email Package Reference

Release 2.5

Barry Warsaw

March 5, 2006

barry@zope.com

Abstract

The `email` package provides classes and utilities to create, parse, generate, and modify email messages, conforming to all the relevant email and MIME related RFCs.

Contents

1 Introduction

The `email` package provides classes and utilities to create, parse, generate, and modify email messages, conforming to all the relevant email and MIME related RFCs.

This document describes the current version of the `email` package, which is available to Python programmers in a number of ways. Python 2.2.2 and 2.3 come with `email` version 2, while earlier versions of Python 2.2.x come with `email` version 1. Python 2.1.x and earlier do not come with any version of the `email` package.

The email package is also available as a standalone distutils package, and is compatible with Python 2.1.3 and beyond. Thus, if you're using Python 2.1.3 you can download the standalone package and install it in your 'site-packages' directory. The standalone email package is available on the SourceForge mimelib project.

The documentation that follows was written for the Python project, so if you're reading this as part of the standalone email package documentation, there are a few notes to be aware of:

- Deprecation and "version added" notes are relative to the Python version a feature was added or deprecated. To find out what version of the email package a particular item was added, changed, or removed, refer to the package's 'NEWS' file.
- The code samples are written with Python 2.2 in mind. For Python 2.1.3, some adjustments are necessary. For example, this code snippet;

  ```
  if isinstance(s, str):
      # ...
  ```

  would need to be written this way in Python 2.1.3:

  ```
  from types import StringType
  # ...
  if isinstance(s, StringType):
      # ...
  ```

- If you're reading this documentation as part of the standalone email package, some of the internal links to other sections of the Python standard library may not resolve.

2 email — An email and MIME handling package

New in version 2.2.

The email package is a library for managing email messages, including MIME and other RFC 2822-based message documents. It subsumes most of the functionality in several older standard modules such as rfc822, mimetools, multifile, and other non-standard packages such as mimecntl. It is specifically *not* designed to do any sending of email messages to SMTP (RFC 2821) servers; that is the function of the smtplib module. The email package attempts to be as RFC-compliant as possible, supporting in addition to RFC 2822, such MIME-related RFCs as RFC 2045-RFC 2047, and RFC 2231.

The primary distinguishing feature of the email package is that it splits the parsing and generating of email messages from the internal *object model* representation of email. Applications using the email package deal primarily with objects; you can add sub-objects to messages, remove sub-objects from messages, completely re-arrange the contents, etc. There is a separate parser and a separate generator which handles the transformation from flat text to the object model, and then back to flat text again. There are also handy subclasses for some common MIME object types, and a few miscellaneous utilities that help with such common tasks as extracting and parsing message field values, creating RFC-compliant dates, etc.

The following sections describe the functionality of the email package. The ordering follows a progression that should be common in applications: an email message is read as flat text from a file or other source, the text is parsed to produce the object structure of the email message, this structure is manipulated, and finally rendered back into flat text.

It is perfectly feasible to create the object structure out of whole cloth — i.e. completely from scratch. From there, a similar progression can be taken as above.

Also included are detailed specifications of all the classes and modules that the `email` package provides, the exception classes you might encounter while using the `email` package, some auxiliary utilities, and a few examples. For users of the older `mimelib` package, or previous versions of the `email` package, a section on differences and porting is provided.

See Also:

Module `smtplib` (section **??**):
SMTP protocol client

2.1 Representing an email message

The central class in the `email` package is the `Message` class; it is the base class for the `email` object model. `Message` provides the core functionality for setting and querying header fields, and for accessing message bodies.

Conceptually, a `Message` object consists of *headers* and *payloads*. Headers are RFC 2822 style field names and values where the field name and value are separated by a colon. The colon is not part of either the field name or the field value.

Headers are stored and returned in case-preserving form but are matched case-insensitively. There may also be a single envelope header, also known as the *Unix-From* header or the `From_` header. The payload is either a string in the case of simple message objects or a list of `Message` objects for MIME container documents (e.g. multipart/* and message/rfc822).

`Message` objects provide a mapping style interface for accessing the message headers, and an explicit interface for accessing both the headers and the payload. It provides convenience methods for generating a flat text representation of the message object tree, for accessing commonly used header parameters, and for recursively walking over the object tree.

Here are the methods of the `Message` class:

class `Message`()
The constructor takes no arguments.

`as_string`([*unixfrom*])
Return the entire message flatten as a string. When optional *unixfrom* is `True`, the envelope header is included in the returned string. *unixfrom* defaults to `False`.

Note that this method is provided as a convenience and may not always format the message the way you want. For more flexibility, instantiate a `Generator` instance and use its `flatten()` method directly. For example:

```
from cStringIO import StringIO
from email.Generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=False, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

`__str__`()
Equivalent to `as_string(unixfrom=True)`.

`is_multipart`()
Return `True` if the message's payload is a list of sub-`Message` objects, otherwise return `False`. When `is_multipart()` returns False, the payload should be a string object.

`set_unixfrom`(*unixfrom*)
Set the message's envelope header to *unixfrom*, which should be a string.

get_unixfrom()
Return the message's envelope header. Defaults to None if the envelope header was never set.

attach(*payload*)
Add the given *payload* to the current payload, which must be None or a list of Message objects before the call. After the call, the payload will always be a list of Message objects. If you want to set the payload to a scalar object (e.g. a string), use set_payload() instead.

get_payload([*i*[, *decode*]])
Return a reference the current payload, which will be a list of Message objects when is_multipart() is True, or a string when is_multipart() is False. If the payload is a list and you mutate the list object, you modify the message's payload in place.

With optional argument *i*, get_payload() will return the *i*-th element of the payload, counting from zero, if is_multipart() is True. An IndexError will be raised if *i* is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. is_multipart() is False) and *i* is given, a TypeError is raised.

Optional *decode* is a flag indicating whether the payload should be decoded or not, according to the Content-Transfer-Encoding: header. When True and the message is not a multipart, the payload will be decoded if this header's value is '`quoted-printable`' or '`base64`'. If some other encoding is used, or Content-Transfer-Encoding: header is missing, or if the payload has bogus base64 data, the payload is returned as-is (undecoded). If the message is a multipart and the *decode* flag is True, then None is returned. The default for *decode* is False.

set_payload(*payload*[, *charset*])
Set the entire message object's payload to *payload*. It is the client's responsibility to ensure the payload invariants. Optional *charset* sets the message's default character set; see set_charset() for details.

Changed in version 2.2.2: *charset* argument added.

set_charset(*charset*)
Set the character set of the payload to *charset*, which can either be a Charset instance (see email.Charset), a string naming a character set, or None. If it is a string, it will be converted to a Charset instance. If *charset* is None, the charset parameter will be removed from the Content-Type: header. Anything else will generate a TypeError.

The message will be assumed to be of type text/* encoded with charset.input_charset. It will be converted to charset.output_charset and encoded properly, if needed, when generating the plain text representation of the message. MIME headers (MIME-Version:, Content-Type:, Content-Transfer-Encoding:) will be added as needed.

New in version 2.2.2.

get_charset()
Return the Charset instance associated with the message's payload. New in version 2.2.2.

The following methods implement a mapping-like interface for accessing the message's RFC 2822 headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by keys(), but in a Message object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

__len__()
Return the total number of headers, including duplicates.

__contains__(*name*)
Return true if the message object has a field named *name*. Matching is done case-insensitively and *name* should

not include the trailing colon. Used for the `in` operator, e.g.:

```
if 'message-id' in myMessage:
    print 'Message-ID:', myMessage['message-id']
```

__getitem__(*name*)
Return the value of the named header field. *name* should not include the colon field separator. If the header is missing, `None` is returned; a `KeyError` is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the `get_all()` method to get the values of all the extant named headers.

__setitem__(*name*, *val*)
Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing fields.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)
Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

has_key(*name*)
Return true if the message contains a header field named *name*, otherwise return false.

keys()
Return a list of all the message's header field names.

values()
Return a list of all the message's field values.

items()
Return a list of 2-tuples containing all the message's field headers and values.

get(*name*[, *failobj*])
Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (defaults to `None`).

Here are some additional useful methods:

get_all(*name*[, *failobj*])
Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)
Extended header setting. This method is similar to `__setitem__()` except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added.

Here's an example:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

replace_header(*_name, _value*)
Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case. If no matching header was found, a `KeyError` is raised.

New in version 2.2.2.

get_content_type()
Return the message's content type. The returned string is coerced to lower case of the form maintype/subtype. If there was no Content-Type: header in the message the default type as given by `get_default_type()` will be returned. Since according to RFC 2045, messages always have a default type, `get_content_type()` will always return a value.

RFC 2045 defines a message's default type to be text/plain unless it appears inside a multipart/digest container, in which case it would be message/rfc822. If the Content-Type: header has an invalid type specification, RFC 2045 mandates that the default type be text/plain.

New in version 2.2.2.

get_content_maintype()
Return the message's main content type. This is the maintype part of the string returned by `get_content_type()`.

New in version 2.2.2.

get_content_subtype()
Return the message's sub-content type. This is the subtype part of the string returned by `get_content_type()`.

New in version 2.2.2.

get_default_type()
Return the default content type. Most messages have a default content type of text/plain, except for messages that are subparts of multipart/digest containers. Such subparts have a default content type of message/rfc822.

New in version 2.2.2.

set_default_type(*ctype*)
Set the default content type. *ctype* should either be text/plain or message/rfc822, although this is not enforced. The default content type is not stored in the Content-Type: header.

New in version 2.2.2.

get_params([*failobj*[, *header*[, *unquote*]]])
Return the message's Content-Type: parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the '=' sign. The left hand side of the '=' is the key, while the right hand side is the value. If there is no '=' sign in the parameter the value is the empty string, otherwise the value is as described in `get_param()` and is unquoted if optional *unquote* is `True` (the default).

Optional *failobj* is the object to return if there is no Content-Type: header. Optional *header* is the header to search instead of Content-Type:.

Changed in version 2.2.2: *unquote* argument added.

get_param(*param*[, *failobj*[, *header*[, *unquote*]]])
Return the value of the Content-Type: header's parameter *param* as a string. If the message has no Content-Type: header or if there is no such parameter, then *failobj* is returned (defaults to `None`).

Optional *header* if given, specifies the message header to use instead of Content-Type:.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was RFC 2231 encoded. When it's a 3-tuple, the elements of the value are of the form `(CHARSET, LANGUAGE, VALUE)`. Note that both `CHARSET` and `LANGUAGE` can be `None`, in which case you should consider `VALUE` to be encoded in the `us-ascii` charset. You can usually ignore `LANGUAGE`.

Your application should be prepared to deal with 3-tuple return values, and can convert the parameter to a Unicode string like so:

```
param = msg.get_param('foo')
if isinstance(param, tuple):
    param = unicode(param[2], param[0] or 'us-ascii')
```

In any case, the parameter value (either the returned string, or the VALUE item in the 3-tuple) is always unquoted, unless *unquote* is set to `False`.

Changed in version 2.2.2: *unquote* argument added, and 3-tuple return value possible.

`set_param`(*param, value*[, *header*[, *requote*[, *charset*[, *language*]]]])
Set a parameter in the Content-Type: header. If the parameter already exists in the header, its value will be replaced with *value*. If the Content-Type: header as not yet been defined for this message, it will be set to text/plain and the new parameter value will be appended as per RFC 2045.

Optional *header* specifies an alternative header to Content-Type:, and all parameters will be quoted as necessary unless optional *requote* is `False` (the default is `True`).

If optional *charset* is specified, the parameter will be encoded according to RFC 2231. Optional *language* specifies the RFC 2231 language, defaulting to the empty string. Both *charset* and *language* should be strings.

New in version 2.2.2.

`del_param`(*param*[, *header*[, *requote*]])
Remove the given parameter completely from the Content-Type: header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless *requote* is `False` (the default is `True`). Optional *header* specifies an alternative to Content-Type:.

New in version 2.2.2.

`set_type`(*type*[, *header*][, *requote*])
Set the main type and subtype for the Content-Type: header. *type* must be a string in the form maintype/subtype, otherwise a `ValueError` is raised.

This method replaces the Content-Type: header, keeping all the parameters in place. If *requote* is `False`, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the *header* argument. When the Content-Type: header is set a MIME-Version: header is also added.

New in version 2.2.2.

`get_filename`([*failobj*])
Return the value of the `filename` parameter of the Content-Disposition: header of the message. If the header does not have a `filename` parameter, this method falls back to looking for the `name` parameter. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per `Utils.unquote()`.

`get_boundary`([*failobj*])
Return the value of the `boundary` parameter of the Content-Type: header of the message, or *failobj* if either the header is missing, or has no `boundary` parameter. The returned string will always be unquoted as per `Utils.unquote()`.

`set_boundary`(*boundary*)
Set the `boundary` parameter of the Content-Type: header to *boundary*. `set_boundary()` will always quote *boundary* if necessary. A `HeaderParseError` is raised if the message object has no Content-Type: header.

Note that using this method is subtly different than deleting the old Content-Type: header and adding a new one with the new boundary via `add_header()`, because `set_boundary()` preserves the order of the Content-Type: header in the list of headers. However, it does *not* preserve any continuation lines which may have been present in the original Content-Type: header.

get_content_charset([*failobj*])
Return the charset parameter of the Content-Type: header, coerced to lower case. If there is no Content-Type: header, or if that header has no charset parameter, *failobj* is returned.

Note that this method differs from get_charset() which returns the Charset instance for the default encoding of the message body.

New in version 2.2.2.

get_charsets([*failobj*])
Return a list containing the character set names in the message. If the message is a multipart, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the charset parameter in the Content-Type: header for the represented subpart. However, if the subpart has no Content-Type: header, no charset parameter, or is not of the text main MIME type, then that item in the returned list will be *failobj*.

walk()
The walk() method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use walk() as the iterator in a for loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
>>>     print part.get_content_type()
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
```

Message objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

preamble
The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the Parser discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the Generator is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See email.Parser and email.Generator for details.

Note that if the message object has no preamble, the *preamble* attribute will be None.

epilogue
The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message.

One note: when generating the flat text for a multipart message that has no *epilogue* (using the standard Generator class), no newline is added after the closing boundary line. If the message object has an *epilogue* and its value does not start with a newline, a newline is printed after the closing boundary. This seems a little clumsy, but it makes the most practical sense. The upshot is that if you want to ensure that a newline get printed after your closing multipart boundary, set the *epilogue* to the empty string.

Deprecated methods

The following methods are deprecated in `email` version 2. They are documented here for completeness.

`add_payload`(*payload*)
Add *payload* to the message object's existing payload. If, prior to calling this method, the object's payload was `None` (i.e. never before set), then after this method is called, the payload will be the argument *payload*.

If the object's payload was already a list (i.e. `is_multipart()` returns 1), then *payload* is appended to the end of the existing payload list.

For any other type of existing payload, `add_payload()` will transform the new payload into a list consisting of the old payload and *payload*, but only if the document is already a MIME multipart document. This condition is satisfied if the message's Content-Type: header's main type is either multipart, or there is no Content-Type: header. In any other situation, `MultipartConversionError` is raised.

Deprecated since release 2.2.2. Use the `attach()` method instead.

`get_type`([*failobj*])
Return the message's content type, as a string of the form maintype/subtype as taken from the Content-Type: header. The returned string is coerced to lowercase.

If there is no Content-Type: header in the message, *failobj* is returned (defaults to `None`).

Deprecated since release 2.2.2. Use the `get_content_type()` method instead.

`get_main_type`([*failobj*])
Return the message's *main* content type. This essentially returns the *maintype* part of the string returned by `get_type()`, with the same semantics for *failobj*.

Deprecated since release 2.2.2. Use the `get_content_maintype()` method instead.

`get_subtype`([*failobj*])
Return the message's sub-content type. This essentially returns the *subtype* part of the string returned by `get_type()`, with the same semantics for *failobj*.

Deprecated since release 2.2.2. Use the `get_content_subtype()` method instead.

2.2 Parsing email messages

Message object structures can be created in one of two ways: they can be created from whole cloth by instantiating `Message` objects and stringing them together via `attach()` and `set_payload()` calls, or they can be created by parsing a flat text 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 string or a file object, and the parser will return to you the root `Message` 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 `get_payload()` and `walk()` methods.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. There is no magical connection between the `email` package's bundled parser and the `Message` class, so your custom parser can create message object trees any way it finds necessary.

The primary parser class is `Parser` which parses both the headers and the payload of the message. In the case of multipart messages, it will recursively parse the body of the container message. Two modes of parsing are supported, *strict* parsing, which will usually reject any non-RFC compliant message, and *lax* parsing, which attempts to adjust for common MIME formatting problems.

The `email.Parser` module also provides a second class, called `HeaderParser` which can be used if you're only interested in the headers of the message. `HeaderParser` can be much faster in these situations, since it does not

attempt to parse the message body, instead setting the payload to the raw body as a string. `HeaderParser` has the same API as the `Parser` class.

Parser class API

class `Parser`(*[_class[, strict]]*)

The constructor for the `Parser` class takes an optional argument *_class*. This must be a callable factory (such as a function or a class), and it is used whenever a sub-message object needs to be created. It defaults to `Message` (see `email.Message`). The factory will be called without arguments.

The optional *strict* flag specifies whether strict or lax parsing should be performed. When things like MIME terminating boundaries are missing, or when messages contain other formatting problems, the `Parser` will raise a `MessageParseError`, if the *strict* flag is `True`. However, when lax parsing is enabled (i.e. *strict* is `False`), the `Parser` will attempt to work around such broken formatting to produce a usable message structure (this doesn't mean `MessageParseError`s are never raised; some ill-formatted messages just can't be parsed). The *strict* flag defaults to `False` since lax parsing usually provides the most convenient behavior.

Changed in version 2.2.2: The *strict* flag was added.

The other public `Parser` methods are:

`parse`(*fp[, headersonly]*)

Read all the data from the 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.

The text contained in *fp* must be formatted as a block of RFC 2822 style headers and header continuation lines, optionally preceded by a 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).

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

Changed in version 2.2.2: The *headersonly* flag was added.

`parsestr`(*text[, headersonly]*)

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

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.

Changed in version 2.2.2: The *headersonly* flag was added.

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

`message_from_string`(*s[, _class[, strict]]*)

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

Changed in version 2.2.2: The *strict* flag was added.

`message_from_file`(*fp[, _class[, strict]]*)

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

Changed in version 2.2.2: The *strict* flag was added.

Here's an example of how you might use this at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_string(myString)
```

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()`. Their `get_payload()` method will return a string object.
- 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 their `get_payload()` method will return the list of `Message` subparts.
- Most messages with a content type of message/* (e.g. 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 in the list payload will be a sub-message object.

2.3 Generating MIME documents

One of the most common tasks is to generate the flat text of the email message represented by a message object structure. You will need to do this if you want to send your message via the `smtplib` module or the `nntplib` module, or print the message on the console. Taking a message object structure and producing a flat text document is the job of the `Generator` class.

Again, as with the `email.Parser` module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the transformation from flat text, to a message structure via the `Parser` class, and back to flat text, is idempotent (the input is identical to the output).

Here are the public methods of the `Generator` class:

class `Generator`(*outfp*[, *mangle_from_*[, *maxheaderlen*]])

The constructor for the `Generator` class takes a file-like object called *outfp* for an argument. *outfp* must support the `write()` method and be usable as the output file in a Python extended print statement.

Optional *mangle_from_* is a flag that, when `True`, puts a '>' character in front of any line in the body that starts exactly as '`From `', i.e. `From` followed by a space at the beginning of the line. This is the only guaranteed portable way to avoid having such lines be mistaken for a Unix mailbox format envelope header separator (see WHY THE CONTENT-LENGTH FORMAT IS BAD for details). *mangle_from_* defaults to `True`, but you might want to set this to `False` if you are not writing Unix mailbox format files.

Optional *maxheaderlen* specifies the longest length for a non-continued header. When a header line is longer than *maxheaderlen* (in characters, with tabs expanded to 8 spaces), the header will be split as defined in the `email.Header` class. Set to zero to disable header wrapping. The default is 78, as recommended (but not required) by RFC 2822.

The other public `Generator` methods are:

`flatten`(*msg*[, *unixfrom*])

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the `Generator` instance was created. Subparts are visited depth-first and the resulting text will be properly MIME encoded.

Optional *unixfrom* is a flag that forces the printing of the envelope header delimiter before the first RFC 2822 header of the root message object. If the root object has no envelope header, a standard one is crafted. By default, this is set to `False` to inhibit the printing of the envelope delimiter.

Note that for subparts, no envelope header is ever printed.

New in version 2.2.2.

clone(*fp*)
Return an independent clone of this `Generator` instance with the exact same options.

New in version 2.2.2.

write(*s*)
Write the string *s* to the underlying file object, i.e. *outfp* passed to `Generator`'s constructor. This provides just enough file-like API for `Generator` instances to be used in extended print statements.

As a convenience, see the methods `Message.as_string()` and `str(aMessage)`, a.k.a. `Message.__str__()`, which simplify the generation of a formatted string representation of a message object. For more detail, see `email.Message`.

The `email.Generator` module also provides a derived class, called `DecodedGenerator` which is like the `Generator` base class, except that non-text parts are substituted with a format string representing the part.

class DecodedGenerator(*outfp*[, *mangle_from_*[, *maxheaderlen*[, *fmt*]]])
This class, derived from `Generator` walks through all the subparts of a message. If the subpart is of main type text, then it prints the decoded payload of the subpart. Optional *_mangle_from_* and *maxheaderlen* are as with the `Generator` base class.

If the subpart is not of main type text, optional *fmt* is a format string that is used instead of the message payload. *fmt* is expanded with the following keywords, '`%(keyword)s`' format:

- `type` – Full MIME type of the non-text part
- `maintype` – Main MIME type of the non-text part
- `subtype` – Sub-MIME type of the non-text part
- `filename` – Filename of the non-text part
- `description` – Description associated with the non-text part
- `encoding` – Content transfer encoding of the non-text part

The default value for *fmt* is `None`, meaning

```
[Non-text (%(type)s) part of message omitted, filename %(filename)s]
```

New in version 2.2.2.

Deprecated methods

The following methods are deprecated in `email` version 2. They are documented here for completeness.

__call__(*msg*[, *unixfrom*])
This method is identical to the `flatten()` method.

Deprecated since release 2.2.2. Use the `flatten()` method instead.

2.4 Creating email and MIME objects from scratch

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual `Message` objects by hand. In fact, you can also take an existing structure and add new `Message` objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating `Message` instances, adding attachments and all the appropriate headers manually. For MIME messages though, the `email` package provides some convenient subclasses to make things easier. Each of these classes should be imported from a module with the same name as the class, from within the `email` package. E.g.:

```
import email.MIMEImage.MIMEImage
```

or

```
from email.MIMEText import MIMEText
```

Here are the classes:

class `MIMEBase`(*_maintype*, *_subtype*, ***_params*)
This is the base class for all the MIME-specific subclasses of `Message`. Ordinarily you won't create instances specifically of `MIMEBase`, although you could. `MIMEBase` is provided primarily as a convenient base class for more specific MIME-aware subclasses.

_maintype is the Content-Type: major type (e.g. text or image), and *_subtype* is the Content-Type: minor type (e.g. plain or gif). *_params* is a parameter key/value dictionary and is passed directly to `Message.add_header()`.

The `MIMEBase` class always adds a Content-Type: header (based on *_maintype*, *_subtype*, and *_params*), and a MIME-Version: header (always set to `1.0`).

class `MIMENonMultipart`()
A subclass of `MIMEBase`, this is an intermediate base class for MIME messages that are not multipart. The primary purpose of this class is to prevent the use of the `attach()` method, which only makes sense for multipart messages. If `attach()` is called, a `MultipartConversionError` exception is raised.

New in version 2.2.2.

class `MIMEMultipart`([*subtype*[, *boundary*[, *_subparts*[, *_params*]]]])
A subclass of `MIMEBase`, this is an intermediate base class for MIME messages that are multipart. Optional *_subtype* defaults to mixed, but can be used to specify the subtype of the message. A Content-Type: header of multipart/*_subtype* will be added to the message object. A MIME-Version: header will also be added.

Optional *boundary* is the multipart boundary string. When `None` (the default), the boundary is calculated when needed.

_subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the `Message.attach()` method.

Additional parameters for the Content-Type: header are taken from the keyword arguments, or passed into the *_params* argument, which is a keyword dictionary.

New in version 2.2.2.

class `MIMEAudio`(*_audiodata*[, *_subtype*[, *_encoder*[, ***_params*]]])
A subclass of `MIMENonMultipart`, the `MIMEAudio` class is used to create MIME message objects of major type audio. *_audiodata* is a string containing the raw audio data. If this data can be decoded by the standard Python module `sndhdr`, then the subtype will be automatically included in the Content-Type: header. Otherwise you can explicitly specify the audio subtype via the *_subtype* parameter. If the minor type could not be guessed and *_subtype* was not given, then `TypeError` is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the `MIMEAudio` instance. It should use `get_payload()` and `set_payload()` to change the payload to encoded form. It should also add any Content-Transfer-Encoding: or other headers to the message object as necessary. The default encoding is base64. See the `email.Encoders` module for a list of the built-in encoders.

_params are passed straight through to the base class constructor.

class `MIMEImage`(*_imagedata*[, *_subtype*[, *_encoder*[, ***_params*]]])
A subclass of `MIMENonMultipart`, the `MIMEImage` class is used to create MIME message objects of major type image. *_imagedata* is a string containing the raw image data. If this data can be decoded by the standard

Python module `imghdr`, then the subtype will be automatically included in the Content-Type: header. Otherwise you can explicitly specify the image subtype via the *_subtype* parameter. If the minor type could not be guessed and *_subtype* was not given, then `TypeError` is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the `MIMEImage` instance. It should use `get_payload()` and `set_payload()` to change the payload to encoded form. It should also add any Content-Transfer-Encoding: or other headers to the message object as necessary. The default encoding is base64. See the `email.Encoders` module for a list of the built-in encoders.

_params are passed straight through to the `MIMEBase` constructor.

class `MIMEMessage`(*_msg*[, *_subtype*]**)**

A subclass of `MIMENonMultipart`, the `MIMEMessage` class is used to create MIME objects of main type message. *_msg* is used as the payload, and must be an instance of class `Message` (or a subclass thereof), otherwise a `TypeError` is raised.

Optional *_subtype* sets the subtype of the message; it defaults to rfc822.

class `MIMEText`(*_text*[, *_subtype*[, *_charset*[, *_encoder*]]]**)**

A subclass of `MIMENonMultipart`, the `MIMEText` class is used to create MIME objects of major type text. *_text* is the string for the payload. *_subtype* is the minor type and defaults to plain. *_charset* is the character set of the text and is passed as a parameter to the `MIMENonMultipart` constructor; it defaults to `us-ascii`. No guessing or encoding is performed on the text data.

Deprecated since release 2.2.2. The *_encoding* argument has been deprecated. Encoding now happens implicitly based on the *_charset* argument.

2.5 Internationalized headers

RFC 2822 is the base standard that describes the format of email messages. It derives from the older RFC 822 standard which came into widespread use at a time when most email was composed of ASCII characters only. RFC 2822 is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transfered using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into RFC 2822-compliant format. These RFCs include RFC 2045, RFC 2046, RFC 2047, and RFC 2231. The `email` package supports these standards in its `email.Header` and `email.Charset` modules.

If you want to include non-ASCII characters in your email headers, say in the Subject: or To: fields, you should use the `Header` class and assign the field in the `Message` object to an instance of `Header` instead of using a string for the header value. For example:

```
>>> from email.Message import Message
>>> from email.Header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> print msg.as_string()
Subject: =?iso-8859-1?q?p=F6stal?=
```

Notice here how we wanted the Subject: field to contain a non-ASCII character? We did this by creating a `Header` instance and passing in the character set that the byte string was encoded in. When the subsequent `Message` instance was flattened, the Subject: field was properly RFC 2047 encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

New in version 2.2.2.

Here is the `Header` class description:

class `Header`(`[`*s*`[`, *charset*`[`, *maxlinelen*`[`, *header_name*`[`, *continuation_ws*`[`, *errors*`]]]]]]`)
Create a MIME-compliant header that can contain strings in different character sets.

Optional *s* is the initial header value. If `None` (the default), the initial header value is not set. You can later append to the header with `append()` method calls. *s* may be a byte string or a Unicode string, but see the `append()` documentation for semantics.

Optional *charset* serves two purposes: it has the same meaning as the *charset* argument to the `append()` method. It also sets the default character set for all subsequent `append()` calls that omit the *charset* argument. If *charset* is not provided in the constructor (the default), the `us-ascii` character set is used both as *s*'s initial charset and as the default for subsequent `append()` calls.

The maximum line length can be specified explicit via *maxlinelen*. For splitting the first line to a shorter value (to account for the field header which isn't included in *s*, e.g. `Subject`:) pass in the name of the field in *header_name*. The default *maxlinelen* is 76, and the default value for *header_name* is `None`, meaning it is not taken into account for the first line of a long, split header.

Optional *continuation_ws* must be RFC 2822-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines.

Optional *errors* is passed straight through to the `append()` method.

`append`(*s*`[`, *charset*`[`, *errors*`]]`)
Append the string *s* to the MIME header.

Optional *charset*, if given, should be a `Charset` instance (see `email.Charset`) or the name of a character set, which will be converted to a `Charset` instance. A value of `None` (the default) means that the *charset* given in the constructor is used.

s may be a byte string or a Unicode string. If it is a byte string (i.e. `isinstance(s, str)` is true), then *charset* is the encoding of that byte string, and a `UnicodeError` will be raised if the string cannot be decoded with that character set.

If *s* is a Unicode string, then *charset* is a hint specifying the character set of the characters in the string. In this case, when producing an RFC 2822-compliant header using RFC 2047 rules, the Unicode string will be encoded using the following charsets in order: `us-ascii`, the *charset* hint, `utf-8`. The first character set to not provoke a `UnicodeError` is used.

Optional *errors* is passed through to any `unicode()` or `ustr.encode()` call, and defaults to "strict".

`encode`(`[`*splitchars*`]`)
Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings. Optional *splitchars* is a string containing characters to split long ASCII lines on, in rough support of RFC 2822's *highest level syntactic breaks*. This doesn't affect RFC 2047 encoded lines.

The `Header` class also provides a number of methods to support standard operators and built-in functions.

`__str__`()
A synonym for `Header.encode()`. Useful for `str(aHeader)`.

`__unicode__`()
A helper for the built-in `unicode()` function. Returns the header as a Unicode string.

`__eq__`(*other*)
This method allows you to compare two `Header` instances for equality.

`__ne__`(*other*)
This method allows you to compare two `Header` instances for inequality.

The `email.Header` module also provides the following convenient functions.

decode_header(*header*)

Decode a message header value without converting the character set. The header value is in *header*.

This function returns a list of `(decoded_string, charset)` pairs containing each of the decoded parts of the header. *charset* is `None` for non-encoded parts of the header, otherwise a lower case string containing the name of the character set specified in the encoded string.

Here's an example:

```
>>> from email.Header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[('p\\xf6stal', 'iso-8859-1')]
```

make_header(*decoded_seq*[, *maxlinelen*[, *header_name*[, *continuation_ws*]]])

Create a `Header` instance from a sequence of pairs as returned by `decode_header()`.

`decode_header()` takes a header value string and returns a sequence of pairs of the format `(decoded_string, charset)` where *charset* is the name of the character set.

This function takes one of those sequence of pairs and returns a `Header` instance. Optional *maxlinelen*, *header_name*, and *continuation_ws* are as in the `Header` constructor.

2.6 Representing character sets

This module provides a class `Charset` for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of `Charset` are used in several other modules within the `email` package.

New in version 2.2.2.

class Charset([*input_charset*])

Map character sets to their email properties.

This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.

Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional *input_charset* is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if *input_charset* is `iso-8859-1`, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If *input_charset* is `euc-jp`, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the `euc-jp` character set to the `iso-2022-jp` character set.

`Charset` instances have the following data attributes:

input_charset

The initial character set specified. Common aliases are converted to their *official* email names (e.g. `latin_1` is converted to `iso-8859-1`). Defaults to 7-bit `us-ascii`.

header_encoding

If the character set must be encoded before it can be used in an email header, this attribute will be set to `Charset.QP` (for quoted-printable), `Charset.BASE64` (for base64 encoding), or `Charset.SHORTEST` for the shortest of QP or BASE64 encoding. Otherwise, it will be `None`.

body_encoding
Same as *header_encoding*, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. `Charset.SHORTEST` is not allowed for *body_encoding*.

output_charset
Some character sets must be converted before they can be used in email headers or bodies. If the *input_charset* is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be `None`.

input_codec
The name of the Python codec used to convert the *input_charset* to Unicode. If no conversion codec is necessary, this attribute will be `None`.

output_codec
The name of the Python codec used to convert Unicode to the *output_charset*. If no conversion codec is necessary, this attribute will have the same value as the *input_codec*.

`Charset` instances also have the following methods:

get_body_encoding()
Return the content transfer encoding used for body encoding.

This is either the string '`quoted-printable`' or '`base64`' depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the Content-Transfer-Encoding: header itself to whatever is appropriate.

Returns the string '`quoted-printable`' if *body_encoding* is `QP`, returns the string '`base64`' if *body_encoding* is `BASE64`, and returns the string '`7bit`' otherwise.

convert(*s*)
Convert the string *s* from the *input_codec* to the *output_codec*.

to_splittable(*s*)
Convert a possibly multibyte string to a safely splittable format. *s* is the string to split.

Uses the *input_codec* to try and convert the string to Unicode, so it can be safely split on character boundaries (even for multibyte characters).

Returns the string as-is if it isn't known how to convert *s* to Unicode with the *input_charset*.

Characters that could not be converted to Unicode will be replaced with the Unicode replacement character '`U+FFFD`'.

from_splittable(*ustr*[, *to_output*])
Convert a splittable string back into an encoded string. *ustr* is a Unicode string to "unsplit".

This method uses the proper codec to try and convert the string from Unicode back into an encoded format. Return the string as-is if it is not Unicode, or if it could not be converted from Unicode.

Characters that could not be converted from Unicode will be replaced with an appropriate character (usually '?').

If *to_output* is `True` (the default), uses *output_codec* to convert to an encoded format. If *to_output* is `False`, it uses *input_codec*.

get_output_charset()
Return the output character set.

This is the *output_charset* attribute if that is not `None`, otherwise it is *input_charset*.

encoded_header_len()
Return the length of the encoded header string, properly calculating for quoted-printable or base64 encoding.

header_encode(*s*[, *convert*])
Header-encode the string *s*.

If *convert* is `True`, the string will be converted from the input charset to the output charset automatically. This is not useful for multibyte character sets, which have line length issues (multibyte characters must be split on a character, not a byte boundary); use the higher-level `Header` class to deal with these issues (see `email.Header`). *convert* defaults to `False`.

The type of encoding (base64 or quoted-printable) will be based on the *header_encoding* attribute.

body_encode(*s*[, *convert*])
Body-encode the string *s*.

If *convert* is `True` (the default), the string will be converted from the input charset to output charset automatically. Unlike `header_encode()`, there are no issues with byte boundaries and multibyte charsets in email bodies, so this is usually pretty safe.

The type of encoding (base64 or quoted-printable) will be based on the *body_encoding* attribute.

The `Charset` class also provides a number of methods to support standard operations and built-in functions.

__str__()
Returns *input_charset* as a string coerced to lower case. `__repr__()` is an alias for `__str__()`.

__eq__(*other*)
This method allows you to compare two `Charset` instances for equality.

__ne__(*other*)
This method allows you to compare two `Charset` instances for inequality.

The `email.Charset` module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

add_charset(*charset*[, *header_enc*[, *body_enc*[, *output_charset*]]])
Add character properties to the global registry.

charset is the input character set, and must be the canonical name of a character set.

Optional *header_enc* and *body_enc* is either `Charset.QP` for quoted-printable, `Charset.BASE64` for base64 encoding, `Charset.SHORTEST` for the shortest of quoted-printable or base64 encoding, or `None` for no encoding. `SHORTEST` is only valid for *header_enc*. The default is `None` for no encoding.

Optional *output_charset* is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method `Charset.convert()` is called. The default is to output in the same character set as the input.

Both *input_charset* and *output_charset* must have Unicode codec entries in the module's character set-to-codec mapping; use `add_codec()` to add codecs the module does not know about. See the `codecs` module's documentation for more information.

The global character set registry is kept in the module global dictionary `CHARSETS`.

add_alias(*alias*, *canonical*)
Add a character set alias. *alias* is the alias name, e.g. `latin-1`. *canonical* is the character set's canonical name, e.g. `iso-8859-1`.

The global charset alias registry is kept in the module global dictionary `ALIASES`.

add_codec(*charset*, *codecname*)
Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. *codecname* is the name of a Python codec, as appropriate for the second argument to the `unicode()` built-in, or to the `encode()` method of a Unicode string.

2.7 Encoders

When creating `Message` objects from scratch, you often need to encode the payloads for transport through compliant mail servers. This is especially true for image/* and text/* type messages containing binary data.

The email package provides some convenient encodings in its Encoders module. These encoders are actually used by the MIMEImage and MIMEText class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the Content-Transfer-Encoding: header as appropriate.

Here are the encoding functions provided:

encode_quopri(*msg*)
: Encodes the payload into quoted-printable form and sets the Content-Transfer-Encoding: header to quoted-printable[1]. This is a good encoding to use when most of your payload is normal printable data, but contains a few unprintable characters.

encode_base64(*msg*)
: Encodes the payload into base64 form and sets the Content-Transfer-Encoding: header to base64. This is a good encoding to use when most of your payload is unprintable data since it is a more compact form than quoted-printable. The drawback of base64 encoding is that it renders the text non-human readable.

encode_7or8bit(*msg*)
: This doesn't actually modify the message's payload, but it does set the Content-Transfer-Encoding: header to either 7bit or 8bit as appropriate, based on the payload data.

encode_noop(*msg*)
: This does nothing; it doesn't even set the Content-Transfer-Encoding: header.

2.8 Exception classes

The following exception classes are defined in the email.Errors module:

exception MessageError()
: This is the base class for all exceptions that the email package can raise. It is derived from the standard Exception class and defines no additional methods.

exception MessageParseError()
: This is the base class for exceptions thrown by the Parser class. It is derived from MessageError.

exception HeaderParseError()
: Raised under some error conditions when parsing the RFC 2822 headers of a message, this class is derived from MessageParseError. It can be raised from the Parser.parse() or Parser.parsestr() methods.

  Situations where it can be raised include finding an envelope header after the first RFC 2822 header of the message, finding a continuation line before the first RFC 2822 header is found, or finding a line in the headers which is neither a header or a continuation line.

exception BoundaryError()
: Raised under some error conditions when parsing the RFC 2822 headers of a message, this class is derived from MessageParseError. It can be raised from the Parser.parse() or Parser.parsestr() methods.

  Situations where it can be raised include not being able to find the starting or terminating boundary in a multipart/* message when strict parsing is used.

exception MultipartConversionError()
: Raised when a payload is added to a Message object using add_payload(), but the payload is already a scalar and the message's Content-Type: main type is not either multipart or missing. MultipartConversionError multiply inherits from MessageError and the built-in TypeError.

  Since Message.add_payload() is deprecated, this exception is rarely raised in practice. However the exception may also be raised if the attach() method is called on an instance of a class derived from MIMENonMultipart (e.g. MIMEImage).

[1] Note that encoding with encode_quopri() also encodes all tabs and space characters in the data.

2.9 Miscellaneous utilities

There are several useful utilities provided in the `email.Utils` module:

`quote`(*str*)
Return a new string with backslashes in *str* replaced by two backslashes, and double quotes replaced by backslash-double quote.

`unquote`(*str*)
Return a new string which is an *unquoted* version of *str*. If *str* ends and begins with double quotes, they are stripped off. Likewise if *str* ends and begins with angle brackets, they are stripped off.

`parseaddr`(*address*)
Parse address – which should be the value of some address-containing field such as To: or Cc: – into its constituent *realname* and *email address* parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of `('', '')` is returned.

`formataddr`(*pair*)
The inverse of `parseaddr()`, this takes a 2-tuple of the form `(realname, email_address)` and returns the string value suitable for a To: or Cc: header. If the first element of *pair* is false, then the second element is returned unmodified.

`getaddresses`(*fieldvalues*)
This method returns a list of 2-tuples of the form returned by `parseaddr()`. *fieldvalues* is a sequence of header field values as might be returned by `Message.get_all()`. Here's a simple example that gets all the recipients of a message:

```
from email.Utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

`parsedate`(*date*)
Attempts to parse a date according to the rules in RFC 2822. however, some mailers don't follow that format as specified, so `parsedate()` tries to guess correctly in such cases. *date* is a string containing an RFC 2822 date, such as `"Mon, 20 Nov 1995 19:12:08 -0500"`. If it succeeds in parsing the date, `parsedate()` returns a 9-tuple that can be passed directly to `time.mktime()`; otherwise `None` will be returned. Note that fields 6, 7, and 8 of the result tuple are not usable.

`parsedate_tz`(*date*)
Performs the same function as `parsedate()`, but returns either `None` or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to `time.mktime()`, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time)[2]. If the input string has no timezone, the last element of the tuple returned is `None`. Note that fields 6, 7, and 8 of the result tuple are not usable.

`mktime_tz`(*tuple*)
Turn a 10-tuple as returned by `parsedate_tz()` into a UTC timestamp. It the timezone item in the tuple is `None`, assume local time. Minor deficiency: `mktime_tz()` interprets the first 8 elements of *tuple* as a local time and then compensates for the timezone difference. This may yield a slight error around changes in daylight savings time, though not worth worrying about for common use.

`formatdate`([*timeval*[, *localtime*]])
Returns a date string as per RFC 2822, e.g.:

[2] Note that the sign of the timezone offset is the opposite of the sign of the `time.timezone` variable for the same timezone; the latter variable follows the POSIX standard while this module follows RFC 2822.

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional *timeval* if given is a floating point time value as accepted by `time.gmtime()` and `time.localtime()`, otherwise the current time is used.

Optional *localtime* is a flag that when `True`, interprets *timeval*, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is `False` meaning UTC is used.

`make_msgid`(*[idstring]*)
Returns a string suitable for an RFC 2822-compliant Message-ID: header. Optional *idstring* if given, is a string used to strengthen the uniqueness of the message id.

`decode_rfc2231`(*s*)
Decode the string *s* according to RFC 2231.

`encode_rfc2231`(*s*[, *charset*[, *language*]])
Encode the string *s* according to RFC 2231. Optional *charset* and *language*, if given is the character set name and language name to use. If neither is given, *s* is returned as-is. If *charset* is given but *language* is not, the string is encoded using the empty string for *language*.

`decode_params`(*params*)
Decode parameters list according to RFC 2231. *params* is a sequence of 2-tuples containing elements of the form `(content-type, string-value)`.

The following functions have been deprecated:

`dump_address_pair`(*pair*)
Deprecated since release 2.2.2. Use `formataddr()` instead.

`decode`(*s*)
Deprecated since release 2.2.2. Use `Header.decode_header()` instead.

`encode`(*s*[, *charset*[, *encoding*]])
Deprecated since release 2.2.2. Use `Header.encode()` instead.

2.10 Iterators

Iterating over a message object tree is fairly easy with the `Message.walk()` method. The `email.Iterators` module provides some useful higher level iterations over message object trees.

`body_line_iterator`(*msg*[, *decode*])
This iterates over all the payloads in all the subparts of *msg*, returning the string payloads line-by-line. It skips over all the subpart headers, and it skips over any subpart with a payload that isn't a Python string. This is somewhat equivalent to reading the flat text representation of the message from a file using `readline()`, skipping over all the intervening headers.

Optional *decode* is passed through to `Message.get_payload()`.

`typed_subpart_iterator`(*msg*[, *maintype*[, *subtype*]])
This iterates over all the subparts of *msg*, returning only those subparts that match the MIME type specified by *maintype* and *subtype*.

Note that *subtype* is optional; if omitted, then subpart MIME type matching is done only with the main type. *maintype* is optional too; it defaults to text.

Thus, by default `typed_subpart_iterator()` returns each subpart that has a MIME type of text/*.

The following function has been added as a useful debugging tool. It should *not* be considered part of the supported public interface for the package.

_structure(*msg*[, *fp*[, *level*]])

Prints an indented representation of the content types of the message object structure. For example:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

Optional *fp* is a file-like object to print the output to. It must be suitable for Python's extended print statement. *level* is used internally.

2.11 Differences from email v1 (up to Python 2.2.1)

Version 1 of the email package was bundled with Python releases up to Python 2.2.1. Version 2 was developed for the Python 2.3 release, and backported to Python 2.2.2. It was also available as a separate distutils based package. email version 2 is almost entirely backward compatible with version 1, with the following differences:

- The email.Header and email.Charset modules have been added.
- The pickle format for Message instances has changed. Since this was never (and still isn't) formally defined, this isn't considered a backward incompatibility. However if your application pickles and unpickles Message instances, be aware that in email version 2, Message instances now have private variables *_charset* and *_default_type*.
- Several methods in the Message class have been deprecated, or their signatures changed. Also, many new methods have been added. See the documentation for the Message class for details. The changes should be completely backward compatible.
- The object structure has changed in the face of message/rfc822 content types. In email version 1, such a type would be represented by a scalar payload, i.e. the container message's is_multipart() returned false, get_payload() was not a list object, but a single Message instance.

  This structure was inconsistent with the rest of the package, so the object representation for message/rfc822 content types was changed. In email version 2, the container *does* return True from is_multipart(), and get_payload() returns a list containing a single Message item.

  Note that this is one place that backward compatibility could not be completely maintained. However, if you're already testing the return type of get_payload(), you should be fine. You just need to make sure your code doesn't do a set_payload() with a Message instance on a container with a content type of message/rfc822.
- The Parser constructor's *strict* argument was added, and its parse() and parsestr() methods grew a *headersonly* argument. The *strict* flag was also added to functions email.message_from_file() and email.message_from_string().

- `Generator.__call__()` is deprecated; use `Generator.flatten()` instead. The `Generator` class has also grown the `clone()` method.
- The `DecodedGenerator` class in the `email.Generator` module was added.
- The intermediate base classes `MIMENonMultipart` and `MIMEMultipart` have been added, and interposed in the class hierarchy for most of the other MIME-related derived classes.
- The *_encoder* argument to the `MIMEText` constructor has been deprecated. Encoding now happens implicitly based on the *_charset* argument.
- The following functions in the `email.Utils` module have been deprecated: `dump_address_pairs()`, `decode()`, and `encode()`. The following functions have been added to the module: `make_msgid()`, `decode_rfc2231()`, `encode_rfc2231()`, and `decode_params()`.
- The non-public function `email.Iterators._structure()` was added.

2.12 Differences from `mimelib`

The `email` package was originally prototyped as a separate library called `mimelib`. Changes have been made so that method names are more consistent, and some methods or modules have either been added or removed. The semantics of some of the methods have also changed. For the most part, any functionality available in `mimelib` is still available in the `email` package, albeit often in a different way. Backward compatibility between the `mimelib` package and the `email` package was not a priority.

Here is a brief description of the differences between the `mimelib` and the `email` packages, along with hints on how to port your applications.

Of course, the most visible difference between the two packages is that the package name has been changed to `email`. In addition, the top-level package has the following differences:

- `messageFromString()` has been renamed to `message_from_string()`.
- `messageFromFile()` has been renamed to `message_from_file()`.

The `Message` class has the following differences:

- The method `asString()` was renamed to `as_string()`.
- The method `ismultipart()` was renamed to `is_multipart()`.
- The `get_payload()` method has grown a *decode* optional argument.
- The method `getall()` was renamed to `get_all()`.
- The method `addheader()` was renamed to `add_header()`.
- The method `gettype()` was renamed to `get_type()`.
- The method`getmaintype()` was renamed to `get_main_type()`.
- The method `getsubtype()` was renamed to `get_subtype()`.
- The method `getparams()` was renamed to `get_params()`. Also, whereas `getparams()` returned a list of strings, `get_params()` returns a list of 2-tuples, effectively the key/value pairs of the parameters, split on the '`=`' sign.
- The method `getparam()` was renamed to `get_param()`.

- The method `getcharsets()` was renamed to `get_charsets()`.
- The method `getfilename()` was renamed to `get_filename()`.
- The method `getboundary()` was renamed to `get_boundary()`.
- The method `setboundary()` was renamed to `set_boundary()`.
- The method `getdecodedpayload()` was removed. To get similar functionality, pass the value 1 to the *decode* flag of the get_payload() method.
- The method `getpayloadastext()` was removed. Similar functionality is supported by the `DecodedGenerator` class in the `email.Generator` module.
- The method `getbodyastext()` was removed. You can get similar functionality by creating an iterator with `typed_subpart_iterator()` in the `email.Iterators` module.

The `Parser` class has no differences in its public interface. It does have some additional smarts to recognize message/delivery-status type messages, which it represents as a `Message` instance containing separate `Message` subparts for each header block in the delivery status notification[3].

The `Generator` class has no differences in its public interface. There is a new class in the `email.Generator` module though, called `DecodedGenerator` which provides most of the functionality previously available in the `Message.getpayloadastext()` method.

The following modules and classes have been changed:

- The `MIMEBase` class constructor arguments *_major* and *_minor* have changed to *_maintype* and *_subtype* respectively.
- The `Image` class/module has been renamed to `MIMEImage`. The *_minor* argument has been renamed to *_subtype*.
- The `Text` class/module has been renamed to `MIMEText`. The *_minor* argument has been renamed to *_subtype*.
- The `MessageRFC822` class/module has been renamed to `MIMEMessage`. Note that an earlier version of `mimelib` called this class/module `RFC822`, but that clashed with the Python standard library module `rfc822` on some case-insensitive file systems.

  Also, the `MIMEMessage` class now represents any kind of MIME message with main type message. It takes an optional argument *_subtype* which is used to set the MIME subtype. *_subtype* defaults to rfc822.

`mimelib` provided some utility functions in its `address` and `date` modules. All of these functions have been moved to the `email.Utils` module.

The `MsgReader` class/module has been removed. Its functionality is most closely supported in the `body_line_iterator()` function in the `email.Iterators` module.

2.13 Examples

Here are a few examples of how to use the `email` package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let's see how to create and send a simple text message:

[3] Delivery Status Notifications (DSN) are defined in RFC 1894.

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.MIMEText import MIMEText

# Open a plain text file for reading.  For this example, assume that
# the text file contains only ASCII characters.
fp = open(textfile, 'rb')
# Create a text/plain message
msg = MIMEText(fp.read())
fp.close()

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server, but don't include the
# envelope header.
s = smtplib.SMTP()
s.connect()
s.sendmail(me, [you], msg.as_string())
s.close()
```

Here's an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```
# Import smtplib for the actual sending function
import smtplib

# Here are the email pacakge modules we'll need
from email.MIMEImage import MIMEImage
from email.MIMEMultipart import MIMEMultipart

COMMASPACE = ', '

# Create the container (outer) email message.
msg = MIMEMultipart()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = COMMASPACE.join(family)
msg.preamble = 'Our family reunion'
# Guarantees the message ends in a newline
msg.epilogue = ''

# Assume we know that the image files are all in PNG format
for file in pngfiles:
    # Open the files in binary mode.  Let the MIMEImage class automatically
    # guess the specific image type.
    fp = open(file, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

# Send the email via our own SMTP server.
```

```
s = smtplib.SMTP()
s.connect()
s.sendmail(me, family, msg.as_string())
s.close()
```

Here's an example of how to send the entire contents of a directory as an email message: [4]

```
#!/usr/bin/env python

"""Send the contents of a directory as a MIME message.

Usage: dirmail [options] from to [to ...]*

Options:
    -h / --help
        Print this message and exit.

    -d directory
    --directory=directory
        Mail the contents of the specified directory, otherwise use the
        current directory.  Only the regular files in the directory are sent,
        and we don't recurse to subdirectories.

'from' is the email address of the sender of the message.

'to' is the email address of the recipient of the message, and multiple
recipients may be given.

The email is sent by forwarding to your local SMTP server, which then does the
normal delivery process.  Your local machine must be running an SMTP server.
"""

import sys
import os
import getopt
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from email import Encoders
from email.Message import Message
from email.MIMEAudio import MIMEAudio
from email.MIMEBase import MIMEBase
from email.MIMEMultipart import MIMEMultipart
from email.MIMEImage import MIMEImage
from email.MIMEText import MIMEText

COMMASPACE = ', '


def usage(code, msg=''):
    print >> sys.stderr, __doc__
    if msg:
        print >> sys.stderr, msg
    sys.exit(code)
```

[4] Thanks to Matthew Dixon Cowles for the original inspiration and examples.

```
def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'hd:', ['help', 'directory='])
    except getopt.error, msg:
        usage(1, msg)

    dir = os.curdir
    for opt, arg in opts:
        if opt in ('-h', '--help'):
            usage(0)
        elif opt in ('-d', '--directory'):
            dir = arg

    if len(args) < 2:
        usage(1)

    sender = args[0]
    recips = args[1:]

    # Create the enclosing (outer) message
    outer = MIMEMultipart()
    outer['Subject'] = 'Contents of directory %s' % os.path.abspath(dir)
    outer['To'] = COMMASPACE.join(recips)
    outer['From'] = sender
    outer.preamble = 'You will not see this in a MIME-aware mail reader.\n'
    # To guarantee the message ends with a newline
    outer.epilogue = ''

    for filename in os.listdir(dir):
        path = os.path.join(dir, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        if maintype == 'text':
            fp = open(path)
            # Note: we should handle calculating the charset
            msg = MIMEText(fp.read(), _subtype=subtype)
            fp.close()
        elif maintype == 'image':
            fp = open(path, 'rb')
            msg = MIMEImage(fp.read(), _subtype=subtype)
            fp.close()
        elif maintype == 'audio':
            fp = open(path, 'rb')
            msg = MIMEAudio(fp.read(), _subtype=subtype)
            fp.close()
        else:
            fp = open(path, 'rb')
            msg = MIMEBase(maintype, subtype)
            msg.set_payload(fp.read())
            fp.close()
```

```
            # Encode the payload using Base64
            Encoders.encode_base64(msg)
        # Set the filename parameter
        msg.add_header('Content-Disposition', 'attachment', filename=filename)
        outer.attach(msg)

    # Now send the message
    s = smtplib.SMTP()
    s.connect()
    s.sendmail(sender, recips, outer.as_string())
    s.close()


if __name__ == '__main__':
    main()
```

And finally, here's an example of how to unpack a MIME message like the one above, into a directory of files:

```
#!/usr/bin/env python

"""Unpack a MIME message into a directory of files.

Usage: unpackmail [options] msgfile

Options:
    -h / --help
        Print this message and exit.

    -d directory
    --directory=directory
        Unpack the MIME message into the named directory, which will be
        created if it doesn't already exist.

msgfile is the path to the file containing the MIME message.
"""

import sys
import os
import getopt
import errno
import mimetypes
import email


def usage(code, msg=''):
    print >> sys.stderr, __doc__
    if msg:
        print >> sys.stderr, msg
    sys.exit(code)


def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'hd:', ['help', 'directory='])
    except getopt.error, msg:
        usage(1, msg)

    dir = os.curdir
    for opt, arg in opts:
```

```
            if opt in ('-h', '--help'):
                usage(0)
            elif opt in ('-d', '--directory'):
                dir = arg

        try:
            msgfile = args[0]
        except IndexError:
            usage(1)

        try:
            os.mkdir(dir)
        except OSError, e:
            # Ignore directory exists error
            if e.errno <> errno.EEXIST: raise

        fp = open(msgfile)
        msg = email.message_from_file(fp)
        fp.close()

        counter = 1
        for part in msg.walk():
            # multipart/* are just containers
            if part.get_content_maintype() == 'multipart':
                continue
            # Applications should really sanitize the given filename so that an
            # email message can't be used to overwrite important files
            filename = part.get_filename()
            if not filename:
                ext = mimetypes.guess_extension(part.get_type())
                if not ext:
                    # Use a generic bag-of-bits extension
                    ext = '.bin'
                filename = 'part-%03d%s' % (counter, ext)
            counter += 1
            fp = open(os.path.join(dir, filename), 'wb')
            fp.write(part.get_payload(decode=1))
            fp.close()


    if __name__ == '__main__':
        main()
```