

File Formats

Overview

This chapter describes a number of modules that are used to parse different file formats.

Markup Languages

Python comes with extensive support for the *Extensible Markup Language* XML and *Hypertext Markup Language* (HTML) file formats. Python also provides basic support for *Standard Generalized Markup Language* (SGML).

All these formats share the same basic structure (this isn't so strange, since both HTML and XML are derived from SGML). Each document contains a mix of *start tags*, *end tags*, plain text (also called character data), and *entity references*.

```
<document name="sample.xml">
  <header>This is a header</header>
  <body>This is the body text. The text can contain
  plain text ("character data"), tags, and
  entities.
</body>
</document>
```

In the above example, **<document>**, **<header>**, and **<body>** are start tags. For each start tag, there's a corresponding end tag which looks similar, but has a slash before the tag name. The start tag can also contain one or more *attributes*, like the **name** attribute in this example.

Everything between a start tag and its matching end tag is called an *element*. In the above example, the **document** element contains two other elements, **header** and **body**.

Finally, **"** is a character entity. It is used to represent reserved characters in the text sections (in this case, it's an ampersand (&) which is used to start the entity itself. Other common entities include **<** for "less than" (<), and **>** for "greater than" (>).

While XML, HTML, and SGML all share the same building blocks, there are important differences between them. In XML, all elements must have both start tags and end tags, and the tags must be properly nested (if they are, the document is said to be *well-formed*). In addition, XML is case-sensitive, so **<document>** and **<Document>** are two different element types.

HTML, in contrast, is much more flexible. The HTML parser can often fill in missing tags; for example, if you open a new paragraph in HTML using the `<P>` tag without closing the previous paragraph, the parser automatically adds a `</P>` end tag. HTML is also case-insensitive. On the other hand, XML allows you to define your own elements, while HTML uses a fixed element set, as defined by the HTML specifications.

SGML is even more flexible. In its full incarnation, you can use a custom *declaration* to define how to translate the source text into an element structure, and a *document type description* (DTD) to validate the structure, and fill in missing tags. Technically, both HTML and XML are *SGML applications*; they both have their own SGML declaration, and HTML also has a standard DTD.

Python comes with parsers for all markup flavors. While SGML is the most flexible of the formats, Python's **sgmlib** parser is actually pretty simple. It avoids most of the problems by only understanding enough of the SGML standard to be able to deal with HTML. It doesn't handle document type descriptions either; instead, you can customize the parser via subclassing.

The HTML support is built on top of the SGML parser. The **htmlib** parser delegates the actual rendering to a formatter object. The **formatter** module contains a couple of standard formatters.

The XML support is most complex. In Python 1.5.2, the built-in support was limited to the **xmllib** parser, which is pretty similar to the **sgmlib** module (with one important difference; **xmllib** actually tries to support the entire XML standard).

Python 2.0 comes with more advanced XML tools, based on the optional **expat** parser.

Configuration Files

The **ConfigParser** module reads and writes a simple configuration file format, similar to Windows INI files.

The **netrc** file reads **.netrc** configuration files, and the **shlex** module can be used to read any configuration file using a shell script-like syntax.

Archive Formats

Python's standard library also provides support for the popular GZIP and ZIP (2.0 only) formats. The **gzip** module can read and write GZIP files, and the **zipfile** reads and writes ZIP files. Both modules depend on the **zlib** data compression module.

The xmllib module

This module provides a simple XML parser, using regular expressions to pull the XML data apart. The parser does basic checks on the document, such as checking that there is only one top-level element, and checking that all tags are balanced.

You feed XML data to this parser piece by piece (as data arrives over a network, for example). The parser calls methods in itself for start tags, data sections, end tags, and entities, among other things.

If you're only interested in a few tags, you can define special **start_tag** and **end_tag** methods, where **tag** is the tag name. The **start** functions are called with the attributes given as a dictionary.

Example: Using the xmllib module to extract information from an element

```
# File:xmllib-example-1.py

import xmllib

class Parser(xmllib.XMLParser):
    # get quotation number

    def __init__(self, file=None):
        xmllib.XMLParser.__init__(self)
        if file:
            self.load(file)

    def load(self, file):
        while 1:
            s = file.read(512)
            if not s:
                break
            self.feed(s)
        self.close()

    def start_quotation(self, attrs):
        print "id =>", attrs.get("id")
        raise EOFError

try:
    c = Parser()
    c.load(open("samples/sample.xml"))
except EOFError:
    pass
```

```
id => 031
```

The second example contains a simple (and incomplete) rendering engine. The parser maintains an element stack (**__tags**), which it passes to the renderer, together with text fragments. The renderer looks the current tag hierarchy up in a style dictionary, and if it isn't already there, it creates a new style descriptor by combining bits and pieces from the style sheet.

Example: Using the xmllib module

```
# File: xmllib-example-2.py

import xmllib
import string, sys

STYLESHEET = {
    # each element can contribute one or more style elements
    "quotation": {"style": "italic"},
    "lang": {"weight": "bold"},
    "name": {"weight": "medium"},
}

class Parser(xmllib.XMLParser):
    # a simple styling engine

    def __init__(self, renderer):
        xmllib.XMLParser.__init__(self)
        self.__data = []
        self.__tags = []
        self.__renderer = renderer

    def load(self, file):
        while 1:
            s = file.read(8192)
            if not s:
                break
            self.feed(s)
        self.close()

    def handle_data(self, data):
        self.__data.append(data)

    def unknown_starttag(self, tag, attrs):
        if self.__data:
            text = string.join(self.__data, "")
            self.__renderer.text(self.__tags, text)
        self.__tags.append(tag)
        self.__data = []

    def unknown_endtag(self, tag):
        self.__tags.pop()
        if self.__data:
            text = string.join(self.__data, "")
            self.__renderer.text(self.__tags, text)
        self.__data = []
```

```

class DumbRenderer:

    def __init__(self):
        self.cache = {}

    def text(self, tags, text):
        # render text in the style given by the tag stack
        tags = tuple(tags)
        style = self.cache.get(tags)
        if style is None:
            # figure out a combined style
            style = {}
            for tag in tags:
                s = STYLESHEET.get(tag)
                if s:
                    style.update(s)
            self.cache[tags] = style # update cache
        # write to standard output
        sys.stdout.write("%s =>\n" % style)
        sys.stdout.write(" " + repr(text) + "\n")

#
# try it out

r = DumbRenderer()
c = Parser(r)
c.load(open("samples/sample.xml"))

```

```

{'style': 'italic'} =>
'I\ve had a lot of developers come up to me and\012say,
"I haven\t had this much fun in a long time. It sure
beats\012writing '
{'style': 'italic', 'weight': 'bold'} =>
'Cobol'
{'style': 'italic'} =>
'__ '
{'style': 'italic', 'weight': 'medium'} =>
'James Gosling'
{'style': 'italic'} =>
', on\012'
{'weight': 'bold'} =>
'Java'
{'style': 'italic'} =>
''

```

The `xml.parsers.expat` module

(Optional). This is an interface to James Clark's Expat XML parser. This is a full-featured and fast parser, and an excellent choice for production use.

Example: Using the `xml.parsers.expat` module

```
# File:xml-parsers-expat-example-1.py

from xml.parsers import expat

class Parser:

    def __init__(self):
        self._parser = expat.ParserCreate()
        self._parser.StartElementHandler = self.start
        self._parser.EndElementHandler = self.end
        self._parser.CharacterDataHandler = self.data

    def feed(self, data):
        self._parser.Parse(data, 0)

    def close(self):
        self._parser.Parse("", 1) # end of data
        del self._parser # get rid of circular references

    def start(self, tag, attrs):
        print "START", repr(tag), attrs

    def end(self, tag):
        print "END", repr(tag)

    def data(self, data):
        print "DATA", repr(data)

p = Parser()
p.feed("<tag>data</tag>")
p.close()
```

```
START u'tag' {}
DATA u'data'
END u'tag'
```

Note that the parser returns Unicode strings, even if you pass it ordinary text. By default, the parser interprets the source text as UTF-8 (as per the XML standard). To use other encodings, make sure the XML file contains an *encoding* directive.

Example: Using the xml.parsers.expat module to read ISO Latin-1 text

```
# File:xml-parsers-expat-example-2.py

from xml.parsers import expat

class Parser:

    def __init__(self):
        self._parser = expat.ParserCreate()
        self._parser.StartElementHandler = self.start
        self._parser.EndElementHandler = self.end
        self._parser.CharacterDataHandler = self.data

    def feed(self, data):
        self._parser.Parse(data, 0)

    def close(self):
        self._parser.Parse("", 1) # end of data
        del self._parser # get rid of circular references

    def start(self, tag, attrs):
        print "START", repr(tag), attrs

    def end(self, tag):
        print "END", repr(tag)

    def data(self, data):
        print "DATA", repr(data)

p = Parser()
p.feed("""\
<?xml version='1.0' encoding='iso-8859-1'?>
<author>
<name>fredrik lundh</name>
<city>linköping</city>
</author>
""")
p.close()
```

```
START u'author' {}
DATA u'\012'
START u'name' {}
DATA u'fredrik lundh'
END u'name'
DATA u'\012'
START u'city' {}
DATA u'link\366ping'
END u'city'
DATA u'\012'
END u'author'
```

The sgmlib module

This module provides an basic SGML parser. It works pretty much like the **xmlib** parser, but is less restrictive (and less complete).

Like in **xmlib**, this parser calls methods in itself to deal with things like start tags, data sections, end tags, and entities. If you're only interested in a few tags, you can define special **start** and **end** methods:

Example: Using the sgmlib module to extract the title element

```
# File:sgmlib-example-1.py

import sgmlib
import string

class FoundTitle(Exception):
    pass

class ExtractTitle(sgmlib.SGMLParser):

    def __init__(self, verbose=0):
        sgmlib.SGMLParser.__init__(self, verbose)
        self.title = self.data = None

    def handle_data(self, data):
        if self.data is not None:
            self.data.append(data)

    def start_title(self, attrs):
        self.data = []

    def end_title(self):
        self.title = string.join(self.data, "")
        raise FoundTitle # abort parsing!

def extract(file):
    # extract title from an HTML/SGML stream
    p = ExtractTitle()
    try:
        while 1:
            # read small chunks
            s = file.read(512)
            if not s:
                break
            p.feed(s)
        p.close()
    except FoundTitle:
        return p.title
    return None
```

```
#
# try it out

print "html", ">", extract(open("samples/sample.htm"))
print "sgml", ">", extract(open("samples/sample.sgml"))

html => A Title.
sgml => Quotations
```

To handle all tags, overload the **unknown_starttag** and **unknown_endtag** methods instead:

Example: Using the sgmlib module to format an SGML document

```
# File:sgmlib-example-2.py

import sgmlib
import cgi, sys

class PrettyPrinter(sgmlib.SGMLParser):
    # A simple SGML pretty printer

    def __init__(self):
        # initialize base class
        sgmlib.SGMLParser.__init__(self)
        self.flag = 0

    def newline(self):
        # force newline, if necessary
        if self.flag:
            sys.stdout.write("\n")
            self.flag = 0

    def unknown_starttag(self, tag, attrs):
        # called for each start tag

        # the attrs argument is a list of (attr, value)
        # tuples. convert it to a string.
        text = ""
        for attr, value in attrs:
            text = text + " %s='%s'" % (attr, cgi.escape(value))

        self.newline()
        sys.stdout.write("<%s%s>\n" % (tag, text))

    def handle_data(self, text):
        # called for each text section
        sys.stdout.write(text)
        self.flag = (text[-1:] != "\n")

    def handle_entityref(self, text):
        # called for each entity
        sys.stdout.write("&%s;" % text)

    def unknown_endtag(self, tag):
        # called for each end tag
        self.newline()
        sys.stdout.write("<%s>" % tag)
```

```
#
# try it out

file = open("samples/sample.sgm")

p = PrettyPrinter()
p.feed(file.read())
p.close()
```

```
<chapter>
<title>
Quotations
<title>
<epigraph>
<attribution>
eff-bot, June 1997
<attribution>
<para>
<quote>
Nobody expects the Spanish Inquisition! Amongst
our weaponry are such diverse elements as fear, surprise,
ruthless efficiency, and an almost fanatical devotion to
Guido, and nice red uniforms &mdash; oh, damn!
<quote>
<para>
<epigraph>
<chapter>
```

The following example checks if an SGML document is "well-formed", in the XML sense. In a well-formed document, all elements are properly nested, and there's one end tag for each start tag. To check this, we simply keep a list of open tags, and check that each end tag closes a matching start tag, and that there are no open tags when we reach the end of the document.

Example: Using the sgmlib module to check if an SGML document is well-formed

```
# File:sgmlib-example-3.py

import sgmlib

class WellFormednessChecker(sgmlib.SGMLParser):
    # check that an SGML document is 'well formed'
    # (in the XML sense).

    def __init__(self, file=None):
        sgmlib.SGMLParser.__init__(self)
        self.tags = []
        if file:
            self.load(file)

    def load(self, file):
        while 1:
            s = file.read(8192)
            if not s:
                break
            self.feed(s)
        self.close()
```

```

def close(self):
    sgmlib.SGMLParser.close(self)
    if self.tags:
        raise SyntaxError, "start tag %s not closed" % self.tags[-1]

def unknown_starttag(self, start, attrs):
    self.tags.append(start)

def unknown_endtag(self, end):
    start = self.tags.pop()
    if end != start:
        raise SyntaxError, "end tag %s doesn't match start tag %s" %\
            (end, start)

try:
    c = WellFormednessChecker()
    c.load(open("samples/sample.htm"))
except SyntaxError:
    raise # report error
else:
    print "document is wellformed"

```

Traceback (innermost last):

```

...
SyntaxError: end tag head doesn't match start tag meta

```

Finally, here's a class that allows you to filter HTML and SGML documents. To use this class, create your own base class, and implement the **start** and **end** methods.

Example: Using the `sgmlib` module to filter SGML documents

```

# File:sgmlib-example-4.py

import sgmlib
import cgi, string, sys

class SGMLFilter(sgmlib.SGMLParser):
    # sgml filter.  override start/end to manipulate
    # document elements

    def __init__(self, outfile=None, infile=None):
        sgmlib.SGMLParser.__init__(self)
        if not outfile:
            outfile = sys.stdout
            self.write = outfile.write
        if infile:
            self.load(infile)

    def load(self, file):
        while 1:
            s = file.read(8192)
            if not s:
                break
            self.feed(s)
        self.close()

```

```
def handle_entityref(self, name):
    self.write("&%s;" % name)

def handle_data(self, data):
    self.write(cgi.escape(data))

def unknown_starttag(self, tag, attrs):
    tag, attrs = self.start(tag, attrs)
    if tag:
        if not attrs:
            self.write("<%s>" % tag)
        else:
            self.write("<%s" % tag)
            for k, v in attrs:
                self.write(" %s=%s" % (k, repr(v)))
            self.write(">")

def unknown_endtag(self, tag):
    tag = self.end(tag)
    if tag:
        self.write("</%s>" % tag)

def start(self, tag, attrs):
    return tag, attrs # override

def end(self, tag):
    return tag # override

class Filter(SGMLFilter):

    def fixtag(self, tag):
        if tag == "em":
            tag = "i"
        if tag == "string":
            tag = "b"
        return string.upper(tag)

    def start(self, tag, attrs):
        return self.fixtag(tag), attrs

    def end(self, tag):
        return self.fixtag(tag)

c = Filter()
c.load(open("samples/sample.htm"))
```

The `htmlib` module

This module contains a tag-driven HTML parser, which sends data to a formatting object. For more examples on how to parse HTML files using this module, see the descriptions of the **formatter** module.

Example: Using the `htmlib` module

```
# File:htmlib-example-1.py

import htmlib
import formatter
import string

class Parser(htmlib.HTMLParser):
    # return a dictionary mapping anchor texts to lists
    # of associated hyperlinks

    def __init__(self, verbose=0):
        self.anchors = {}
        f = formatter.NullFormatter()
        htmlib.HTMLParser.__init__(self, f, verbose)

    def anchor_bgn(self, href, name, type):
        self.save_bgn()
        self.anchor = href

    def anchor_end(self):
        text = string.strip(self.save_end())
        if self.anchor and text:
            self.anchors[text] = self.anchors.get(text, []) + [self.anchor]

file = open("samples/sample.htm")
html = file.read()
file.close()

p = Parser()
p.feed(html)
p.close()

for k, v in p.anchors.items():
    print k, "=>", v

print

link => ['http://www.python.org']
```

If you're only out to parse an HTML file, and not render it to an output device, it's usually easier to use the **sgmlib** module instead.

The `htmlentitydefs` module

This module contains a dictionary with lots of ISO Latin 1 character entities used by HTML.

Example: Using the `htmlentitydefs` module

```
# File:htmlentitydefs-example-1.py

import htmlentitydefs

entities = htmlentitydefs.entitydefs

for entity in "amp", "quot", "copy", "yen":
    print entity, "=", entities[entity]

amp = &
quot = "
copy = ©
yen = ¥
```

The following example shows how to combine regular expressions with this dictionary to translate entities in a string (the opposite of `cgi.escape`):

Example: Using the `htmlentitydefs` module to translate entities

```
# File:htmlentitydefs-example-2.py

import htmlentitydefs
import re
import cgi

pattern = re.compile("&(\w+?);")

def descapse_entity(m, defs=htmlentitydefs.entitydefs):
    # callback: translate one entity to its ISO Latin value
    try:
        return defs[m.group(1)]
    except KeyError:
        return m.group(0) # use as is

def descapse(string):
    return pattern.sub(descapse_entity, string)

print descapse("&lt;spam&amp;eggs&gt;")
print descapse(cgi.escape("<spam&eggs>"))

<spam&eggs>
<spam&eggs>
```

Finally, the following example shows how to use translate reserved XML characters and ISO Latin 1 characters to an XML string. This is similar to **cgi.escape**, but it also replaces non-ASCII characters.

Example: Escaping ISO Latin 1 entities

```
# File:htmlentitydefs-example-3.py

import htmlentitydefs
import re, string

# this pattern matches substrings of reserved and non-ASCII characters
pattern = re.compile(r"[\&<>\"'\x80-\xff]+")

# create character map
entity_map = {}

for i in range(256):
    entity_map[chr(i)] = "&#%d;" % i

for entity, char in htmlentitydefs.entitydefs.items():
    if entity_map.has_key(char):
        entity_map[char] = "%s;" % entity

def escape_entity(m, get=entity_map.get):
    return string.join(map(get, m.group()), "")

def escape(string):
    return pattern.sub(escape_entity, string)

print escape("<spam&eggs>")
print escape("å i å ä e ö")

<spam&eggs>
&aring; i &aring;a &auml; e &ouml;
```

The formatter module

This module provides formatter classes that can be used together with the **htmllib** module.

This module provides two class families, *formatters* and *writers*. The former convert a stream of tags and data strings from the HTML parser into an event stream suitable for an output device, and the latter renders that event stream on an output device.

In most cases, you can use the **AbstractFormatter** class to do the formatting. It calls methods on the writer object, representing different kinds of formatting events. The **AbstractWriter** class simply prints a message for each method call.

Example: Using the formatter module to convert HTML to an event stream

```
# File:formatter-example-1.py

import formatter
import htmllib

w = formatter.AbstractWriter()
f = formatter.AbstractFormatter(w)

file = open("samples/sample.htm")

p = htmllib.HTMLParser(f)
p.feed(file.read())
p.close()

file.close()

send_paragraph(1)
new_font(('h1', 0, 1, 0))
send_flowling_data('A Chapter.')
send_line_break()
send_paragraph(1)
new_font(None)
send_flowling_data('Some text. Some more text. Some')
send_flowling_data(' ')
new_font((None, 1, None, None))
send_flowling_data('emphasised')
new_font(None)
send_flowling_data(' text. A')
send_flowling_data(' link')
send_flowling_data('[1]')
send_flowling_data('.')
```

In addition to the **AbstractWriter** class, the **formatter** module provides an **NullWriter** class, which ignores all events passed to it, and a **DumbWriter** class that converts the event stream to a plain text document:

Example: Using the formatter module convert HTML to plain text

```
# File:formatter-example-2.py

import formatter
import htmllib

w = formatter.DumbWriter() # plain text
f = formatter.AbstractFormatter(w)

file = open("samples/sample.htm")

# print html body as plain text
p = htmllib.HTMLParser(f)
p.feed(file.read())
p.close()

file.close()

# print links
print
print
i = 1
for link in p.anchorlist:
    print i, "=>", link
    i = i + 1
```

A Chapter.

Some text. Some more text. Some emphasised text. A link[1].

1 => <http://www.python.org>

The following example provides a custom **Writer**, which in this case is subclassed from the **DumbWriter** class. This version keeps track of the current font style, and tweaks the output somewhat depending on the font.

Example: Using the formatter module with a custom writer

```
# File:formatter-example-3.py

import formatter
import htmllib, string

class Writer(formatter.DumbWriter):

    def __init__(self):
        formatter.DumbWriter.__init__(self)
        self.tag = ""
        self.bold = self.italic = 0
        self.fonts = []

    def new_font(self, font):
        if font is None:
            font = self.fonts.pop()
            self.tag, self.bold, self.italic = font
        else:
            self.fonts.append((self.tag, self.bold, self.italic))
            tag, bold, italic, typewriter = font
            if tag is not None:
                self.tag = tag
            if bold is not None:
                self.bold = bold
            if italic is not None:
                self.italic = italic

    def send_flowling_data(self, data):
        if not data:
            return
        atbreak = self.atbreak or data[0] in string.whitespace
        for word in string.split(data):
            if atbreak:
                self.file.write(" ")
            if self.tag in ("h1", "h2", "h3"):
                word = string.upper(word)
            if self.bold:
                word = "*" + word + "*"
            if self.italic:
                word = "_" + word + "_"
            self.file.write(word)
            atbreak = 1
        self.atbreak = data[-1] in string.whitespace
```

```
w = Writer()
f = formatter.AbstractFormatter(w)

file = open("samples/sample.htm")

# print html body as plain text
p = htmllib.HTMLParser(f)
p.feed(file.read())
p.close()
```

```
_A_ _CHAPTER._
```

```
Some text. Some more text. Some *emphasised* text. A link[1].
```

The ConfigParser module

This module reads configuration files.

The files should be written in a format similar to Windows INI files. The file contains one or more sections, separated by section names written in brackets. Each section can contain one or more configuration items.

Here's an example:

```
[book]
title: The Python Standard Library
author: Fredrik Lundh
email: fredrik@pythonware.com
version: 2.0-001115
```

```
[ematter]
pages: 250
```

```
[hardcopy]
pages: 350
```

Example: Using the ConfigParser module

```
# File: configparser-example-1.py

import ConfigParser
import string

config = ConfigParser.ConfigParser()

config.read("samples/sample.ini")

# print summary
print
print string.upper(config.get("book", "title"))
print "by", config.get("book", "author"),
print "(" + config.get("book", "email") + ")"
print
print config.get("ematter", "pages"), "pages"
print

# dump entire config file
for section in config.sections():
    print section
    for option in config.options(section):
        print " ", option, "=", config.get(section, option)
```

```
THE PYTHON STANDARD LIBRARY
by Fredrik Lundh (fredrik@pythonware.com)

250 pages

book
title = Python Standard Library
email = fredrik@pythonware.com
author = Fredrik Lundh
version = 2.0-010504
__name__ = book
ematter
__name__ = ematter
pages = 250
hardcopy
__name__ = hardcopy
pages = 300
```

In Python 2.0, this module also allows you to write configuration data to a file.

Example: Using the ConfigParser module to write configuration data

```
# File:configparser-example-2.py

import ConfigParser
import sys

config = ConfigParser.ConfigParser()

# set a number of parameters
config.add_section("book")
config.set("book", "title", "the python standard library")
config.set("book", "author", "fredrik lundh")

config.add_section("ematter")
config.set("ematter", "pages", 250)

# write to screen
config.write(sys.stdout)

[book]
title = the python standard library
author = fredrik lundh

[ematter]
pages = 250
```

The netrc module

This module parses **.netrc** configuration files. Such files are used to store FTP user names and passwords in a user's home directory (don't forget to configure things so that the file can only be read by the user: "**chmod 0600 ~/.netrc**", in other words).

Example: Using the netrc module

```
# File:netrc-example-1.py

import netrc

# default is $HOME/.netrc
info = netrc.netrc("samples/sample.netrc")

login, account, password = info.authenticators("secret.fbi")
print "login", "=>", repr(login)
print "account", "=>", repr(account)
print "password", "=>", repr(password)

login => 'mulder'
account => None
password => 'trustno1'
```

The shlex module

This module provides a simple lexer (also known as tokenizer) for languages based on the Unix shell syntax.

Example: Using the shlex module

```
# File:shlex-example-1.py

import shlex

lexer = shlex.shlex(open("samples/sample.netrc", "r"))
lexer.wordchars = lexer.wordchars + "._"

while 1:
    token = lexer.get_token()
    if not token:
        break
    print repr(token)
```

```
'machine'
'secret.fbi'
'login'
'mulder'
'password'
'trustno1'
'machine'
'non.secret.fbi'
'login'
'scully'
'password'
'noway'
```

The zipfile module

(New in 2.0) This module allows you to read and write files in the popular ZIP archive format.

Listing the contents

To list the contents of an existing archive, you can use the **namelist** and **infolist** methods. The former returns a list of filenames, the latter a list of **ZipInfo** instances.

Example: Using the zipfile module to list files in a ZIP file

```
# File:zipfile-example-1.py

import zipfile

file = zipfile.ZipFile("samples/sample.zip", "r")

# list filenames
for name in file.namelist():
    print name,
print

# list file information
for info in file.infolist():
    print info.filename, info.date_time, info.file_size

sample.txt sample.jpg
sample.txt (1999, 9, 11, 20, 11, 8) 302
sample.jpg (1999, 9, 18, 16, 9, 44) 4762
```

Reading data from a ZIP file

To read data from an archive, simply use the **read** method. It takes a filename as an argument, and returns the data as a string.

Example: Using the zipfile module to read data from a ZIP file

```
# File:zipfile-example-2.py

import zipfile

file = zipfile.ZipFile("samples/sample.zip", "r")

for name in file.namelist():
    data = file.read(name)
    print name, len(data), repr(data[:10])

sample.txt 302 'We will pe'
sample.jpg 4762 '\377\330\377\340\000\020JFIF'
```

Writing data to a ZIP file

Adding files to an archive is easy. Just pass the file name, and the name you want that file to have in the archive, to the **write** method.

The following script creates a ZIP file containing all files in the **samples** directory.

Example: Using the `zipfile` module to store files in a ZIP file

```
# File: zipfile-example-3.py

import zipfile
import glob, os

# open the zip file for writing, and write stuff to it

file = zipfile.ZipFile("test.zip", "w")

for name in glob.glob("samples/*"):
    file.write(name, os.path.basename(name), zipfile.ZIP_DEFLATED)

file.close()

# open the file again, to see what's in it

file = zipfile.ZipFile("test.zip", "r")
for info in file.infolist():
    print info.filename, info.date_time, info.file_size, info.compress_size

sample.wav (1999, 8, 15, 21, 26, 46) 13260 10985
sample.jpg (1999, 9, 18, 16, 9, 44) 4762 4626
sample.au (1999, 7, 18, 20, 57, 34) 1676 1103
...
```

The third, optional argument to the **write** method controls what compression method to use. Or rather, it controls whether data should be compressed at all. The default is **zipfile.ZIP_STORED**, which stores the data in the archive without any compression at all. If the **zlib** module is installed, you can also use **zipfile.ZIP_DEFLATED**, which gives you "deflate" compression.

The **zipfile** module also allows you to add strings to the archive. However, adding data from a string is a bit tricky; instead of just passing in the archive name and the data, you have to create a **ZipInfo** instance and configure it correctly. Here's a simple example:

Example: Using the zipfile module to store strings in a ZIP file

```
# File:zipfile-example-4.py

import zipfile
import glob, os, time

file = zipfile.ZipFile("test.zip", "w")

now = time.localtime(time.time())[6]

for name in ("life", "of", "brian"):
    info = zipfile.ZipInfo(name)
    info.date_time = now
    info.compress_type = zipfile.ZIP_DEFLATED
    file.writestr(info, name*1000)

file.close()

# open the file again, to see what's in it

file = zipfile.ZipFile("test.zip", "r")

for info in file.infolist():
    print info.filename, info.date_time, info.file_size, info.compress_size

life (2000, 12, 1, 0, 12, 1) 4000 26
of (2000, 12, 1, 0, 12, 1) 2000 18
brian (2000, 12, 1, 0, 12, 1) 5000 31
```

The gzip module

This module allows you to read and write gzip-compressed files as if they were ordinary files.

Example: Using the gzip module to read a compressed file

```
# File: gzip-example-1.py

import gzip

file = gzip.GzipFile("samples/sample.gz")

print file.read()
```

```
Well it certainly looks as though we're in for
a splendid afternoon's sport in this the 127th
Upperclass Twit of the Year Show.
```

The standard implementation doesn't support the **seek** and **tell** methods. The following example shows how to add forward seeking:

Example: Extending the gzip module to support seek/tell

```
# File: gzip-example-2.py

import gzip

class gzipFile(gzip.GzipFile):
    # adds seek/tell support to GzipFile

    offset = 0

    def read(self, size=None):
        data = gzip.GzipFile.read(self, size)
        self.offset = self.offset + len(data)
        return data

    def seek(self, offset, whence=0):
        # figure out new position (we can only seek forwards)
        if whence == 0:
            position = offset
        elif whence == 1:
            position = self.offset + offset
        else:
            raise IOError, "Illegal argument"
        if position < self.offset:
            raise IOError, "Cannot seek backwards"

        # skip forward, in 16k blocks
        while position > self.offset:
            if not self.read(min(position - self.offset, 16384)):
                break

    def tell(self):
        return self.offset
```

```
#  
# try it  
  
file = gzipFile("samples/sample.gz")  
file.seek(80)  
  
print file.read()
```

```
this the 127th  
Upperclass Twit of the Year Show.
```