`!email.headerregistry`: Custom Header Objects

email.headerregistry

Source code: `Lib/email/headerregistry.py`

3.6[^1]

Headers are represented by customized subclasses of `str`. The particular class used to represent a given header is determined by the `~email.policy.EmailPolicy.header_factory` of the `~email.policy` in effect when the headers are created. This section documents the particular `header_factory` implemented by the email package for handling RFC 5322 compliant email messages, which not only provides customized header objects for various header types, but also provides an extension mechanism for applications to add their own custom header types.

When using any of the policy objects derived from `~email.policy.EmailPolicy`, all headers are produced by `.HeaderRegistry` and have `.BaseHeader` as their last base class. Each header class has an additional base class that is determined by the type of the header. For example, many headers have the class `.UnstructuredHeader` as their other base class. The specialized second class for a header is determined by the name of the header, using a lookup table stored in the `.HeaderRegistry`. All of this is managed transparently for the typical application program, but interfaces are provided for modifying the default behavior for use by more complex applications.

The sections below first document the header base classes and their attributes, followed by the API for modifying the behavior of `.HeaderRegistry`, and finally the support classes used to represent the data parsed from structured headers.

name and *value* are passed to `BaseHeader` from the `~email.policy.EmailPolicy.header_factory` call. The string value of any header object is the *value* fully decoded to a string.

This base class defines the following read-only properties:

name

The name of the header (the portion of the field before the ':'). This is exactly the value passed in the `~email.policy.EmailPolicy.header_factory` call for *name*; that is, case is preserved.

defects

A tuple of `~email.errors.HeaderDefect` instances reporting any RFC compliance problems found during parsing. The email package tries to be complete about detecting compliance issues. See the `~email.errors` module for a discussion of the types of defects that may be reported.

max_count

The maximum number of headers of this type that can have the same `name`. A value of `None` means unlimited. The `BaseHeader` value for this attribute is `None`; it is expected that specialized header classes will override this value as needed.

`BaseHeader` also provides the following method, which is called by the email library code and should not in general be called by application programs:

fold(*, policy)

Return a string containing `~email.policy.Policy.linesep` characters as required to correctly fold the header according to *policy*. A `~email.policy.Policy.cte_type` of `8bit` will be treated as if it were `7bit`, since headers may not contain arbitrary binary data. If `~email.policy.EmailPolicy.utf8` is `False`, non-ASCII data will be `2047` encoded.

`BaseHeader` by itself cannot be used to create a header object. It defines a protocol that each specialized header cooperates with in order to produce the header object. Specifically, `BaseHeader` requires that the specialized class provide a `classmethod` named `parse`. This method is called as follows:

```
parse(string, kwds)
```

`kwds` is a dictionary containing one pre-initialized key, `defects`. `defects` is an empty list. The parse method should append any detected defects to this list. On return, the `kwds` dictionary *must* contain values for at least the keys `decoded`, `defects` and `parse_tree`. `decoded` should be the string value for the header (that is, the header value fully decoded to a string). `parse_tree` is set to the parse tree obtained from parsing the header. The parse method should assume that *string* may contain content-transfer-encoded parts, but should correctly handle all valid Unicode characters as well so that it can parse un-encoded header values.

`BaseHeader`'s `__new__` then creates the header instance, and calls its `init` method. The specialized class only needs to provide an `init` method if it wishes to set additional attributes beyond those provided by `BaseHeader` itself. Such an `init` method should look like this:

```
def init(self, /, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

That is, anything extra that the specialized class puts in to the `kwds` dictionary should be removed and handled, and the remaining contents of `kw` (and `args`) passed to the `BaseHeader` `init` method.

An "unstructured" header is the default type of header in `5322`. Any header that does not have a specified syntax is treated as unstructured. The classic example of an unstructured header is the `Subject` header.

In `5322`, an unstructured header is a run of arbitrary text in the ASCII character set. `2047`, however, has an `5322` compatible mechanism for encoding non-ASCII text as ASCII characters within a header value. When a *value* containing encoded words is passed to the constructor, the `UnstructuredHeader` parser converts such encoded words into a string, following the `2047` rules for unstructured text. The parser uses heuristics to attempt to decode certain non-compliant encoded words. Defects are registered in such cases, as well as defects for issues such as invalid characters within the encoded words or the non-encoded text.

This header type provides no additional attributes.

`5322` specifies a very specific format for dates within email headers. The `DateHeader` parser recognizes that date format, as well as recognizing a number of variant forms that are sometimes found "in the wild".

This header type provides the following additional attributes:

datetime

If the header value can be recognized as a valid date of one form or another, this attribute will contain a `~datetime.datetime` instance representing that date. If the timezone of the input date is specified as `-0000` (indicating it is in UTC but contains no information about the source timezone), then `.datetime` will be a naive `~datetime.datetime`. If a specific timezone offset is found (including `+0000`), then `.datetime` will contain an aware `datetime` that uses `datetime.timezone` to record the timezone offset.

The `decoded` value of the header is determined by formatting the `datetime` according to the `5322` rules; that is, it is set to:

```
email.utils.format_datetime(self.datetime)
```

When creating a `DateHeader`, *value* may be `~datetime.datetime` instance. This means, for example, that the following code is valid and does what one would expect:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

Because this is a naive `datetime` it will be interpreted as a UTC timestamp, and the resulting value will have a timezone of `-0000`. Much more useful is to use the `~email.utils.localtime` function from the `~email.utils` module:

```
msg['Date'] = utils.localtime()
```

This example sets the date header to the current time and date using the current timezone offset.

Address headers are one of the most complex structured header types. The `AddressHeader` class provides a generic interface to any address header.

This header type provides the following additional attributes:

groups

A tuple of `.Group` objects encoding the addresses and groups found in the header value. Addresses that are not part of a group are represented in this list as single-address `Groups` whose `~.Group.display_name` is `None`.

addresses

A tuple of `.Address` objects encoding all of the individual addresses from the header value. If the header value contains any groups, the individual addresses from the group are included in the list at the point where the group occurs in the value (that is, the list of addresses is "flattened" into a one dimensional list).

The `decoded` value of the header will have all encoded words decoded to a string. `~encodings.idna` encoded domain names are also decoded to a string. The `decoded` value is set by `joining <meth-str-join>` the `str` value of the elements of the `groups` attribute with `', '`.

A list of `.Address` and `.Group` objects in any combination may be used to set the value of an address header. `Group` objects whose `display_name` is `None` will be interpreted as single addresses, which allows an address list to be copied with groups intact by using the list obtained from the `groups` attribute of the source header.

A subclass of `.AddressHeader` that adds one additional attribute:

address

The single address encoded by the header value. If the header value actually contains more than one address (which would be a violation of the RFC under the default `~email.policy`), accessing this attribute will result in a `ValueError`.

Many of the above classes also have a `Unique` variant (for example, `UniqueUnstructuredHeader`). The only difference is that in the `Unique` variant, `~.BaseHeader.max_count` is set to 1.

There is really only one valid value for the `MIME-Version` header, and that is `1.0`. For future proofing, this header class supports other valid version numbers. If a version number has a valid value per `2045`, then the header object will have non-`None` values for the following attributes:

version

The version number as a string, with any whitespace and/or comments removed.

major

The major version number as an integer

minor

The minor version number as an integer

MIME headers all start with the prefix 'Content-'. Each specific header has a certain value, described under the class for that header. Some can also take a list of supplemental parameters, which have a common format. This class serves as a base for all the MIME headers that take parameters.

params

A dictionary mapping parameter names to parameter values.

3.15 It is now a `frozendict` instead of a `types.MappingProxyType`.

A `ParameterizedMIMEHeader` class that handles the `Content-Type` header.

content_type

The content type string, in the form `maintype/subtype`.

maintype

subtype

A `ParameterizedMIMEHeader` class that handles the `Content-Disposition` header.

content_disposition

`inline` and `attachment` are the only valid values in common use.

Handles the `Content-Transfer-Encoding` header.

cte

Valid values are `7bit`, `8bit`, `base64`, and `quoted-printable`. See `2045` for more information.

This is the factory used by `~email.policy.EmailPolicy` by default. `HeaderRegistry` builds the class used to create a header instance dynamically, using *base_class* and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by *default_class* is used as the specialized class. When *use_default_map* is `True` (the default), the standard mapping of header names to classes is copied in to the registry during initialization. *base_class* is always the last class in the generated class's `~type.__bases__` list.

The default mappings are:

> subject
> UniqueUnstructuredHeader
>
> date
> UniqueDateHeader
>
> resent-date
> DateHeader
>
> orig-date
> UniqueDateHeader
>
> sender
> UniqueSingleAddressHeader
>
> resent-sender
> SingleAddressHeader
>
> to
> UniqueAddressHeader
>
> resent-to
> AddressHeader
>
> cc
> UniqueAddressHeader
>
> resent-cc
> AddressHeader

> bcc
> UniqueAddressHeader
>
> resent-bcc
> AddressHeader
>
> from
> UniqueAddressHeader
>
> resent-from
> AddressHeader
>
> reply-to
> UniqueAddressHeader
>
> mime-version
> MIMEVersionHeader
>
> content-type
> ContentTypeHeader
>
> content-disposition
> ContentDispositionHeader
>
> content-transfer-encoding
> ContentTransferEncodingHeader
>
> message-id
> MessageIDHeader

`HeaderRegistry` has the following methods:

map_to_type(self, name, cls)

name is the name of the header to be mapped. It will be converted to lower case in the registry. *cls* is the specialized class to be used, along with *base_class*, to create the class used to instantiate headers that match *name*.

__getitem__(name)

Construct and return a class to handle creating a *name* header.

__call__(name, value)

Retrieves the specialized header associated with *name* from the registry (using *default_class* if *name* does not appear in the registry) and composes it with *base_class* to produce a class, calls the constructed class's constructor, passing it the same argument list, and finally returns the class instance created thereby.

The following classes are the classes used to represent data parsed from structured headers and can, in general, be used by an application program to construct structured values to assign to specific headers.

The class used to represent an email address. The general form of an address is:

```
[display_name] <username@domain>
```

or:

```
username@domain
```

where each part must conform to specific syntax rules spelled out in `5322`.

As a convenience *addr_spec* can be specified instead of *username* and *domain*, in which case *username* and *domain* will be parsed from the *addr_spec*. An *addr_spec* must be a properly RFC quoted string; if it is not `Address` will raise an error. Unicode characters are allowed and will be property encoded when serialized. However, per the RFCs, Unicode is *not* allowed in the username portion of the address.

display_name

The display name portion of the address, if any, with all quoting removed. If the address does not have a display name, this attribute will be an empty string.

username

The `username` portion of the address, with all quoting removed.

domain

The `domain` portion of the address.

addr_spec

The `username@domain` portion of the address, correctly quoted for use as a bare address (the second form shown above). This attribute is not mutable.

__str__()

The `str` value of the object is the address quoted according to `5322` rules, but with no Content Transfer Encoding of any non-ASCII characters.

To support SMTP (`5321`), `Address` handles one special case: if `username` and `domain` are both the empty string (or `None`), then the string value of the `Address` is `<>`.

The class used to represent an address group. The general form of an address group is:

```
display_name: [address-list];
```

As a convenience for processing lists of addresses that consist of a mixture of groups and single addresses, a `Group` may also be used to represent single addresses that are not part of a group by setting *display_name* to `None` and providing a list of the single address as *addresses*.

display_name

The `display_name` of the group. If it is `None` and there is exactly one `Address` in `addresses`, then the `Group` represents a single address that is not in a group.

addresses

A possibly empty tuple of `.Address` objects representing the addresses in the group.

__str__()

The `str` value of a `Group` is formatted according to `5322`, but with no Content Transfer Encoding of any non-ASCII characters. If `display_name` is none and there is a single `Address` in the `addresses` list, the `str` value will be the same as the `str` of that single `Address`.

Footnotes

[^1]: Originally added in 3.3 as a `provisional module <provisional package>`

□□□ #2
□ □□□ □□□ 2026-10-31 17:09:11 UTC
□ □□□ □□□ 2026-10-31 17:09:15 UTC