19.1.2. `email.parser`: Parsing email messages

Source code: Lib/email/parser.py

Message object structures can be created in one of two ways: they can be created from whole cloth by creating an `EmailMessage` object, adding headers using the dictionary interface, and adding payload(s) using `set_content()` and related methods, or they can be created by parsing a serialized representation of the email message.

The `email` package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a bytes, string or file object, and the parser will return to you the root `EmailMessage` instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return `True` from its `is_multipart()` method, and the subparts can be accessed via the payload manipulation methods, such as `get_body()`, `iter_parts()`, and `walk()`.

There are actually two parser interfaces available for use, the `Parser` API and the incremental `FeedParser` API. The `Parser` API is most useful if you have the entire text of the message in memory, or if the entire message lives in a file on the file system. `FeedParser` is more appropriate when you are reading the message from a stream which might block waiting for more input (such as reading an email message from a socket). The `FeedParser` can consume and parse the message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. All of the logic that connects the `email` package's bundled parser and the `EmailMessage` class is embodied in the `policy` class, so a custom parser can create message object trees any way it finds necessary by implementing custom versions of the appropriate `policy` methods.

19.1.2.1. FeedParser API

The `BytesFeedParser`, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (such as a socket). The `BytesFeedParser` can of course be used to parse an email message fully contained in a bytes-like object, string, or file, but the `BytesParser`

API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The `BytesFeedParser`'s API is simple; you create an instance, feed it a bunch of bytes until there's no more to feed it, then close the parser to retrieve the root message object. The `BytesFeedParser` is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's `defects` attribute with a list of any problems it found in a message. See the `email.errors` module for the list of defects that it can find.

Here is the API for the `BytesFeedParser`:

class `email.parser.`**`BytesFeedParser`**(*_factory=None*, ***, *policy=policy.compat32*)

Create a `BytesFeedParser` instance. Optional *_factory* is a no-argument callable; if not specified use the `message_factory` from the *policy*. Call *_factory* whenever a new message object is needed.

If *policy* is specified use the rules it specifies to update the representation of the message. If *policy* is not set, use the `compat32` policy, which maintains backward compatibility with the Python 3.2 version of the email package and provides `Message` as the default factory. All other policies provide `EmailMessage` as the default *_factory*. For more information on what else *policy* controls, see the `policy` documentation.

Note: **The policy keyword should always be specified**; The default will change to `email.policy.default` in a future version of Python.

New in version 3.2.

Changed in version 3.3: Added the *policy* keyword.

Changed in version 3.6: *_factory* defaults to the policy `message_factory`.

`feed`(*data*)

Feed the parser some more data. *data* should be a bytes-like object containing one or more lines. The lines can be partial and the parser will stitch such partial lines together properly. The lines can have any of the three common line endings: carriage return, newline, or carriage return and newline (they can even be mixed).

`close`()

Complete the parsing of all previously fed data and return the root message object. It is undefined what happens if `feed()` is called after this method has been called.

class `email.parser.`**`FeedParser`**(*_factory=None*, ***, *policy=policy.compat32*)

Works like `BytesFeedParser` except that the input to the `feed()` method must be a string. This is of limited utility, since the only way for such a message to be valid is for it to contain only ASCII text or, if `utf8` is `True`, no binary attachments.

Changed in version 3.3: Added the *policy* keyword.

19.1.2.2. Parser API

The `BytesParser` class, imported from the `email.parser` module, provides an API that can be used to parse a message when the complete contents of the message are available in a bytes-like object or file. The `email.parser` module also provides `Parser` for parsing strings, and header-only parsers, `BytesHeaderParser` and `HeaderParser`, which can be used if you're only interested in the headers of the message. `BytesHeaderParser` and `HeaderParser` can be much faster in these situations, since they do not attempt to parse the message body, instead setting the payload to the raw body.

class `email.parser.`**`BytesParser`**(*_class=None*, ***, *policy=policy.compat32*)

Create a `BytesParser` instance. The *_class* and *policy* arguments have the same meaning and semantics as the *_factory* and *policy* arguments of `BytesFeedParser`.

Note: **The policy keyword should always be specified**; The default will change to `email.policy.default` in a future version of Python.

Changed in version 3.3: Removed the *strict* argument that was deprecated in 2.4. Added the *policy* keyword.

Changed in version 3.6: *_class* defaults to the policy `message_factory`.

`parse`(*fp*, *headersonly=False*)

Read all the data from the binary file-like object *fp*, parse the resulting bytes, and return the message object. *fp* must support both the `readline()` and the `read()` methods.

The bytes contained in *fp* must be formatted as a block of **RFC 5322** (or, if `utf8` is `True`, **RFC 6532**) style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts, including subparts with a *Content-Transfer-Encoding* of `8bit`.

Optional *headersonly* is a flag specifying whether to stop parsing after reading the headers or not. The default is `False`, meaning it parses the entire contents of the file.

`parsebytes`(*bytes*, *headersonly=False*)

Similar to the `parse()` method, except it takes a bytes-like object instead of a file-like object. Calling this method on a bytes-like object is equivalent to wrapping *bytes* in a `BytesIO` instance first and calling `parse()`.

Optional *headersonly* is as with the `parse()` method.

New in version 3.2.

class `email.parser.`**`BytesHeaderParser`**(*_class=None*, ***, *policy=policy.compat32*)

Exactly like `BytesParser`, except that *headersonly* defaults to `True`.

New in version 3.3.

class `email.parser.`**`Parser`**(*_class=None*, ***, *policy=policy.compat32*)

This class is parallel to `BytesParser`, but handles string input.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

Changed in version 3.6: *_class* defaults to the policy `message_factory`.

`parse`(*fp*, *headersonly=False*)

Read all the data from the text-mode file-like object *fp*, parse the resulting text, and return the root message object. *fp* must support both the `readline()` and the `read()` methods on file-like objects.

Other than the text mode requirement, this method operates like `BytesParser.parse()`.

`parsestr`(*text*, *headersonly=False*)

Similar to the `parse()` method, except it takes a string object instead of a file-like object. Calling this method on a string is equivalent to wrapping *text* in a `StringIO` instance first and calling `parse()`.

Optional *headersonly* is as with the `parse()` method.

class `email.parser.`**`HeaderParser`**(*_class=None*, ***, *policy=policy.compat32*)

Exactly like `Parser`, except that *headersonly* defaults to `True`.

Since creating a message object structure from a string or a file object is such a common task, four functions are provided as a convenience. They are available in the top-level `email` package namespace.

`email.`**`message_from_bytes`**(*s*, *_class=None*, ***, *policy=policy.compat32*)

Return a message object structure from a bytes-like object. This is equivalent to `BytesParser().parsebytes(s)`. Optional *_class* and *strict* are interpreted as with the `BytesParser` class constructor.

New in version 3.2.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

`message_from_binary_file(fp, _class=None, *, policy=policy.compat32)`

Return a message object structure tree from an open binary file object. This is equivalent to `BytesParser().parse(fp)`. *_class* and *policy* are interpreted as with the `BytesParser` class constructor.

New in version 3.2.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

`email.`**`message_from_string`**(*s*, *_class=None*, ***, *policy=policy.compat32*)

Return a message object structure from a string. This is equivalent to `Parser().parsestr(s)`. *_class* and *policy* are interpreted as with the `Parser` class constructor.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

`email.`**`message_from_file`**(*fp*, *_class=None*, ***, *policy=policy.compat32*)

Return a message object structure tree from an open file object. This is equivalent to `Parser().parse(fp)`. *_class* and *policy* are interpreted as with the `Parser` class constructor.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

Changed in version 3.6: *_class* defaults to the policy `message_factory`.

Here’s an example of how you might use `message_from_bytes()` at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

19.1.2.3. Additional notes

Here are some notes on the parsing semantics:

- Most non-*multipart* type messages are parsed as a single message object with a string payload. These objects will return `False` for `is_multipart()`, and `iter_parts()` will yield an empty list.
- All *multipart* type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for `is_multipart()`, and `iter_parts()` will yield a list of subparts.
- Most messages with a content type of *message/** (such as *message/delivery-status* and *message/rfc822*) will also be parsed as container object containing a list payload of length 1. Their `is_multipart()` method will return `True`. The single element yielded by `iter_parts()` will be a sub-message object.
- Some non-standards-compliant messages may not be internally consistent about their *multipart*-edness. Such messages may have a *Content-Type* header of type *multipart*, but their `is_multipart()` method may return `False`. If such messages were parsed with the `FeedParser`, they will have an instance of the `MultipartInvariantViolationDefect` class in their *defects* attribute list. See `email.errors` for details.