19.1.4. `email.policy`: Policy Objects

New in version 3.3.

Source code: Lib/email/policy.py

The `email` package's prime focus is the handling of email messages as described by the various email and MIME RFCs. However, the general format of email messages (a block of header fields each consisting of a name followed by a colon followed by a value, the whole block followed by a blank line and an arbitrary 'body'), is a format that has found utility outside of the realm of email. Some of these uses conform fairly closely to the main email RFCs, some do not. Even when working with email, there are times when it is desirable to break strict compliance with the RFCs, such as generating emails that interoperate with email servers that do not themselves follow the standards, or that implement extensions you want to use in ways that violate the standards.

Policy objects give the email package the flexibility to handle all these disparate use cases.

A `Policy` object encapsulates a set of attributes and methods that control the behavior of various components of the email package during use. `Policy` instances can be passed to various classes and methods in the email package to alter the default behavior. The settable values and their defaults are described below.

There is a default policy used by all classes in the email package. For all of the `parser` classes and the related convenience functions, and for the `Message` class, this is the `Compat32` policy, via its corresponding pre-defined instance `compat32`. This policy provides for complete backward compatibility (in some cases, including bug compatibility) with the pre-Python3.3 version of the email package.

This default value for the *policy* keyword to `EmailMessage` is the `EmailPolicy` policy, via its pre-defined instance `default`.

When a `Message` or `EmailMessage` object is created, it acquires a policy. If the message is created by a `parser`, a policy passed to the parser will be the policy used by the message it creates. If the message is created by the program, then the policy can be specified when it is created. When a message is passed to a `generator`, the generator uses the policy from the message by default, but you can also pass a specific policy to the generator that will override the one stored on the message object.

The default value for the *policy* keyword for the `email.parser` classes and the parser convenience functions **will be changing** in a future version of Python. Therefore you should **always specify explicitly which policy you want to use** when calling any of the classes and functions described in the `parser` module.

The first part of this documentation covers the features of `Policy`, an abstract base class that defines the features that are common to all policy objects, including `compat32`. This includes certain hook methods that are called internally by the email package, which a custom policy could override to obtain different behavior. The second part describes the concrete classes `EmailPolicy` and `Compat32`, which implement the hooks that provide the standard behavior and the backward compatible behavior and features, respectively.

`Policy` instances are immutable, but they can be cloned, accepting the same keyword arguments as the class constructor and returning a new `Policy` instance that is a copy of the original but with the specified attributes values changed.

As an example, the following code could be used to read an email message from a file on disk and pass it to the system `sendmail` program on a Unix system:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Here we are telling `BytesGenerator` to use the RFC correct line separator characters when creating the binary string to feed into `sendmail's stdin`, where the default policy would use `\n` line separators.

Some email package methods accept a *policy* keyword argument, allowing the policy to be overridden for that method. For example, the following code uses the `as_bytes()` method of the *msg* object from the previous example and writes the message to a file using the native line separators for the platform on which it is running:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linese
17
```

Policy objects can also be combined using the addition operator, producing a policy object whose settings are a combination of the non-default values of the summed objects:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

This operation is not commutative; that is, the order in which the objects are added matters. To illustrate:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class `email.policy.`**`Policy`**(***kw*)

This is the abstract base class for all policy classes. It provides default implementations for a couple of trivial methods, as well as the implementation of the immutability property, the `clone()` method, and the constructor semantics.

The constructor of a policy class can be passed various keyword arguments. The arguments that may be specified are any non-method properties on this class, plus any additional non-method properties on the concrete class. A value specified in the constructor will override the default value for the corresponding attribute.

This class defines the following properties, and thus values for the following may be passed in the constructor of any policy class:

`max_line_length`

The maximum length of any line in the serialized output, not counting the end of line character(s). Default is 78, per **RFC 5322**. A value of `0` or `None` indicates that no line wrapping should be done at all.

`linesep`

The string to be used to terminate lines in serialized output. The default is `\n` because that's the internal end-of-line discipline used by Python, though `\r\n` is required by the RFCs.

`cte_type`

Controls the type of Content Transfer Encodings that may be or are required to be used. The possible values are:

`7bit`	all data must be “7 bit clean” (ASCII-only). This means that where necessary data will be encoded using either quoted-printable or base64 encoding.
`8bit`	data is not constrained to be 7 bit clean. Data in headers is still required to be ASCII-only and so will be encoded (see `fold_binary()` and `utf8` below for exceptions), but body parts may use the `8bit` CTE.

A `cte_type` value of `8bit` only works with `BytesGenerator`, not `Generator`, because strings cannot contain binary data. If a `Generator` is operating under a policy that specifies `cte_type=8bit`, it will act as if `cte_type` is `7bit`.

`raise_on_defect`

If `True`, any defects encountered will be raised as errors. If `False` (the default), defects will be passed to the `register_defect()` method.

`mangle_from_`

If `True`, lines starting with *“From “* in the body are escaped by putting a `>` in front of them. This parameter is used when the message is being serialized by a generator. Default: `False`.

New in version 3.5: The *mangle_from_* parameter.

`message_factory`

A factory function for constructing a new empty message object. Used by the parser when building messages. Defaults to `None`, in which case `Message` is used.

New in version 3.6.

The following `Policy` method is intended to be called by code using the email library to create policy instances with custom settings:

`clone`(***kw*)

Return a new `Policy` instance whose attributes have the same values as the current instance, except where those attributes are given new values by the keyword arguments.

The remaining `Policy` methods are called by the email package code, and are not intended to be called by an application using the email package. A custom policy must implement all of these methods.

handle_defect(*obj*, *defect*)

Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of `Defect`.

The default implementation checks the `raise_on_defect` flag. If it is `True`, *defect* is raised as an exception. If it is `False` (the default), *obj* and *defect* are passed to `register_defect()`.

register_defect(*obj*, *defect*)

Register a *defect* on *obj*. In the email package, *defect* will always be a subclass of `Defect`.

The default implementation calls the `append` method of the `defects` attribute of *obj*. When the email package calls `handle_defect`, *obj* will normally have a `defects` attribute that has an `append` method. Custom object types used with the email package (for example, custom `Message` objects) should also provide such an attribute, otherwise defects in parsed messages will raise unexpected errors.

header_max_count(*name*)

Return the maximum allowed number of headers named *name*.

Called when a header is added to an `EmailMessage` or `Message` object. If the returned value is not `0` or `None`, and there are already a number of headers with the name *name* greater than or equal to the value returned, a `ValueError` is raised.

Because the default behavior of `Message.__setitem__` is to append the value to the list of headers, it is easy to create duplicate headers without realizing it. This method allows certain headers to be limited in the number of instances of that header that may be added to a `Message` programmatically. (The limit is not observed by the parser, which will faithfully produce as many headers as exist in the message being parsed.)

The default implementation returns `None` for all header names.

header_source_parse(*sourcelines*)

The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace

in the source is preserved. The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the parsed header.

If an implementation wishes to retain compatibility with the existing email package policies, *name* should be the case preserved name (all characters up to the '`:`' separator), while *value* should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.

sourcelines may contain surrogateescaped binary data.

There is no default implementation

`header_store_parse`(*name*, *value*)

The email package calls this method with the name and value provided by the application program when the application program is modifying a `Message` programmatically (as opposed to a `Message` created by a parser). The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the header.

If an implementation wishes to retain compatibility with the existing email package policies, the *name* and *value* should be strings or string subclasses that do not change the content of the passed in arguments.

There is no default implementation

`header_fetch_parse`(*name*, *value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` when that header is requested by the application program, and whatever the method returns is what is passed back to the application as the value of the header being retrieved. Note that there may be more than one header with the same name stored in the `Message`; the method is passed the specific name and value of the header destined to be returned to the application.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the value returned by the method.

There is no default implementation

`fold`(*name*, *value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` for a given header. The method should return a string that represents that header "folded" correctly (according to the policy settings) by composing the *name* with the *value* and inserting `linesep` charac-

ters at the appropriate places. See **RFC 5322** for a discussion of the rules for folding email headers.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the string returned by the method.

fold_binary(*name*, *value*)

The same as `fold()`, except that the returned value should be a bytes object rather than a string.

value may contain surrogateescaped binary data. These could be converted back into binary data in the returned bytes object.

class `email.policy.`**EmailPolicy**(***kw*)

This concrete `Policy` provides behavior that is intended to be fully compliant with the current email RFCs. These include (but are not limited to) **RFC 5322**, **RFC 2047**, and the current MIME RFCs.

This policy adds new header parsing and folding algorithms. Instead of simple strings, headers are `str` subclasses with attributes that depend on the type of the field. The parsing and folding algorithm fully implement **RFC 2047** and **RFC 5322**.

The default value for the `message_factory` attribute is `EmailMessage`.

In addition to the settable attributes listed above that apply to all policies, this policy adds the following additional attributes:

New in version 3.6: [1]

utf8

If `False`, follow **RFC 5322**, supporting non-ASCII characters in headers by encoding them as "encoded words". If `True`, follow **RFC 6532** and use `utf-8` encoding for headers. Messages formatted in this way may be passed to SMTP servers that support the `SMTPUTF8` extension (**RFC 6531**).

refold_source

If the value for a header in the `Message` object originated from a `parser` (as opposed to being set by a program), this attribute indicates whether or not a generator should refold that value when transforming the message back into serialized form. The possible values are:

`none`	all source values use original folding
`long`	source values that have any line that is longer than `max_line_length` will be refolded

`all`	all values are refolded.

The default is `long`.

`header_factory`

A callable that takes two arguments, `name` and `value`, where `name` is a header field name and `value` is an unfolded header field value, and returns a string subclass that represents that header. A default `header_factory` (see `headerregistry`) is provided that supports custom parsing for the various address and date **RFC 5322** header field types, and the major MIME header field stypes. Support for additional custom parsing will be added in the future.

`content_manager`

An object with at least two methods: get_content and set_content. When the `get_content()` or `set_content()` method of an `EmailMessage` object is called, it calls the corresponding method of this object, passing it the message object as its first argument, and any arguments or keywords that were passed to it as additional arguments. By default `content_manager` is set to `raw_data_manager`.

New in version 3.4.

The class provides the following concrete implementations of the abstract methods of `Policy`:

`header_max_count`(*name*)

Returns the value of the `max_count` attribute of the specialized class used to represent the header with the given name.

`header_source_parse`(*sourcelines*)

The name is parsed as everything up to the '`:`' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

`header_store_parse`(*name*, *value*)

The name is returned unchanged. If the input value has a `name` attribute and it matches *name* ignoring case, the value is returned unchanged. Otherwise the *name* and *value* are passed to `header_factory`, and the resulting header object is returned as the value. In this case a `ValueError` is raised if the input value contains CR or LF characters.

`header_fetch_parse`(*name*, *value*)

If the value has a `name` attribute, it is returned to unmodified. Otherwise the *name*, and the *value* with any CR or LF characters removed, are passed to the `header_factory`, and the resulting header object is returned. Any surrogateescaped bytes get turned into the unicode unknown-character glyph.

fold(*name*, *value*)

Header folding is controlled by the `refold_source` policy setting. A value is considered to be a 'source value' if and only if it does not have a `name` attribute (having a `name` attribute means it is a header object of some sort). If a source value needs to be refolded according to the policy, it is converted into a header object by passing the *name* and the *value* with any CR and LF characters removed to the `header_factory`. Folding of a header object is done by calling its `fold` method with the current policy.

Source values are split into lines using `splitlines()`. If the value is not to be refolded, the lines are rejoined using the `linesep` from the policy and returned. The exception is lines containing non-ascii binary data. In that case the value is refolded regardless of the `refold_source` setting, which causes the binary data to be CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)

The same as `fold()` if `cte_type` is `7bit`, except that the returned value is bytes.

If `cte_type` is `8bit`, non-ASCII binary data is converted back into bytes. Headers with binary data are not refolded, regardless of the `refold_header` setting, since there is no way to know whether the binary data consists of single byte characters or multibyte characters.

The following instances of `EmailPolicy` provide defaults suitable for specific application domains. Note that in the future the behavior of these instances (in particular the `HTTP` instance) may be adjusted to conform even more closely to the RFCs relevant to their domains.

`email.policy.`**default**

An instance of `EmailPolicy` with all defaults unchanged. This policy uses the standard Python `\n` line endings rather than the RFC-correct `\r\n`.

`email.policy.`**SMTP**

Suitable for serializing messages in conformance with the email RFCs. Like `default`, but with `linesep` set to `\r\n`, which is RFC compliant.

`email.policy.`**SMTPUTF8**

The same as `SMTP` except that `utf8` is `True`. Useful for serializing messages to a message store without using encoded words in the headers. Should only be used for SMTP transmission if the sender or recipient addresses have non-ASCII characters (the `smtplib.SMTP.send_message()` method handles this automatically).

`email.policy.`**`HTTP`**

Suitable for serializing headers with for use in HTTP traffic. Like `SMTP` except that `max_line_length` is set to `None` (unlimited).

`email.policy.`**`strict`**

Convenience instance. The same as `default` except that `raise_on_defect` is set to `True`. This allows any policy to be made strict by writing:

```
somepolicy + policy.strict
```

With all of these `EmailPolicies`, the effective API of the email package is changed from the Python 3.2 API in the following ways:

- Setting a header on a `Message` results in that header being parsed and a header object created.
- Fetching a header value from a `Message` results in that header being parsed and a header object created and returned.
- Any header object, or any header that is refolded due to the policy settings, is folded using an algorithm that fully implements the RFC folding algorithms, including knowing where encoded words are required and allowed.

From the application view, this means that any header obtained through the `EmailMessage` is a header object with extra attributes, whose string value is the fully decoded unicode value of the header. Likewise, a header may be assigned a new value, or a new header created, using a unicode string, and the policy will take care of converting the unicode string into the correct RFC encoded form.

The header objects and their attributes are described in `headerregistry`.

class `email.policy.`**`Compat32`**(***kw*)

This concrete `Policy` is the backward compatibility policy. It replicates the behavior of the email package in Python 3.2. The `policy` module also defines an instance of this class, `compat32`, that is used as the default policy. Thus the default behavior of the email package is to maintain compatibility with Python 3.2.

The following attributes have values that are different from the `Policy` default:

`mangle_from_`

The default is `True`.

The class provides the following concrete implementations of the abstract methods of `Policy`:

`header_source_parse`(*sourcelines*)

The name is parsed as everything up to the ':' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

`header_store_parse`(*name*, *value*)

The name and value are returned unmodified.

`header_fetch_parse`(*name*, *value*)

If the value contains binary data, it is converted into a `Header` object using the `unknown-8bit` charset. Otherwise it is returned unmodified.

`fold`(*name*, *value*)

Headers are folded using the `Header` folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. Non-ASCII binary data are CTE encoded using the `unknown-8bit` charset.

`fold_binary`(*name*, *value*)

Headers are folded using the `Header` folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. If `cte_type` is `7bit`, non-ascii binary data is CTE encoded using the `unknown-8bit` charset. Otherwise the original source header is used, with its existing line breaks and any (RFC invalid) binary data it may contain.

`email.policy.`**`compat32`**

An instance of `Compat32`, providing backward compatibility with the behavior of the email package in Python 3.2.

Footnotes

[1] Originally added in 3.3 as a provisional feature.