Chapter 21. Email, MIME, and Other Network Encodings

What travels on a network are streams of bytes, also known in networking jargon as *octets*. Bytes can, of course, represent text, via any of several possible encodings. However, what you want to send over the network often has more structure than just a stream of text or bytes. The Multipurpose Internet Mail Extensions (**MIME**) and other encoding standards bridge the gap, by specifying how to represent structured data as bytes or text. While often originally designed for email, such encodings are also used on the web and in many other networked systems. Python supports such encodings through various library modules, such as `base64`, `quopri`, and `uu` (covered in **"Encoding Binary Data as ASCII Text"**), and the modules of the `email` package (covered in the following section). These encodings allow us, for example, to seamlessly create messages in one encoding containing attachments in another, avoiding many awkward tasks along the way.

MIME and Email Format Handling

The `email` package handles parsing, generation, and manipulation of MIME files such as email messages, Network News Transfer Protocol (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, need have nothing to do with receiving or sending email; for such tasks, see the modules `imaplib`, `poplib`, and `smtplib`, covered in **"Email Protocols"**. Rather, `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 four factory functions that return an instance *m* of the class `email.message.Message` from a string or file (see **Table 21-1**). 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.

Table 21-1. `email` factory functions that build message objects from strings or files

`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
`message_from_bytes`	`message_from_bytes(`*s*`)` Builds *m* by parsing bytestring *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
`message_from_string`	`message_from_string(`*s*`)` Builds *m* by parsing string *s*

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). *m* is a mapping, with header names as keys, and header value strings as values.

To create an initially empty *m*, call `Message` with no arguments. More often, you create *m* by parsing via one of the factory functions in **Table 21-1**, or 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 a *multipart message* (a recursively nested 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**; you can find a summary of many other RFCs about headers in nonnormative **RFC 2076**.

To make *m* more convenient, its semantics 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 (not views!) 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., it behaves like `m.get(key)`): **`del`** `m[key]` does nothing in this case, and `m.get_all(key)` returns **`None`**. You can loop directly on *m*: it's just like looping on `m.keys()` instead.

An instance *m* of `Message` supplies various attributes and methods that deal with *m*'s headers and payload, listed in **Table 21-2**.

Table 21-2. Attributes and methods of an instance *m* of `Message`

`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 any underscores in *pname* to dashes, then appends to the header's value a string of the form: ; *pname*="*pvalue*" When *pvalue* is **None**, add_header appends only a string of the form: ; *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; *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 line, normally starting with 'From ', known as the *envelope header* of the message.
attach	*m*.attach(*payload*) Adds *payload*, a message, to *m*'s payload. When *m*'s payload is **None**, *m*'s payload is now the single-item list [*payload*]. When *m*'s payload is a list of messages, appends *payload* to the list. When *m*'s payload is 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 any *m*, 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 are 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 header. When *m* is multipart, *L* has one item per part; otherwise, *L* has length `1`. For parts that have no Content-Type header, 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 lowercase string `'`*maintype*`'` taken from header Content-Type. For example, when Content-Type is `'Text/Html'`, `get_content_maintype` returns `'text'`. When *m* has no Content-Type header, `get_content_maintype` returns `default`.
`get_content_subtype`	`m.get_content_subtype(default=None)` Returns *m*'s content subtype: a lowercase string `'`*subtype*`'` taken from header Content-Type. For example, when Content-Type is `'Text/Html'`, `get_content_subtype` returns `'html'`. When *m* has

	no Content-Type header, `get_content_subtype` returns `default`.
`get_content_type`	*`m`*`.get_content_type(default=`**`None`**`)` Returns *m*’s content type: a lowercase 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 Content-Type header, `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 header, 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 `''` 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 `''` 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, returns **`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 a list of `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 enough, but `Generator` gives more flexibility. Instantiate the `Generator` class with a mandatory argument, *outfp*, and two optional arguments:

`Generator`	**`class`** `Generator(`*`outfp`*`, mangle_from_=`**`False`**`, maxheaderlen=78)` *outfp* is a file or file-like object that supplies the method `write`. When `mangle_from_` is true, *g* prepends a greater-than sign (>) to any line in the payload that starts with `'From '`, 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`; for example:

```
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 subpackage `email.mime` supplies various modules, each with a subclass of `Message` named like the module. The modules' names are lowercase (e.g., `email.mime.text`), while the class names are in mixed case. These classes, listed in **Table 21-3**, help you create `Message` instances of different MIME types.

Table 21-3. Classes supplied by `email.mime`

`MIMEAudio`	**`class`** `MIMEAudio(`*`_audiodata`*`, _subtype=`**`None`**`, _encoder=`**`None`**`, **`*`_params`*`)` Creates MIME message objects of major type 'audio'. *`_audiodata`* is a bytestring of audio data to pack in a message of MIME type `'audio/_subtype'`. When `_subty` is **None**, *`_audiodata`* must be parsable by standard Pytho library module `sndhdr` to determine the subtype; otherwise, `MIMEAudio` raises `TypeError`. **3.11+** Since `snd` is deprecated, you should always specify the `_subtype`. When `_encoder` is **None**, `MIMEAudio` encodes data as Base which is usually optimal. Otherwise, `_encoder` must be callable with one parameter, *m*, which is the message be constructed; `_encoder` must then call *m*`.get_payload` to the payload, encode the payload, put the encoded form back by calling *m*`.set_payload`, and set *m*'s Content-

	Transfer-Encoding header. MIMEAudio passes the *_param* dictionary of named argument names and values to *m*.add_header to construct *m*'s Content-Type header.

MIMEBase	**class** MIMEBase(*_maintype*, *_subtype*, ***_params*)
	Base class of all MIME classes; extends Message.
	Instantiating:
	`m = MIMEBase(mainsub, **params)`
	is equivalent to the longer and slightly less convenient idiom:
	`m = Message()`
	`m.add_header('Content-Type', f'{main}/{sub}',`
	`             **params)`
	`m.add_header('Mime-Version', '1.0')`

MIMEImage	**class** MIMEImage(*_imagedata*, _subtype=**None**, _encoder=**None**, ***_params*) Like MIMEAudio, but with main type 'image'; uses stand Python module imghdr to determine the subtype, if need **3.11+** Since imghdr is deprecated, you should always specify the _subtype.
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 te which is generally the best choice. 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, p the encoded form back by calling *`m`*`.set_payload`, and se *m*’s Content-Transfer-Encoding header appropriately.

The email.encoders Module

The `email.encoders` module supplies functions that take a *nonmultipart* message *`m`* as their only argument, encode *m*’s payload, and set *m*’s headers appropriately. These functions are listed in **Table 21-4**.

Table 21-4. Functions of the `email.encoders` module

`encode_base64`	`encode_base64(`*`m`*`)` Uses Base64 encoding, usually optimal for arbitrary binary data (see **“The base64 Module”**).
`encode_noop`	`encode_noop(`*`m`*`)` Does nothing to *m*’s payload and headers.
`encode_quopri`	`encode_quopri(`*`m`*`)` Uses Quoted Printable encoding, usually 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, but sets the header Content-Transfer-Encoding to `'8bit'` when any byte of *m*’s payload has the high bit set; otherwise, sets it to `'7bit'`.

The email.utils Module

The `email.utils` module supplies several functions for email processing, listed in Table 21-5.

Table 21-5. Functions of the `email.utils` module

`formataddr`	`formataddr(`*`pair`*`)` Takes a pair of strings `(`*`realname`*`, `*`email_address`*`)` and 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`**`)` Returns a string with the time instant formatted as specified by RFC 2822. `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.
`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`*`, `*`address`*`)`. When `getaddresses` cannot parse an item of *`L`* as an email address, it sets `('', '')` as the corresponding item in the list.
`mktime_tz`	`mktime_tz(`*`t`*`)` Returns a `float` representing the number of seconds since the epoch, in UTC, corresponding to the instant that *`t`* denotes. *`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.
`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, address`*`)`. When `parseaddr` cannot parse *s* as an address, it 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 commonly encountered mailers use. When `parsedate` cannot parse *s*, it 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 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:

```python
import pathlib, 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.
    '''
    dest_dir_path = pathlib.Path(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 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 = f'part-{part_number}'
        # in real life, make sure that dest is a reasonable filename
        # for your OS; otherwise, mangle that name until it is
        part_payload = part.get_payload(decode=True)
        (dest_dir_path / dest).write_text(part_payload)
```

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:

```python
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.
    '''
    source_dir_path = pathlib.Path(source_dir)
    msg = email.message.Message()
    for name, value in headers.items():
        msg[name] = value
    msg['Content-type'] = 'multipart/mixed'
    filepaths = [path for path in source_dir_path.iterdir()
                 if path.is_file()]
    for filepath in filepaths:
        m = email.message.Message()
        m.add_header('Content-type', 'text/plain', name=filename)
        m.set_payload(filepath.read_text())
        msg.attach(m)
    return msg.as_string()
```

Encoding Binary Data as ASCII Text

Several kinds of media (e.g., email messages) can contain only ASCII text. When you want to transmit arbitrary binary data via such media, you need to encode the data as ASCII text strings. The Python standard library supplies modules that support the standard encodings known as Base64, Quoted Printable, and Unix-to-Unix, described in the following sections.

The base64 Module

The `base64` module supports the encodings specified in **RFC 3548** as Base16, Base32, and Base64. Each of these encodings is a compact way to represent arbitrary binary data as ASCII text, without any attempt to produce human-readable results. `base64` supplies 10 functions: 6 for Base64, plus 2 each for Base32 and Base16. The six Base64 functions are listed in **Table 21-6**.

Table 21-6. Base64 functions of the `base64` module

`b64decode`	`b64decode(`*`s`*`, altchars=`**`None`**`, validate=`**`False`**`)` Decodes B64-encoded bytestring *`s`*, and returns the decoded bytestring. `altchars`, if not **`None`**, must be a bytestring of at least two characters (extra characters are ignored) specifying the two nonstandard characters to use instead of `+` and `/` (potentially useful to decode URL-safe or filesystem-safe B64-encoded strings). When `validate` is **`True`**, the call raises an exception if *`s`* contains any bytes that are not valid in B64-encoded strings (by default, such bytes are just ignored and skipped). Also raises an exception when *`s`* is improperly padded according to the Base64 standard.
`b64encode`	`b64encode(`*`s`*`, altchars=`**`None`**`)` Encodes bytestring *`s`* and returns the bytestring with the corresponding B64-encoded data. `altchars`, if not **`None`**, must be a bytestring of at least two characters (extra characters are ignored) specifying the two nonstandard characters to use instead of `+` and `/` (potentially useful to make URL-safe or filesystem-safe B64-encoded strings).
`standard_b64decode`	`standard_b64decode(`*`s`*`)` Like `b64decode(`*`s`*`)`.
`standard_b64encode`	`standard_b64encode(`*`s`*`)` Like `b64encode(`*`s`*`)`.
`urlsafe_b64decode`	`urlsafe_b64decode(`*`s`*`)` Like `b64decode(`*`s`*`, '-_')`.
`urlsafe_b64encode`	`urlsafe_b64encode(`*`s`*`)` Like `b64encode(`*`s`*`, '-_')`.

The four Base16 and Base32 functions are listed in Table 21-7.

Table 21-7. Base16 and Base32 functions of the `base64` module

`b16decode`	`b16decode(s, casefold=False)` Decodes B16-encoded bytestring *s*, and returns the decoded bytestring. When `casefold` is **`True`**, lowercase characters in *s* are treated like their uppercase equivalents; by default, when lowercase characters are present, the call raises an exception.
`b16encode`	`b16encode(s)` Encodes bytestring *s*, and returns the bytestring with the corresponding B16-encoded data.
`b32decode`	`b32decode(s, casefold=False, map01=None)` Decodes B32-encoded bytestring *s*, and returns the decoded bytestring. When `casefold` is **`True`**, lowercase characters in *s* are treated like their uppercase equivalents; by default, when lowercase characters are present, the call raises an exception. When `map01` is **`None`**, characters `0` and `1` are not allowed in the input; when not **`None`**, it must be a single-character bytestring specifying what `1` is mapped to (lowercase `'l'` or uppercase `'L'`); `0` is then always mapped to uppercase `'O'`.
`b32encode`	`b32encode(s)` Encodes bytestring *s* and returns the bytestring with the corresponding B32-encoded data.

The module also supplies functions to encode and decode the nonstandard but popular encodings Base85 and Ascii85, which, while not codified in RFCs or compatible with each other, can offer space savings of 15% by using larger alphabets for encoded bytestrings. See the online docs for details on those functions.

The quopri Module

The quopri module supports the encoding specified in RFC 1521 as *Quoted Printable* (QP). QP can represent any binary data as ASCII text, but it's mainly intended for data that is mostly text, with a small amount of characters with the high bit set (i.e., characters outside the ASCII range). For such data, QP produces results that are both compact and human-readable. The quopri module supplies four functions, listed in **Table 21-8**.

Table 21-8. Functions of the quopri module

decode	decode(*infile*, *outfile*, header=**False**) Reads the binary file-like object *infile* by calling *infile*.readline until end-of-file (i.e., until a call to *infile*.readline returns an empty string), decodes the QP-encoded ASCII text thus read, and writes the results to binary file-like object *outfile*. When header is true, decode also turns _ (underscores) into spaces (per RFC 1522).
decodestring	decodestring(*s*, header=**False**) Decodes bytestring *s*, QP-encoded ASCII text, and returns the bytestring with the decoded data. When header is true, decodestring also turns _ (underscores) into spaces.
encode	encode(*infile*, *outfile*, *quotetabs*, header=**False**) Reads binary file-like object *infile* by calling *infile*.readline until end-of-file (i.e., until a call to *infile*.readline returns an empty string), encodes the data thus read in QP, and writes the encoded ASCII text to binary file-like object *outfile*. When *quotetabs* is true, encode also encodes spaces and tabs. When header is true, encode encodes spaces as _ (underscores).

encodestring	encodestring(*s*, quotetabs=**False**, header=**False**) Encodes bytestring *s*, which contains arbitrary bytes, and returns a bytestring with QP-encoded ASCII text. When quotetabs is true, encodestring also encodes spaces and tabs. When header is true, encodestring encodes spaces as _ (underscores).

The uu Module

The uu module[1] supports the classic *Unix-to-Unix* (UU) encoding, as implemented by the Unix programs *uuencode* and *uudecode*. UU starts encoded data with a begin line, which includes the filename and permissions of the file being encoded, and ends it with an end line. Therefore, UU encoding lets you embed encoded data in otherwise unstructured text, while Base64 encoding (discussed in **"The base64 Module"**) relies on the existence of other indications of where the encoded data starts and finishes. The uu module supplies two functions, listed in **Table 21-9**.

Table 21-9. Functions of the uu module

decode	decode(*infile*, outfile=**None**, mode=**None**) Reads the file-like object *infile* by calling *infile*.readline until end-of-file (i.e., until a call to *infile*.readline returns an empty string) or until a terminator line (the string 'end' surrounded by any amount of whitespace). decode decodes the UU-encoded text thus read and writes the decoded data to the file-like object outfile. When outfile is **None**, decode creates the file specified in the UU-format begin line, with the permission bits given by mode (the permission bits specified in the begin line, when mode is **None**). In this case, decode raises an exception if the file already exists.
encode	encode(*infile*, *outfile*, name='-', mode=0o666) Reads the file-like object *infile* by calling *infile*.read

(45 bytes at a time, which is the amount of data that UU encodes into 60 characters in each output line) until end-of-file (i.e., until a call to *infile*.read returns an empty string). It encodes the data thus read in UU and writes the encoded text to file-like object *outfile*. encode also writes a UU-format begin line before the text and a UU-format end line after the text. In the begin line, encode specifies the filename as name and the mode as mode.

[1] Deprecated in Python 3.11, to be removed in Python 3.13; the online docs direct users to update existing code to use the base64 module for data content and MIME headers for metadata.