actual functions; not names of functions, but actual functions. When they get assigned in the `for` loop, then `matchesRule` and `applyRule` are actual functions that you can call. So on the first iteration of the `for` loop, this is equivalent to calling `matches_sxz(noun)`.
- ❹ On the first iteration of the `for` loop, this is equivalent to calling `apply_sxz(noun)`, and so forth.

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

Example 17.7. Unrolling the `plural` function

```

def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)

```

```

if match_y(noun):
    return apply_y(noun)
if match_default(noun):
    return apply_default(noun)

```

The benefit here is that that `plural` function is now simplified. It takes a list of rules, defined elsewhere, and iterates through them in a generic fashion. Get a match rule; does it match? Then call the apply rule. The rules could be defined anywhere, in any way. The `plural` function doesn't care.

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

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

17.4. plural.py, stage 3

Defining separate named functions for each match and apply rule isn't really necessary. You never call them directly; you define them in the `rules` list and call them through there. Let's streamline the rules definition by anonymizing those functions.

Example 17.8. plural3.py

```

import re

rules = \
(
    (
        lambda word: re.search('[sxz]$', word),
        lambda word: re.sub('$', 'es', word)
    ),
    (
        lambda word: re.search('[^aeioudgkprt]h$', word),
        lambda word: re.sub('$', 'es', word)
    ),
    (
        lambda word: re.search('[^aeiouly]$', word),
        lambda word: re.sub('y$', 'ies', word)
    ),
    (
        lambda word: re.search('$', word),
        lambda word: re.sub('$', 's', word)
    )
)

def plural(noun):
    for matchesRule, applyRule in rules:
        if matchesRule(noun):
            return applyRule(noun)

```

❶

❷

- ❶ This is the same set of rules as you defined in stage 2. The only difference is that instead of defining named functions like `match_sxz` and `apply_sxz`, you have "inlined" those function definitions directly into the `rules` list itself, using lambda functions.
- ❷ Note that the `plural` function hasn't changed at all. It iterates through a set of rule functions, checks

the first rule, and if it returns a true value, calls the second rule and returns the value. Same as above, word for word. The only difference is that the rule functions were defined inline, anonymously, using lambda functions. But the `plural` function doesn't care how they were defined; it just gets a list of rules and blindly works through them.

Now to add a new rule, all you need to do is define the functions directly in the `rules` list itself: one match rule, and one apply rule. But defining the rule functions inline like this makes it very clear that you have some unnecessary duplication here. You have four pairs of functions, and they all follow the same pattern. The match function is a single call to `re.search`, and the apply function is a single call to `re.sub`. Let's factor out these similarities.

17.5. `plural.py`, stage 4

Let's factor out the duplication in the code so that defining new rules can be easier.

Example 17.9. `plural4.py`

```
import re

def buildMatchAndApplyFunctions((pattern, search, replace)):
    matchFunction = lambda word: re.search(pattern, word)
    applyFunction = lambda word: re.sub(search, replace, word)
    return (matchFunction, applyFunction)
```

- ❶ `buildMatchAndApplyFunctions` is a function that builds other functions dynamically. It takes `pattern`, `search` and `replace` (actually it takes a tuple, but more on that in a minute), and you can build the match function using the lambda syntax to be a function that takes one parameter (`word`) and calls `re.search` with the `pattern` that was passed to the `buildMatchAndApplyFunctions` function, and the `word` that was passed to the match function you're building. Whoa.
- ❷ Building the apply function works the same way. The apply function is a function that takes one parameter, and calls `re.sub` with the `search` and `replace` parameters that were passed to the `buildMatchAndApplyFunctions` function, and the `word` that was passed to the apply function you're building. This technique of using the values of outside parameters within a dynamic function is called *closures*. You're essentially defining constants within the apply function you're building: it takes one parameter (`word`), but it then acts on that plus two other values (`search` and `replace`) which were set when you defined the apply function.
- ❸ Finally, the `buildMatchAndApplyFunctions` function returns a tuple of two values: the two functions you just created. The constants you defined within those functions (`pattern` within `matchFunction`, and `search` and `replace` within `applyFunction`) stay with those functions, even after you return from `buildMatchAndApplyFunctions`. That's insanely cool.

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

Example 17.10. `plural4.py` continued

```
patterns = \
(
    ('[sxz]$', '$', 'es'),
    ('^aeioudgkprt|h$', '$', 'es'),
    ('(qu|^aeiou)y$', 'y$', 'ies'),
    ('$', '$', 's')
)
rules = map(buildMatchAndApplyFunctions, patterns)
```

- ❶ Our pluralization rules are now defined as a series of strings (not functions). The first string is the regular expression that you would use in `re.search` to see if this rule matches; the second and third are the search and replace expressions you would use in `re.sub` to actually apply the rule to turn a noun into its plural.
- ❷ This line is magic. It takes the list of strings in `patterns` and turns them into a list of functions. How? By mapping the strings to the `buildMatchAndApplyFunctions` function, which just happens to take three strings as parameters and return a tuple of two functions. This means that `rules` ends up being exactly the same as the previous example: a list of tuples, where each tuple is a pair of functions, where the first function is the match function that calls `re.search`, and the second function is the apply function that calls `re.sub`.

I swear I am not making this up: `rules` ends up with exactly the same list of functions as the previous example. Unroll the `rules` definition, and you'll get this:

Example 17.11. Unrolling the rules definition

```
rules = \
(
    (
        lambda word: re.search('[sxz]$', word),
        lambda word: re.sub('$', 'es', word)
    ),
    (
        lambda word: re.search('[^aeioudgkprt]h$', word),
        lambda word: re.sub('$', 'es', word)
    ),
    (
        lambda word: re.search('[^aeiouly]$', word),
        lambda word: re.sub('y$', 'ies', word)
    ),
    (
        lambda word: re.search('$', word),
        lambda word: re.sub('$', 's', word)
    )
)
```

Example 17.12. `plural14.py`, finishing up

```
def plural(noun):
    for matchesRule, applyRule in rules:
        if matchesRule(noun):
            return applyRule(noun)
```

❶

- ❶ Since the `rules` list is the same as the previous example, it should come as no surprise that the `plural` function hasn't changed. Remember, it's completely generic; it takes a list of rule functions and calls them in order. It doesn't care how the rules are defined. In stage 2, they were defined as separate named functions. In stage 3, they were defined as anonymous lambda functions. Now in stage 4, they are built dynamically by mapping the `buildMatchAndApplyFunctions` function onto a list of raw strings. Doesn't matter; the `plural` function still works the same way.

Just in case that wasn't mind-blowing enough, I must confess that there was a subtlety in the definition of `buildMatchAndApplyFunctions` that I skipped over. Let's go back and take another look.

Example 17.13. Another look at `buildMatchAndApplyFunctions`

```
def buildMatchAndApplyFunctions((pattern, search, replace)):
```

❶

- ❶ Notice the double parentheses? This function doesn't actually take three parameters; it actually takes one parameter, a tuple of three elements. But the tuple is expanded when the function is called, and the three elements of the tuple are each assigned to different variables: `pattern`, `search`, and `replace`. Confused yet? Let's see it in action.

Example 17.14. Expanding tuples when calling functions

```
>>> def foo((a, b, c)):  
...     print c  
...     print b  
...     print a  
>>> parameters = ('apple', 'bear', 'catnap')  
>>> foo(parameters) ❶  
catnap  
bear  
apple
```

- ❶ The proper way to call the function `foo` is with a tuple of three elements. When the function is called, the elements are assigned to different local variables within `foo`.

Now let's go back and see why this auto-tuple-expansion trick was necessary. `patterns` was a list of tuples, and each tuple had three elements. When you called `map(buildMatchAndApplyFunctions, patterns)`, that means that `buildMatchAndApplyFunctions` is *not* getting called with three parameters. Using `map` to map a single list onto a function always calls the function with a single parameter: each element of the list. In the case of `patterns`, each element of the list is a tuple, so `buildMatchAndApplyFunctions` always gets called with the tuple, and you use the auto-tuple-expansion trick in the definition of `buildMatchAndApplyFunctions` to assign the elements of that tuple to named variables that you can work with.

17.6. plural.py, stage 5

You've factored out all the duplicate code and added enough abstractions so that the pluralization rules are defined in a list of strings. The next logical step is to take these strings and put them in a separate file, where they can be maintained separately from the code that uses them.

First, let's create a text file that contains the rules you want. No fancy data structures, just space- (or tab-)delimited strings in three columns. You'll call it `rules.en`; "en" stands for English. These are the rules for pluralizing English nouns. You could add other rule files for other languages later.

Example 17.15. rules.en

```
[sxz]$          $          es  
[^aeiou dgkprt]h$  $          es  
[^aeiouly]$       y$        ies  
$                $          s
```

Now let's see how you can use this rules file.

Example 17.16. plural5.py

```
import re  
import string  
  
def buildRule((pattern, search, replace)):  
    return lambda word: re.search(pattern, word) and re.sub(search, replace, word) ❶
```



```
def plural(noun, language='en'):
    lines = file('rules.%s' % language).readlines()
    patterns = map(string.split, lines)
    rules = map(buildRule, patterns)
    for rule in rules:
        result = rule(noun)
        if result: return result
```

②
③
④
⑤
⑥

- ① You're still using the closures technique here (building a function dynamically that uses variables defined outside the function), but now you've combined the separate match and apply functions into one. (The reason for this change will become clear in the next section.) This will let you accomplish the same thing as having two functions, but you'll need to call it differently, as you'll see in a minute.
- ② Our `plural` function now takes an optional second parameter, `language`, which defaults to `en`.
- ③ You use the `language` parameter to construct a filename, then open the file and read the contents into a list. If `language` is `en`, then you'll open the `rules.en` file, read the entire thing, break it up by carriage returns, and return a list. Each line of the file will be one element in the list.
- ④ As you saw, each line in the file really has three values, but they're separated by whitespace (tabs or spaces, it makes no difference). Mapping the `string.split` function onto this list will create a new list where each element is a tuple of three strings. So a line like `[sxz]$ $ es` will be broken up into the tuple `('[sxz]$', '$', 'es')`. This means that `patterns` will end up as a list of tuples, just like you hard-coded it in stage 4.
- ⑤ If `patterns` is a list of tuples, then `rules` will be a list of the functions created dynamically by each call to `buildRule`. Calling `buildRule(('[sxz]$', '$', 'es'))` returns a function that takes a single parameter, `word`. When this returned function is called, it will execute `re.search('[sxz]$', word)` and `re.sub('$', 'es', word)`.
- ⑥ Because you're now building a combined match-and-apply function, you need to call it differently. Just call the function, and if it returns something, then that's the plural; if it returns nothing (`None`), then the rule didn't match and you need to try another rule.

So the improvement here is that you've completely separated the pluralization rules into an external file. Not only can the file be maintained separately from the code, but you've set up a naming scheme where the same `plural` function can use different rule files, based on the `language` parameter.

The downside here is that you're reading that file every time you call the `plural` function. I thought I could get through this entire book without using the phrase "left as an exercise for the reader", but here you go: building a caching mechanism for the language-specific rule files that auto-refreshes itself if the rule files change between calls *is left as an exercise for the reader*. Have fun.

17.7. plural.py, stage 6

Now you're ready to talk about generators.

Example 17.17. plural6.py

```
import re

def rules(language):
    for line in file('rules.%s' % language):
        pattern, search, replace = line.split()
        yield lambda word: re.search(pattern, word) and re.sub(search, replace, word)

def plural(noun, language='en'):
```

```

for applyRule in rules(language):
    result = applyRule(noun)
    if result: return result

```

This uses a technique called generators, which I'm not even going to try to explain until you look at a simpler example first.

Example 17.18. Introducing generators

```

>>> def make_counter(x):
...     print 'entering make_counter'
...     while 1:
...         yield x
...         print 'incrementing x'
...         x = x + 1
...
>>> counter = make_counter(2)
>>> counter
<generator object at 0x001C9C10>
>>> counter.next()
entering make_counter
2
>>> counter.next()
incrementing x
3
>>> counter.next()
incrementing x
4

```

- ❶ The presence of the `yield` keyword in `make_counter` means that this is not a normal function. It is a special kind of function which generates values one at a time. You can think of it as a resumable function. Calling it will return a generator that can be used to generate successive values of `x`.
- ❷ To create an instance of the `make_counter` generator, just call it like any other function. Note that this does not actually execute the function code. You can tell this because the first line of `make_counter` is a `print` statement, but nothing has been printed yet.
- ❸ The `make_counter` function returns a generator object.
- ❹ The first time you call the `next()` method on the generator object, it executes the code in `make_counter` up to the first `yield` statement, and then returns the value that was yielded. In this case, that will be 2, because you originally created the generator by calling `make_counter(2)`.
- ❺ Repeatedly calling `next()` on the generator object *resumes where you left off* and continues until you hit the next `yield` statement. The next line of code waiting to be executed is the `print` statement that prints `incrementing x`, and then after that the `x = x + 1` statement that actually increments it. Then you loop through the `while` loop again, and the first thing you do is `yield x`, which returns the current value of `x` (now 3).
- ❻ The second time you call `counter.next()`, you do all the same things again, but this time `x` is now 4. And so forth. Since `make_counter` sets up an infinite loop, you could theoretically do this forever, and it would just keep incrementing `x` and spitting out values. But let's look at more productive uses of generators instead.

Example 17.19. Using generators instead of recursion

```

def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b

```

- ❶ The Fibonacci sequence is a sequence of numbers where each number is the sum of the two numbers before it. It starts with 0 and 1, goes up slowly at first, then more and more rapidly. To start the sequence, you need two variables: `a` starts at 0, and `b` starts at 1.
- ❷ `a` is the current number in the sequence, so yield it.
- ❸ `b` is the next number in the sequence, so assign that to `a`, but also calculate the next value (`a+b`) and assign that to `b` for later use. Note that this happens in parallel; if `a` is 3 and `b` is 5, then `a, b = b, a+b` will set `a` to 5 (the previous value of `b`) and `b` to 8 (the sum of the previous values of `a` and `b`).

So you have a function that spits out successive Fibonacci numbers. Sure, you could do that with recursion, but this way is easier to read. Also, it works well with `for` loops.

Example 17.20. Generators in `for` loops

```
>>> for n in fibonacci(1000): ❶
...     print n,              ❷
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

- ❶ You can use a generator like `fibonacci` in a `for` loop directly. The `for` loop will create the generator object and successively call the `next()` method to get values to assign to the `for` loop index variable (`n`).
- ❷ Each time through the `for` loop, `n` gets a new value from the `yield` statement in `fibonacci`, and all you do is print it out. Once `fibonacci` runs out of numbers (`a` gets bigger than `max`, which in this case is 1000), then the `for` loop exits gracefully.

OK, let's go back to the `plural` function and see how you're using this.

Example 17.21. Generators that generate dynamic functions

```
def rules(language):
    for line in file('rules.%s' % language): ❶
        pattern, search, replace = line.split() ❷
        yield lambda word: re.search(pattern, word) and re.sub(search, replace, word) ❸

def plural(noun, language='en'):
    for applyRule in rules(language): ❹
        result = applyRule(noun)
        if result: return result
```

- ❶ `for line in file(...)` is a common idiom for reading lines from a file, one line at a time. It works because *file actually returns a generator* whose `next()` method returns the next line of the file. That is so insanely cool, I wet myself just thinking about it.
- ❷ No magic here. Remember that the lines of the rules file have three values separated by whitespace, so `line.split()` returns a tuple of 3 values, and you assign those values to 3 local variables.
- ❸ *And then you yield.* What do you yield? A function, built dynamically with `lambda`, that is actually a closure (it uses the local variables `pattern`, `search`, and `replace` as constants). In other words, `rules` is a generator that spits out rule functions.
- ❹ Since `rules` is a generator, you can use it directly in a `for` loop. The first time through the `for` loop, you will call the `rules` function, which will open the rules file, read the first line out of it, dynamically build a function that matches and applies the first rule defined in the rules file, and yields the dynamically built function. The second time through the `for` loop, you will pick up where you left off in `rules` (which was in the middle of the `for line in file(...)` loop), read the second line of the rules file, dynamically build another function that matches and applies the second rule defined in the rules file, and yields it. And so forth.

What have you gained over stage 5? In stage 5, you read the entire rules file and built a list of all the possible rules before you even tried the first one. Now with generators, you can do everything lazily: you open the first and read the first rule and create a function to try it, but if that works you don't ever read the rest of the file or create any other functions.

Further reading

- PEP 255 (<http://www.python.org/peps/pep-0255.html>) defines generators.
- Python Cookbook (<http://www.activestate.com/ASPN/Python/Cookbook/>) has many more examples of generators (<http://www.google.com/search?q=generators+cookbook+site:aspn.activestate.com>).

17.8. Summary

You talked about several different advanced techniques in this chapter. Not all of them are appropriate for every situation.

You should now be comfortable with all of these techniques:

- Performing string substitution with regular expressions.
- Treating functions as objects, storing them in lists, assigning them to variables, and calling them through those variables.
- Building dynamic functions with `lambda`.
- Building closures, dynamic functions that contain surrounding variables as constants.
- Building generators, resumable functions that perform incremental logic and return different values each time you call them.

Adding abstractions, building functions dynamically, building closures, and using generators can all make your code simpler, more readable, and more flexible. But they can also end up making it more difficult to debug later. It's up to you to find the right balance between simplicity and power.

Chapter 18. Performance Tuning

Performance tuning is a many-splendored thing. Just because Python is an interpreted language doesn't mean you shouldn't worry about code optimization. But don't worry about it *too* much.

18.1. Diving in

There are so many pitfalls involved in optimizing your code, it's hard to know where to start.

Let's start here: *are you sure you need to do it at all?* Is your code really so bad? Is it worth the time to tune it? Over the lifetime of your application, how much time is going to be spent running that code, compared to the time spent waiting for a remote database server, or waiting for user input?

Second, *are you sure you're done coding?* Premature optimization is like spreading frosting on a half-baked cake. You spend hours or days (or more) optimizing your code for performance, only to discover it doesn't do what you need it to do. That's time down the drain.

This is not to say that code optimization is worthless, but you need to look at the whole system and decide whether it's the best use of your time. Every minute you spend optimizing code is a minute you're not spending adding new features, or writing documentation, or playing with your kids, or writing unit tests.

Oh yes, unit tests. It should go without saying that you need a complete set of unit tests before you begin performance tuning. The last thing you need is to introduce new bugs while fiddling with your algorithms.

With these caveats in place, let's look at some techniques for optimizing Python code. The code in question is an implementation of the Soundex algorithm. Soundex was a method used in the early 20th century for categorizing surnames in the United States census. It grouped similar-sounding names together, so even if a name was misspelled, researchers had a chance of finding it. Soundex is still used today for much the same reason, although of course we use computerized database servers now. Most database servers include a Soundex function.

There are several subtle variations of the Soundex algorithm. This is the one used in this chapter:

1. Keep the first letter of the name as-is.
2. Convert the remaining letters to digits, according to a specific table:
 - ◆ B, F, P, and V become 1.
 - ◆ C, G, J, K, Q, S, X, and Z become 2.
 - ◆ D and T become 3.
 - ◆ L becomes 4.
 - ◆ M and N become 5.
 - ◆ R becomes 6.
 - ◆ All other letters become 9.
3. Remove consecutive duplicates.
4. Remove all 9s altogether.
5. If the result is shorter than four characters (the first letter plus three digits), pad the result with trailing zeros.
6. If the result is longer than four characters, discard everything after the fourth character.

For example, my name, *Pilgrim*, becomes P942695. That has no consecutive duplicates, so nothing to do there. Then you remove the 9s, leaving P4265. That's too long, so you discard the excess character, leaving P426.

Another example: *Woo* becomes W99, which becomes W9, which becomes W, which gets padded with zeros to become W000.

Here's a first attempt at a Soundex function:

Example 18.1. `soundex/stage1/soundex1a.py`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
import string, re

charToSoundex = {"A": "9",
                  "B": "1",
                  "C": "2",
                  "D": "3",
                  "E": "9",
                  "F": "1",
                  "G": "2",
                  "H": "9",
                  "I": "9",
                  "J": "2",
                  "K": "2",
                  "L": "4",
                  "M": "5",
                  "N": "5",
                  "O": "9",
                  "P": "1",
                  "Q": "2",
                  "R": "6",
                  "S": "2",
                  "T": "3",
                  "U": "9",
                  "V": "1",
                  "W": "9",
                  "X": "2",
                  "Y": "9",
                  "Z": "2"}

def soundex(source):
    "convert string to Soundex equivalent"

    # Soundex requirements:
    # source string must be at least 1 character
    # and must consist entirely of letters
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"

    # Soundex algorithm:
    # 1. make first character uppercase
    source = source[0].upper() + source[1:]

    # 2. translate all other characters to Soundex digits
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]

    # 3. remove consecutive duplicates
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
```

```

# 4. remove all "9"s
digits3 = re.sub('9', '', digits2)

# 5. pad end with "0"s to 4 characters
while len(digits3) < 4:
    digits3 += "0"

# 6. return first 4 characters
return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())

```

Further Reading on Soundex

- Soundexing and Genealogy (<http://www.avotaynu.com/soundex.html>) gives a chronology of the evolution of the Soundex and its regional variations.

18.2. Using the `timeit` Module

The most important thing you need to know about optimizing Python code is that you shouldn't write your own timing function.

Timing short pieces of code is incredibly complex. How much processor time is your computer devoting to running this code? Are there things running in the background? Are you sure? Every modern computer has background processes running, some all the time, some intermittently. Cron jobs fire off at consistent intervals; background services occasionally "wake up" to do useful things like check for new mail, connect to instant messaging servers, check for application updates, scan for viruses, check whether a disk has been inserted into your CD drive in the last 100 nanoseconds, and so on. Before you start your timing tests, turn everything off and disconnect from the network. Then turn off all the things you forgot to turn off the first time, then turn off the service that's incessantly checking whether the network has come back yet, then ...

And then there's the matter of the variations introduced by the timing framework itself. Does the Python interpreter cache method name lookups? Does it cache code block compilations? Regular expressions? Will your code have side effects if run more than once? Don't forget that you're dealing with small fractions of a second, so small mistakes in your timing framework will irreparably skew your results.

The Python community has a saying: "Python comes with batteries included." Don't write your own timing framework. Python 2.3 comes with a perfectly good one called `timeit`.

Example 18.2. Introducing `timeit`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```

>>> import timeit
>>> t = timeit.Timer("soundex.soundex('Pilgrim')",
...                  "import soundex") ❶
>>> t.timeit() ❷

```

```
8.21683733547
>>> t.repeat(3, 2000000)
[16.48319309109, 16.46128984923, 16.44203948912]
```

- ❶ The `timeit` module defines one class, `Timer`, which takes two arguments. Both arguments are strings. The first argument is the statement you wish to time; in this case, you are timing a call to the `Soundex` function within the `soundex` with an argument of `'Pilgrim'`. The second argument to the `Timer` class is the import statement that sets up the environment for the statement. Internally, `timeit` sets up an isolated virtual environment, manually executes the setup statement (importing the `soundex` module), then manually compiles and executes the timed statement (calling the `Soundex` function).
- ❷ Once you have the `Timer` object, the easiest thing to do is call `timeit()`, which calls your function 1 million times and returns the number of seconds it took to do it.
- ❸ The other major method of the `Timer` object is `repeat()`, which takes two optional arguments. The first argument is the number of times to repeat the entire test, and the second argument is the number of times to call the timed statement within each test. Both arguments are optional, and they default to 3 and 1000000 respectively. The `repeat()` method returns a list of the times each test cycle took, in seconds.

You can use the `timeit` module on the command line to test an existing Python program, without modifying the code. See <http://docs.python.org/lib/node396.html> for documentation on the command-line flags.

Note that `repeat()` returns a list of times. The times will almost never be identical, due to slight variations in how much processor time the Python interpreter is getting (and those pesky background processes that you can't get rid of). Your first thought might be to say "Let's take the average and call that The True Number."

In fact, that's almost certainly wrong. The tests that took longer didn't take longer because of variations in your code or in the Python interpreter; they took longer because of those pesky background processes, or other factors outside of the Python interpreter that you can't fully eliminate. If the different timing results differ by more than a few percent, you still have too much variability to trust the results. Otherwise, take the minimum time and discard the rest.

Python has a handy `min` function that takes a list and returns the smallest value:

```
>>> min(t.repeat(3, 1000000))
8.22203948912
```

The `timeit` module only works if you already know what piece of code you need to optimize. If you have a larger Python program and don't know where your performance problems are, check out the `hotshot` module. (<http://docs.python.org/lib/module-hotshot.html>)

18.3. Optimizing Regular Expressions

The first thing the `Soundex` function checks is whether the input is a non-empty string of letters. What's the best way to do this?

If you answered "regular expressions", go sit in the corner and contemplate your bad instincts. Regular expressions are almost never the right answer; they should be avoided whenever possible. Not only for performance reasons, but simply because they're difficult to debug and maintain. Also for performance reasons.

This code fragment from `soundex/stage1/soundex1a.py` checks whether the function argument `source` is a word made entirely of letters, with at least one letter (not the empty string):

```
allChars = string.uppercase + string.lowercase
if not re.search('^[%s]+$' % allChars, source):
    return "0000"
```


How does `soundex1a.py` perform? For convenience, the `__main__` section of the script contains this code that calls the `timeit` module, sets up a timing test with three different names, tests each name three times, and displays the minimum time for each:

```
if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

So how does `soundex1a.py` perform with this regular expression?

```
C:\samples\soundex\stage1>python soundex1a.py
Woo          W000 19.3356647283
Pilgrim      P426 24.0772053431
Flingjingwaller F452 35.0463220884
```

As you might expect, the algorithm takes significantly longer when called with longer names. There will be a few things we can do to narrow that gap (make the function take less relative time for longer input), but the nature of the algorithm dictates that it will never run in constant time.

The other thing to keep in mind is that we are testing a representative sample of names. `Woo` is a kind of trivial case, in that it gets shorted down to a single letter and then padded with zeros. `Pilgrim` is a normal case, of average length and a mixture of significant and ignored letters. `Flingjingwaller` is extraordinarily long and contains consecutive duplicates. Other tests might also be helpful, but this hits a good range of different cases.

So what about that regular expression? Well, it's inefficient. Since the expression is testing for ranges of characters (A–Z in uppercase, and a–z in lowercase), we can use a shorthand regular expression syntax. Here is `soundex/stage1/soundex1b.py`:

```
if not re.search('[A-Za-z]+$', source):
    return "0000"
```

`timeit` says `soundex1b.py` is slightly faster than `soundex1a.py`, but nothing to get terribly excited about:

```
C:\samples\soundex\stage1>python soundex1b.py
Woo          W000 17.1361133887
Pilgrim      P426 21.8201693232
Flingjingwaller F452 32.7262294509
```

We saw in Section 15.3, *Refactoring* that regular expressions can be compiled and reused for faster results. Since this regular expression never changes across function calls, we can compile it once and use the compiled version. Here is `soundex/stage1/soundex1c.py`:

```
isOnlyChars = re.compile('[A-Za-z]+$').search
def soundex(source):
    if not isOnlyChars(source):
        return "0000"
```

Using a compiled regular expression in `soundex1c.py` is significantly faster:

```
C:\samples\soundex\stage1>python soundex1c.py
Woo          W000 14.5348347346
Pilgrim      P426 19.2784703084
Flingjingwaller F452 30.0893873383
```

But is this the wrong path? The logic here is simple: the input `source` needs to be non-empty, and it needs to be composed entirely of letters. Wouldn't it be faster to write a loop checking each character, and do away with regular expressions altogether?

Here is `soundex/stage1/soundex1d.py`:

```
if not source:
    return "0000"
for c in source:
    if not ('A' <= c <= 'Z') and not ('a' <= c <= 'z'):
        return "0000"
```

It turns out that this technique in `soundex1d.py` is *not* faster than using a compiled regular expression (although it is faster than using a non-compiled regular expression):

```
C:\samples\soundex\stage1>python soundex1d.py
Woo          W000 15.4065058548
Pilgrim      P426 22.2753567842
Flingjingwaller F452 37.5845122774
```

Why isn't `soundex1d.py` faster? The answer lies in the interpreted nature of Python. The regular expression engine is written in C, and compiled to run natively on your computer. On the other hand, this loop is written in Python, and runs through the Python interpreter. Even though the loop is relatively simple, it's not simple enough to make up for the overhead of being interpreted. Regular expressions are never the right answer... except when they are.

It turns out that Python offers an obscure string method. You can be excused for not knowing about it, since it's never been mentioned in this book. The method is called `isalpha()`, and it checks whether a string contains only letters.

This is `soundex/stage1/soundex1e.py`:

```
if (not source) and (not source.isalpha()):
    return "0000"
```

How much did we gain by using this specific method in `soundex1e.py`? Quite a bit.

```
C:\samples\soundex\stage1>python soundex1e.py
Woo          W000 13.5069504644
Pilgrim      P426 18.2199394057
Flingjingwaller F452 28.9975225902
```

Example 18.3. Best Result So Far: `soundex/stage1/soundex1e.py`

```
import string, re

charToSoundex = { "A": "9",
                  "B": "1",
                  "C": "2",
                  "D": "3",
                  "E": "9",
                  "F": "1",
                  "G": "2",
                  "H": "9",
                  "I": "9",
                  "J": "2",
                  "K": "2",
                  "L": "4",
                  "M": "5",
```

```

"N": "5",
"O": "9",
"P": "1",
"Q": "2",
"R": "6",
"S": "2",
"T": "3",
"U": "9",
"V": "1",
"W": "9",
"X": "2",
"Y": "9",
"Z": "2"}

```

```

def soundex(source):
    if (not source) and (not source.isalpha()):
        return "0000"
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())

```

18.4. Optimizing Dictionary Lookups

The second step of the Soundex algorithm is to convert characters to digits in a specific pattern. What's the best way to do this?

The most obvious solution is to define a dictionary with individual characters as keys and their corresponding digits as values, and do dictionary lookups on each character. This is what we have in `soundex/stage1/soundex1c.py` (the current best result so far):

```

charToSoundex = { "A": "9",
                  "B": "1",
                  "C": "2",
                  "D": "3",
                  "E": "9",
                  "F": "1",
                  "G": "2",
                  "H": "9",
                  "I": "9",
                  "J": "2",
                  "K": "2",
                  "L": "4",
                  "M": "5",

```

```

"N": "5",
"O": "9",
"P": "1",
"Q": "2",
"R": "6",
"S": "2",
"T": "3",
"U": "9",
"V": "1",
"W": "9",
"X": "2",
"Y": "9",
"Z": "2"}

```

```

def soundex(source):
    # ... input check omitted for brevity ...
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]

```

You timed `soundex1c.py` already; this is how it performs:

```

C:\samples\soundex\stage1>python soundex1c.py
Woo          W000 14.5341678901
Pilgrim      P426 19.2650071448
Flingjingwaller F452 30.1003563302

```

This code is straightforward, but is it the best solution? Calling `upper()` on each individual character seems inefficient; it would probably be better to call `upper()` once on the entire string.

Then there's the matter of incrementally building the `digits` string. Incrementally building strings like this is horribly inefficient; internally, the Python interpreter needs to create a new string each time through the loop, then discard the old one.

Python is good at lists, though. It can treat a string as a list of characters automatically. And lists are easy to combine into strings again, using the string method `join()`.

Here is `soundex/stage2/soundex2a.py`, which converts letters to digits by using `|` and `lambda`:

```

def soundex(source):
    # ...
    source = source.upper()
    digits = source[0] + "".join(map(lambda c: charToSoundex[c], source[1:]))

```

Surprisingly, `soundex2a.py` is not faster:

```

C:\samples\soundex\stage2>python soundex2a.py
Woo          W000 15.0097526362
Pilgrim      P426 19.254806407
Flingjingwaller F452 29.3790847719

```

The overhead of the anonymous `lambda` function kills any performance you gain by dealing with the string as a list of characters.

`soundex/stage2/soundex2b.py` uses a list comprehension instead of `|` and `lambda`:

```
source = source.upper()
digits = source[0] + "".join([charToSoundex[c] for c in source[1:]])
```

Using a list comprehension in `soundex2b.py` is faster than using `l` and `lambda` in `soundex2a.py`, but still not faster than the original code (incrementally building a string in `soundex1c.py`):

```
C:\samples\soundex\stage2>python soundex2b.py
Woo          W000 13.4221324219
Pilgrim      P426 16.4901234654
Flingjingwaller F452 25.8186157738
```

It's time for a radically different approach. Dictionary lookups are a general purpose tool. Dictionary keys can be any length string (or many other data types), but in this case we are only dealing with single-character keys *and* single-character values. It turns out that Python has a specialized function for handling exactly this situation: the `string.maketrans` function.

This is `soundex/stage2/soundex2c.py`:

```
allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
def soundex(source):
    # ...
    digits = source[0].upper() + source[1:].translate(charToSoundex)
```

What the heck is going on here? `string.maketrans` creates a translation matrix between two strings: the first argument and the second argument. In this case, the first argument is the string `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`, and the second argument is the string `9123912992245591262391929291239129922455912623919292`. See the pattern? It's the same conversion pattern we were setting up longhand with a dictionary. A maps to 9, B maps to 1, C maps to 2, and so forth. But it's not a dictionary; it's a specialized data structure that you can access using the string method `translate`, which translates each character into the corresponding digit, according to the matrix defined by `string.maketrans`.

`timeit` shows that `soundex2c.py` is significantly faster than defining a dictionary and looping through the input and building the output incrementally:

```
C:\samples\soundex\stage2>python soundex2c.py
Woo          W000 11.437645008
Pilgrim      P426 13.2825062962
Flingjingwaller F452 18.5570110168
```

You're not going to get much better than that. Python has a specialized function that does exactly what you want to do; use it and move on.

Example 18.4. Best Result So Far: `soundex/stage2/soundex2c.py`

```
import string, re

allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
isOnlyChars = re.compile('^[A-Za-z]+$').search

def soundex(source):
    if not isOnlyChars(source):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
```

```

digits2 = digits[0]
for d in digits[1:]:
    if digits2[-1] != d:
        digits2 += d
digits3 = re.sub('9', '', digits2)
while len(digits3) < 4:
    digits3 += "0"
return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())

```

18.5. Optimizing List Operations

The third step in the Soundex algorithm is eliminating consecutive duplicate digits. What's the best way to do this?

Here's the code we have so far, in `soundex/stage2/soundex2c.py`:

```

digits2 = digits[0]
for d in digits[1:]:
    if digits2[-1] != d:
        digits2 += d

```

Here are the performance results for `soundex2c.py`:

```

C:\samples\soundex\stage2>python soundex2c.py
Woo          W000 12.6070768771
Pilgrim      P426 14.4033353401
Flingjingwaller F452 19.7774882003

```

The first thing to consider is whether it's efficient to check `digits[-1]` each time through the loop. Are list indexes expensive? Would we be better off maintaining the last digit in a separate variable, and checking that instead?

To answer this question, here is `soundex/stage3/soundex3a.py`:

```

digits2 = ''
last_digit = ''
for d in digits:
    if d != last_digit:
        digits2 += d
        last_digit = d

```

`soundex3a.py` does not run any faster than `soundex2c.py`, and may even be slightly slower (although it's not enough of a difference to say for sure):

```

C:\samples\soundex\stage3>python soundex3a.py
Woo          W000 11.5346048171
Pilgrim      P426 13.3950636184
Flingjingwaller F452 18.6108927252

```

Why isn't `soundex3a.py` faster? It turns out that list indexes in Python are extremely efficient. Repeatedly accessing `digits2[-1]` is no problem at all. On the other hand, manually maintaining the last seen digit in a separate variable means we have *two* variable assignments for each digit we're storing, which wipes out any small

gains we might have gotten from eliminating the list lookup.

Let's try something radically different. If it's possible to treat a string as a list of characters, it should be possible to use a list comprehension to iterate through the list. The problem is, the code needs access to the previous character in the list, and that's not easy to do with a straightforward list comprehension.

However, it is possible to create a list of index numbers using the built-in `range()` function, and use those index numbers to progressively search through the list and pull out each character that is different from the previous character. That will give you a list of characters, and you can use the string method `join()` to reconstruct a string from that.

Here is `soundex/stage3/soundex3b.py`:

```
digits2 = "".join([digits[i] for i in range(len(digits))
                    if i == 0 or digits[i-1] != digits[i]])
```

Is this faster? In a word, no.

```
C:\samples\soundex\stage3>python soundex3b.py
```

```
Woo          W000 14.2245271396
Pilgrim      P426 17.8337165757
Flingjingwaller F452 25.9954005327
```

It's possible that the techniques so far as have been "string-centric". Python can convert a string into a list of characters with a single command: `list('abc')` returns `['a', 'b', 'c']`. Furthermore, lists can be *modified in place* very quickly. Instead of incrementally building a new list (or string) out of the source string, why not move elements around within a single list?

Here is `soundex/stage3/soundex3c.py`, which modifies a list in place to remove consecutive duplicate elements:

```
digits = list(source[0].upper() + source[1:].translate(charToSoundex))
i=0
for item in digits:
    if item==digits[i]: continue
    i+=1
    digits[i]=item
del digits[i+1:]
digits2 = "".join(digits)
```

Is this faster than `soundex3a.py` or `soundex3b.py`? No, in fact it's the slowest method yet:

```
C:\samples\soundex\stage3>python soundex3c.py
```

```
Woo          W000 14.1662554878
Pilgrim      P426 16.0397885765
Flingjingwaller F452 22.1789341942
```

We haven't made any progress here at all, except to try and rule out several "clever" techniques. The fastest code we've seen so far was the original, most straightforward method (`soundex2c.py`). Sometimes it doesn't pay to be clever.

Example 18.5. Best Result So Far: `soundex/stage2/soundex2c.py`

```
import string, re
```

```

allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
isOnlyChars = re.compile('^[A-Za-z]+$').search

def soundex(source):
    if not isOnlyChars(source):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())

```

18.6. Optimizing String Manipulation

The final step of the Soundex algorithm is padding short results with zeros, and truncating long results. What is the best way to do this?

This is what we have so far, taken from `soundex/stage2/soundex2c.py`:

```

digits3 = re.sub('9', '', digits2)
while len(digits3) < 4:
    digits3 += "0"
return digits3[:4]

```

These are the results for `soundex2c.py`:

```

C:\samples\soundex\stage2>python soundex2c.py
Woo          W000 12.6070768771
Pilgrim      P426 14.4033353401
Flingjingwaller F452 19.7774882003

```

The first thing to consider is replacing that regular expression with a loop. This code is from `soundex/stage4/soundex4a.py`:

```

digits3 = ''
for d in digits2:
    if d != '9':
        digits3 += d

```

Is `soundex4a.py` faster? Yes it is:

```

C:\samples\soundex\stage4>python soundex4a.py
Woo          W000 6.62865531792
Pilgrim      P426 9.02247576158
Flingjingwaller F452 13.6328416042

```


But wait a minute. A loop to remove characters from a string? We can use a simple string method for that. Here's `soundex/stage4/soundex4b.py`:

```
digits3 = digits2.replace('9', '')
```

Is `soundex4b.py` faster? That's an interesting question. It depends on the input:

```
C:\samples\soundex\stage4>python soundex4b.py
Woo          W000 6.75477414029
Pilgrim      P426 7.56652144337
Flingjingwaller F452 10.8727729362
```

The string method in `soundex4b.py` is faster than the loop for most names, but it's actually slightly slower than `soundex4a.py` in the trivial case (of a very short name). Performance optimizations aren't always uniform; tuning that makes one case faster can sometimes make other cases slower. In this case, the majority of cases will benefit from the change, so let's leave it at that, but the principle is an important one to remember.

Last but not least, let's examine the final two steps of the algorithm: padding short results with zeros, and truncating long results to four characters. The code you see in `soundex4b.py` does just that, but it's horribly inefficient. Take a look at `soundex/stage4/soundex4c.py` to see why:

```
digits3 += '000'
return digits3[:4]
```

Why do we need a `while` loop to pad out the result? We know in advance that we're going to truncate the result to four characters, and we know that we already have at least one character (the initial letter, which is passed unchanged from the original source variable). That means we can simply add three zeros to the output, then truncate it. Don't get stuck in a rut over the exact wording of the problem; looking at the problem slightly differently can lead to a simpler solution.

How much speed do we gain in `soundex4c.py` by dropping the `while` loop? It's significant:

```
C:\samples\soundex\stage4>python soundex4c.py
Woo          W000 4.89129791636
Pilgrim      P426 7.30642134685
Flingjingwaller F452 10.689832367
```

Finally, there is still one more thing you can do to these three lines of code to make them faster: you can combine them into one line. Take a look at `soundex/stage4/soundex4d.py`:

```
return (digits2.replace('9', '') + '000')[:4]
```

Putting all this code on one line in `soundex4d.py` is barely faster than `soundex4c.py`:

```
C:\samples\soundex\stage4>python soundex4d.py
Woo          W000 4.93624105857
Pilgrim      P426 7.19747593619
Flingjingwaller F452 10.5490700634
```

It is also significantly less readable, and for not much performance gain. Is that worth it? I hope you have good comments. Performance isn't everything. Your optimization efforts must always be balanced against threats to your program's readability and maintainability.

18.7. Summary

This chapter has illustrated several important aspects of performance tuning in Python, and performance tuning in general.

- If you need to choose between regular expressions and writing a loop, choose regular expressions. The regular expression engine is compiled in C and runs natively on your computer; your loop is written in Python and runs through the Python interpreter.
- If you need to choose between regular expressions and string methods, choose string methods. Both are compiled in C, so choose the simpler one.
- General-purpose dictionary lookups are fast, but specialty functions such as `string.maketrans` and string methods such as `isalpha()` are faster. If Python has a custom-tailored function for you, use it.
- Don't be too clever. Sometimes the most obvious algorithm is also the fastest.
- Don't sweat it too much. Performance isn't everything.

I can't emphasize that last point strongly enough. Over the course of this chapter, you made this function three times faster and saved 20 seconds over 1 million function calls. Great. Now think: over the course of those million function calls, how many seconds will your surrounding application wait for a database connection? Or wait for disk I/O? Or wait for user input? Don't spend too much time over-optimizing one algorithm, or you'll ignore obvious improvements somewhere else. Develop an instinct for the sort of code that Python runs well, correct obvious blunders if you find them, and leave the rest alone.

Appendix A. Further reading

Chapter 1. Installing Python

Chapter 2. Your First Python Program

- 2.3. Documenting Functions

- ◆ PEP 257 (<http://www.python.org/peps/pep-0257.html>) defines `doc string` conventions.
- ◆ *Python Style Guide* (<http://www.python.org/doc/essays/styleguide.html>) discusses how to write a good `doc string`.
- ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses conventions for spacing in `doc strings` (<http://www.python.org/doc/current/tut/node6.html#SECTION00675000000000000000>).

- 2.4.2. What's an Object?

- ◆ *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) explains exactly what it means to say that everything in Python is an object (<http://www.python.org/doc/current/ref/objects.html>), because some people are pedantic and like to discuss this sort of thing at great length.
- ◆ *eff-bot* (<http://www.effbot.org/guides/>) summarizes Python objects (<http://www.effbot.org/guides/python-objects.htm>).

- 2.5. Indenting Code

- ◆ *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) discusses cross-platform indentation issues and shows various indentation errors (<http://www.python.org/doc/current/ref/indentation.html>).
- ◆ *Python Style Guide* (<http://www.python.org/doc/essays/styleguide.html>) discusses good indentation style.

- 2.6. Testing Modules

- ◆ *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) discusses the low-level details of importing modules (<http://www.python.org/doc/current/ref/import.html>).

Chapter 3. Native Datatypes

- 3.1.3. Deleting Items From Dictionaries

- ◆ *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) teaches about dictionaries and shows how to use dictionaries to model sparse matrices (<http://www.ibiblio.org/obp/thinkCSpy/chap10.htm>).
- ◆ Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) has a lot of example code using dictionaries (<http://www.faqs.com/knowledge-base/index.phtml/fid/541>).
- ◆ Python Cookbook (<http://www.activestate.com/ASPN/Python/Cookbook/>) discusses how to sort the values of a dictionary by key (<http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52306>).
- ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the dictionary methods (<http://www.python.org/doc/current/lib/typesmapping.html>).

- 3.2.5. Using List Operators

- ◆ *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) teaches about lists and makes an important point about passing lists as function arguments (<http://www.ibiblio.org/obp/thinkCSpy/chap08.htm>).

- ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) shows how to use lists as stacks and queues (<http://www.python.org/doc/current/tut/node7.html#SECTION00711000000000000000>).
- ◆ Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) answers common questions about lists (<http://www.faqs.com/knowledge-base/index.phtml/fid/534>) and has a lot of example code using lists (<http://www.faqs.com/knowledge-base/index.phtml/fid/540>).
- ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the list methods (<http://www.python.org/doc/current/lib/typesesq-mutable.html>).
- 3.3. Introducing Tuples
 - ◆ *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) teaches about tuples and shows how to concatenate tuples (<http://www.ibiblio.org/obp/thinkCSpy/chap10.htm>).
 - ◆ Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) shows how to sort a tuple (<http://www.faqs.com/knowledge-base/view.phtml/aid/4553/fid/587>).
 - ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) shows how to define a tuple with one element (<http://www.python.org/doc/current/tut/node7.html#SECTION00730000000000000000>).
- 3.4.2. Assigning Multiple Values at Once
 - ◆ *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) shows examples of when you can skip the line continuation character (<http://www.python.org/doc/current/ref/implicit-joining.html>) and when you need to use it (<http://www.python.org/doc/current/ref/explicit-joining.html>).
 - ◆ *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) shows how to use multi-variable assignment to swap the values of two variables (<http://www.ibiblio.org/obp/thinkCSpy/chap09.htm>).
- 3.5. Formatting Strings
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the string formatting format characters (<http://www.python.org/doc/current/lib/typesesq-strings.html>).
 - ◆ *Effective AWK Programming* ([http://www-gnats.gnu.org:8080/cgi-bin/info2www?\(gawk\)Top](http://www-gnats.gnu.org:8080/cgi-bin/info2www?(gawk)Top)) discusses all the format characters ([http://www-gnats.gnu.org:8080/cgi-bin/info2www?\(gawk\)Control+Letters](http://www-gnats.gnu.org:8080/cgi-bin/info2www?(gawk)Control+Letters)) and advanced string formatting techniques like specifying width, precision, and zero-padding ([http://www-gnats.gnu.org:8080/cgi-bin/info2www?\(gawk\)Format+Modifiers](http://www-gnats.gnu.org:8080/cgi-bin/info2www?(gawk)Format+Modifiers)).
- 3.6. Mapping Lists
 - ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses another way to map lists using the built-in map function (<http://www.python.org/doc/current/tut/node7.html#SECTION00713000000000000000>).
 - ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) shows how to do nested list comprehensions (<http://www.python.org/doc/current/tut/node7.html#SECTION00714000000000000000>).
- 3.7. Joining Lists and Splitting Strings
 - ◆ Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) answers common questions about strings (<http://www.faqs.com/knowledge-base/index.phtml/fid/480>) and has a lot of example code using strings (<http://www.faqs.com/knowledge-base/index.phtml/fid/539>).
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the string methods (<http://www.python.org/doc/current/lib/string-methods.html>).
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the string module (<http://www.python.org/doc/current/lib/module-string.html>).
 - ◆ *The Whole Python FAQ* (<http://www.python.org/doc/FAQ.html>) explains why join is a string method

(<http://www.python.org/cgi-bin/faqw.py?query=4.96&querytype=simple&casefold=yes&req=search>) instead of a list method.

Chapter 4. The Power Of Introspection

• 4.2. Using Optional and Named Arguments

- ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses exactly when and how default arguments are evaluated (<http://www.python.org/doc/current/tut/node6.html#SECTION00671000000000000000>), which matters when the default value is a list or an expression with side effects.

• 4.3.3. Built-In Functions

- ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents all the built-in functions (<http://www.python.org/doc/current/lib/built-in-funcs.html>) and all the built-in exceptions (<http://www.python.org/doc/current/lib/module-exceptions.html>).

• 4.5. Filtering Lists

- ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses another way to filter lists using the built-in `filter` function (<http://www.python.org/doc/current/tut/node7.html#SECTION00713000000000000000>).

• 4.6.1. Using the and-or Trick

- ◆ *Python Cookbook* (<http://www.activestate.com/ASPN/Python/Cookbook/>) discusses alternatives to the and-or trick (<http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52310>).

• 4.7.1. Real-World lambda Functions

- ◆ *Python Knowledge Base* (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) discusses using `lambda` to call functions indirectly (<http://www.faqs.com/knowledge-base/view.phtml/aid/6081/fid/241>).
- ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) shows how to access outside variables from inside a `lambda` function (<http://www.python.org/doc/current/tut/node6.html#SECTION00674000000000000000>). (PEP 227 (<http://python.sourceforge.net/peps/pep-0227.html>) explains how this will change in future versions of Python.)
- ◆ *The Whole Python FAQ* (<http://www.python.org/doc/FAQ.html>) has examples of obfuscated one-liners using `lambda` (<http://www.python.org/cgi-bin/faqw.py?query=4.15&querytype=simple&casefold=yes&req=search>).

Chapter 5. Objects and Object-Oriented

• 5.2. Importing Modules Using `from module import`

- ◆ *eff-bot* (<http://www.effbot.org/guides/>) has more to say on `import module` vs. `from module import` (<http://www.effbot.org/guides/import-confusion.htm>).
- ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses advanced import techniques, including `from module import *` (<http://www.python.org/doc/current/tut/node8.html#SECTION00841000000000000000>).

• 5.3.2. Knowing When to Use `self` and `__init__`

- ◆ *Learning to Program* (<http://www.freenetpages.co.uk/hp/alan.gauld/>) has a gentler introduction to classes (<http://www.freenetpages.co.uk/hp/alan.gauld/tutclass.htm>).

- ◆ *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) shows how to use classes to model compound datatypes (<http://www.ibiblio.org/obp/thinkCSpy/chap12.htm>).
- ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) has an in-depth look at classes, namespaces, and inheritance (<http://www.python.org/doc/current/tut/node11.html>).
- ◆ Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) answers common questions about classes (<http://www.faqs.com/knowledge-base/index.phtml/fid/242/>).
- 5.4.1. Garbage Collection
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes built-in attributes like `__class__` (<http://www.python.org/doc/current/lib/specialattrs.html>).
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `gc` module (<http://www.python.org/doc/current/lib/module-gc.html>), which gives you low-level control over Python's garbage collection.
- 5.5. Exploring UserDict: A Wrapper Class
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `UserDict` module (<http://www.python.org/doc/current/lib/module-UserDict.html>) and the `copy` module (<http://www.python.org/doc/current/lib/module-copy.html>).
- 5.7. Advanced Special Class Methods
 - ◆ *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) documents all the special class methods (<http://www.python.org/doc/current/ref/specialnames.html>).
- 5.9. Private Functions
 - ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses the inner workings of private variables (<http://www.python.org/doc/current/tut/node11.html#SECTION00116000000000000000>).

Chapter 6. Exceptions and File Handling

- 6.1.1. Using Exceptions For Other Purposes
 - ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses defining and raising your own exceptions, and handling multiple exceptions at once (<http://www.python.org/doc/current/tut/node10.html#SECTION00104000000000000000>).
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the built-in exceptions (<http://www.python.org/doc/current/lib/module-exceptions.html>).
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `getpass` (<http://www.python.org/doc/current/lib/module-getpass.html>) module.
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `traceback` module (<http://www.python.org/doc/current/lib/module-traceback.html>), which provides low-level access to exception attributes after an exception is raised.
 - ◆ *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) discusses the inner workings of the `try...except` block (<http://www.python.org/doc/current/ref/try.html>).
- 6.2.4. Writing to Files
 - ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses reading and writing files, including how to read a file one line at a time into a list (<http://www.python.org/doc/current/tut/node9.html#SECTION00921000000000000000>).
 - ◆ *eff-bot* (<http://www.effbot.org/guides/>) discusses efficiency and performance of various ways of reading a file (<http://www.effbot.org/guides/readline-performance.htm>).
 - ◆ Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) answers

- common questions about files (<http://www.faqs.com/knowledge-base/index.phtml/fid/552>).
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the file object methods (<http://www.python.org/doc/current/lib/bltin-file-objects.html>).
- 6.4. Using `sys.modules`
 - ◆ *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses exactly when and how default arguments are evaluated (<http://www.python.org/doc/current/tut/node6.html#SECTION0067100000000000000000>).
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `sys` (<http://www.python.org/doc/current/lib/module-sys.html>) module.
- 6.5. Working with Directories
 - ◆ Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) answers questions about the `os` module (<http://www.faqs.com/knowledge-base/index.phtml/fid/240>).
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `os` (<http://www.python.org/doc/current/lib/module-os.html>) module and the `os.path` (<http://www.python.org/doc/current/lib/module-os.path.html>) module.

Chapter 7. Regular Expressions

- 7.6. Case study: Parsing Phone Numbers
 - ◆ Regular Expression HOWTO (<http://py-howto.sourceforge.net/regex/regex.html>) teaches about regular expressions and how to use them in Python.
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes the `re` module (<http://www.python.org/doc/current/lib/module-re.html>).

Chapter 8. HTML Processing

- 8.4. Introducing `BaseHTMLProcessor.py`
 - ◆ W3C (<http://www.w3.org/>) discusses character and entity references (<http://www.w3.org/TR/REC-html40/charset.html#entities>).
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) confirms your suspicions that the `htmlentitydefs` module (<http://www.python.org/doc/current/lib/module-htmlentitydefs.html>) is exactly what it sounds like.
- 8.9. Putting it all together
 - ◆ You thought I was kidding about the server-side scripting idea. So did I, until I found this web-based dialectizer (<http://rinkworks.com/dialect/>). Unfortunately, source code does not appear to be available.

Chapter 9. XML Processing

- 9.4. Unicode
 - ◆ Unicode.org (<http://www.unicode.org/>) is the home page of the unicode standard, including a brief technical introduction (<http://www.unicode.org/standard/principles.html>).
 - ◆ Unicode Tutorial (http://www.reportlab.com/i18n/python_unicode_tutorial.html) has some more examples of how to use Python's unicode functions, including how to force Python to coerce unicode into ASCII even when it doesn't really want to.
 - ◆ PEP 263 (<http://www.python.org/peps/pep-0263.html>) goes into more detail about how and when to define a character encoding in your `.py` files.

Chapter 10. Scripts and Streams

Chapter 11. HTTP Web Services

- 11.1. Diving in
 - ◆ Paul Prescod believes that pure HTTP web services are the future of the Internet (<http://webservices.xml.com/pub/a/ws/2002/02/06/rest.html>).

Chapter 12. SOAP Web Services

- 12.1. Diving In
 - ◆ <http://www.xmethods.net/> is a repository of public access SOAP web services.
 - ◆ The SOAP specification (<http://www.w3.org/TR/soap/>) is surprisingly readable, if you like that sort of thing.
- 12.8. Troubleshooting SOAP Web Services
 - ◆ New developments for SOAPpy (<http://www-106.ibm.com/developerworks/webservices/library/ws-pyth17.html>) steps through trying to connect to another SOAP service that doesn't quite work as advertised.

Chapter 13. Unit Testing

- 13.1. Introduction to Roman numerals
 - ◆ This site (<http://www.wilkiecollins.demon.co.uk/roman/front.htm>) has more on Roman numerals, including a fascinating history (<http://www.wilkiecollins.demon.co.uk/roman/intro.htm>) of how Romans and other civilizations really used them (short answer: haphazardly and inconsistently).
- 13.3. Introducing `romantest.py`
 - ◆ The PyUnit home page (<http://pyunit.sourceforge.net/>) has an in-depth discussion of using the `unittest` framework (<http://pyunit.sourceforge.net/pyunit.html>), including advanced features not covered in this chapter.
 - ◆ The PyUnit FAQ (<http://pyunit.sourceforge.net/pyunit.html>) explains why test cases are stored separately (<http://pyunit.sourceforge.net/pyunit.html#WHERE>) from the code they test.
 - ◆ *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes the `unittest` (<http://www.python.org/doc/current/lib/module-unittest.html>) module.
 - ◆ `ExtremeProgramming.org` (<http://www.extremeprogramming.org/>) discusses why you should write unit tests (<http://www.extremeprogramming.org/rules/unittests.html>).
 - ◆ The Portland Pattern Repository (<http://www.c2.com/cgi/wiki>) has an ongoing discussion of unit tests (<http://www.c2.com/cgi/wiki?UnitTests>), including a standard definition (<http://www.c2.com/cgi/wiki?StandardDefinitionOfUnitTest>), why you should code unit tests first (<http://www.c2.com/cgi/wiki?CodeUnitTestFirst>), and several in-depth case studies (<http://www.c2.com/cgi/wiki?UnitTestTrial>).

Chapter 14. Test-First Programming

Chapter 15. Refactoring

- 15.5. Summary

- ◆ XProgramming.com (<http://www.xprogramming.com/>) has links to download unit testing frameworks (<http://www.xprogramming.com/software.htm>) for many different languages.

Chapter 16. Functional Programming

Chapter 17. Dynamic functions

- 17.7. plural.py, stage 6
 - ◆ PEP 255 (<http://www.python.org/peps/pep-0255.html>) defines generators.
 - ◆ Python Cookbook (<http://www.activestate.com/ASPN/Python/Cookbook/>) has many more examples of generators (<http://www.google.com/search?q=generators+cookbook+site:aspn.activestate.com>).

Chapter 18. Performance Tuning

- 18.1. Diving in
 - ◆ Soundexing and Genealogy (<http://www.avotaynu.com/soundex.html>) gives a chronology of the evolution of the Soundex and its regional variations.

Appendix B. A 5-minute review

Chapter 1. Installing Python

- 1.1. Which Python is right for you?

The first thing you need to do with Python is install it. Or do you?

- 1.2. Python on Windows

On Windows, you have a couple choices for installing Python.

- 1.3. Python on Mac OS X

On Mac OS X, you have two choices for installing Python: install it, or don't install it. You probably want to install it.

- 1.4. Python on Mac OS 9

Mac OS 9 does not come with any version of Python, but installation is very simple, and there is only one choice.

- 1.5. Python on RedHat Linux

Download the latest Python RPM by going to <http://www.python.org/ftp/python/> and selecting the highest version number listed, then selecting the `rpms/` directory within that. Then download the RPM with the highest version number. You can install it with the **rpm** command, as shown here:

- 1.6. Python on Debian GNU/Linux

If you are lucky enough to be running Debian GNU/Linux, you install Python through the **apt** command.

- 1.7. Python Installation from Source

If you prefer to build from source, you can download the Python source code from <http://www.python.org/ftp/python/>. Select the highest version number listed, download the `.tar.gz` file), and then do the usual **configure, make, make install** dance.

- 1.8. The Interactive Shell

Now that you have Python installed, what's this interactive shell thing you're running?

- 1.9. Summary

You should now have a version of Python installed that works for you.

Chapter 2. Your First Python Program

- 2.1. Diving in

Here is a complete, working Python program.

- 2.2. Declaring Functions

Python has functions like most other languages, but it does not have separate header files like C++ or `interface/implementation` sections like Pascal. When you need a function, just declare it, like this:

- 2.3. Documenting Functions

You can document a Python function by giving it a `doc string`.

- 2.4. Everything Is an Object

A function, like everything else in Python, is an object.

- 2.5. Indenting Code

Python functions have no explicit `begin` or `end`, and no curly braces to mark where the function code starts and stops. The only delimiter is a colon (`:`) and the indentation of the code itself.

- 2.6. Testing Modules

Python modules are objects and have several useful attributes. You can use this to easily test your modules as you write them. Here's an example that uses the `if __name__` trick.

Chapter 3. Native Datatypes

- 3.1. Introducing Dictionaries

One of Python's built-in datatypes is the dictionary, which defines one-to-one relationships between keys and values.

- 3.2. Introducing Lists

Lists are Python's workhorse datatype. If your only experience with lists is arrays in Visual Basic or (God forbid) the datastore in Powerbuilder, brace yourself for Python lists.

- 3.3. Introducing Tuples

A tuple is an immutable list. A tuple can not be changed in any way once it is created.

- 3.4. Declaring variables

Python has local and global variables like most other languages, but it has no explicit variable declarations. Variables spring into existence by being assigned a value, and they are automatically destroyed when they go out of scope.

- 3.5. Formatting Strings

Python supports formatting values into strings. Although this can include very complicated expressions, the most basic usage is to insert values into a string with the `%s` placeholder.

- 3.6. Mapping Lists

One of the most powerful features of Python is the list comprehension, which provides a compact way of mapping a list into another list by applying a function to each of the elements of the list.

- 3.7. Joining Lists and Splitting Strings

You have a list of key-value pairs in the form `key=value`, and you want to join them into a single string. To join any list of strings into a single string, use the `join` method of a string object.

- 3.8. Summary

The `odbc helper.py` program and its output should now make perfect sense.

Chapter 4. The Power Of Introspection

- 4.1. Diving In

Here is a complete, working Python program. You should understand a good deal about it just by looking at it. The numbered lines illustrate concepts covered in Chapter 2, *Your First Python Program*. Don't worry if the rest of the code looks intimidating; you'll learn all about it throughout this chapter.

- 4.2. Using Optional and Named Arguments

Python allows function arguments to have default values; if the function is called without the argument, the argument gets its default value. Furthermore, arguments can be specified in any order by using named arguments. Stored procedures in SQL Server Transact/SQL can do this, so if you're a SQL Server scripting guru, you can skim this part.

- 4.3. Using type, str, dir, and Other Built-In Functions

Python has a small set of extremely useful built-in functions. All other functions are partitioned off into modules. This was actually a conscious design decision, to keep the core language from getting bloated like other scripting languages (cough cough, Visual Basic).

- 4.4. Getting Object References With `getattr`

You already know that Python functions are objects. What you don't know is that you can get a reference to a function without knowing its name until run-time, by using the `getattr` function.

- 4.5. Filtering Lists

As you know, Python has powerful capabilities for mapping lists into other lists, via list comprehensions (Section 3.6, Mapping Lists). This can be combined with a filtering mechanism, where some elements in the list are mapped while others are skipped entirely.

- 4.6. The Peculiar Nature of `and` and `or`

In Python, `and` and `or` perform boolean logic as you would expect, but they do not return boolean values; instead, they return one of the actual values they are comparing.

- 4.7. Using `lambda` Functions

Python supports an interesting syntax that lets you define one-line mini-functions on the fly. Borrowed from Lisp, these so-called `lambda` functions can be used anywhere a function is required.

- 4.8. Putting It All Together

The last line of code, the only one you haven't deconstructed yet, is the one that does all the work. But by now the work is easy, because everything you need is already set up just the way you need it. All the dominoes are in place; it's time to knock them down.

- 4.9. Summary

The `apihelper.py` program and its output should now make perfect sense.

Chapter 5. Objects and Object-Oriented

- 5.1. Diving In

Here is a complete, working Python program. Read the `doc strings` of the module, the classes, and the functions to get an overview of what this program does and how it works. As usual, don't worry about the stuff you don't understand; that's what the rest of the chapter is

for.

- 5.2. Importing Modules Using `from module import`

Python has two ways of importing modules. Both are useful, and you should know when to use each. One way, `import module`, you've already seen in Section 2.4, *Everything Is an Object*. The other way accomplishes the same thing, but it has subtle and important differences.

- 5.3. Defining Classes

Python is fully object-oriented: you can define your own classes, inherit from your own or built-in classes, and instantiate the classes you've defined.

- 5.4. Instantiating Classes

Instantiating classes in Python is straightforward. To instantiate a class, simply call the class as if it were a function, passing the arguments that the `__init__` method defines. The return value will be the newly created object.

- 5.5. Exploring `UserDict`: A Wrapper Class

As you've seen, `FileInfo` is a class that acts like a dictionary. To explore this further, let's look at the `UserDict` class in the `UserDict` module, which is the ancestor of the `FileInfo` class. This is nothing special; the class is written in Python and stored in a `.py` file, just like any other Python code. In particular, it's stored in the `lib` directory in your Python installation.

- 5.6. Special Class Methods

In addition to normal class methods, there are a number of special methods that Python classes can define. Instead of being called directly by your code (like normal methods), special methods are called for you by Python in particular circumstances or when specific syntax is used.

- 5.7. Advanced Special Class Methods

Python has more special methods than just `__getitem__` and `__setitem__`. Some of them let you emulate functionality that you may not even know about.

- 5.8. Introducing Class Attributes

You already know about data attributes, which are variables owned by a specific instance of a class. Python also supports class attributes, which are variables owned by the class itself.

- 5.9. Private Functions

Unlike in most languages, whether a Python function, method, or attribute is private or public is determined entirely by its name.

- 5.10. Summary

That's it for the hard-core object trickery. You'll see a real-world application of special class methods in Chapter 12, which uses `getattr` to create a proxy to a remote web service.

Chapter 6. Exceptions and File Handling

- 6.1. Handling Exceptions

Like many other programming languages, Python has exception handling via `try...except` blocks.

- 6.2. Working with File Objects

Python has a built-in function, `open`, for opening a file on disk. `open` returns a file object, which has methods and attributes for getting information about and manipulating the opened file.

- 6.3. Iterating with `for` Loops

Like most other languages, Python has `for` loops. The only reason you haven't seen them until now is that Python is good at so many other things that you don't need them as often.

- 6.4. Using `sys.modules`

Modules, like everything else in Python, are objects. Once imported, you can always get a reference to a module through the global dictionary `sys.modules`.

- 6.5. Working with Directories

The `os.path` module has several functions for manipulating files and directories. Here, we're looking at handling pathnames and listing the contents of a directory.

- 6.6. Putting It All Together

Once again, all the dominoes are in place. You've seen how each line of code works. Now let's step back and see how it all fits together.

- 6.7. Summary

The `fileinfo.py` program introduced in Chapter 5 should now make perfect sense.

Chapter 7. Regular Expressions

- 7.1. Diving In

If what you're trying to do can be accomplished with string functions, you should use them. They're fast and simple and easy to read, and there's a lot to be said for fast, simple, readable code. But if you find yourself using a lot of different string functions with `if` statements to handle special cases, or if you're combining them with `split` and `join` and list comprehensions in weird unreadable ways, you may need to move up to regular expressions.

- 7.2. Case Study: Street Addresses

This series of examples was inspired by a real-life problem I had in my day job several years ago, when I needed to scrub and standardize street addresses exported from a legacy system before importing them into a newer system. (See, I don't just make this stuff up; it's actually useful.) This example shows how I approached the problem.

- 7.3. Case Study: Roman Numerals

You've most likely seen Roman numerals, even if you didn't recognize them. You may have seen them in copyrights of old movies and television shows ("Copyright MCMXLVI" instead of "Copyright 1946"), or on the dedication walls of libraries or universities ("established MDCCCLXXXVII" instead of "established 1888"). You may also have seen them in outlines and bibliographical references. It's a system of representing numbers that really does date back to the ancient Roman empire (hence the name).

- 7.4. Using the `{n,m}` Syntax

In the previous section, you were dealing with a pattern where the same character could be repeated up to three times. There is another way to express this in regular expressions, which

some people find more readable. First look at the method we already used in the previous example.

- 7.5. Verbose Regular Expressions

So far you've just been dealing with what I'll call "compact" regular expressions. As you've seen, they are difficult to read, and even if you figure out what one does, that's no guarantee that you'll be able to understand it six months later. What you really need is inline documentation.

- 7.6. Case study: Parsing Phone Numbers

So far you've concentrated on matching whole patterns. Either the pattern matches, or it doesn't. But regular expressions are much more powerful than that. When a regular expression *does* match, you can pick out specific pieces of it. You can find out what matched where.

- 7.7. Summary

This is just the tiniest tip of the iceberg of what regular expressions can do. In other words, even though you're completely overwhelmed by them now, believe me, you ain't seen nothing yet.

Chapter 8. HTML Processing

- 8.1. Diving in

I often see questions on `comp.lang.python` (<http://groups.google.com/groups?group=comp.lang.python>) like "How can I list all the [headers|images|links] in my HTML document?" "How do I parse/translate/munge the text of my HTML document but leave the tags alone?" "How can I add/remove/quote attributes of all my HTML tags at once?" This chapter will answer all of these questions.

- 8.2. Introducing `sgmlib.py`

HTML processing is broken into three steps: breaking down the HTML into its constituent pieces, fiddling with the pieces, and reconstructing the pieces into HTML again. The first step is done by `sgmlib.py`, a part of the standard Python library.

- 8.3. Extracting data from HTML documents

To extract data from HTML documents, subclass the `SGMLParser` class and define methods for each tag or entity you want to capture.

- 8.4. Introducing `BaseHTMLProcessor.py`

`SGMLParser` doesn't produce anything by itself. It parses and parses and parses, and it calls a method for each interesting thing it finds, but the methods don't do anything. `SGMLParser` is an HTML *consumer*: it takes HTML and breaks it down into small, structured pieces. As you saw in the previous section, you can subclass `SGMLParser` to define classes that catch specific tags and produce useful things, like a list of all the links on a web page. Now you'll take this one step further by defining a class that catches everything `SGMLParser` throws at it and reconstructs the complete HTML document. In technical terms, this class will be an HTML *producer*.

- 8.5. locals and globals

Let's digress from HTML processing for a minute and talk about how Python handles variables. Python has two built-in functions, `locals` and `globals`, which provide

dictionary-based access to local and global variables.

- 8.6. Dictionary-based string formatting

There is an alternative form of string formatting that uses dictionaries instead of tuples of values.

- 8.7. Quoting attribute values

A common question on `comp.lang.python`

(<http://groups.google.com/groups?group=comp.lang.python>) is "I have a bunch of HTML documents with unquoted attribute values, and I want to properly quote them all. How can I do this?"^[4] (This is generally precipitated by a project manager who has found the HTML-is-a-standard religion joining a large project and proclaiming that all pages must validate against an HTML validator. Unquoted attribute values are a common violation of the HTML standard.) Whatever the reason, unquoted attribute values are easy to fix by feeding HTML through `BaseHTMLProcessor`.

- 8.8. Introducing `dialect.py`

`Dialectizer` is a simple (and silly) descendant of `BaseHTMLProcessor`. It runs blocks of text through a series of substitutions, but it makes sure that anything within a `<pre> . . . </pre>` block passes through unaltered.

- 8.9. Putting it all together

It's time to put everything you've learned so far to good use. I hope you were paying attention.

- 8.10. Summary

Python provides you with a powerful tool, `sgmllib.py`, to manipulate HTML by turning its structure into an object model. You can use this tool in many different ways.

Chapter 9. XML Processing

- 9.1. Diving in

There are two basic ways to work with XML. One is called SAX ("Simple API for XML"), and it works by reading the XML a little bit at a time and calling a method for each element it finds. (If you read Chapter 8, *HTML Processing*, this should sound familiar, because that's how the `sgmllib` module works.) The other is called DOM ("Document Object Model"), and it works by reading in the entire XML document at once and creating an internal representation of it using native Python classes linked in a tree structure. Python has standard modules for both kinds of parsing, but this chapter will only deal with using the DOM.

- 9.2. Packages

Actually parsing an XML document is very simple: one line of code. However, before you get to that line of code, you need to take a short detour to talk about packages.

- 9.3. Parsing XML

As I was saying, actually parsing an XML document is very simple: one line of code. Where you go from there is up to you.

- 9.4. Unicode

Unicode is a system to represent characters from all the world's different languages. When Python parses an XML document, all data is stored in memory as unicode.

- 9.5. Searching for elements

Traversing XML documents by stepping through each node can be tedious. If you're looking for something in particular, buried deep within your XML document, there is a shortcut you can use to find it quickly: `getElementsByTagName`.

- 9.6. Accessing element attributes

XML elements can have one or more attributes, and it is incredibly simple to access them once you have parsed an XML document.

- 9.7. Segue

OK, that's it for the hard-core XML stuff. The next chapter will continue to use these same example programs, but focus on other aspects that make the program more flexible: using streams for input processing, using `getattr` for method dispatching, and using command-line flags to allow users to reconfigure the program without changing the code.

Chapter 10. Scripts and Streams

- 10.1. Abstracting input sources

One of Python's greatest strengths is its dynamic binding, and one powerful use of dynamic binding is the *file-like object*.

- 10.2. Standard input, output, and error

UNIX users are already familiar with the concept of standard input, standard output, and standard error. This section is for the rest of you.

- 10.3. Caching node lookups

`kgp.py` employs several tricks which may or may not be useful to you in your XML processing. The first one takes advantage of the consistent structure of the input documents to build a cache of nodes.

- 10.4. Finding direct children of a node

Another useful technique when parsing XML documents is finding all the direct child elements of a particular element. For instance, in the grammar files, a `ref` element can have several `p` elements, each of which can contain many things, including other `p` elements. You want to find just the `p` elements that are children of the `ref`, not `p` elements that are children of other `p` elements.

- 10.5. Creating separate handlers by node type

The third useful XML processing tip involves separating your code into logical functions, based on node types and element names. Parsed XML documents are made up of various types of nodes, each represented by a Python object. The root level of the document itself is represented by a `Document` object. The `Document` then contains one or more `Element` objects (for actual XML tags), each of which may contain other `Element` objects, `Text` objects (for bits of text), or `Comment` objects (for embedded comments). Python makes it easy to write a dispatcher to separate the logic for each node type.

- 10.6. Handling command-line arguments

Python fully supports creating programs that can be run on the command line, complete with command-line arguments and either short- or long-style flags to specify various options. None of this is XML-specific, but this script makes good use of command-line processing, so it seemed like a good time to mention it.

- 10.7. Putting it all together

You've covered a lot of ground. Let's step back and see how all the pieces fit together.

- 10.8. Summary

Python comes with powerful libraries for parsing and manipulating XML documents. The `minidom` takes an XML file and parses it into Python objects, providing for random access to arbitrary elements. Furthermore, this chapter shows how Python can be used to create a "real" standalone command-line script, complete with command-line flags, command-line arguments, error handling, even the ability to take input from the piped result of a previous program.

Chapter 11. HTTP Web Services

- 11.1. Diving in

You've learned about HTML processing and XML processing, and along the way you saw how to download a web page and how to parse XML from a URL, but let's dive into the more general topic of HTTP web services.

- 11.2. How not to fetch data over HTTP

Let's say you want to download a resource over HTTP, such as a syndicated Atom feed. But you don't just want to download it once; you want to download it over and over again, every hour, to get the latest news from the site that's offering the news feed. Let's do it the quick-and-dirty way first, and then see how you can do better.

- 11.3. Features of HTTP

There are five important features of HTTP which you should support.

- 11.4. Debugging HTTP web services

First, let's turn on the debugging features of Python's HTTP library and see what's being sent over the wire. This will be useful throughout the chapter, as you add more and more features.

- 11.5. Setting the User-Agent

The first step to improving your HTTP web services client is to identify yourself properly with a `User-Agent`. To do that, you need to move beyond the basic `urllib` and dive into `urllib2`.

- 11.6. Handling Last-Modified and ETag

Now that you know how to add custom HTTP headers to your web service requests, let's look at adding support for `Last-Modified` and `ETag` headers.

- 11.7. Handling redirects

You can support permanent and temporary redirects using a different kind of custom URL handler.

- 11.8. Handling compressed data

The last important HTTP feature you want to support is compression. Many web services have the ability to send data compressed, which can cut down the amount of data sent over the wire by 60% or more. This is especially true of XML web services, since XML data compresses very well.

- 11.9. Putting it all together

You've seen all the pieces for building an intelligent HTTP web services client. Now let's see

how they all fit together.

- 11.10. Summary

The `openanything.py` and its functions should now make perfect sense.

Chapter 12. SOAP Web Services

- 12.1. Diving In

You use Google, right? It's a popular search engine. Have you ever wished you could programmatically access Google search results? Now you can. Here is a program to search Google from Python.

- 12.2. Installing the SOAP Libraries

Unlike the other code in this book, this chapter relies on libraries that do not come pre-installed with Python.

- 12.3. First Steps with SOAP

The heart of SOAP is the ability to call remote functions. There are a number of public access SOAP servers that provide simple functions for demonstration purposes.

- 12.4. Debugging SOAP Web Services

The SOAP libraries provide an easy way to see what's going on behind the scenes.

- 12.5. Introducing WSDL

The `SOAPProxy` class proxies local method calls and transparently turns them into invocations of remote SOAP methods. As you've seen, this is a lot of work, and `SOAPProxy` does it quickly and transparently. What it doesn't do is provide any means of method introspection.

- 12.6. Introspecting SOAP Web Services with WSDL

Like many things in the web services arena, WSDL has a long and checkered history, full of political strife and intrigue. I will skip over this history entirely, since it bores me to tears. There were other standards that tried to do similar things, but WSDL won, so let's learn how to use it.

- 12.7. Searching Google

Let's finally turn to the sample code that you saw at the beginning of this chapter, which does something more useful and exciting than get the current temperature.

- 12.8. Troubleshooting SOAP Web Services

Of course, the world of SOAP web services is not all happiness and light. Sometimes things go wrong.

- 12.9. Summary

SOAP web services are very complicated. The specification is very ambitious and tries to cover many different use cases for web services. This chapter has touched on some of the simpler use cases.

Chapter 13. Unit Testing

- 13.1. Introduction to Roman numerals

In previous chapters, you "dived in" by immediately looking at code and trying to understand it as quickly as possible. Now that you have some Python under your belt, you're going to step back and look at the steps that happen *before* the code gets written.

- 13.2. Diving in

Now that you've completely defined the behavior you expect from your conversion functions, you're going to do something a little unexpected: you're going to write a test suite that puts these functions through their paces and makes sure that they behave the way you want them to. You read that right: you're going to write code that tests code that you haven't written yet.

- 13.3. Introducing `romantest.py`

This is the complete test suite for your Roman numeral conversion functions, which are yet to be written but will eventually be in `roman.py`. It is not immediately obvious how it all fits together; none of these classes or methods reference any of the others. There are good reasons for this, as you'll see shortly.

- 13.4. Testing for success

The most fundamental part of unit testing is constructing individual test cases. A test case answers a single question about the code it is testing.

- 13.5. Testing for failure

It is not enough to test that functions succeed when given good input; you must also test that they fail when given bad input. And not just any sort of failure; they must fail in the way you expect.

- 13.6. Testing for sanity

Often, you will find that a unit of code contains a set of reciprocal functions, usually in the form of conversion functions where one converts A to B and the other converts B to A. In these cases, it is useful to create a "sanity check" to make sure that you can convert A to B and back to A without losing precision, incurring rounding errors, or triggering any other sort of bug.

Chapter 14. Test-First Programming

- 14.1. `roman.py`, stage 1

Now that the unit tests are complete, it's time to start writing the code that the test cases are attempting to test. You're going to do this in stages, so you can see all the unit tests fail, then watch them pass one by one as you fill in the gaps in `roman.py`.

- 14.2. `roman.py`, stage 2

Now that you have the framework of the `roman` module laid out, it's time to start writing code and passing test cases.

- 14.3. `roman.py`, stage 3

Now that `toRoman` behaves correctly with good input (integers from 1 to 3999), it's time to make it behave correctly with bad input (everything else).

- 14.4. `roman.py`, stage 4

Now that `toRoman` is done, it's time to start coding `fromRoman`. Thanks to the rich data structure that maps individual Roman numerals to integer values, this is no more difficult than the `toRoman` function.

- 14.5. roman.py, stage 5

Now that `fromRoman` works properly with good input, it's time to fit in the last piece of the puzzle: making it work properly with bad input. That means finding a way to look at a string and determine if it's a valid Roman numeral. This is inherently more difficult than validating numeric input in `toRoman`, but you have a powerful tool at your disposal: regular expressions.

Chapter 15. Refactoring

- 15.1. Handling bugs

Despite your best efforts to write comprehensive unit tests, bugs happen. What do I mean by "bug"? A bug is a test case you haven't written yet.

- 15.2. Handling changing requirements

Despite your best efforts to pin your customers to the ground and extract exact requirements from them on pain of horrible nasty things involving scissors and hot wax, requirements will change. Most customers don't know what they want until they see it, and even if they do, they aren't that good at articulating what they want precisely enough to be useful. And even if they do, they'll want more in the next release anyway. So be prepared to update your test cases as requirements change.

- 15.3. Refactoring

The best thing about comprehensive unit testing is not the feeling you get when all your test cases finally pass, or even the feeling you get when someone else blames you for breaking their code and you can actually *prove* that you didn't. The best thing about unit testing is that it gives you the freedom to refactor mercilessly.

- 15.4. Postscript

A clever reader read the previous section and took it to the next level. The biggest headache (and performance drain) in the program as it is currently written is the regular expression, which is required because you have no other way of breaking down a Roman numeral. But there's only 5000 of them; why don't you just build a lookup table once, then simply read that? This idea gets even better when you realize that you don't need to use regular expressions at all. As you build the lookup table for converting integers to Roman numerals, you can build the reverse lookup table to convert Roman numerals to integers.

- 15.5. Summary

Unit testing is a powerful concept which, if properly implemented, can both reduce maintenance costs and increase flexibility in any long-term project. It is also important to understand that unit testing is not a panacea, a Magic Problem Solver, or a silver bullet. Writing good test cases is hard, and keeping them up to date takes discipline (especially when customers are screaming for critical bug fixes). Unit testing is not a replacement for other forms of testing, including functional testing, integration testing, and user acceptance testing. But it is feasible, and it does work, and once you've seen it work, you'll wonder how you ever got along without it.

Chapter 16. Functional Programming

- 16.1. Diving in

In Chapter 13, *Unit Testing*, you learned about the philosophy of unit testing. In Chapter 14, *Test–First Programming*, you stepped through the implementation of basic unit tests in Python. In Chapter 15, *Refactoring*, you saw how unit testing makes large–scale refactoring easier. This chapter will build on those sample programs, but here we will focus more on advanced Python–specific techniques, rather than on unit testing itself.

- 16.2. Finding the path

When running Python scripts from the command line, it is sometimes useful to know where the currently running script is located on disk.

- 16.3. Filtering lists revisited

You're already familiar with using list comprehensions to filter lists. There is another way to accomplish this same thing, which some people feel is more expressive.

- 16.4. Mapping lists revisited

You're already familiar with using list comprehensions to map one list into another. There is another way to accomplish the same thing, using the built–in `map` function. It works much the same way as the `filter` function.

- 16.5. Data–centric programming

By now you're probably scratching your head wondering why this is better than using `for` loops and straight function calls. And that's a perfectly valid question. Mostly, it's a matter of perspective. Using `map` and `filter` forces you to center your thinking around your data.

- 16.6. Dynamically importing modules

OK, enough philosophizing. Let's talk about dynamically importing modules.

- 16.7. Putting it all together

You've learned enough now to deconstruct the first seven lines of this chapter's code sample: reading a directory and importing selected modules within it.

- 16.8. Summary

The `regression.py` program and its output should now make perfect sense.

Chapter 17. Dynamic functions

- 17.1. Diving in

I want to talk about plural nouns. Also, functions that return other functions, advanced regular expressions, and generators. Generators are new in Python 2.3. But first, let's talk about how to make plural nouns.

- 17.2. `plural.py`, stage 1

So you're looking at words, which at least in English are strings of characters. And you have rules that say you need to find different combinations of characters, and then do different things to them. This sounds like a job for regular expressions.

- 17.3. `plural.py`, stage 2

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

- 17.4. `plural.py`, stage 3

Defining separate named functions for each match and apply rule isn't really necessary. You never call them directly; you define them in the `rules` list and call them through there. Let's streamline the rules definition by anonymizing those functions.

- 17.5. `plural.py`, stage 4

Let's factor out the duplication in the code so that defining new rules can be easier.

- 17.6. `plural.py`, stage 5

You've factored out all the duplicate code and added enough abstractions so that the pluralization rules are defined in a list of strings. The next logical step is to take these strings and put them in a separate file, where they can be maintained separately from the code that uses them.

- 17.7. `plural.py`, stage 6

Now you're ready to talk about generators.

- 17.8. Summary

You talked about several different advanced techniques in this chapter. Not all of them are appropriate for every situation.

Chapter 18. Performance Tuning

- 18.1. Diving in

There are so many pitfalls involved in optimizing your code, it's hard to know where to start.

- 18.2. Using the `timeit` Module

The most important thing you need to know about optimizing Python code is that you shouldn't write your own timing function.

- 18.3. Optimizing Regular Expressions

The first thing the Soundex function checks is whether the input is a non-empty string of letters. What's the best way to do this?

- 18.4. Optimizing Dictionary Lookups

The second step of the Soundex algorithm is to convert characters to digits in a specific pattern. What's the best way to do this?

- 18.5. Optimizing List Operations

The third step in the Soundex algorithm is eliminating consecutive duplicate digits. What's the best way to do this?

- 18.6. Optimizing String Manipulation

The final step of the Soundex algorithm is padding short results with zeros, and truncating long results. What is the best way to do this?

- 18.7. Summary

This chapter has illustrated several important aspects of performance tuning in Python, and performance tuning in general.

Appendix C. Tips and tricks

Chapter 1. Installing Python

Chapter 2. Your First Python Program

- 2.1. Diving in

In the ActivePython IDE on Windows, you can run the Python program you're editing by choosing File->Run... (**Ctrl-R**). Output is displayed in the interactive window.

In the Python IDE on Mac OS, you can run a Python program with Python->Run window... (**Cmd-R**), but there is an important option you must set first. Open the .py file in the IDE, pop up the options menu by clicking the black triangle in the upper-right corner of the window, and make sure the Run as `__main__` option is checked. This is a per-file setting, but you'll only need to do it once per file.

On UNIX-compatible systems (including Mac OS X), you can run a Python program from the command line: **`python odbchelper.py`**

- 2.2. Declaring Functions

In Visual Basic, functions (that return a value) start with `function`, and subroutines (that do not return a value) start with `sub`. There are no subroutines in Python. Everything is a function, all functions return a value (even if it's `None`), and all functions start with `def`.

In Java, C++, and other statically-typed languages, you must specify the datatype of the function return value and each function argument. In Python, you never explicitly specify the datatype of anything. Based on what value you assign, Python keeps track of the datatype internally.

- 2.3. Documenting Functions

Triple quotes are also an easy way to define a string with both single and double quotes, like `qq / . . . /` in Perl.

Many Python IDEs use the `doc` string to provide context-sensitive documentation, so that when you type a function name, its `doc` string appears as a tooltip. This can be incredibly helpful, but it's only as good as the `doc` strings you write.

- 2.4. Everything Is an Object

`import` in Python is like `require` in Perl. Once you `import` a Python module, you access its functions with `module.function`; once you `require` a Perl module, you access its functions with `module::function`.

- 2.5. Indenting Code

Python uses carriage returns to separate statements and a colon and indentation to separate code blocks. C++ and Java use semicolons to separate statements and curly braces to separate code blocks.

- 2.6. Testing Modules

Like C, Python uses `==` for comparison and `=` for assignment. Unlike C, Python does not support in-line assignment, so there's no chance of accidentally assigning the value you thought you were comparing.

On MacPython, there is an additional step to make the `if __name__` trick work. Pop up the module's options menu by clicking the black triangle in the upper-right corner of the window, and make sure Run as `__main__` is checked.

Chapter 3. Native Datatypes

- 3.1. Introducing Dictionaries

A dictionary in Python is like a hash in Perl. In Perl, variables that store hashes always start with a `%` character. In Python, variables can be named anything, and Python keeps track of the datatype internally.

A dictionary in Python is like an instance of the `Hashtable` class in Java.

A dictionary in Python is like an instance of the `Scripting.Dictionary` object in Visual Basic.

- 3.1.2. Modifying Dictionaries

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered. This is an important distinction that will annoy you when you want to access the elements of a dictionary in a specific, repeatable order (like alphabetical order by key). There are ways of doing this, but they're not built into the dictionary.

- 3.2. Introducing Lists

A list in Python is like an array in Perl. In Perl, variables that store arrays always start with the `@` character; in Python, variables can be named anything, and Python keeps track of the datatype internally.

A list in Python is much more than an array in Java (although it can be used as one if that's really all you want out of life). A better analogy would be to the `ArrayList` class, which can hold arbitrary objects and can expand dynamically as new items are added.

- 3.2.3. Searching Lists

Before version 2.2.1, Python had no separate boolean datatype. To compensate for this, Python accepted almost anything in a boolean context (like an `if` statement), according to the following rules:

- ◆ 0 is false; all other numbers are true.
- ◆ An empty string (`" "`) is false; all other strings are true.
- ◆ An empty list (`[]`) is false; all other lists are true.
- ◆ An empty tuple (`()`) is false; all other tuples are true.
- ◆ An empty dictionary (`{}`) is false; all other dictionaries are true.

These rules still apply in Python 2.2.1 and beyond, but now you can also use an actual boolean, which has a value of `True` or `False`. Note the capitalization; these values, like everything else in Python, are case-sensitive.

- 3.3. Introducing Tuples

Tuples can be converted into lists, and vice-versa. The built-in `tuple` function takes a list and returns a tuple with the same elements, and the `list` function takes a tuple and returns a list. In effect, `tuple` freezes a list, and `list` thaws a tuple.

- 3.4. Declaring variables

When a command is split among several lines with the line-continuation marker (`"\"`), the continued lines can be indented in any manner; Python's normally stringent indentation rules do not apply. If your Python IDE auto-indents the continued line, you should probably accept its default unless you have a burning reason not to.

- 3.5. Formatting Strings

String formatting in Python uses the same syntax as the `sprintf` function in C.

- 3.7. Joining Lists and Splitting Strings

`join` works only on lists of strings; it does not do any type coercion. Joining a list that has one or more non-string elements will raise an exception.

`anystring.split(delimiter, 1)` is a useful technique when you want to search a string for a substring and then work with everything before the substring (which ends up in the first element of the returned list) and everything after it (which ends up in the second element).

Chapter 4. The Power Of Introspection

- 4.2. Using Optional and Named Arguments

The only thing you need to do to call a function is specify a value (somehow) for each required argument; the manner and order in which you do that is up to you.

- 4.3.3. Built-In Functions

Python comes with excellent reference manuals, which you should peruse thoroughly to learn all the modules Python has to offer. But unlike most languages, where you would find yourself referring back to the manuals or man pages to remind yourself how to use these modules, Python is largely self-documenting.

- 4.7. Using lambda Functions

lambda functions are a matter of style. Using them is never required; anywhere you could use them, you could define a separate normal function and use that instead. I use them in places where I want to encapsulate specific, non-reusable code without littering my code with a lot of little one-line functions.

- 4.8. Putting It All Together

In SQL, you must use `IS NULL` instead of `= NULL` to compare a null value. In Python, you can use either `== None` or `is None`, but `is None` is faster.

Chapter 5. Objects and Object-Oriented

- 5.2. Importing Modules Using `from module import`

`from module import *` in Python is like use `module` in Perl; `import module` in Python is like `require module` in Perl.

`from module import *` in Python is like `import module.*` in Java; `import module` in Python is like `import module` in Java.

Use `from module import *` sparingly, because it makes it difficult to determine where a particular function or attribute came from, and that makes debugging and refactoring more difficult.

- 5.3. Defining Classes

The `pass` statement in Python is like an empty set of braces (`{ }`) in Java or C.

In Python, the ancestor of a class is simply listed in parentheses immediately after the class name. There is no special keyword like `extends` in Java.

- 5.3.1. Initializing and Coding Classes

By convention, the first argument of any Python class method (the reference to the current instance) is called `self`. This argument fills the role of the reserved word `this` in C++ or Java, but `self` is not a reserved word in Python, merely a naming convention. Nonetheless, please don't call it anything but `self`; this is a very strong convention.

- 5.3.2. Knowing When to Use `self` and `__init__`

`__init__` methods are optional, but when you define one, you must remember to explicitly call the ancestor's `__init__` method (if it defines one). This is more generally true: whenever a descendant wants to extend the behavior of the ancestor, the descendant method must explicitly call the ancestor method at the proper time, with the proper arguments.

- 5.4. Instantiating Classes

In Python, simply call a class as if it were a function to create a new instance of the class. There is no explicit new operator like C++ or Java.

- 5.5. Exploring `UserDict`: A Wrapper Class

In the ActivePython IDE on Windows, you can quickly open any module in your library path by selecting `File->Locate...` (**Ctrl-L**).

Java and Powerbuilder support function overloading by argument list, *i.e.* one class can have multiple methods with the same name but a different number of arguments, or arguments of different types. Other languages (most notably PL/SQL) even support function overloading by argument name; *i.e.* one class can

have multiple methods with the same name and the same number of arguments of the same type but different argument names. Python supports neither of these; it has no form of function overloading whatsoever. Methods are defined solely by their name, and there can be only one method per class with a given name. So if a descendant class has an `__init__` method, it *always* overrides the ancestor `__init__` method, even if the descendant defines it with a different argument list. And the same rule applies to any other method.

Guido, the original author of Python, explains method overriding this way: "Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class, may in fact end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)" If that doesn't make sense to you (it confuses the hell out of me), feel free to ignore it. I just thought I'd pass it along.

Always assign an initial value to all of an instance's data attributes in the `__init__` method. It will save you hours of debugging later, tracking down `AttributeError` exceptions because you're referencing uninitialized (and therefore non-existent) attributes.

In versions of Python prior to 2.2, you could not directly subclass built-in datatypes like strings, lists, and dictionaries. To compensate for this, Python comes with wrapper classes that mimic the behavior of these built-in datatypes: `UserString`, `UserList`, and `UserDict`. Using a combination of normal and special methods, the `UserDict` class does an excellent imitation of a dictionary. In Python 2.2 and later, you can inherit classes directly from built-in datatypes like `dict`. An example of this is given in the examples that come with this book, in `fileinfo_fromdict.py`.

- 5.6.1. Getting and Setting Items

When accessing data attributes within a class, you need to qualify the attribute name: `self.attribute`. When calling other methods within a class, you need to qualify the method name: `self.method`.

- 5.7. Advanced Special Class Methods

In Java, you determine whether two string variables reference the same physical memory location by using `str1 == str2`. This is called *object identity*, and it is written in Python as `str1 is str2`. To compare string values in Java, you would use `str1.equals(str2)`; in Python, you would use `str1 == str2`. Java programmers who have been taught to believe that the world is a better place because `==` in Java compares by identity instead of by value may have a difficult time adjusting to Python's lack of such "gotchas".

While other object-oriented languages only let you define the physical model of an object ("this object has a `GetLength` method"), Python's special class methods like `__len__` allow you to define the logical model of an object ("this object has a length").

- 5.8. Introducing Class Attributes

In Java, both static variables (called class attributes in Python) and instance variables (called data attributes in Python) are defined immediately after the class definition (one with the `static` keyword, one without). In Python, only class attributes can be defined here; data attributes are defined in the `__init__` method.

There are no constants in Python. Everything can be changed if you try hard enough. This fits with one of the core principles of Python: bad behavior should be discouraged but not banned. If you really want to change the value of `None`, you can do it, but don't come running to me when your code is impossible to debug.

- 5.9. Private Functions

In Python, all special methods (like `__setitem__`) and built-in attributes (like `__doc__`) follow a standard naming convention: they both start with and end with two underscores. Don't name your own methods and attributes this way, because it will only confuse you (and others) later.

Chapter 6. Exceptions and File Handling

- 6.1. Handling Exceptions

Python uses `try...except` to handle exceptions and `raise` to generate them. Java and C++ use `try...catch` to handle exceptions, and `throw` to generate them.

- 6.5. Working with Directories

Whenever possible, you should use the functions in `os` and `os.path` for file, directory, and path manipulations. These modules are wrappers for platform-specific modules, so functions like `os.path.split` work on UNIX, Windows, Mac OS, and any other platform supported by Python.

Chapter 7. Regular Expressions

- 7.4. Using the `{n,m}` Syntax

There is no way to programmatically determine that two regular expressions are equivalent. The best you can do is write a lot of test cases to make sure they behave the same way on all relevant inputs. You'll talk more about writing test cases later in this book.

Chapter 8. HTML Processing

- 8.2. Introducing `sgmlib.py`

Python 2.0 had a bug where `SGMLParser` would not recognize declarations at all (`handle_decl` would never be called), which meant that DOCTYPEs were silently ignored. This is fixed in Python 2.1.

In the ActivePython IDE on Windows, you can specify command line arguments in the "Run script" dialog. Separate multiple arguments with spaces.

- 8.4. Introducing `BaseHTMLProcessor.py`

The HTML specification requires that all non-HTML (like client-side JavaScript) must be enclosed in HTML comments, but not all web pages do this properly (and all modern web browsers are forgiving if they don't). `BaseHTMLProcessor` is not forgiving; if script is improperly embedded, it will be parsed as if it were HTML. For instance, if the script contains less-than and equals signs, `SGMLParser` may incorrectly think that it has found tags and attributes. `SGMLParser` always converts tags and attribute names to lowercase, which may break the script, and `BaseHTMLProcessor` always encloses attribute values in double quotes (even if the original HTML document used single quotes or no quotes), which will certainly break the script. Always protect your client-side script within HTML comments.

- 8.5. locals and globals

Python 2.2 introduced a subtle but important change that affects the namespace search order: nested scopes. In versions of Python prior to 2.2, when you reference a variable within a nested function or `lambda` function, Python will search for that variable in the current (nested or `lambda`) function's namespace, then in the module's namespace. Python 2.2 will search for the variable in the current (nested or `lambda`) function's namespace, *then in the parent function's namespace*, then in the module's namespace. Python 2.1 can work either way; by default, it works like Python 2.0, but you can add the following line of code at the top of your module to make your module work like Python 2.2:

```
from __future__ import nested_scopes
```

Using the `locals` and `globals` functions, you can get the value of arbitrary variables dynamically, providing the variable name as a string. This mirrors the functionality of the `getattr` function, which allows you to access arbitrary functions dynamically by providing the function name as a string.

- 8.6. Dictionary-based string formatting

Using dictionary-based string formatting with `locals` is a convenient way of making complex string formatting expressions more readable, but it comes with a price. There is a slight performance hit in making the call to `locals`, since `locals` builds a copy of the local namespace.

Chapter 9. XML Processing

- 9.2. Packages

A package is a directory with the special `__init__.py` file in it. The `__init__.py` file defines the attributes and methods of the package. It doesn't need to define anything; it can just be an empty file, but it has to exist. But if `__init__.py` doesn't exist, the directory is just a directory, not a package, and it can't be imported or contain modules or nested packages.

- 9.6. Accessing element attributes

This section may be a little confusing, because of some overlapping terminology. Elements in an XML document have attributes, and Python objects also have attributes. When you parse an XML document, you get a bunch of Python objects that represent all the pieces of the XML document, and some of these Python objects represent attributes of the XML elements. But the (Python) objects that represent the (XML) attributes also have (Python) attributes, which are used to access various parts of the (XML) attribute that the object represents. I told you it was confusing. I am open to suggestions on how to distinguish these more clearly.

Like a dictionary, attributes of an XML element have no ordering. Attributes may *happen to be* listed in a certain order in the original XML document, and the `Attr` objects may *happen to be* listed in a certain order when the XML document is parsed into Python objects, but these orders are arbitrary and should carry no special meaning. You should always access individual attributes by name, like the keys of a dictionary.

Chapter 10. Scripts and Streams

Chapter 11. HTTP Web Services

- 11.6. Handling Last-Modified and ETag

In these examples, the HTTP server has supported both Last-Modified and ETag headers, but not all servers do. As a web services client, you should be prepared to support both, but you must code defensively in case a server only supports one or the other, or neither.

Chapter 12. SOAP Web Services

Chapter 13. Unit Testing

- 13.2. Diving in

`unittest` is included with Python 2.1 and later. Python 2.0 users can download it from `pyunit.sourceforge.net` (<http://pyunit.sourceforge.net/>).

Chapter 14. Test-First Programming

- 14.3. `roman.py`, stage 3

The most important thing that comprehensive unit testing can tell you is when to stop coding. When all the unit tests for a function pass, stop coding the function. When all the unit tests for an entire module pass, stop coding the module.

- 14.5. `roman.py`, stage 5

When all of your tests pass, stop coding.

Chapter 15. Refactoring

- 15.3. Refactoring

Whenever you are going to use a regular expression more than once, you should compile it to get a pattern object, then call the methods on the pattern object directly.

Chapter 16. Functional Programming

- 16.2. Finding the path

The pathnames and filenames you pass to `os.path.abspath` do not need to exist.

`os.path.abspath` not only constructs full path names, it also normalizes them. That means that if you are in the `/usr/` directory, `os.path.abspath('bin/../local/bin')` will return `/usr/local/bin`. It normalizes the path by making it as simple as possible. If you just want to normalize a pathname like this without turning it into a full pathname, use `os.path.normpath` instead.

Like the other functions in the `os` and `os.path` modules, `os.path.abspath` is cross-platform. Your results will look slightly different than my examples if you're running on Windows (which uses backslash as a path separator) or Mac OS (which uses colons), but they'll still work. That's the whole point of the `os` module.

Chapter 17. Dynamic functions

Chapter 18. Performance Tuning

- 18.2. Using the `timeit` Module

You can use the `timeit` module on the command line to test an existing Python program, without modifying the code. See <http://docs.python.org/lib/node396.html> for documentation on the command-line flags.

The `timeit` module only works if you already know what piece of code you need to optimize. If you have a larger Python program and don't know where your performance problems are, check out the `hotshot` module. (<http://docs.python.org/lib/module-hotshot.html>)

Appendix D. List of examples

Chapter 1. Installing Python

- 1.3. Python on Mac OS X
 - ◆ Example 1.1. Two versions of Python
- 1.5. Python on RedHat Linux
 - ◆ Example 1.2. Installing on RedHat Linux 9
- 1.6. Python on Debian GNU/Linux
 - ◆ Example 1.3. Installing on Debian GNU/Linux
- 1.7. Python Installation from Source
 - ◆ Example 1.4. Installing from source
- 1.8. The Interactive Shell
 - ◆ Example 1.5. First Steps in the Interactive Shell

Chapter 2. Your First Python Program

- 2.1. Diving in
 - ◆ Example 2.1. odbchelper.py
- 2.3. Documenting Functions
 - ◆ Example 2.2. Defining the buildConnectionString Function's doc string
- 2.4. Everything Is an Object
 - ◆ Example 2.3. Accessing the buildConnectionString Function's doc string
- 2.4.1. The Import Search Path
 - ◆ Example 2.4. Import Search Path
- 2.5. Indenting Code
 - ◆ Example 2.5. Indenting the buildConnectionString Function
 - ◆ Example 2.6. if Statements

Chapter 3. Native Datatypes

- 3.1.1. Defining Dictionaries
 - ◆ Example 3.1. Defining a Dictionary
- 3.1.2. Modifying Dictionaries
 - ◆ Example 3.2. Modifying a Dictionary
 - ◆ Example 3.3. Dictionary Keys Are Case–Sensitive
 - ◆ Example 3.4. Mixing Datatypes in a Dictionary
- 3.1.3. Deleting Items From Dictionaries
 - ◆ Example 3.5. Deleting Items from a Dictionary

- 3.2.1. Defining Lists
 - ◆ Example 3.6. Defining a List
 - ◆ Example 3.7. Negative List Indices
 - ◆ Example 3.8. Slicing a List
 - ◆ Example 3.9. Slicing Shorthand
- 3.2.2. Adding Elements to Lists
 - ◆ Example 3.10. Adding Elements to a List
 - ◆ Example 3.11. The Difference between extend and append
- 3.2.3. Searching Lists
 - ◆ Example 3.12. Searching a List
- 3.2.4. Deleting List Elements
 - ◆ Example 3.13. Removing Elements from a List
- 3.2.5. Using List Operators
 - ◆ Example 3.14. List Operators
- 3.3. Introducing Tuples
 - ◆ Example 3.15. Defining a tuple
 - ◆ Example 3.16. Tuples Have No Methods
- 3.4. Declaring variables
 - ◆ Example 3.17. Defining the myParams Variable
- 3.4.1. Referencing Variables
 - ◆ Example 3.18. Referencing an Unbound Variable
- 3.4.2. Assigning Multiple Values at Once
 - ◆ Example 3.19. Assigning multiple values at once
 - ◆ Example 3.20. Assigning Consecutive Values
- 3.5. Formatting Strings
 - ◆ Example 3.21. Introducing String Formatting
 - ◆ Example 3.22. String Formatting vs. Concatenating
 - ◆ Example 3.23. Formatting Numbers
- 3.6. Mapping Lists
 - ◆ Example 3.24. Introducing List Comprehensions
 - ◆ Example 3.25. The keys, values, and items Functions
 - ◆ Example 3.26. List Comprehensions in buildConnectionString, Step by Step
- 3.7. Joining Lists and Splitting Strings
 - ◆ Example 3.27. Output of odbchelper.py
 - ◆ Example 3.28. Splitting a String

Chapter 4. The Power Of Introspection

- 4.1. Diving In

- ◆ Example 4.1. apihelper.py
 - ◆ Example 4.2. Sample Usage of apihelper.py
 - ◆ Example 4.3. Advanced Usage of apihelper.py
- 4.2. Using Optional and Named Arguments
 - ◆ Example 4.4. Valid Calls of info
- 4.3.1. The type Function
 - ◆ Example 4.5. Introducing type
- 4.3.2. The str Function
 - ◆ Example 4.6. Introducing str
 - ◆ Example 4.7. Introducing dir
 - ◆ Example 4.8. Introducing callable
- 4.3.3. Built-In Functions
 - ◆ Example 4.9. Built-in Attributes and Functions
- 4.4. Getting Object References With getattr
 - ◆ Example 4.10. Introducing getattr
- 4.4.1. getattr with Modules
 - ◆ Example 4.11. The getattr Function in apihelper.py
- 4.4.2. getattr As a Dispatcher
 - ◆ Example 4.12. Creating a Dispatcher with getattr
 - ◆ Example 4.13. getattr Default Values
- 4.5. Filtering Lists
 - ◆ Example 4.14. Introducing List Filtering
- 4.6. The Peculiar Nature of and and or
 - ◆ Example 4.15. Introducing and
 - ◆ Example 4.16. Introducing or
- 4.6.1. Using the and-or Trick
 - ◆ Example 4.17. Introducing the and-or Trick
 - ◆ Example 4.18. When the and-or Trick Fails
 - ◆ Example 4.19. Using the and-or Trick Safely
- 4.7. Using lambda Functions
 - ◆ Example 4.20. Introducing lambda Functions
- 4.7.1. Real-World lambda Functions
 - ◆ Example 4.21. split With No Arguments
- 4.8. Putting It All Together
 - ◆ Example 4.22. Getting a doc string Dynamically
 - ◆ Example 4.23. Why Use str on a doc string?
 - ◆ Example 4.24. Introducing ljust
 - ◆ Example 4.25. Printing a List

Chapter 5. Objects and Object–Orientation

- 5.1. Diving In
 - ◆ Example 5.1. fileinfo.py
- 5.2. Importing Modules Using from module import
 - ◆ Example 5.2. import module vs. from module import
- 5.3. Defining Classes
 - ◆ Example 5.3. The Simplest Python Class
 - ◆ Example 5.4. Defining the FileInfo Class
- 5.3.1. Initializing and Coding Classes
 - ◆ Example 5.5. Initializing the FileInfo Class
 - ◆ Example 5.6. Coding the FileInfo Class
- 5.4. Instantiating Classes
 - ◆ Example 5.7. Creating a FileInfo Instance
- 5.4.1. Garbage Collection
 - ◆ Example 5.8. Trying to Implement a Memory Leak
- 5.5. Exploring UserDict: A Wrapper Class
 - ◆ Example 5.9. Defining the UserDict Class
 - ◆ Example 5.10. UserDict Normal Methods
 - ◆ Example 5.11. Inheriting Directly from Built–In Datatype dict
- 5.6.1. Getting and Setting Items
 - ◆ Example 5.12. The __getitem__ Special Method
 - ◆ Example 5.13. The __setitem__ Special Method
 - ◆ Example 5.14. Overriding __setitem__ in MP3FileInfo
 - ◆ Example 5.15. Setting an MP3FileInfo's name
- 5.7. Advanced Special Class Methods
 - ◆ Example 5.16. More Special Methods in UserDict
- 5.8. Introducing Class Attributes
 - ◆ Example 5.17. Introducing Class Attributes
 - ◆ Example 5.18. Modifying Class Attributes
- 5.9. Private Functions
 - ◆ Example 5.19. Trying to Call a Private Method

Chapter 6. Exceptions and File Handling

- 6.1. Handling Exceptions
 - ◆ Example 6.1. Opening a Non–Existent File
- 6.1.1. Using Exceptions For Other Purposes
 - ◆ Example 6.2. Supporting Platform–Specific Functionality

- 6.2. Working with File Objects
 - ◆ Example 6.3. Opening a File
- 6.2.1. Reading Files
 - ◆ Example 6.4. Reading a File
- 6.2.2. Closing Files
 - ◆ Example 6.5. Closing a File
- 6.2.3. Handling I/O Errors
 - ◆ Example 6.6. File Objects in MP3FileInfo
- 6.2.4. Writing to Files
 - ◆ Example 6.7. Writing to Files
- 6.3. Iterating with for Loops
 - ◆ Example 6.8. Introducing the for Loop
 - ◆ Example 6.9. Simple Counters
 - ◆ Example 6.10. Iterating Through a Dictionary
 - ◆ Example 6.11. for Loop in MP3FileInfo
- 6.4. Using sys.modules
 - ◆ Example 6.12. Introducing sys.modules
 - ◆ Example 6.13. Using sys.modules
 - ◆ Example 6.14. The __module__ Class Attribute
 - ◆ Example 6.15. sys.modules in fileinfo.py
- 6.5. Working with Directories
 - ◆ Example 6.16. Constructing Pathnames
 - ◆ Example 6.17. Splitting Pathnames
 - ◆ Example 6.18. Listing Directories
 - ◆ Example 6.19. Listing Directories in fileinfo.py
 - ◆ Example 6.20. Listing Directories with glob
- 6.6. Putting It All Together
 - ◆ Example 6.21. listDirectory

Chapter 7. Regular Expressions

- 7.2. Case Study: Street Addresses
 - ◆ Example 7.1. Matching at the End of a String
 - ◆ Example 7.2. Matching Whole Words
- 7.3.1. Checking for Thousands
 - ◆ Example 7.3. Checking for Thousands
- 7.3.2. Checking for Hundreds
 - ◆ Example 7.4. Checking for Hundreds
- 7.4. Using the {n,m} Syntax

- ◆ Example 7.5. The Old Way: Every Character Optional
 - ◆ Example 7.6. The New Way: From n o m
- 7.4.1. Checking for Tens and Ones
 - ◆ Example 7.7. Checking for Tens
 - ◆ Example 7.8. Validating Roman Numerals with {n,m}
- 7.5. Verbose Regular Expressions
 - ◆ Example 7.9. Regular Expressions with Inline Comments
- 7.6. Case study: Parsing Phone Numbers
 - ◆ Example 7.10. Finding Numbers
 - ◆ Example 7.11. Finding the Extension
 - ◆ Example 7.12. Handling Different Separators
 - ◆ Example 7.13. Handling Numbers Without Separators
 - ◆ Example 7.14. Handling Leading Characters
 - ◆ Example 7.15. Phone Number, Wherever I May Find Ye
 - ◆ Example 7.16. Parsing Phone Numbers (Final Version)

Chapter 8. HTML Processing

- 8.1. Diving in
 - ◆ Example 8.1. BaseHTMLProcessor.py
 - ◆ Example 8.2. dialect.py
 - ◆ Example 8.3. Output of dialect.py
- 8.2. Introducing sgmlib.py
 - ◆ Example 8.4. Sample test of sgmlib.py
- 8.3. Extracting data from HTML documents
 - ◆ Example 8.5. Introducing urllib
 - ◆ Example 8.6. Introducing urlister.py
 - ◆ Example 8.7. Using urlister.py
- 8.4. Introducing BaseHTMLProcessor.py
 - ◆ Example 8.8. Introducing BaseHTMLProcessor
 - ◆ Example 8.9. BaseHTMLProcessor output
- 8.5. locals and globals
 - ◆ Example 8.10. Introducing locals
 - ◆ Example 8.11. Introducing globals
 - ◆ Example 8.12. locals is read-only, globals is not
- 8.6. Dictionary-based string formatting
 - ◆ Example 8.13. Introducing dictionary-based string formatting
 - ◆ Example 8.14. Dictionary-based string formatting in BaseHTMLProcessor.py
 - ◆ Example 8.15. More dictionary-based string formatting
- 8.7. Quoting attribute values
 - ◆ Example 8.16. Quoting attribute values
- 8.8. Introducing dialect.py

- ◆ Example 8.17. Handling specific tags
- ◆ Example 8.18. SGMLParser
- ◆ Example 8.19. Overriding the `handle_data` method
- 8.9. Putting it all together
 - ◆ Example 8.20. The `translate` function, part 1
 - ◆ Example 8.21. The `translate` function, part 2: `curiouser` and `curiouser`
 - ◆ Example 8.22. The `translate` function, part 3

Chapter 9. XML Processing

- 9.1. Diving in
 - ◆ Example 9.1. `kgp.py`
 - ◆ Example 9.2. `toolbox.py`
 - ◆ Example 9.3. Sample output of `kgp.py`
 - ◆ Example 9.4. Simpler output from `kgp.py`
- 9.2. Packages
 - ◆ Example 9.5. Loading an XML document (a sneak peek)
 - ◆ Example 9.6. File layout of a package
 - ◆ Example 9.7. Packages are modules, too
- 9.3. Parsing XML
 - ◆ Example 9.8. Loading an XML document (for real this time)
 - ◆ Example 9.9. Getting child nodes
 - ◆ Example 9.10. `toxml` works on any node
 - ◆ Example 9.11. Child nodes can be text
 - ◆ Example 9.12. Drilling down all the way to text
- 9.4. Unicode
 - ◆ Example 9.13. Introducing unicode
 - ◆ Example 9.14. Storing non-ASCII characters
 - ◆ Example 9.15. `sitcustomize.py`
 - ◆ Example 9.16. Effects of setting the default encoding
 - ◆ Example 9.17. Specifying encoding in `.py` files
 - ◆ Example 9.18. `russiansample.xml`
 - ◆ Example 9.19. Parsing `russiansample.xml`
- 9.5. Searching for elements
 - ◆ Example 9.20. `binary.xml`
 - ◆ Example 9.21. Introducing `getElementsByTagName`
 - ◆ Example 9.22. Every element is searchable
 - ◆ Example 9.23. Searching is actually recursive
- 9.6. Accessing element attributes
 - ◆ Example 9.24. Accessing element attributes
 - ◆ Example 9.25. Accessing individual attributes

Chapter 10. Scripts and Streams

- 10.1. Abstracting input sources

- ◆ Example 10.1. Parsing XML from a file
- ◆ Example 10.2. Parsing XML from a URL
- ◆ Example 10.3. Parsing XML from a string (the easy but inflexible way)
- ◆ Example 10.4. Introducing StringIO
- ◆ Example 10.5. Parsing XML from a string (the file–like object way)
- ◆ Example 10.6. openAnything
- ◆ Example 10.7. Using openAnything in kgp.py
- 10.2. Standard input, output, and error
 - ◆ Example 10.8. Introducing stdout and stderr
 - ◆ Example 10.9. Redirecting output
 - ◆ Example 10.10. Redirecting error information
 - ◆ Example 10.11. Printing to stderr
 - ◆ Example 10.12. Chaining commands
 - ◆ Example 10.13. Reading from standard input in kgp.py
- 10.3. Caching node lookups
 - ◆ Example 10.14. loadGrammar
 - ◆ Example 10.15. Using the ref element cache
- 10.4. Finding direct children of a node
 - ◆ Example 10.16. Finding direct child elements
- 10.5. Creating separate handlers by node type
 - ◆ Example 10.17. Class names of parsed XML objects
 - ◆ Example 10.18. parse, a generic XML node dispatcher
 - ◆ Example 10.19. Functions called by the parse dispatcher
- 10.6. Handling command–line arguments
 - ◆ Example 10.20. Introducing sys.argv
 - ◆ Example 10.21. The contents of sys.argv
 - ◆ Example 10.22. Introducing getopt
 - ◆ Example 10.23. Handling command–line arguments in kgp.py

Chapter 11. HTTP Web Services

- 11.1. Diving in
 - ◆ Example 11.1. openanything.py
- 11.2. How not to fetch data over HTTP
 - ◆ Example 11.2. Downloading a feed the quick–and–dirty way
- 11.4. Debugging HTTP web services
 - ◆ Example 11.3. Debugging HTTP
- 11.5. Setting the User–Agent
 - ◆ Example 11.4. Introducing urllib2
 - ◆ Example 11.5. Adding headers with the Request
- 11.6. Handling Last–Modified and ETag
 - ◆ Example 11.6. Testing Last–Modified

- ◆ Example 11.7. Defining URL handlers
 - ◆ Example 11.8. Using custom URL handlers
 - ◆ Example 11.9. Supporting ETag/If-None-Match
- 11.7. Handling redirects
 - ◆ Example 11.10. Accessing web services without a redirect handler
 - ◆ Example 11.11. Defining the redirect handler
 - ◆ Example 11.12. Using the redirect handler to detect permanent redirects
 - ◆ Example 11.13. Using the redirect handler to detect temporary redirects
- 11.8. Handling compressed data
 - ◆ Example 11.14. Telling the server you would like compressed data
 - ◆ Example 11.15. Decompressing the data
 - ◆ Example 11.16. Decompressing the data directly from the server
- 11.9. Putting it all together
 - ◆ Example 11.17. The openanything function
 - ◆ Example 11.18. The fetch function
 - ◆ Example 11.19. Using openanything.py

Chapter 12. SOAP Web Services

- 12.1. Diving In
 - ◆ Example 12.1. search.py
 - ◆ Example 12.2. Sample Usage of search.py
- 12.2.1. Installing PyXML
 - ◆ Example 12.3. Verifying PyXML Installation
- 12.2.2. Installing fpconst
 - ◆ Example 12.4. Verifying fpconst Installation
- 12.2.3. Installing SOAPpy
 - ◆ Example 12.5. Verifying SOAPpy Installation
- 12.3. First Steps with SOAP
 - ◆ Example 12.6. Getting the Current Temperature
- 12.4. Debugging SOAP Web Services
 - ◆ Example 12.7. Debugging SOAP Web Services
- 12.6. Introspecting SOAP Web Services with WSDL
 - ◆ Example 12.8. Discovering The Available Methods
 - ◆ Example 12.9. Discovering A Method's Arguments
 - ◆ Example 12.10. Discovering A Method's Return Values
 - ◆ Example 12.11. Calling A Web Service Through A WSDL Proxy
- 12.7. Searching Google
 - ◆ Example 12.12. Introspecting Google Web Services
 - ◆ Example 12.13. Searching Google
 - ◆ Example 12.14. Accessing Secondary Information From Google

- 12.8. Troubleshooting SOAP Web Services
 - ◆ Example 12.15. Calling a Method With an Incorrectly Configured Proxy
 - ◆ Example 12.16. Calling a Method With the Wrong Arguments
 - ◆ Example 12.17. Calling a Method and Expecting the Wrong Number of Return Values
 - ◆ Example 12.18. Calling a Method With An Application–Specific Error

Chapter 13. Unit Testing

- 13.3. Introducing `romantest.py`
 - ◆ Example 13.1. `romantest.py`
- 13.4. Testing for success
 - ◆ Example 13.2. `testToRomanKnownValues`
- 13.5. Testing for failure
 - ◆ Example 13.3. Testing bad input to `toRoman`
 - ◆ Example 13.4. Testing bad input to `fromRoman`
- 13.6. Testing for sanity
 - ◆ Example 13.5. Testing `toRoman` against `fromRoman`
 - ◆ Example 13.6. Testing for case

Chapter 14. Test–First Programming

- 14.1. `roman.py`, stage 1
 - ◆ Example 14.1. `roman1.py`
 - ◆ Example 14.2. Output of `romantest1.py` against `roman1.py`
- 14.2. `roman.py`, stage 2
 - ◆ Example 14.3. `roman2.py`
 - ◆ Example 14.4. How `toRoman` works
 - ◆ Example 14.5. Output of `romantest2.py` against `roman2.py`
- 14.3. `roman.py`, stage 3
 - ◆ Example 14.6. `roman3.py`
 - ◆ Example 14.7. Watching `toRoman` handle bad input
 - ◆ Example 14.8. Output of `romantest3.py` against `roman3.py`
- 14.4. `roman.py`, stage 4
 - ◆ Example 14.9. `roman4.py`
 - ◆ Example 14.10. How `fromRoman` works
 - ◆ Example 14.11. Output of `romantest4.py` against `roman4.py`
- 14.5. `roman.py`, stage 5
 - ◆ Example 14.12. `roman5.py`
 - ◆ Example 14.13. Output of `romantest5.py` against `roman5.py`

Chapter 15. Refactoring

- 15.1. Handling bugs
 - ◆ Example 15.1. The bug
 - ◆ Example 15.2. Testing for the bug (romantest61.py)
 - ◆ Example 15.3. Output of romantest61.py against roman61.py
 - ◆ Example 15.4. Fixing the bug (roman62.py)
 - ◆ Example 15.5. Output of romantest62.py against roman62.py
- 15.2. Handling changing requirements
 - ◆ Example 15.6. Modifying test cases for new requirements (romantest71.py)
 - ◆ Example 15.7. Output of romantest71.py against roman71.py
 - ◆ Example 15.8. Coding the new requirements (roman72.py)
 - ◆ Example 15.9. Output of romantest72.py against roman72.py
- 15.3. Refactoring
 - ◆ Example 15.10. Compiling regular expressions
 - ◆ Example 15.11. Compiled regular expressions in roman81.py
 - ◆ Example 15.12. Output of romantest81.py against roman81.py
 - ◆ Example 15.13. roman82.py
 - ◆ Example 15.14. Output of romantest82.py against roman82.py
 - ◆ Example 15.15. roman83.py
 - ◆ Example 15.16. Output of romantest83.py against roman83.py
- 15.4. Postscript
 - ◆ Example 15.17. roman9.py
 - ◆ Example 15.18. Output of romantest9.py against roman9.py

Chapter 16. Functional Programming

- 16.1. Diving in
 - ◆ Example 16.1. regression.py
 - ◆ Example 16.2. Sample output of regression.py
- 16.2. Finding the path
 - ◆ Example 16.3. fullpath.py
 - ◆ Example 16.4. Further explanation of os.path.abspath
 - ◆ Example 16.5. Sample output from fullpath.py
 - ◆ Example 16.6. Running scripts in the current directory
- 16.3. Filtering lists revisited
 - ◆ Example 16.7. Introducing filter
 - ◆ Example 16.8. filter in regression.py
 - ◆ Example 16.9. Filtering using list comprehensions instead
- 16.4. Mapping lists revisited
 - ◆ Example 16.10. Introducing map
 - ◆ Example 16.11. map with lists of mixed datatypes
 - ◆ Example 16.12. map in regression.py
- 16.6. Dynamically importing modules
 - ◆ Example 16.13. Importing multiple modules at once

- ◆ Example 16.14. Importing modules dynamically
- ◆ Example 16.15. Importing a list of modules dynamically
- 16.7. Putting it all together
 - ◆ Example 16.16. The regressionTest function
 - ◆ Example 16.17. Step 1: Get all the files
 - ◆ Example 16.18. Step 2: Filter to find the files you care about
 - ◆ Example 16.19. Step 3: Map filenames to module names
 - ◆ Example 16.20. Step 4: Mapping module names to modules
 - ◆ Example 16.21. Step 5: Loading the modules into a test suite
 - ◆ Example 16.22. Step 6: Telling unittest to use your test suite

Chapter 17. Dynamic functions

- 17.2. plural.py, stage 1
 - ◆ Example 17.1. plural1.py
 - ◆ Example 17.2. Introducing re.sub
 - ◆ Example 17.3. Back to plural1.py
 - ◆ Example 17.4. More on negation regular expressions
 - ◆ Example 17.5. More on re.sub
- 17.3. plural.py, stage 2
 - ◆ Example 17.6. plural2.py
 - ◆ Example 17.7. Unrolling the plural function
- 17.4. plural.py, stage 3
 - ◆ Example 17.8. plural3.py
- 17.5. plural.py, stage 4
 - ◆ Example 17.9. plural4.py
 - ◆ Example 17.10. plural4.py continued
 - ◆ Example 17.11. Unrolling the rules definition
 - ◆ Example 17.12. plural4.py, finishing up
 - ◆ Example 17.13. Another look at buildMatchAndApplyFunctions
 - ◆ Example 17.14. Expanding tuples when calling functions
- 17.6. plural.py, stage 5
 - ◆ Example 17.15. rules.en
 - ◆ Example 17.16. plural5.py
- 17.7. plural.py, stage 6
 - ◆ Example 17.17. plural6.py
 - ◆ Example 17.18. Introducing generators
 - ◆ Example 17.19. Using generators instead of recursion
 - ◆ Example 17.20. Generators in for loops
 - ◆ Example 17.21. Generators that generate dynamic functions

Chapter 18. Performance Tuning

- 18.1. Diving in

- ♦ Example 18.1. `soundex/stage1/soundex1a.py`
- 18.2. Using the `timeit` Module
 - ♦ Example 18.2. Introducing `timeit`
- 18.3. Optimizing Regular Expressions
 - ♦ Example 18.3. Best Result So Far: `soundex/stage1/soundex1e.py`
- 18.4. Optimizing Dictionary Lookups
 - ♦ Example 18.4. Best Result So Far: `soundex/stage2/soundex2c.py`
- 18.5. Optimizing List Operations
 - ♦ Example 18.5. Best Result So Far: `soundex/stage2/soundex2c.py`

Appendix E. Revision history

Revision History	
Revision 5.4	2004-05-20
<ul style="list-style-type: none">• Added Section 12.1, Diving In .• Added Section 12.2, Installing the SOAP Libraries .• Added Section 12.3, First Steps with SOAP .• Added Section 12.4, Debugging SOAP Web Services .• Added Section 12.5, Introducing WSDL .• Added Section 12.6, Introspecting SOAP Web Services with WSDL .• Added Section 12.7, Searching Google .• Added Section 12.8, Troubleshooting SOAP Web Services .• Added Section 12.9, Summary .• Incorporated technical reviewer revisions in Chapter 16, <i>Functional Programming</i> and Chapter 18, <i>Performance Tuning</i>.	
Revision 5.3	2004-05-12
<ul style="list-style-type: none">• Added <code>isalpha()</code> example to Section 18.3, Optimizing Regular Expressions . Thanks, Paul.• Incorporated copyediting revisions into Chapter 5, <i>Objects and Object–Orientation</i> and Chapter 6, <i>Exceptions and File Handling</i>.• Fixed URL of Section 9.7, Segue .	
Revision 5.2	2004-05-09
<ul style="list-style-type: none">• Fixed URL of Section 14.1, <code>roman.py</code>, stage 1 .• Added Section 18.1, Diving in .• Added Section 18.2, Using the <code>timeit</code> Module .• Added Section 18.3, Optimizing Regular Expressions .• Added Section 18.4, Optimizing Dictionary Lookups .• Added Section 18.5, Optimizing List Operations .• Added Section 18.6, Optimizing String Manipulation .• Added Section 18.7, Summary .	
Revision 5.1	2004-05-05
<ul style="list-style-type: none">• Clarified Example 7.7, Checking for Tens and Example 7.8, Validating Roman Numerals with <code>{n,m}</code> .• Clarified Example 7.10, Finding Numbers .• Fixed typo in Example 11.6, Testing Last–Modified . Thanks, Jesir.• Fixed typo in Example 3.11, The Difference between <code>extend</code> and <code>append</code> . Thanks, Daniel.• Incorporated technical reviewer revisions.• Incorporated copy editor revisions in Chapter 1, <i>Installing Python</i>, Chapter 2, <i>Your First Python Program</i>, Chapter 3, <i>Native Datatypes</i>, and Chapter 4, <i>The Power Of Introspection</i>.	
Revision 5.0	2004-04-16
<ul style="list-style-type: none">• Added Section 11.1, Diving in .• Added Section 11.2, How not to fetch data over HTTP .• Added Section 11.3, Features of HTTP .• Added Section 11.4, Debugging HTTP web services .• Added Section 11.5, Setting the User–Agent .• Added Section 11.6, Handling Last–Modified and ETag .	

- Added Section 11.7, Handling redirects .
- Added Section 11.8, Handling compressed data .
- Added Section 11.9, Putting it all together .
- Added Section 11.10, Summary .
- Added Example 3.11, The Difference between extend and append .
- Changed descriptions of how to download Python throughout Chapter 1, *Installing Python* to be more generic and less version-specific.
- Changed references of "module" to "program" in Section 2.1, Diving in and Section 2.4, Everything Is an Object since we haven't explained modules yet.
- Added explicit instructions in Section 2.4, Everything Is an Object for the reader to open their Python IDE and follow along with the examples.
- Changed all examples and descriptions that referred to truth values 1 and 0 to refer to True and False.
- Updated Example 3.22, String Formatting vs. Concatenating to show new Python 2.3 `TypeError` message.
- Fixed typo in Example 17.19, Using generators instead of recursion .
- Fixed typo in Section 7.7, Summary .
- Fixed typo in Example 17.9, `plural4.py` .

Revision 4.9

2004-03-25

- Finished Section 16.7, Putting it all together .
- Added Section 16.8, Summary .
- Split unit testing introduction into two chapters, Chapter 13, *Unit Testing* and Chapter 14, *Test-First Programming*.
- Fixed typo in Example 17.12, `plural4.py`, finishing up .
- Fixed typo in Example 17.18, Introducing generators .

Revision 4.8

2004-03-25

- Finished Section 17.7, `plural.py`, stage 6 .
- Finished Section 17.8, Summary .
- Fixed broken links in Appendix A, *Further reading*, Appendix B, *A 5-minute review*, Appendix C, *Tips and tricks*, Appendix D, *List of examples*.

Revision 4.7

2004-03-21

- Added Section 17.1, Diving in .
- Added Section 17.2, `plural.py`, stage 1 .
- Added Section 17.3, `plural.py`, stage 2 .
- Added Section 17.4, `plural.py`, stage 3 .
- Added Section 17.5, `plural.py`, stage 4 .
- Added Section 17.6, `plural.py`, stage 5 .
- Added Section 17.7, `plural.py`, stage 6 (unfinished).
- Added Section 17.8, Summary (unfinished).

Revision 4.6

2004-03-14

- Finished Section 7.4, Using the `{n,m}` Syntax .
- Finished Section 7.5, Verbose Regular Expressions .
- Finished Section 7.6, Case study: Parsing Phone Numbers .
- Expanded Section 7.7, Summary .

Revision 4.5

2004-03-07

- Added Section 7.1, Diving In .

- Added Section 7.4, Using the {n,m} Syntax (incomplete).
- Added Section 7.5, Verbose Regular Expressions (incomplete).
- Added Section 7.6, Case study: Parsing Phone Numbers (incomplete).
- Added Section 7.7, Summary .
- Moved Section 7.2, Case Study: Street Addresses and Section 7.3, Case Study: Roman Numerals to regular expressions chapter.
- Added Example 6.20, Listing Directories with glob .
- Added Example 6.7, Writing to Files .
- Added Example 5.11, Inheriting Directly from Built-In Datatype dict .
- Added Example 10.11, Printing to stderr .
- Added Example 4.12, Creating a Dispatcher with getattr and Example 4.13, getattr Default Values .
- Added Example 2.6, if Statements .
- Added Example 3.23, Formatting Numbers .
- Split Chapter 5, *Objects and Object–Orientation* into 2 chapters: Chapter 5, *Objects and Object–Orientation* and Chapter 6, *Exceptions and File Handling*.
- Split Chapter 9, *XML Processing* into 2 chapters: Chapter 9, *XML Processing* and Chapter 10, *Scripts and Streams*.
- Split Chapter 13, *Unit Testing* into 2 chapters: Chapter 13, *Unit Testing* and Chapter 15, *Refactoring*.
- Renamed help to info in Chapter 4, *The Power Of Introspection*.
- Fixed incorrect back–reference in Section 8.5, locals and globals .
- Fixed broken example links in Section 8.1, Diving in .
- Fixed missing line in example in Section 9.1, Diving in .
- Fixed typo in Section 8.2, Introducing sgmlib.py .

Revision 4.4

2003–10–08

- Added Section 1.1, Which Python is right for you? .
- Added Section 1.2, Python on Windows .
- Added Section 1.3, Python on Mac OS X .
- Added Section 1.4, Python on Mac OS 9 .
- Added Section 1.5, Python on RedHat Linux .
- Added Section 1.6, Python on Debian GNU/Linux .
- Added Section 1.7, Python Installation from Source .
- Added Section 1.9, Summary .
- Removed preface.
- Fixed typo in Example 3.27, Output of odbchelper.py .
- Added link to PEP 257 in Section 2.3, Documenting Functions .
- Fixed link to *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) in Section 3.4.2, Assigning Multiple Values at Once .
- Added note about implied assert in Section 3.3, Introducing Tuples .

Revision 4.3

2003–09–28

- Added Section 16.6, Dynamically importing modules .
- Added Section 16.7, Putting it all together (incomplete).
- Fixed links in Appendix F, *About the book*.

Revision 4.2.1

2003–09–17

- Fixed links on index page.
- Fixed syntax highlighting.

Revision 4.2

2003–09–12

- Fixed typos in Section 16.4, Mapping lists revisited , Section 16.3, Filtering lists revisited , Section 7.2, Case Study: Street Addresses , and Section 10.6, Handling command–line arguments . Thanks, Doug.
- Fixed external link in Section 5.3, Defining Classes . Thanks, Harry.
- Changed wording at the end of Section 4.5, Filtering Lists . Thanks, Paul.
- Added sentence in Section 13.5, Testing for failure to make it clearer that we're passing a function to `assertRaises`, not a function name as a string. Thanks, Stephen.
- Fixed typo in Section 8.8, Introducing `dialect.py` . Thanks, Wellie.
- Fixed links to dialectized examples.
- Fixed external link to the history of Roman numerals. Thanks to many concerned Roman numeral fans around the world.

Revision 4.1

2002–07–28

- Added Section 10.3, Caching node lookups .
- Added Section 10.4, Finding direct children of a node .
- Added Section 10.5, Creating separate handlers by node type .
- Added Section 10.6, Handling command–line arguments .
- Added Section 10.7, Putting it all together .
- Added Section 10.8, Summary .
- Fixed typo in Section 6.5, Working with Directories . It's `os.getcwd()` , not `os.path.getcwd()` . Thanks, Abhishek.
- Fixed typo in Section 3.7, Joining Lists and Splitting Strings . When evaluated (instead of printed), the Python IDE will display single quotes around the output.
- Changed `str` example in Section 4.8, Putting It All Together to use a user–defined function, since Python 2.2 obsoleted the old example by defining a `doc string` for the built–in dictionary methods. Thanks Eric.
- Fixed typo in Section 9.4, Unicode , "anyway" to "anywhere". Thanks Frank.
- Fixed typo in Section 13.6, Testing for sanity , doubled word "accept". Thanks Ralph.
- Fixed typo in Section 15.3, Refactoring , `C?C?C?` matches 0 to 3 `C` characters, not 4. Thanks Ralph.
- Clarified and expanded explanation of implied slice indices in Example 3.9, Slicing Shorthand . Thanks Petr.
- Added historical note in Section 5.5, Exploring `UserDict`: A Wrapper Class now that Python 2.2 supports subclassing built–in datatypes directly.
- Added explanation of `update` dictionary method in Example 5.9, Defining the `UserDict` Class . Thanks Petr.
- Clarified Python's lack of overloading in Section 5.5, Exploring `UserDict`: A Wrapper Class . Thanks Petr.
- Fixed typo in Example 8.8, Introducing `BaseHTMLProcessor` . HTML comments end with two dashes and a bracket, not one. Thanks Petr.
- Changed tense of note about nested scopes in Section 8.5, locals and globals now that Python 2.2 is out. Thanks Petr.
- Fixed typo in Example 8.14, Dictionary–based string formatting in `BaseHTMLProcessor.py` ; a space should have been a non–breaking space. Thanks Petr.
- Added title to note on derived classes in Section 5.5, Exploring `UserDict`: A Wrapper Class . Thanks Petr.
- Added title to note on downloading `unittest` in Section 15.3, Refactoring . Thanks Petr.
- Fixed typesetting problem in Example 16.6, Running scripts in the current directory ; tabs should have been spaces, and the line numbers were misaligned. Thanks Petr.
- Fixed capitalization typo in the tip on truth values in Section 3.2, Introducing Lists . It's `True` and `False`, not `true` and `false`. Thanks to everyone who pointed this out.
- Changed section titles of Section 3.1, Introducing Dictionaries , Section 3.2, Introducing Lists , and Section 3.3, Introducing Tuples . "Dictionaries 101" was a cute way of saying that this section was an beginner's introduction to dictionaries. American colleges tend to use this numbering scheme to indicate introductory courses with no prerequisites, but apparently this is a distinctly American tradition, and it was unnecessarily confusing my international readers. In my defense, when I initially wrote these sections a year and a half ago, it never occurred to me that I would have international readers.

- Upgraded to version 1.52 of the DocBook XSL stylesheets.
- Upgraded to version 6.52 of the SAXON XSLT processor from Michael Kay.
- Various accessibility–related stylesheet tweaks.
- Somewhere between this revision and the last one, she said yes. The wedding will be next spring.

Revision 4.0–2

2002–04–26

- Fixed typo in Example 4.15, `Introducing and` .
- Fixed typo in Example 2.4, `Import Search Path` .
- Fixed Windows help file (missing table of contents due to base stylesheet changes).

Revision 4.0

2002–04–19

- Expanded Section 2.4, `Everything Is an Object` to include more about import search paths.
- Fixed typo in Example 3.7, `Negative List Indices` . Thanks to Brian for the correction.
- Rewrote the tip on truth values in Section 3.2, `Introducing Lists` , now that Python has a separate boolean datatype.
- Fixed typo in Section 5.2, `Importing Modules Using from module import` when comparing syntax to Java. Thanks to Rick for the correction.
- Added note in Section 5.5, `Exploring UserDict: A Wrapper Class` about derived classes always overriding ancestor classes.
- Fixed typo in Example 5.18, `Modifying Class Attributes` . Thanks to Kevin for the correction.
- Added note in Section 6.1, `Handling Exceptions` that you can define and raise your own exceptions. Thanks to Rony for the suggestion.
- Fixed typo in Example 8.17, `Handling specific tags` . Thanks for Rick for the correction.
- Added note in Example 8.18, `SGMLParser` about what the return codes mean. Thanks to Howard for the suggestion.
- Added `str` function when creating `StringIO` instance in Example 10.6, `openAnything` . Thanks to Ganesan for the idea.
- Added link in Section 13.3, `Introducing romantest.py` to explanation of why test cases belong in a separate file.
- Changed Section 16.2, `Finding the path` to use `os.path.dirname` instead of `os.path.split`. Thanks to Marc for the idea.
- Added code samples (`piglatin.py`, `parsephone.py`, and `plural.py`) for the upcoming regular expressions chapter.
- Updated and expanded list of Python distributions on home page.

Revision 3.9

2002–01–01

- Added Section 9.4, `Unicode` .
- Added Section 9.5, `Searching for elements` .
- Added Section 9.6, `Accessing element attributes` .
- Added Section 10.1, `Abstracting input sources` .
- Added Section 10.2, `Standard input, output, and error` .
- Added simple counter `for` loop examples (good usage and bad usage) in Section 6.3, `Iterating with for Loops` . Thanks to Kevin for the idea.
- Fixed typo in Example 3.25, `The keys, values, and items Functions` (two elements of `params.values()` were reversed).
- Fixed mistake in Section 4.3, `Using type, str, dir, and Other Built–In Functions` with regards to the name of the `__builtin__` module. Thanks to Denis for the correction.
- Added additional example in Section 16.2, `Finding the path` to show how to run unit tests in the current working directory, instead of the directory where `regression.py` is located.
- Modified explanation of how to derive a negative list index from a positive list index in Example 3.7, `Negative List Indices` . Thanks to Renauld for the suggestion.

<ul style="list-style-type: none"> • Updated links on home page for downloading latest version of Python. • Added link on home page to Bruce Eckel's preliminary draft of <i>Thinking in Python</i> (http://www.mindview.net/Books/TIPython), a marvelous (and advanced) book on design patterns and how to implement them in Python. 	
Revision 3.8	2001-11-18
<ul style="list-style-type: none"> • Added Section 16.2, Finding the path . • Added Section 16.3, Filtering lists revisited . • Added Section 16.4, Mapping lists revisited . • Added Section 16.5, Data-centric programming . • Expanded sample output in Section 16.1, Diving in . • Finished Section 9.3, Parsing XML . 	
Revision 3.7	2001-09-30
<ul style="list-style-type: none"> • Added Section 9.2, Packages . • Added Section 9.3, Parsing XML . • Cleaned up introductory paragraph in Section 9.1, Diving in . Thanks to Matt for this suggestion. • Added Java tip in Section 5.2, Importing Modules Using from module import . Thanks to Ori for this suggestion. • Fixed mistake in Section 4.8, Putting It All Together where I implied that you could not use <code>is None</code> to compare to a null value in Python. In fact, you can, and it's faster than <code>== None</code>. Thanks to Ori pointing this out. • Clarified in Section 3.2, Introducing Lists where I said that <code>li = li + other</code> was equivalent to <code>li.extend(other)</code>. The result is the same, but <code>extend</code> is faster because it doesn't create a new list. Thanks to Denis pointing this out. • Fixed mistake in Section 3.2, Introducing Lists where I said that <code>li += other</code> was equivalent to <code>li = li + other</code>. In fact, it's equivalent to <code>li.extend(other)</code>, since it doesn't create a new list. Thanks to Denis pointing this out. • Fixed typographical laziness in Chapter 2, <i>Your First Python Program</i>; when I was writing it, I had not yet standardized on putting string literals in single quotes within the text. They were set off by typography, but this is lost in some renditions of the book (like plain text), making it difficult to read. Thanks to Denis for this suggestion. • Fixed mistake in Section 2.2, Declaring Functions where I said that statically typed languages always use explicit variable + datatype declarations to enforce static typing. Most do, but there are some statically typed languages where the compiler figures out what type the variable is based on usage within the code. Thanks to Tony for pointing this out. • Added link to Spanish translation (http://es.diveintopython.org/). 	
Revision 3.6.4	2001-09-06
<ul style="list-style-type: none"> • Added code in <code>BaseHTMLProcessor</code> to handle non-HTML entity references, and added a note about it in Section 8.4, Introducing <code>BaseHTMLProcessor.py</code> . • Modified Example 8.11, Introducing globals to include <code>htmlentitydefs</code> in the output. 	
Revision 3.6.3	2001-09-04
<ul style="list-style-type: none"> • Fixed typo in Section 9.1, Diving in . • Added link to Korean translation (http://kr.diveintopython.org/html/index.htm). 	
Revision 3.6.2	2001-08-31
<ul style="list-style-type: none"> • Fixed typo in Section 13.6, Testing for sanity (the last requirement was listed twice). 	
Revision 3.6	2001-08-31

- Finished Chapter 8, *HTML Processing* with Section 8.9, Putting it all together and Section 8.10, Summary .
- Added Section 15.4, Postscript .
- Started Chapter 9, *XML Processing* with Section 9.1, Diving in .
- Started Chapter 16, *Functional Programming* with Section 16.1, Diving in .
- Fixed long-standing bug in colorizing script that improperly colorized the examples in Chapter 8, *HTML Processing*.
- Added link to French translation (<http://fr.diveintopython.org/toc.html>). They did the right thing and translated the source XML, so they can re-use all my build scripts and make their work available in six different formats.
- Upgraded to version 1.43 of the DocBook XSL stylesheets.
- Upgraded to version 6.43 of the SAXON XSLT processor from Michael Kay.
- Massive stylesheet changes, moving away from a table-based layout and towards more appropriate use of cascading style sheets. Unfortunately, CSS has as many compatibility problems as anything else, so there are still some tables used in the header and footer. The resulting HTML version looks worse in Netscape 4, but better in modern browsers, including Netscape 6, Mozilla, Internet Explorer 5, Opera 5, Konqueror, and iCab. And it's still completely readable in Lynx. I love Lynx. It was my first web browser. You never forget your first.
- Moved to Ant (<http://jakarta.apache.org/ant/>) to have better control over the build process, which is especially important now that I'm juggling six output formats and two languages.
- Consolidated the available downloadable archives; previously, I had different files for each platform, because the .zip files that Python's `zipfile` module creates are non-standard and can't be opened by Aladdin Expander on Mac OS. But the .zip files that Ant creates are completely standard and cross-platform. Go Ant!
- Now hosting the complete XML source, XSL stylesheets, and associated scripts and libraries on SourceForge. There's also CVS access for the really adventurous.
- Re-licensed the example code under the new-and-improved GPL-compatible Python 2.1.1 license (<http://www.python.org/2.1.1/license.html>). Thanks, Guido; people really do care, and it really does matter.

Revision 3.5

2001-06-26

- Added explanation of strong/weak/static/dynamic datatypes in Section 2.2, Declaring Functions .
- Added case-sensitivity example in Section 3.1, Introducing Dictionaries .
- Use `os.path.normcase` in Chapter 5, *Objects and Object-Oriented* to compensate for inferior operating systems whose files aren't case-sensitive.
- Fixed indentation problems in code samples in PDF version.

Revision 3.4

2001-05-31

- Added Section 14.5, `roman.py`, stage 5 .
- Added Section 15.1, Handling bugs .
- Added Section 15.2, Handling changing requirements .
- Added Section 15.3, Refactoring .
- Added Section 15.5, Summary .
- Fixed yet another stylesheet bug that was dropping nested `` tags.

Revision 3.3

2001-05-24

- Added Section 13.2, Diving in .
- Added Section 13.3, Introducing `romantest.py` .
- Added Section 13.4, Testing for success .
- Added Section 13.5, Testing for failure .
- Added Section 13.6, Testing for sanity .
- Added Section 14.1, `roman.py`, stage 1 .
- Added Section 14.2, `roman.py`, stage 2 .

<ul style="list-style-type: none"> • Added Section 14.3, <code>roman.py</code>, stage 3 . • Added Section 14.4, <code>roman.py</code>, stage 4 . • Tweaked stylesheets in an endless quest for complete Netscape/Mozilla compatibility. 	
Revision 3.2	2001-05-03
<ul style="list-style-type: none"> • Added Section 8.8, <code>Introducing dialect.py</code> . • Added Section 7.2, Case Study: Street Addresses . • Fixed bug in <code>handle_decl</code> method that would produce incorrect declarations (adding a space where it couldn't be). • Fixed bug in CSS (introduced in 2.9) where body background color was missing. 	
Revision 3.1	2001-04-18
<ul style="list-style-type: none"> • Added code in <code>BaseHTMLProcessor.py</code> to handle declarations, now that Python 2.1 supports them. • Added note about nested scopes in Section 8.5, locals and globals . • Fixed obscure bug in Example 8.1, <code>BaseHTMLProcessor.py</code> where attribute values with character entities would not be properly escaped. • Now recommending (but not requiring) Python 2.1, due to its support of declarations in <code>sgmllib.py</code>. • Updated download links on the home page (http://diveintopython.org/) to point to Python 2.1, where available. • Moved to versioned filenames, to help people who redistribute the book. 	
Revision 3.0	2001-04-16
<ul style="list-style-type: none"> • Fixed minor bug in code listing in Chapter 8, <i>HTML Processing</i>. • Added link to Chinese translation on home page (http://diveintopython.org/). 	
Revision 2.9	2001-04-13
<ul style="list-style-type: none"> • Added Section 8.5, locals and globals . • Added Section 8.6, Dictionary-based string formatting . • Tightened code in Chapter 8, <i>HTML Processing</i>, specifically <code>ChefDialectizer</code>, to use fewer and simpler regular expressions. • Fixed a stylesheet bug that was inserting blank pages between chapters in the PDF version. • Fixed a script bug that was stripping the <code>DOCTYPE</code> from the home page (http://diveintopython.org/). • Added link to Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/), and added a few links to individual recipes in Appendix A, <i>Further reading</i>. • Switched to Google (http://www.google.com/services/free.html) for searching on http://diveintopython.org/. • Upgraded to version 1.36 of the DocBook XSL stylesheets, which was much more difficult than it sounds. There may still be lingering bugs. 	
Revision 2.8	2001-03-26
<ul style="list-style-type: none"> • Added Section 8.3, Extracting data from HTML documents . • Added Section 8.4, Introducing <code>BaseHTMLProcessor.py</code> . • Added Section 8.7, Quoting attribute values . • Tightened up code in Chapter 4, <i>The Power Of Introspection</i>, using the built-in function <code>callable</code> instead of manually checking types. • Moved Section 5.2, Importing Modules Using <code>from module import</code> from Chapter 4, <i>The Power Of Introspection</i> to Chapter 5, <i>Objects and Object-Orientation</i>. • Fixed typo in code example in Section 5.1, Diving In (added colon). • Added several additional downloadable example scripts. • Added Windows Help output format. 	

Revision 2.7	2001-03-16
<ul style="list-style-type: none"> • Added Section 8.2, Introducing <code>sgmllib.py</code> . • Tightened up code in Chapter 8, <i>HTML Processing</i>. • Changed code in Chapter 2, <i>Your First Python Program</i> to use <code>items</code> method instead of <code>keys</code>. • Moved Section 3.4.2, Assigning Multiple Values at Once section to Chapter 2, <i>Your First Python Program</i>. • Edited note about <code>join</code> string method, and provided a link to the new entry in <i>The Whole Python FAQ</i> (http://www.python.org/doc/FAQ.html) that explains why <code>join</code> is a string method (http://www.python.org/cgi-bin/faqw.py?query=4.96&querytype=simple&casefold=yes&req=search) instead of a list method. • Rewrote Section 4.6, The Peculiar Nature of <code>and</code> and <code>or</code> to emphasize the fundamental nature of <code>and</code> and <code>or</code> and de-emphasize the <code>and-or</code> trick. • Reorganized language comparisons into notes. 	
Revision 2.6	2001-02-28
<ul style="list-style-type: none"> • The PDF and Word versions now have colorized examples, an improved table of contents, and properly indented tips and notes. • The Word version is now in native Word format, compatible with Word 97. • The PDF and text versions now have fewer problems with improperly converted special characters (like trademark symbols and curly quotes). • Added link to download Word version for UNIX, in case some twisted soul wants to import it into StarOffice or something. • Fixed several notes which were missing titles. • Fixed stylesheets to work around bug in Internet Explorer 5 for Mac OS which caused colorized words in the examples to be displayed in the wrong font. (Hello?!? Microsoft? Which part of <code><pre></code> don't you understand?) • Fixed archive corruption in Mac OS downloads. • In first section of each chapter, added link to download examples. (My access logs show that people skim or skip the two pages where they could have downloaded them (the home page (http://diveintopython.org/) and preface), then scramble to find a download link once they actually start reading.) • Tightened the home page (http://diveintopython.org/) and preface even more, in the hopes that someday someone will read them. • Soon I hope to get back to actually writing this book instead of debugging it. 	
Revision 2.5	2001-02-23
<ul style="list-style-type: none"> • Added Section 6.4, Using <code>sys.modules</code> . • Added Section 6.5, Working with Directories . • Moved Example 6.17, Splitting Pathnames from Section 3.4.2, Assigning Multiple Values at Once to Section 6.5, Working with Directories . • Added Section 6.6, Putting It All Together . • Added Section 5.10, Summary . • Added Section 8.1, Diving in . • Fixed program listing in Example 6.10, Iterating Through a Dictionary which was missing a colon. 	
Revision 2.4.1	2001-02-12
<ul style="list-style-type: none"> • Changed newsgroup links to use "news:" protocol, now that <code>deja.com</code> is defunct. • Added file sizes to download links. 	
Revision 2.4	2001-02-12

<ul style="list-style-type: none"> • Added "further reading" links in most sections, and collated them in Appendix A, <i>Further reading</i>. • Added URLs in parentheses next to external links in text version. 	
Revision 2.3	2001-02-09
<ul style="list-style-type: none"> • Rewrote some of the code in Chapter 5, <i>Objects and Object–Orientation</i> to use class attributes and a better example of multi-variable assignment. • Reorganized Chapter 5, <i>Objects and Object–Orientation</i> to put the class sections first. • Added Section 5.8, <i>Introducing Class Attributes</i>. • Added Section 6.1, <i>Handling Exceptions</i>. • Added Section 6.2, <i>Working with File Objects</i>. • Merged the "review" section in Chapter 5, <i>Objects and Object–Orientation</i> into Section 5.1, <i>Diving In</i>. • Colorized all program listings and examples. • Fixed important error in Section 2.2, <i>Declaring Functions</i>: functions that do not explicitly return a value return <code>None</code>, so you <i>can</i> assign the return value of such a function to a variable without raising an exception. • Added minor clarifications to Section 2.3, <i>Documenting Functions</i>, Section 2.4, <i>Everything Is an Object</i>, and Section 3.4, <i>Declaring variables</i>. 	
Revision 2.2	2001-02-02
<ul style="list-style-type: none"> • Edited Section 4.4, <i>Getting Object References With getattr</i>. • Added titles to <code>xref</code> tags, so they can have their cute little tooltips too. • Changed the look of the revision history page. • Fixed problem I introduced yesterday in my HTML post-processing script that was causing invalid HTML character references and breaking some browsers. • Upgraded to version 1.29 of the DocBook XSL stylesheets. 	
Revision 2.1	2001-02-01
<ul style="list-style-type: none"> • Rewrote the example code of Chapter 4, <i>The Power Of Introspection</i> to use <code>getattr</code> instead of <code>exec</code> and <code>eval</code>, and rewrote explanatory text to match. • Added example of list operators in Section 3.2, <i>Introducing Lists</i>. • Added links to relevant sections in the summary lists at the end of each chapter (Section 3.8, <i>Summary</i> and Section 4.9, <i>Summary</i>). 	
Revision 2.0	2001-01-31
<ul style="list-style-type: none"> • Split Section 5.6, <i>Special Class Methods</i> into three sections, Section 5.5, <i>Exploring UserDict: A Wrapper Class</i>, Section 5.6, <i>Special Class Methods</i>, and Section 5.7, <i>Advanced Special Class Methods</i>. • Changed notes on garbage collection to point out that Python 2.0 and later can handle circular references without additional coding. • Fixed UNIX downloads to include all relevant files. 	
Revision 1.9	2001-01-15
<ul style="list-style-type: none"> • Removed introduction to Chapter 2, <i>Your First Python Program</i>. • Removed introduction to Chapter 4, <i>The Power Of Introspection</i>. • Removed introduction to Chapter 5, <i>Objects and Object–Orientation</i>. • Edited text ruthlessly. I tend to ramble. 	
Revision 1.8	2001-01-12
<ul style="list-style-type: none"> • Added more examples to Section 3.4.2, <i>Assigning Multiple Values at Once</i>. • Added Section 5.3, <i>Defining Classes</i>. • Added Section 5.4, <i>Instantiating Classes</i>. 	

<ul style="list-style-type: none"> • Added Section 5.6, <i>Special Class Methods</i> . • More minor stylesheet tweaks, including adding titles to link tags, which, if your browser is cool enough, will display a description of the link target in a cute little tooltip. 	
Revision 1.71	2001-01-03
<ul style="list-style-type: none"> • Made several modifications to stylesheets to improve browser compatibility. 	
Revision 1.7	2001-01-02
<ul style="list-style-type: none"> • Added introduction to Chapter 2, <i>Your First Python Program</i>. • Added introduction to Chapter 4, <i>The Power Of Introspection</i>. • Added review section to Chapter 5, <i>Objects and Object–Orientation</i> [later removed] • Added Section 5.9, <i>Private Functions</i> . • Added Section 6.3, <i>Iterating with for Loops</i> . • Added Section 3.4.2, <i>Assigning Multiple Values at Once</i> . • Wrote scripts to convert book to new output formats: one single HTML file, PDF, Microsoft Word 97, and plain text. • Registered the <code>diveintopython.org</code> domain and moved the book there, along with links to download the book in all available output formats for offline reading. • Modified the XSL stylesheets to change the header and footer navigation that displays on each page. The top of each page is branded with the domain name and book version, followed by a breadcrumb trail to jump back to the chapter table of contents, the main table of contents, or the site home page. 	
Revision 1.6	2000-12-11
<ul style="list-style-type: none"> • Added Section 4.8, <i>Putting It All Together</i> . • Finished Chapter 4, <i>The Power Of Introspection</i> with Section 4.9, <i>Summary</i> . • Started Chapter 5, <i>Objects and Object–Orientation</i> with Section 5.1, <i>Diving In</i> . 	
Revision 1.5	2000-11-22
<ul style="list-style-type: none"> • Added Section 4.6, <i>The Peculiar Nature of and and or</i> . • Added Section 4.7, <i>Using lambda Functions</i> . • Added appendix that lists section abstracts. • Added appendix that lists tips. • Added appendix that lists examples. • Added appendix that lists revision history. • Expanded example of mapping lists in Section 3.6, <i>Mapping Lists</i> . • Encapsulated several more common phrases into entities. • Upgraded to version 1.25 of the DocBook XSL stylesheets. 	
Revision 1.4	2000-11-14
<ul style="list-style-type: none"> • Added Section 4.5, <i>Filtering Lists</i> . • Added <code>dir</code> documentation to Section 4.3, <i>Using type, str, dir, and Other Built–In Functions</i> . • Added <code>in</code> example in Section 3.3, <i>Introducing Tuples</i> . • Added additional note about <code>if __name__</code> trick under MacPython. • Switched to the SAXON XSLT processor from Michael Kay. • Upgraded to version 1.24 of the DocBook XSL stylesheets. • Added db–html processing instructions with explicit filenames of each chapter and section, to allow deep links to content even if I add or re–arrange sections later. • Made several common phrases into entities for easier reuse. • Changed several <code>literal</code> tags to <code>constant</code>. 	
Revision 1.3	2000-11-09

<ul style="list-style-type: none"> • Added section on dynamic code execution. • Added links to relevant section/example wherever I refer to previously covered concepts. • Expanded introduction of chapter 2 to explain what the function actually does. • Explicitly placed example code under the GNU General Public License and added appendix to display license. [Note 8/16/2001: code has been re-licensed under GPL-compatible Python license] • Changed links to licenses to use xref tags, now that I know how to use them. 	
Revision 1.2	2000-11-06
<ul style="list-style-type: none"> • Added first four sections of chapter 2. • Tightened up preface even more, and added link to Mac OS version of Python. • Filled out examples in "Mapping lists" and "Joining strings" to show logical progression. • Added output in chapter 1 summary. 	
Revision 1.1	2000-10-31
<ul style="list-style-type: none"> • Finished chapter 1 with sections on mapping and joining, and a chapter summary. • Toned down the preface, added links to introductions for non-programmers. • Fixed several typos. 	
Revision 1.0	2000-10-30
<ul style="list-style-type: none"> • Initial publication 	

Appendix F. About the book

This book was written in DocBook XML (<http://www.oasis-open.org/docbook/>) using Emacs (<http://www.gnu.org/software/emacs/>), and converted to HTML using the SAXON XSLT processor from Michael Kay (<http://saxon.sourceforge.net/>) with a customized version of Norman Walsh's XSL stylesheets (<http://www.nwalsh.com/xsl/>). From there, it was converted to PDF using HTMLDoc (<http://www.easysw.com/htmldoc/>), and to plain text using w3m (<http://ei5nazha.yz.yamagata-u.ac.jp/~aito/w3m/eng/>). Program listings and examples were colorized using an updated version of Just van Rossum's `pyfontify.py`, which is included in the example scripts.

If you're interested in learning more about DocBook for technical writing, you can download the XML source (<http://diveintopython.org/download/diveintopython-xml-5.4.zip>) and the build scripts (<http://diveintopython.org/download/diveintopython-common-5.4.zip>), which include the customized XSL stylesheets used to create all the different formats of the book. You should also read the canonical book, *DocBook: The Definitive Guide* (<http://www.docbook.org/>). If you're going to do any serious writing in DocBook, I would recommend subscribing to the DocBook mailing lists (<http://lists.oasis-open.org/archives/>).

Appendix G. GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

G.0. Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

G.1. Applicability and definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats

suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

G.2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

G.3. Copying in quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

G.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the

previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

G.5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

G.6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

G.7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

G.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

G.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

G.10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/> (<http://www.gnu.org/copyleft/>).

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

G.11. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix H. Python license

H.A. History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands as a successor of a language called ABC. Guido is Python's principal author, although it includes many contributions from others. The last version released from CWI was Python 1.2. In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI) in Reston, Virginia where he released several versions of the software. Python 1.6 was the last of the versions released by CNRI. In 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. Python 2.0 was the first and only release from BeOpen.com.

Following the release of Python 1.6, and after Guido van Rossum left CNRI to work with commercial software developers, it became clear that the ability to use Python with software available under the GNU Public License (GPL) was very desirable. CNRI and the Free Software Foundation (FSF) interacted to develop enabling wording changes to the Python license. Python 1.6.1 is essentially the same as Python 1.6, with a few minor bug fixes, and with a different license that enables later versions to be GPL-compatible. Python 2.1 is a derivative work of Python 1.6.1, as well as of Python 2.0.

After Python 2.0 was released by BeOpen.com, Guido van Rossum and the other PythonLabs developers joined Digital Creations. All intellectual property added from this point on, starting with Python 2.1 and its alpha and beta releases, is owned by the Python Software Foundation (PSF), a non-profit modeled after the Apache Software Foundation. See <http://www.python.org/psf/> for more information about the PSF.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

H.B. Terms and conditions for accessing or otherwise using Python

H.B.1. PSF license agreement

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.1.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.1.1 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001 Python Software Foundation; All Rights Reserved" are retained in Python 2.1.1 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.1.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.1.1.
4. PSF is making Python 2.1.1 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.1.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.1.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.1.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.1.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.2. BeOpen Python open source license agreement version 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.3. CNRI open source GPL-compatible license agreement

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright (c) 1995–2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement

may also be obtained from a proxy server on the Internet using the following URL:
<http://hdl.handle.net/1895.22/1013>".

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.4. CWI permissions statement and disclaimer

Copyright (c) 1991 – 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.