

19.1.9. `email.message.Message`: Representing an email message using the `compat32` API

The `Message` class is very similar to the `EmailMessage` class, without the methods added by that class, and with the default behavior of certain other methods being slightly different. We also document here some methods that, while supported by the `EmailMessage` class, are not recommended unless you are dealing with legacy code.

The philosophy and structure of the two classes is otherwise the same.

This document describes the behavior under the default (for `Message`) policy `Compat32`. If you are going to use another policy, you should be using the `EmailMessage` class instead.

An email message consists of *headers* and a *payload*. Headers must be [RFC 5233](#) style 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. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

The conceptual model provided by a `Message` object is that of an ordered dictionary of headers with additional methods for accessing both specialized information from the headers, for accessing the payload, for generating a serialized version of the message, and for recursively walking over the object tree. Note that duplicate headers are supported but special methods must be used to access them.

The `Message` pseudo-dictionary is indexed by the header names, which must be ASCII values. The values of the dictionary are strings that are supposed to contain only ASCII characters; there is some special handling for non-ASCII input, but it doesn't always produce the correct results. Headers are stored and returned in case-preserving form, but field names 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 or bytes, in the case of simple message objects, or a list of `Message` objects, for MIME container documents (e.g. *multipart/** and *message/rfc822*).

Here are the methods of the `Message` class:

`class email.message.Message(policy=compat32)`

If *policy* is specified (it must be an instance of a [policy](#) class) use the rules it specifies to update and serialize 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. For more information see the [policy](#) documentation.

Changed in version 3.3: The *policy* keyword argument was added.

`as_string(unixfrom=False, maxheaderlen=0, policy=None)`

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to False. For backward compatibility reasons, *maxheaderlen* defaults to 0, so if you want a different value you must override it explicitly (the value specified for *max_line_length* in the policy will be ignored by this method). The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the Generator.

Flattening the message may trigger changes to the [Message](#) if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with From that is required by the unix mbox format. For more flexibility, instantiate a [Generator](#) instance and use its [flatten\(\)](#) method directly. For example:

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

If the message object contains binary data that is not encoded according to RFC standards, the non-compliant data will be replaced by unicode “unknown character” code points. (See also [as_bytes\(\)](#) and [BytesGenerator](#).)

Changed in version 3.4: the *policy* keyword argument was added.

`__str__()`

Equivalent to `as_string()`. Allows `str(msg)` to produce a string containing the formatted message.

as_bytes(*unixfrom=False, policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to False. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the BytesGenerator.

Flattening the message may trigger changes to the `Message` if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with From that is required by the unix mbox format. For more flexibility, instantiate a `BytesGenerator` instance and use its `flatten()` method directly. For example:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

New in version 3.4.

__bytes__()

Equivalent to `as_bytes()`. Allows `bytes(msg)` to produce a bytes object containing the formatted message.

New in version 3.4.

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 (which might be a CTE encoded binary payload. (Note that `is_multipart()` returning True does not necessarily mean that `"msg.get_content_maintype() == 'multipart'"` will return the True. For example, `is_multipart` will return True when the `Message` is of type `message/rfc822`.)

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.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by [set_content\(\)](#) and the related `make` and `add` methods.

get_payload(*i=None*, *decode=False*)

Return 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, the payload is returned as-is (undecoded). In all cases the returned value is binary data. If the message is a multipart and the *decode* flag is True, then None is returned. If the payload is base64 and it was not perfectly formed (missing padding, characters outside the base64 alphabet), then an appropriate defect will be added to the message's defect property (`InvalidBase64PaddingDefect` or `InvalidBase64CharactersDefect`, respectively).

When *decode* is False (the default) the body is returned as a string without decoding the *Content-Transfer-Encoding*. However, for a *Content-Transfer-Encoding* of 8bit, an attempt is made to decode the original bytes using the charset specified by the *Content-Type* header, using the `replace` error

handler. If no charset is specified, or if the charset given is not recognized by the email package, the body is decoded using the default ASCII charset.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by `get_content()` and `iter_parts()`.

set_payload(payload, charset=None)

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.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by `set_content()`.

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 (the message will not be otherwise modified). Anything else will generate a `TypeError`.

If there is no existing *MIME-Version* header one will be added. If there is no existing *Content-Type* header, one will be added with a value of *text/plain*. Whether the *Content-Type* header already exists or not, its charset parameter will be set to *charset.output_charset*. If *charset.input_charset* and *charset.output_charset* differ, the payload will be re-encoded to the *output_charset*. If there is no existing *Content-Transfer-Encoding* header, then the payload will be transfer-encoded, if needed, using the specified `Charset`, and a header with the appropriate value will be added. If a *Content-Transfer-Encoding* header already exists, the payload is assumed to already be correctly encoded using that *Content-Transfer-Encoding* and is not modified.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *charset* parameter of the `email.message.EmailMessage.set_content()` method.

get_charset()

Return the `Charset` instance associated with the message's payload.

This is a legacy method. On the `EmailMessage` class it always returns `None`.

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.

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as `Header` objects with a charset of *unknown-8bit*.

`__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.

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=None)

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=None)

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. If the value contains non-ASCII characters, it can be specified as a

three tuple in the format (CHARSET, LANGUAGE, VALUE), where CHARSET is a string naming the charset to be used to encode the value, LANGUAGE can usually be set to None or the empty string (see [RFC 2231](#) for other possibilities), and VALUE is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in [RFC 2231](#) format using a CHARSET of utf-8 and a LANGUAGE of None.

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"
```

An example with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',  
              filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename*="iso-8859-1'"Fu%DFba
```

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.

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*.

get_content_maintype()

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

get_content_subtype()

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

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*.

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.

get_params(failobj=None, header='content-type', unquote=True)

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*.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

get_param(param, failobj=None, header='content-type', unquote=True)

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`.

If your application doesn't care whether the parameter was encoded as in [RFC 2231](#), you can collapse the parameter value by calling `email.utils.collapse_rfc2231_value()`, passing in the return value from `get_param()`. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn't. For example:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

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`.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

set_param(*param*, *value*, *header*='Content-Type', *requote*=True, *charset*=None, *language*="", *replace*=False)

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 has 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.

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

Changed in version 3.4: *replace* keyword was added.

del_param(*param*, *header*='content-type', *requote*=True)

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*.

set_type(*type*, *header*='Content-Type', *requote*=True)

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.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `make_` and `add_` methods.

get_filename(*failobj*=None)

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 on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per [email.utils.unquote\(\)](#).

get_boundary(*failobj*=None)

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 [email.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*=None)

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.

get_charsets(*failobj*=None)

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*.

get_content_disposition()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or None. The possible values for this method are *inline*, *attachment* or None if the message follows [RFC 2183](#).

New in version 3.5.

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
text/plain
```

`walk` iterates over the subparts of any part where `is_multipart()` returns True, even though `msg.get_content_maintype() == 'multipart'` may

return False. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart'),
...           part.is_multipart())
True True
False False
False True
False False
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
message/delivery-status
    text/plain
    text/plain
message/rfc822
    text/plain
```

Here the message parts are not multipart, but they do contain subparts. `is_multipart()` returns True and `walk` descends into the subparts.

[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.

You do not need to set the epilogue to the empty string in order for the [Generator](#) to print a newline at the end of the file.

defects

The *defects* attribute contains a list of all the problems found when parsing this message. See [email.errors](#) for a detailed description of the possible parsing defects.