21. Email, MIME, and Other Network Encodings

MIME and Email Format Handling

The `email` package handles parsing, generation, and manipulation of MIME files such as email messages, Network News (NNTP) posts, HTTP interactions, and so on. The Python standard library also contains other modules that handle some parts of these jobs. However, the `email` package offers a complete and systematic approach to these important tasks. We suggest you use `email`, not the older modules that partially overlap with parts of `email`'s functionality. `email`, despite its name, has nothing to do with receiving or sending email; for such tasks, see the modules `poplib` and `smtplib`, covered in "Email Protocols". `email` deals with handling MIME messages (which may or may not be mail) after you receive them, or constructing them properly before you send them.

Functions in the email Package

The `email` package supplies two factory functions that return an instance `m` of the class `email.message.Message`. These functions rely on the class `email.parser.Parser`, but the factory functions are handier and simpler. Therefore, we do not cover the `email.parser` module further in this book.

message_from_string	`message_from_string(s)` Builds `m` by parsing string `s`.
message_from_file	`message_from_file(f)` Builds `m` by parsing the contents of text file-like object `f`, which must be open for reading.

v3 only, also supplies two similar factory functions to build message objects from bytestrings and binary files:

message_from_bytes	`message_from_bytes(s)` Builds `m` by parsing bytestring `s`.
message_from_binary_file	`message_from_binary_file(f)` Builds `m` by parsing the contents of binary file-like object `f`, which must be open for reading.

The email.message Module

The `email.message` module supplies the class `Message`. All parts of the `email` package make, modify, or use instances of `Message`. An instance `m` of `Message` models a MIME message, including headers and a *payload* (data content). To create an initially empty `m`, call `Message` with no arguments. More often, you create `m` by parsing via the factory functions `message_from_string` and `message_from_file` of `email`, or by other indirect means such as the classes covered in "Creating Messages". `m`'s payload can be a string, a single other instance of `Message`, or (for a multipart message) a list of other `Message` instances.

You can set arbitrary headers on email messages you're building. Several Internet RFCs specify headers for a wide variety of purposes. The main applicable RFC is RFC 2822. An instance `m` of the `Message` class holds headers as

well as a payload. `m` is a mapping, with header names as keys and header value strings as values.

To make `m` more convenient, the semantics of `m` as a mapping are different from those of a `dict`. `m`'s keys are case-insensitive. `m` keeps headers in the order in which you add them, and the methods `keys`, `values`, and `items` return lists of headers in that order. `m` can have more than one header named `key`: `m[key]` returns an arbitrary such header (or `None` when the header is missing), and `del m[key]` deletes all of them (it's not an error if the header is missing).

To get a list of all headers with a certain name, call *`m.get_all(key)`*. `len(m)` returns the total number of headers, counting duplicates, not just the number of distinct header names. When there is no header named `key`, `m[key]` returns `None` and does not raise `KeyError` (i.e., behaves like `m.get(key)`): `del m[key]` does nothing in this case, and *`m.get_all(key)`* returns an empty list. In v2, you cannot loop directly on `m`; loop on `m.keys()` instead.

An instance `m` of `Message` supplies the following attributes and methods that deal with `m`'s headers and payload.

add_header

`m.add_header(_name,_value,**_params)`

Like `m[_name]=_value`, but you can also supply header parameters as named arguments. For each named argument `pname=pvalue`, `add_header` changes underscores to dashes, then appends to the header's value a parameter of the form:

```
; pname="pvalue"
```

When `pvalue` is `None`, `add_header` appends only a parameter `';pname'`.

When a parameter's value contains non-ASCII characters, specify it as a tuple with three items, *`(CHARSET, LANGUAGE, VALUE)`*. `CHARSET` names the encoding to use for the value, `LANGUAGE` is usually `None` or `''` but can be set any language value per RFC 2231, and `VALUE` is the string value containing non-ASCII characters.

as_string

`m.as_string(unixfrom=False)`

Returns the entire message as a string. When `unixfrom` is true, also includes a first `'From ` line, normally starting with `'From '`, known as the *envelope header* of the message. The class's `__str__` method is the same as `as_string`, but with `unixfrom` set to `True` in v2 only.

attach

`m.attach(payload)`

Adds `payload`, a message, to `m`'s payload. When `m`'s payload was `None`, `m`'s payload is now the single-item list `[payload]`. When `m`'s payload was a list of messages, appends `payload` to the list. When *`m`*'s payload was anything else, `m.attach(payload)` raises `MultipartConversionError`.

epilogue

The attribute `m.epilogue` can be `None` or a string that becomes part of the message's string-form after the last boundary line. Mail programs normally don't display this text. `epilogue` is a normal attribute of `m`: your program can access it when you're handling an `m` built by whatever means, and bind it when you're building or modifying `m`.

get_all

`m.get_all(name,default=None)`

Returns a list with all values of headers named `name` in the order in which the headers were added to `m`. When `m` has no header named `name`, `get_all` returns `default`.

get_boundary	`m.get_boundary(default=None)` Returns the string value of the `boundary` parameter of `m`'s Content-Type header. When `m` has no Content-Type header, or the header has no `boundary` parameter, `get_boundary` returns `default`.
get_charsets	`m.get_charsets(default=None)` Returns the list `L` of string values of parameter `charset` of `m`'s Content-Type headers. When `m` is multipart, `L` has one item per part; otherwise, `L` has length `1`. For parts that have no Content-Type, no `charset` parameter, or a main type different from `'text'`, the corresponding item in `L` is `default`.
get_content_maintype	`m.get_content_maintype(default=None)` Returns `m`'s main content type: a lowercased string `'maintype'` taken from header Content-Type. For example, when Content-Type is `'text/html'`, `get_content_maintype` returns `'text'`. When `m` has no header Content-Type, `get_content_maintype` returns `default`.
get_content_subtype	`m.get_content_subtype(default=None)` Returns `m`'s content subtype: a lowercased string `'subtype'` taken from header Content-Type. For example, when Content-Type is `'text/html'`, `get_content_subtype` returns `'html'`. When `m` has no header Content-Type, `get_content_subtype` returns `default`.
get_content_type	`m.get_content_type(default=None)` Returns `m`'s content type: a lowercased string `'maintype/subtype'` taken from header Content-Type. For example, when Content-Type is `'text/html'`, `get_content_type` returns `'text/html'`. When `m` has no header Content-Type, `get_content_type` returns `default`.
get_filename	`m.get_filename(default=None)` Returns the string value of the `filename` parameter of `m`'s Content-Disposition header. When `m` has no Content-Disposition, or the header has no `filename` parameter, `get_filename` returns `default`.
get_param	`m.get_param(param,default=None,header='Content-Type')` Returns the string value of parameter `param` of `m`'s header `header`. Returns the empty string for a parameter specified just by name (without a value). When `m` has no header `header`, or the header has no parameter named `param`, `get_param` returns `default` .
get_params	`m.get_params(default=None,header='Content-Type')` Returns the parameters of `m`'s header `header`, a list of pairs of strings that give each parameter's name and value. Uses the empty string as the value for parameters specified just by name (without a value). When `m` has no header `header`, `get_params` returns `default`.

get_payload	`m.get_payload(i=None,decode=False)` Returns `m`'s payload. When `m.is_multipart()` is `False`, `i` must be `None`, and `m.get_payload()` returns `m`'s entire payload, a string or `Message` instance. If `decode` is true and the value of header Content-Transfer-Encoding is either `'quoted-printable'` or `'base64'`, `m.get_payload` also decodes the payload. If `decode` is false, or header Content-Transfer-Encoding is missing or has other values, `m.get_payload` returns the payload unchanged. When `m.is_multipart()` is `True`, `decode` must be false. When `i` is `None`, `m.get_payload()` returns `m`'s payload as a list. Otherwise, `m.get_payload(i)` returns the `i`th item of the payload, or raises `TypeError` if `i<0` or `i` is too large.
get_unixfrom	`m.get_unixfrom()` Returns the envelope header string for `m`, or `None` when `m` has no envelope header.
is_multipart	`m.is_multipart()` Returns `True` when `m`'s payload is a list; otherwise, `False`.
preamble	Attribute `m.preamble` can be `None` or a string that becomes part of the message's string form before the first boundary line. A mail program shows this text only if it doesn't support multipart messages, so you can use this attribute to alert the user that your message is multipart and a different mail program is needed to view it. `preamble` is a normal attribute of `m`: your program can access it when you're handling an `m` that is built by whatever means and bind, rebind, or unbind it when you're building or modifying `m`.
set_boundary	`m.set_boundary(boundary)` Sets the `boundary` parameter of `m`'s Content-Type header to `boundary`. When `m` has no Content-Type header, raises `HeaderParseError`.
set_payload	`m.set_payload(payload)` Sets `m`'s payload to `payload`, which must be a string or list or `Message` instances, as appropriate to `m`'s Content-Type.
set_unixfrom	`m.set_unixfrom(unixfrom)` Sets the envelope header string for `m`. `unixfrom` is the entire envelope header line, including the leading `'From ` but *not* including the trailing `'\n'`.
walk	`m.walk()` Returns an iterator on all parts and subparts of `m` to walk the tree of parts depth-first (see "Recursion").

The email.Generator Module

The `email.Generator` module supplies the class `Generator`, which you can use to generate the textual form of a message `m`. `m.as_string()` and `str(m)` may be sufficient, but `Generator` gives you more flexibility. You instantiate the `Generator` class with a mandatory argument and two optional arguments:

Generator `class Generator(outfp,mangle_from_=False,maxheaderlen=78)`

`outfp` is a file or file-like object that supplies method `write`. When `mangle_from_` is true, `g` prepends `'>'` to any line in the payload that starts with `'From '`, in order to make the message's textual form easier to parse. `g` wraps each header line, at semicolons, into physical lines of no more than `maxheaderlen` characters. To use `g`, call `g.flatten`:

```
g.flatten(m, unixfrom=False)
```

This emits `m` as text to `outfp`, like (but consuming less memory than) `outfp.write(m.as_string(unixfrom))`.

Creating Messages

The `email` package supplies modules with names that, in v2, start with `'MIME'`, each module supplying a subclass of `Message` named just like the module. In v3, the modules are in the subpackage `email.mime`, and the modules' names are lowercase (for example, `email.mime.text` in v3, instead of `email.MIMEText` in v2). Although imported from different modules in v2 and v3, the class names are the same in both versions.

These classes make it easier to create `Message` instances of various MIME types. The MIME classes are as follows:

MIMEAudio `class MIMEAudio(_audiodata,_subtype=None,_encoder=None,**_params)`

`_audiodata` is a bytestring of audio data to pack in a message of MIME type `'audio/_subtype'`. When `_subtype` is `None`, `_audiodata` must be parseable by standard Python library module `sndhdr` to determine the subtype; otherwise, `MIMEAudio` raises `TypeError`. When `_encoder` is `None`, `MIMEAudio` encodes data as Base64, typically optimal. Otherwise, `_encoder` must be callable with one parameter `m`, which is the message being constructed; `_encoder` must then call `m.get_payload()` to get the payload, encode the payload, put the encoded form back by calling `m.set_payload`, and set `m`'s `'Content-Transfer-Encoding'` header appropriately. `MIMEAudio` passes the `_params` dictionary of named-argument names and values to `m.add_header` to construct `m`'s `Content-Type`.

MIMEBase `class MIMEBase(_maintype,_subtype,**_params)`

Base class of all MIME classes, directly extends `Message`. Instantiating:

```
m = MIMEBase(main,sub,**parms)
```

is equivalent to the longer and less convenient idiom:

```
m = Message()m.add_header('Content-Type','{}/{}'.  format(main,sub),**parms)m.add_header('Mime-Version','1.0')
```

MIMEImage	`class MIMEImage(_imagedata,_subtype=None,_encoder=None, **_params)` Like `MIMEAudio`, but with main type `'image'`; uses standard Python module `imghdr` to determine the subtype, if needed.
MIMEMessage	`class MIMEMessage(msg,_subtype='rfc822')` Packs `msg`, which must be an instance of `Message` (or a subclass), as the payload of a message of MIME type `'message/_subtype'`.
MIMEText	`class MIMEText(_text,_subtype='plain',_charset='us-ascii', _encoder=None)` Packs text string `_text` as the payload of a message of MIME type `'text/_subtype'` with the given `charset`. When `_encoder` is `None`, `MIMEText` does not encode the text, which is generally optimal. Otherwise, `_encoder` must be callable with one parameter `m`, which is the message being constructed; `_encoder` must then call `m.get_payload()` to get the payload, encode the payload, put the encoded form back by calling `m.set_payload`, and set `m`'s `'Content-Transfer-Encoding'` appropriately.

The email.encoders Module

The `email.encoders` module (in v3) supplies functions that take a *non-multipart* message `m` as their only argument, encode `m`'s payload, and set `m`'s headers appropriately. In v2, the module's name is titlecase, `email.Encoders`.

encode_base64	`encode_base64(m)` Uses Base64 encoding, optimal for arbitrary binary data.
encode_noop	`encode_noop(m)` Does nothing to `m`'s payload and headers.
encode_quopri	`encode_quopri(m)` Uses Quoted Printable encoding, optimal for text that is almost but not fully ASCII (see "The quopri Module").
encode_7or8bit	`encode_7or8bit(m)` Does nothing to `m`'s payload, and sets header Content-Transfer-Encoding to `'8bit'` when any byte of `m`'s payload has the high bit set; otherwise, to `'7bit'`.

The email.utils Module

The `email.utils` module (in v3) supplies several functions useful for email processing. In v2, the module's name is titlecase, `email.Utils`.

formataddr	`formataddr(pair)` `pair` is a pair of strings `(realname,email_address)`. `formataddr` returns a string `s` with the address to insert in header fields such as `To` and `Cc`. When `realname` is false (e.g., the empty string, `''`), `formataddr` returns `email_address`.
formatdate	`formatdate(timeval=None,localtime=False)` `timeval` is a number of seconds since the epoch. When `timeval` is `None`, `formatdate` uses the current time. When `localtime` is true, `formatdate` uses the local time zone; otherwise, it uses UTC. `formatdate` returns a string with the given time instant formatted in the way specified by RFC 2822.
getaddresses	`getaddresses(L)` Parses each item of `L`, a list of address strings as used in header fields such as `To` and `Cc`, and returns a list of pairs of strings `(name,email_address)`. When `getaddresses` cannot parse an item of `L` as an address, `getaddresses` uses `(None,None)` as the corresponding item in the list it returns.
mktime_tz	`mktime_tz(t)` `t` is a tuple with 10 items. The first nine items of `t` are in the same format used in the module `time`, covered in "The time Module". `t[-1]` is a time zone as an offset in seconds from UTC (with the opposite sign from `time.timezone`, as specified by RFC 2822). When `t[-1]` is `None`, `mktime_tz` uses the local time zone. `mktime_tz` returns a float with the number of seconds since the epoch, in UTC, corresponding to the instant that `t` denotes.
parseaddr	`parseaddr(s)` Parses string `s`, which contains an address as typically specified in header fields such as `To` and `Cc`, and returns a pair of strings `(realname,email_address)`. When `parseaddr` cannot parse `s` as an address, `parseaddr` returns `('','')`.
parsedate	`parsedate(s)` Parses string `s` as per the rules in RFC 2822 and returns a tuple `t` with nine items, as used in the module `time`, covered in "The time Module" (the items `t[-3:]` are not meaningful). `parsedate` also attempts to parse some erroneous variations on RFC 2822 that widespread mailers use. When `parsedate` cannot parse `s`, `parsedate` returns `None`.
parsedate_tz	`parsedate_tz(s)` Like `parsedate`, but returns a tuple `t` with 10 items, where `t[-1]` is `s`'s time zone as an offset in seconds from UTC (with the opposite sign from `time.timezone`, as specified by RFC 2822), like in the argument that `mktime_tz` accepts. Items `t[-4:-1]` are not meaningful. When `s` has no time zone, `t[-1]` is `None`.
quote	`quote(s)` Returns a copy of string `s`, where each double quote (`"`) becomes `'\"'` and each existing backslash is repeated.

unquote	`unquote(s)` Returns a copy of string `s` where leading and trailing double-quote characters (`"`) and angle brackets (`<>`) are removed if they surround the rest of `s`.

Example Uses of the email Package

The `email` package helps you both in reading and composing email and email-like messages (but it's not involved in receiving and transmitting such messages: those tasks belong to different and separate modules covered in Chapter 19). Here is an example of how to use `email` to read a possibly multipart message and unpack each part into a file in a given directory:

```
import os, email

def unpack_mail(mail_file, dest_dir):

''' Given file object mail_file, open for reading, and dest_dir,
a
        string that is a path to an existing, writable
directory,
        unpack each part of the mail message from mail_file to
a
        file within
dest_dir.

'''
    with mail_file:
        msg = email.message_from_file(mail_file)
    for part_number, part in enumerate(msg.walk()):
        if part.get_content_maintype() == 'multipart':
            # we'll get each specific part later in the
            loop,
            # so, nothing to do for the 'multipart'
            itself
            continue
        dest = part.get_filename()
        if dest is None: dest = part.get_param('name')
        if dest is None: dest = 'part-{}'.format(part_number)
        # In real life, make sure that dest is a reasonable
        filename
        # for your OS; otherwise, mangle that name until it
        is
        with open(os.path.join(dest_dir, dest), 'wb') as f:
            f.write(part.get_payload(decode=True))
```

And here is an example that performs roughly the reverse task, packaging all files that are directly under a given source directory into a single file suitable for mailing:

```
def pack_mail(source_dir, **headers):
    ''' Given source_dir, a string that is a path to an
    existing,
        readable directory, and arbitrary header name/value
pairs
        passed in as named arguments, packs all the files
directly
        under source_dir (assumed to be plain text files) into
a
        mail message returned as a MIME-formatted
string.

'''
    msg = email.Message.Message()
    for name, value in headers.items():
        msg[name] = value
    msg['Content-type'] = 'multipart/mixed'
    filenames = next(os.walk(source_dir))[-1]
    for filename in filenames:
        m = email.Message.Message()
        m.add_header('Content-type', 'text/plain', name=filename)
        with open(os.path.join(source_dir, filename), 'r') as f:
            m.set_payload(f.read())
        msg.attach(m)
    return msg.as_string()
```

rfc822 and mimetools Modules (v2)

The best way to handle email-like messages is with the `email` package. However, some other modules covered in Chapters 19 and 20, in v2 only, use instances of the class `rfc822.Message` or its subclass, `mimetools.Message`. This section covers the subset of these classes' functionality that you need to make effective use, in v2, of the modules covered in Chapters 19 and 20.

An instance `m` of the class `Message` in either of these v2-only modules is a mapping, with the headers' names as keys and the corresponding header value strings as values. Keys and values are strings, and keys are case-insensitive. `m` supports all mapping methods except `clear`, `copy`, `popitem`, and `update`. `get` and `setdefault` default to `''` instead of `None`. The instance `m` also supplies convenience methods (e.g., to combine getting a header's value and parsing it as a date or an address). For such purposes, we suggest you use the functions of the module `email.utils` (covered in "The email.utils Module"): use `m` just as a mapping.

When `m` is an instance of `mimetools.Message`, `m` supplies additional methods:

getmaintype `m.getmaintype()`

Returns `m`'s main content type, from header Content-Type, in lowercase. When `m` has no header Content-Type, `getmaintype` returns `'text'`.

getparam `m.getparam(param)`

Returns the value of the parameter named `param` of `m`'s Content-Type.

getsubtype	`m.getsubtype()` Returns `m`'s content subtype, taken from Content-Type, in lowercase. When `m` has no Content-Type, `getsubtype` returns `'plain'`.
gettype	`m.gettype()` Returns `m`'s content type, taken from Content-Type, in lowercase. When `m` has no Content-Type, `gettype` returns `'text/plain'`.