19.1.11. `email.header`: Internationalized headers

Source code: Lib/email/header.py

This module is part of the legacy (`Compat32`) email API. In the current API encoding and decoding of headers is handled transparently by the dictionary-like API of the `EmailMessage` class. In addition to uses in legacy code, this module can be useful in applications that need to completely control the character sets used when encoding headers.

The remaining text in this section is the original documentation of the module.

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 transferred 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. Import the `Header` class from the `email.header` module. For example:

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

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.

Here is the `Header` class description:

class `email.header.`**Header**(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

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 an instance of `bytes` or `str`, 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 explicitly 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. *continuation_ws* defaults to a single space character.

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

append(*s*, *charset=None*, *errors='strict'*)

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 an instance of `bytes` or `str`. If it is an instance of `bytes`, 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 an instance of `str`, then *charset* is a hint specifying the character set of the characters in the string.

In either case, when producing an **RFC 2822**-compliant header using **RFC 2047** rules, the string will be encoded using the output codec of the charset. If the string cannot be encoded using the output codec, a UnicodeError will be raised.

Optional *errors* is passed as the errors argument to the decode call if *s* is a byte string.

encode(*splitchars=';\t'*, *maxlinelen=None*, *linesep='\n'*)

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 which should be given extra weight by the splitting algorithm during normal header wrapping. This is in very rough support of **RFC 2822**'s 'higher level syntactic breaks': split points preceded by a splitchar are preferred during line splitting, with the characters preferred in the order in which they appear in the string. Space and tab may be included in the string to indicate whether preference should be given to one over the other as a split point when other split chars do not appear in the line being split. Splitchars does not affect **RFC 2047** encoded lines.

maxlinelen, if given, overrides the instance's value for the maximum line length.

linesep specifies the characters used to separate the lines of the folded header. It defaults to the most useful value for Python application code (`\n`), but `\r\n` can be specified in order to produce headers with RFC-compliant line separators.

Changed in version 3.2: Added the *linesep* argument.

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

__str__()

Returns an approximation of the `Header` as a string, using an unlimited line length. All pieces are converted to unicode using the specified encoding and joined together appropriately. Any pieces with a charset of `'unknown-8bit'` are decoded as ASCII using the `'replace'` error handler.

Changed in version 3.2: Added handling for the `'unknown-8bit'` charset.

__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.

`email.header.`**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?=')
[(b'p\xf6stal', 'iso-8859-1')]
```

`email.header.`**make_header**(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *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.