18.4. `mailbox` — Manipulate mailboxes in various formats

This module defines two classes, `Mailbox` and `Message`, for accessing and manipulating on-disk mailboxes and the messages they contain. `Mailbox` offers a dictionary-like mapping from keys to messages. `Message` extends the `email.message` module's `Message` class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

See also:

Module `email`
Represent and manipulate messages.

18.4.1. `Mailbox` objects

class `mailbox.`**`Mailbox`**
A mailbox, which may be inspected and modified.

The `Mailbox` class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from `Mailbox` and your code should instantiate a particular subclass.

The `Mailbox` interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the `Mailbox` instance with which they will be used and are only meaningful to that `Mailbox` instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a `Mailbox` instance using the set-like method `add()` and removed using a `del` statement or the set-like methods `remove()` and `discard()`.

`Mailbox` interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a `Message` instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a `Mailbox` instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the `Mailbox` instance.

The default `Mailbox` iterator iterates over message representations, not keys as the default dictionary iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a `KeyError` exception if the corresponding message is subsequently removed.

Warning: Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is Maildir; try to avoid using single-file formats such as mbox for concurrent writing. If you're modifying a mailbox, you *must* lock it by calling the `lock()` and `unlock()` methods *before* reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

`Mailbox` instances have the following methods:

add(*message*)

Add *message* to the mailbox and return the key that has been assigned to it.

Parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, or a file-like object (which should be open in text mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

remove(*key*)
__delitem__(*key*)
discard(*key*)

Delete the message corresponding to *key* from the mailbox.

If no such message exists, a `KeyError` exception is raised if the method was called as `remove()` or `__delitem__()` but no exception is raised if the method was called as `discard()`. The behavior of `discard()` may be preferred if the underlying mailbox format supports concurrent modification by other processes.

__setitem__(*key*, *message*)

Replace the message corresponding to *key* with *message*. Raise a `KeyError` exception if no message already corresponds to *key*.

As with `add()`, parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, or a file-like object (which should be open in text mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to *key* is left unchanged.

iterkeys()
keys()

Return an iterator over all keys if called as `iterkeys()` or return a list of keys if called as `keys()`.

itervalues()
__iter__()

values()

Return an iterator over representations of all messages if called as `itervalues()` or `__iter__()` or return a list of such representations if called as `values()`. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

> **Note:** The behavior of `__iter__()` is unlike that of dictionaries, which iterate over keys.

iteritems()
items()

Return an iterator over (*key*, *message*) pairs, where *key* is a key and *message* is a message representation, if called as `iteritems()` or return a list of such pairs if called as `items()`. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get(*key*, *default=None*)
__getitem__(*key*)

Return a representation of the message corresponding to *key*. If no such message exists, *default* is returned if the method was called as `get()` and a `KeyError` exception is raised if the method was called as `__getitem__()`. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get_message(*key*)

Return a representation of the message corresponding to *key* as an instance of the appropriate format-specific `Message` subclass, or raise a `KeyError` exception if no such message exists.

get_string(*key*)

Return a string representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists.

get_file(*key*)

Return a file-like representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

> **Note:** Unlike other representations of messages, file-like representations are not necessarily independent of the `Mailbox` instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

has_key(*key*)
__contains__(*key*)
Return `True` if *key* corresponds to a message, `False` otherwise.

__len__()
Return a count of messages in the mailbox.

clear()
Delete all messages from the mailbox.

pop(*key*[, *default*])
Return a representation of the message corresponding to *key* and delete the message. If no such message exists, return *default* if it was supplied or else raise a **KeyError** exception. The message is represented as an instance of the appropriate format-specific **Message** subclass unless a custom message factory was specified when the **Mailbox** instance was initialized.

popitem()
Return an arbitrary (*key*, *message*) pair, where *key* is a key and *message* is a message representation, and delete the corresponding message. If the mailbox is empty, raise a **KeyError** exception. The message is represented as an instance of the appropriate format-specific **Message** subclass unless a custom message factory was specified when the **Mailbox** instance was initialized.

update(*arg*)
Parameter *arg* should be a *key*-to-*message* mapping or an iterable of (*key*, *message*) pairs. Updates the mailbox so that, for each given *key* and *message*, the message corresponding to *key* is set to *message* as if by using **__setitem__()**. As with **__setitem__()**, each *key* must already correspond to a message in the mailbox or else a **KeyError** exception will be raised, so in general it is incorrect for *arg* to be a **Mailbox** instance.

> **Note:** Unlike with dictionaries, keyword arguments are not supported.

flush()
Write any pending changes to the filesystem. For some **Mailbox** subclasses, changes are always written immediately and **flush()** does nothing, but you should still make a habit of calling this method.

lock()
Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An **ExternalClashError** is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

unlock()
Release the lock on the mailbox, if any.

close()

Flush the mailbox, unlock it if necessary, and close any open files. For some `Mailbox` subclasses, this method does nothing.

18.4.1.1. `Maildir`

class `mailbox.`**Maildir**(*dirname*, *factory=rfc822.Message*, *create=True*)

A subclass of `Mailbox` for mailboxes in Maildir format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, `MaildirMessage` is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

It is for historical reasons that *factory* defaults to `rfc822.Message` and that *dirname* is named as such rather than *path*. For a `Maildir` instance that behaves like instances of other `Mailbox` subclasses, set *factory* to `None`.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: `tmp`, `new`, and `cur`. Messages are created momentarily in the `tmp` subdirectory and then moved to the `new` subdirectory to finalize delivery. A mail user agent may subsequently move the message to the `cur` subdirectory and store information about the state of the message in a special “info” section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if `'.'` is the first character in its name. Folder names are represented by `Maildir` without the leading `'.'`. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using `'.'` to delimit levels, e.g., “Archived.2005.07”.

> **Note:** The Maildir specification requires the use of a colon (`':'`) in certain message file names. However, some operating systems do not permit this character in file names, If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (`'!'`) is a popular choice. For example:
>
> ```
> import mailbox
> mailbox.Maildir.colon = '!'
> ```
>
> The **colon** attribute may also be set on a per-instance basis.

`Maildir` instances have all of the methods of `Mailbox` in addition to the following:

list_folders()
Return a list of the names of all folders.

get_folder(*folder*)
Return a `Maildir` instance representing the folder whose name is *folder*. A `NoSuchMailboxError` exception is raised if the folder does not exist.

add_folder(*folder*)
Create a folder whose name is *folder* and return a `Maildir` instance representing it.

remove_folder(*folder*)
Delete the folder whose name is *folder*. If the folder contains any messages, a `NotEmptyError` exception will be raised and the folder will not be deleted.

clean()
Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

Some `Mailbox` methods implemented by `Maildir` deserve special remarks:

add(*message*)
__setitem__(*key*, *message*)
update(*arg*)

> **Warning:** These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()
All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()
unlock()
Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()
`Maildir` instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file(*key*)
Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

See also:

maildir man page from qmail
The original specification of the format.

Using maildir format
Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on “info” semantics.

maildir man page from Courier
Another specification of the format. Describes a common extension for supporting folders.

18.4.1.2. mbox

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)

A subclass of **Mailbox** for mailboxes in mbox format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is None, **mboxMessage** is used as the default message representation. If *create* is True, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are “From ”.

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, **mbox** implements the original format, which is sometimes referred to as *mboxo*. This means that the *Content-Length* header, if present, is ignored and that any occurrences of “From ” at the beginning of a line in a message body are transformed to “>From ” when storing the message, although occurrences of “>From ” are not transformed to “From ” when reading the message.

Some **Mailbox** methods implemented by **mbox** deserve special remarks:

get_file(*key*)
Using the file after calling **flush()** or **close()** on the **mbox** instance may yield unpredictable results or raise an exception.

lock()
unlock()
Three locking mechanisms are used—dot locking and, if available, the **flock()** and **lockf()** system calls.

See also:

mbox man page from qmail
A specification of the format and its variations.

mbox man page from tin
Another specification of the format, with details on locking.

Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad
An argument for using the original mbox format rather than a variation.

"mbox" is a family of several mutually incompatible mailbox formats
A history of mbox variations.

18.4.1.3. MH

class mailbox.**MH**(*path*, *factory=None*, *create=True*)

A subclass of Mailbox for mailboxes in MH format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is None, MHMessage is used as the default message representation. If *create* is True, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called *folders*) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support *sequences*, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called .mh_sequences in each folder.

The MH class manipulates MH mailboxes, but it does not attempt to emulate all of **mh**'s behaviors. In particular, it does not modify and is not affected by the context or .mh_profile files that are used by **mh** to store its state and configuration.

MH instances have all of the methods of Mailbox in addition to the following:

list_folders()
Return a list of the names of all folders.

get_folder(*folder*)
Return an MH instance representing the folder whose name is *folder*. A NoSuchMailboxError exception is raised if the folder does not exist.

add_folder(*folder*)
Create a folder whose name is *folder* and return an MH instance representing it.

remove_folder(*folder*)
Delete the folder whose name is *folder*. If the folder contains any messages, a NotEmptyError exception will be raised and the folder will not be deleted.

get_sequences()
Return a dictionary of sequence names mapped to key lists. If there are no

sequences, the empty dictionary is returned.

set_sequences(*sequences*)
Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by **get_sequences()**.

pack()
Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

> **Note:** Already-issued keys are invalidated by this operation and should not be subsequently used.

Some **Mailbox** methods implemented by **MH** deserve special remarks:

remove(*key*)
__delitem__(*key*)
discard(*key*)
These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

lock()
unlock()
Three locking mechanisms are used—dot locking and, if available, the **flock()** and **lockf()** system calls. For MH mailboxes, locking the mailbox means locking the `.mh_sequences` file and, only for the duration of any operations that affect them, locking individual message files.

get_file(*key*)
Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

flush()
All changes to MH mailboxes are immediately applied, so this method does nothing.

close()
MH instances do not keep any open files, so this method is equivalent to **unlock()**.

See also:

nmh - Message Handling System
Home page of **nmh**, an updated version of the original **mh**.

MH & nmh: Email for Users & Programmers
A GPL-licensed book on **mh** and **nmh**, with some information on the mailbox format.

18.4.1.4. Babyl

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)

A subclass of Mailbox for mailboxes in Babyl format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is None, BabylMessage is used as the default message representation. If *create* is True, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (`'\037'`) and Control-L (`'\014'`). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (`'\037'`) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

Babyl instances have all of the methods of Mailbox in addition to the following:

get_labels()

Return a list of the names of all user-defined labels used in the mailbox.

> **Note:** The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some Mailbox methods implemented by Babyl deserve special remarks:

get_file(*key*)

In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into a StringIO instance (from the StringIO module), which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

lock()

unlock()

Three locking mechanisms are used—dot locking and, if available, the **flock()** and **lockf()** system calls.

See also:

Format of Version 5 Babyl Files
A specification of the Babyl format.

Reading Mail with Rmail
The Rmail manual, with some information on Babyl semantics.

18.4.1.5. MMDF

class mailbox.**MMDF**(*path*, *factory=None*, *create=True*)

A subclass of **Mailbox** for mailboxes in MMDF format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, **MMDFMessage** is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`'\001'`) characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are “From ”, but additional occurrences of “From ” are not transformed to “>From ” when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some **Mailbox** methods implemented by **MMDF** deserve special remarks:

get_file(*key*)

Using the file after calling **flush()** or **close()** on the **MMDF** instance may yield unpredictable results or raise an exception.

lock()
unlock()

Three locking mechanisms are used—dot locking and, if available, the **flock()** and **lockf()** system calls.

See also:

mmdf man page from tin
A specification of MMDF format from the documentation of tin, a newsreader.

MMDF
A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

18.4.2. Message objects

class mailbox.**Message**([*message*])

A subclass of the email.message module's Message. Subclasses of mailbox.Message add mailbox-format-specific state and behavior.

If *message* is omitted, the new instance is created in a default, empty state. If *message* is an email.message.Message instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if *message* is a Message instance. If *message* is a string or a file, it should contain an **RFC 2822**-compliant message, which is read and parsed.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that Message instances be used to represent messages retrieved using Mailbox instances. In some situations, the time and memory required to generate Message representations might not be acceptable. For such situations, Mailbox instances also offer string and file-like representations, and a custom message factory may be specified when a Mailbox instance is initialized.

18.4.2.1. MaildirMessage

class mailbox.**MaildirMessage**([*message*])

A message with Maildir-specific behaviors. Parameter *message* has the same meaning as with the Message constructor.

Typically, a mail user agent application moves all of the messages in the new subdirectory to the cur subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in cur has an "info" section added to its file name to store information about its state. (Some mail readers may also add an "info" section to messages in new.) The "info" section may take one of two forms: it may contain "2," followed by a list of standardized flags (e.g., "2,FR") or it may contain "1," followed by so-called experimental information. Standard flags for Maildir messages are as follows:

Flag	Meaning	Explanation
D	Draft	Under composition
F	Flagged	Marked as important
P	Passed	Forwarded, resent, or bounced
R	Replied	Replied to
S	Seen	Read

T	Trashed	Marked for subsequent deletion

`MaildirMessage` instances offer the following methods:

`get_subdir`()
: Return either “new” (if the message should be stored in the `new` subdirectory) or “cur” (if the message should be stored in the `cur` subdirectory).

> **Note:** A message is typically moved from `new` to `cur` after its mailbox has been accessed, whether or not the message is has been read. A message `msg` has been read if `"S" in msg.get_flags()` is `True`.

`set_subdir`(*subdir*)
: Set the subdirectory the message should be stored in. Parameter *subdir* must be either “new” or “cur”.

`get_flags`()
: Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of `'D'`, `'F'`, `'P'`, `'R'`, `'S'`, and `'T'`. The empty string is returned if no flags are set or if “info” contains experimental semantics.

`set_flags`(*flags*)
: Set the flags specified by *flags* and unset all others.

`add_flag`(*flag*)
: Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character. The current “info” is overwritten whether or not it contains experimental information rather than flags.

`remove_flag`(*flag*)
: Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character. If “info” contains experimental information rather than flags, the current “info” is not modified.

`get_date`()
: Return the delivery date of the message as a floating-point number representing seconds since the epoch.

`set_date`(*date*)
: Set the delivery date of the message to *date*, a floating-point number representing seconds since the epoch.

`get_info`()
: Return a string containing the “info” for a message. This is useful for accessing

and modifying “info” that is experimental (i.e., not a list of flags).

set_info(*info*)

Set “info” to *info*, which should be a string.

When a `MaildirMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	`mboxMessage` or `MMDFMessage` state
“cur” subdirectory	O flag
F flag	F flag
R flag	A flag
S flag	R flag
T flag	D flag

When a `MaildirMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

Resulting state	`MHMessage` state
“cur” subdirectory	“unseen” sequence
“cur” subdirectory and S flag	no “unseen” sequence
F flag	“flagged” sequence
R flag	“replied” sequence

When a `MaildirMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

Resulting state	`BabylMessage` state
“cur” subdirectory	“unseen” label
“cur” subdirectory and S flag	no “unseen” label
P flag	“forwarded” or “resent” label
R flag	“answered” label
T flag	“deleted” label

18.4.2.2. `mboxMessage`

class `mailbox.`**mboxMessage**([*message*])

A message with mbox-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

Messages in an mbox mailbox are stored together in a single file. The sender’s envelope address and the time of delivery are typically stored in a line beginning with “From ” that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as

important, are typically stored in *Status* and *X-Status* headers.

Conventional flags for mbox messages are as follows:

Flag	Meaning	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

`mboxMessage` instances offer the following methods:

`get_from`()

Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

`set_from`(*from_*, *time_=None*)

Set the “From ” line to *from_*, which should be specified without a leading “From ” or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a `time.struct_time` instance, a tuple suitable for passing to `time.strftime()`, or `True` (to use `time.gmtime()`).

`get_flags`()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

`set_flags`(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

`add_flag`(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

`remove_flag`(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an `mboxMessage` instance is created based upon a `MaildirMessage` instance, a “From ” line is generated based upon the `MaildirMessage` instance’s delivery date, and the

following conversions take place:

Resulting state	`MaildirMessage` state
R flag	S flag
O flag	“cur” subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an `mboxMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

Resulting state	`MHMessage` state
R flag and O flag	no “unseen” sequence
O flag	“unseen” sequence
F flag	“flagged” sequence
A flag	“replied” sequence

When an `mboxMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

Resulting state	`BabylMessage` state
R flag and O flag	no “unseen” label
O flag	“unseen” label
D flag	“deleted” label
A flag	“answered” label

When a `Message` instance is created based upon an `MMDFMessage` instance, the “From ” line is copied and all flags directly correspond:

Resulting state	`MMDFMessage` state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

18.4.2.3. `MHMessage`

class `mailbox.`**`MHMessage`**([*message*])

A message with MH-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard **mh** and **nmh**) use sequences in much

the same way flags are used with other formats, as follows:

Sequence	Explanation
unseen	Not read, but previously detected by MUA
replied	Replied to
flagged	Marked as important

`MHMessage` instances offer the following methods:

`get_sequences`()
Return a list of the names of sequences that include this message.

`set_sequences`(*sequences*)
Set the list of sequences that include this message.

`add_sequence`(*sequence*)
Add *sequence* to the list of sequences that include this message.

`remove_sequence`(*sequence*)
Remove *sequence* from the list of sequences that include this message.

When an `MHMessage` instance is created based upon a `MaildirMessage` instance, the following conversions take place:

Resulting state	`MaildirMessage` state
“unseen” sequence	no S flag
“replied” sequence	R flag
“flagged” sequence	F flag

When an `MHMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	`mboxMessage` or `MMDFMessage` state
“unseen” sequence	no R flag
“replied” sequence	A flag
“flagged” sequence	F flag

When an `MHMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

Resulting state	`BabylMessage` state
“unseen” sequence	“unseen” label
“replied” sequence	“answered” label

18.4.2.4. `BabylMessage`

class `mailbox.`**`BabylMessage`**([*message*])

A message with Babyl-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows:

Label	Explanation
unseen	Not read, but previously detected by MUA
deleted	Marked for subsequent deletion
filed	Copied to another file or mailbox
answered	Replied to
forwarded	Forwarded
edited	Modified by the user
resent	Resent

By default, Rmail displays only visible headers. The `BabylMessage` class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

`BabylMessage` instances offer the following methods:

`get_labels`()

Return a list of labels on the message.

`set_labels`(*labels*)

Set the list of labels on the message to *labels*.

`add_label`(*label*)

Add *label* to the list of labels on the message.

`remove_label`(*label*)

Remove *label* from the list of labels on the message.

`get_visible`()

Return an `Message` instance whose headers are the message's visible headers and whose body is empty.

`set_visible`(*visible*)

Set the message's visible headers to be the same as the headers in *message*. Parameter *visible* should be a `Message` instance, an `email.message.Message` instance, a string, or a file-like object (which should be open in text mode).

`update_visible`()

When a `BabylMessage` instance's original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original

header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of *Date*, *From*, *Reply-To*, *To*, *CC*, and *Subject* that are present in the original headers but not the visible headers are added to the visible headers.

When a `BabylMessage` instance is created based upon a `MaildirMessage` instance, the following conversions take place:

Resulting state	`MaildirMessage` state
“unseen” label	no S flag
“deleted” label	T flag
“answered” label	R flag
“forwarded” label	P flag

When a `BabylMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	`mboxMessage` or `MMDFMessage` state
“unseen” label	no R flag
“deleted” label	D flag
“answered” label	A flag

When a `BabylMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

Resulting state	`MHMessage` state
“unseen” label	“unseen” sequence
“answered” label	“replied” sequence

18.4.2.5. `MMDFMessage`

class `mailbox.`**`MMDFMessage`**([*message*])

A message with MMDF-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender’s address and the delivery date in an initial line beginning with “From ”. Likewise, flags that indicate the state of the message are typically stored in *Status* and *X-Status* headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

Flag	Meaning	Explanation
R	Read	Read
O	Old	Previously detected by MUA

D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

`MMDFMessage` instances offer the following methods, which are identical to those offered by `mboxMessage`:

get_from()

Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

Set the “From ” line to *from_*, which should be specified without a leading “From ” or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a `time.struct_time` instance, a tuple suitable for passing to `time.strftime()`, or `True` (to use `time.gmtime()`).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an `MMDFMessage` instance is created based upon a `MaildirMessage` instance, a “From ” line is generated based upon the `MaildirMessage` instance’s delivery date, and the following conversions take place:

Resulting state	`MaildirMessage` state
R flag	S flag
O flag	“cur” subdirectory
D flag	T flag

F flag	F flag
A flag	R flag

When an `MMDFMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

Resulting state	`MHMessage` state
R flag and O flag	no “unseen” sequence
O flag	“unseen” sequence
F flag	“flagged” sequence
A flag	“replied” sequence

When an `MMDFMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

Resulting state	`BabylMessage` state
R flag and O flag	no “unseen” label
O flag	“unseen” label
D flag	“deleted” label
A flag	“answered” label

When an `MMDFMessage` instance is created based upon an `mboxMessage` instance, the “From ” line is copied and all flags directly correspond:

Resulting state	`mboxMessage` state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

18.4.3. Exceptions

The following exception classes are defined in the `mailbox` module:

exception `mailbox.`**`Error`**

The based class for all other module-specific exceptions.

exception `mailbox.`**`NoSuchMailboxError`**

Raised when a mailbox is expected but is not found, such as when instantiating a `Mailbox` subclass with a path that does not exist (and with the *create* parameter set to `False`), or when opening a folder that does not exist.

exception `mailbox.`**`NotEmptyError`**

Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

exception mailbox.**ExternalClashError**

Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely-generated file name already exists.

exception mailbox.**FormatError**

Raised when the data in a file cannot be parsed, such as when an **MH** instance attempts to read a corrupted `.mh_sequences` file.

18.4.4. Deprecated classes and methods

Deprecated since version 2.6.

Older versions of the **mailbox** module do not support modification of mailboxes, such as adding or removing message, and do not provide classes to represent format-specific message properties. For backward compatibility, the older mailbox classes are still available, but the newer classes should be used in preference to them. The old classes have been removed in Python 3.

Older mailbox objects support only iteration and provide a single public method:

oldmailbox.**next**()

Return the next message in the mailbox, created with the optional *factory* argument passed into the mailbox object's constructor. By default this is an **rfc822.Message** object (see the **rfc822** module). Depending on the mailbox implementation the *fp* attribute of this object may be a true file object or a class instance simulating a file object, taking care of things like message boundaries if multiple mail messages are contained in a single file, etc. If no more messages are available, this method returns `None`.

Most of the older mailbox classes have names that differ from the current mailbox class names, except for **Maildir**. For this reason, the new **Maildir** class defines a **next()** method and its constructor differs slightly from those of the other new mailbox classes.

The older mailbox classes whose names are not the same as their newer counterparts are as follows:

class mailbox.**UnixMailbox**(*fp*[, *factory*])

Access to a classic Unix-style mailbox, where all messages are contained in a single file and separated by `From` (a.k.a. `From_`) lines. The file object *fp* points to the mailbox file. The optional *factory* parameter is a callable that should create new message objects. *factory* is called with one argument, *fp* by the **next()** method of the mailbox object. The default is the **rfc822.Message** class (see the **rfc822** module – and the note below).

Note: For reasons of this module's internal implementation, you will probably want to open the *fp* object in binary mode. This is especially important on

Windows.

For maximum portability, messages in a Unix-style mailbox are separated by any line that begins exactly with the string `'From '` (note the trailing space) if preceded by exactly two newlines. Because of the wide-range of variations in practice, nothing else on the `From_` line should be considered. However, the current implementation doesn't check for the leading two newlines. This is usually fine for most applications.

The **UnixMailbox** class implements a more strict version of `From_` line checking, using a regular expression that usually correctly matched `From_` delimiters. It considers delimiter line to be separated by `From name time` lines. For maximum portability, use the **PortableUnixMailbox** class instead. This class is identical to **UnixMailbox** except that individual messages are separated by only `From` lines.

class `mailbox.`**PortableUnixMailbox**(*fp*[, *factory*])

A less-strict version of **UnixMailbox**, which considers only the `From` at the beginning of the line separating messages. The "*name time*" portion of the From line is ignored, to protect against some variations that are observed in practice. This works since lines in the message which begin with `'From '` are quoted by mail handling software at delivery-time.

class `mailbox.`**MmdfMailbox**(*fp*[, *factory*])

Access an MMDF-style mailbox, where all messages are contained in a single file and separated by lines consisting of 4 control-A characters. The file object *fp* points to the mailbox file. Optional *factory* is as with the **UnixMailbox** class.

class `mailbox.`**MHMailbox**(*dirname*[, *factory*])

Access an MH mailbox, a directory with each message in a separate file with a numeric name. The name of the mailbox directory is passed in *dirname*. *factory* is as with the **UnixMailbox** class.

class `mailbox.`**BabylMailbox**(*fp*[, *factory*])

Access a Babyl mailbox, which is similar to an MMDF mailbox. In Babyl format, each message has two sets of headers, the *original* headers and the *visible* headers. The original headers appear before a line containing only `'*** EOOH ***'` (End-Of-Original-Headers) and the visible headers appear after the `EOOH` line. Babyl-compliant mail readers will show you only the visible headers, and **BabylMailbox** objects will return messages containing only the visible headers. You'll have to do your own parsing of the mailbox file to get at the original headers. Mail messages start with the EOOH line and end with a line containing only `'\037\014'`. *factory* is as with the **UnixMailbox** class.

If you wish to use the older mailbox classes with the **email** module rather than the deprecated **rfc822** module, you can do so as follows:

```
import email
import email.Errors
import mailbox
```

```
def msgfactory(fp):
    try:
        return email.message_from_file(fp)
    except email.Errors.MessageParseError:
        # Don't return None since that will
        # stop the mailbox iterator
        return ''

mbox = mailbox.UnixMailbox(fp, msgfactory)
```

Alternatively, if you know your mailbox contains only well-formed MIME messages, you can simplify this to:

```
import email
import mailbox

mbox = mailbox.UnixMailbox(fp, email.message_from_file)
```

18.4.5. Examples

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print subject
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = dict((name, mailbox.mbox('~/email/%s' % name)) for name in list_names)
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
```

```
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```