eletter

Release 0.6.0.dev1

John Thorvald Wodder II

2024 Apr 23

CONTENTS

GitHub | PyPI | Documentation | Issues | *Changelog*

CHAPTER

ONE

TUTORIAL

1.1 Basic Composition

`eletter` can be used to construct a basic text e-mail using the *`compose()`* function like so:

```
from eletter import compose

msg = compose(
    subject="The subject of the e-mail",
    from_="sender@domain.com",
    to=["recipient@domain.com", "another.recipient@example.nil"],
    text="This is the body of the e-mail.  Write what you want here!\n",
)
```

Note: Observe that the `from_` argument is spelled with an underscore. It has to be this way, because plain old `from` is a keyword in Python.

If you want to construct an HTML e-mail, use the `html` keyword instead of `text`:

```
from eletter import compose

msg = compose(
    subject="The subject of the e-mail",
    from_="sender@domain.com",
    to=["recipient@domain.com", "another.recipient@example.nil"],
    html=(
        "<p>This is the <strong>body</strong> of the <em>e</em>-mail."
        "  <span style='color: red;'>Write what you want here!</span></p>\n"
    ),
)
```

By specifying both `text` and `html`, you'll get an e-mail whose HTML part is displayed if the e-mail reader supports it and whose text part is displayed instead on lower-tech clients.

```
from eletter import compose

msg = compose(
    subject="The subject of the e-mail",
    from_="sender@domain.com",
```

(continues on next page)

(continued from previous page)

```
    to=["recipient@domain.com", "another.recipient@example.nil"],
    text="This is displayed on plain text clients.\n",
    html="<p>This is displayed on graphical clients.<p>\n",
)
```

1.2 Addresses

In the examples so far, e-mail addresses have just been specified as, well, addresses. However, addresses usually belong to people or organizations with names; we can include these names alongside the addresses by constructing *Address* objects from pairs of "display names" and e-mail addresses:

```
from eletter import Address, compose

msg = compose(
    subject="The subject of the e-mail",
    from_=Address("Sender's name goes here", "sender@domain.com"),
    to=[
        Address("Joe Q. Recipient", "recipient@domain.com"),
        Address("Jane Z. Another-Recipient", "another.recipient@example.nil"),
    ],
    text="This is the body of the e-mail.  Write what you want here!\n",
)
```

Sometimes addresses come in named groups. We can represent these with the *Group* class, which takes a name for the group and an iterable of address strings and/or *Address* objects:

```
from eletter import Address, Group, compose

msg = compose(
    subject="The subject of the e-mail",
    from_="sender@domain.com",
    to=[
        Group(
            "friends",
            [
                Address("Joe Q. Recipient", "recipient@domain.com"),
                Address("Jane Z. Another-Recipient", "another.recipient@example.nil"),
                "anonymous@nowhere.nil",
            ],
        ),
        Address("Mr. Not-in-a-Group", "ungrouped@unkno.wn"),
        Group(
            "enemies",
            [
                "that.guy@over.there",
                "knows.what.they.did@ancient.history",
                Address("Anemones", "sea.flora@ocean.net"),
            ],
        ),
    ],
```

(continues on next page)

(continued from previous page)

```
    text="This is the body of the e-mail.  Write what you want here!\n",
)
```

1.3 CC, BCC, etc.

Besides *From* and *To* addresses, *compose()* also accepts optional arguments for *CC*, *BCC*, *Reply-To*, and *Sender* addresses:

```
from eletter import Address, compose

msg = compose(
    from_=Address("Mme E.", "me@here.com"),
    to=["you@there.net", Address("Thaddeus Hem", "them@hither.yon")],
    cc=[Address("Cee Cee Cecil", "ccc@seesaws.cc"), "coco@nu.tz"],
    bcc=[
        "eletter@depository.nil",
        Address("Secret Cabal", "illuminati@new.world.order"),
        "mom@house.home",
    ],
    reply_to="replyee@some.where",
    sender="steven.ender@big.senders",
    subject="To: Everyone",
    text="Meeting tonight!  You know the place.  Bring pizza.\n",
)
```

Note: The `to`, `cc`, and `bcc` arguments always take lists or iterables of addresses. `from_` and `reply_to`, on the other hand, can be set to either a single address or an iterable of addresses. `sender` must always be a single address.

1.4 Attachments

Attachments come in two common types: text and binary. `eletter` provides a class for each, *TextAttachment* and *BytesAttachment*.

We can construct a *BytesAttachment* as follows:

```
from eletter import BytesAttachment

attachment = BytesAttachment(
    b'... binary data goes here ...',
    filename="name-of-attachment.dat"
)
```

This will create an *application/octet-stream* attachment with an "attachment" disposition (meaning that most clients will just display it as a clickable icon). To set the content type to something more informative, set the `content_type` parameter to the relevant MIME type. To have the attachment displayed inline (generally only an option for images & videos), set the `inline` parameter to true. Hence:

```
from eletter import BytesAttachment

attachment = BytesAttachment(
    b'... binary data goes here ...',
    filename="name-of-attachment.png"
    content_type="image/png",
    inline=True,
)
```

If your desired attachment exists as a file on your system, you can construct a *BytesAttachment* from the file directly with the *from_file()* classmethod:

```
from eletter import BytesAttachment

attachment = BytesAttachment.from_file(
    "path/to/file.dat",
    content_type="application/x-custom",
    inline=True,
)
```

The basename of the given file will be used as the filename of the attachment. (If you want to use a different name, set the *filename* attribute on the attachment after creating it.) If `content_type` is not given, the MIME type of the file will be guessed based on its file extension.

The *TextAttachment* class is analogous to *BytesAttachment*, except that it is constructed from a `str` instead of `bytes`, and the `content_type` (which defaults to `"text/plain"`) must be a *text* type.

Once you've created some attachment objects, they can be attached to an e-mail by passing them in a list as the `attachments` argument:

```
from eletter import BytesAttachment, TextAttachment, compose

spreadsheet = TextAttachment.from_file("income.csv")
image = BytesAttachment.from_file("cat.jpg")

msg = compose(
    subject="That data you wanted",
    from_="sender@domain.com",
    to=["recipient@domain.com"],
    text="Here's that data you wanted, sir.  And also the ... other thing.\n",
    attachments=[spreadsheet, image],
)
```

1.5 Attaching E-mails to E-mails

On rare occasions, you may have an e-mail that you want to completely embed in a new e-mail as an attachment. With `eletter`, you can do this with the *EmailAttachment* class. It works the same as *BytesAttachment* and *TextAttachment*, except that the content must be an `email.message.EmailMessage` instance, and you can't set the *Content-Type* (which is always *message/rfc822*). Like the other attachment classes, *EmailAttachment* also has a *from_file()* classmethod for constructing an instance from an e-mail in a file.

1.6 Date and Extra Headers

compose() takes two more parameters that we haven't mentioned yet. First is `date`, which lets you set the *Date* header in an e-mail to a given `datetime.datetime` instance. Second is `headers`, which lets you set arbitrary extra headers on an e-mail by passing in a `dict`. Each value in the `dict` must be either a string or (if you want to set multiple headers with the same name) an iterable of strings.

```
from datetime import datetime
from eletter import compose

msg = compose(
    subject="The subject of the e-mail",
    from_="sender@domain.com",
    to=["recipient@domain.com", "another.recipient@example.nil"],
    text="This is the body of the e-mail.  Write what you want here!\n",
    date=datetime(2021, 3, 10, 17, 56, 36).astimezone(),
    headers={
        "User-Agent": "My Mail Application v.1",
        "Priority": "urgent",
        "Comments": [
            "I like e-mail.",
            "But no one ever looks at e-mails' sources, so no one will ever know.",
        ]
    },
)
```

1.7 *multipart/mixed* Messages

All the e-mails constructed so far, when viewed in an e-mail client, have their attachments listed at the bottom. What if we want to mix & match attachments and text, switching from text to an attachment and then back to text? `eletter` lets you do this by providing *TextBody* and *HTMLBody* classes that can be &-ed with attachments to produce *multipart/mixed* messages, like so:

```
from eletter import BytesAttachment, TextBody

part1 = TextBody("Look at the pretty kitty!\n")

snuffles = BytesAttachment.from_file("snuffles.jpeg", inline=True)

part2 = TextBody("Now look at this dog.\n")

rags = BytesAttachment.from_file("rags.jpeg", inline=True)

part3 = TextBody("Which one is cuter?\n")

mixed = part1 & snuffles & part2 & rags & part3
```

We can then convert `mixed` into an `EmailMessage` by calling its *compose()* method, which takes the same arguments as the *compose()* function, minus `text`, `html`, and `attachments`.

```
msg = mixed.compose(
    subject="The subject of the e-mail",
    from_="sender@domain.com",
    to=["recipient@domain.com", "another.recipient@example.nil"],
)
```

When the resulting e-mail is viewed in a client, you'll see three lines of text with images between them.

Tip: As a shortcut, you can combine a bare `str` with an `eletter` object using `|` or the other overloaded operators described below (`&` and `^`), and that `str` will be automatically converted to a *`TextBody`*. The example above could thus be rewritten:

```
from eletter import BytesAttachment, TextBody

snuffles = BytesAttachment.from_file("snuffles.jpeg", inline=True)

rags = BytesAttachment.from_file("rags.jpeg", inline=True)

mixed = (
    "Look at the pretty kitty!\n"
    & snuffles
    & "Now look at this dog.\n"
    & rags
    & "Which one is cuter?\n"
)
```

1.8 *multipart/alternative* Messages

Now that we know how to construct mixed messages, how do we use them to create messages with both mixed-HTML and mixed-text payloads where the client shows whichever mixed payload it can support? The answer is the `|` operator; using it to combine two `eletter` objects will give you a *`multipart/alternative`* object, representing an e-mail message with two different versions of the same content that the client will then pick between.

```
from eletter import BytesAttachment, HTMLBody, TextBody

text1 = TextBody("Look at the pretty kitty!\n")
text2 = TextBody("Now look at this dog.\n")
text3 = TextBody("Which one is cuter?\n")

html1 = HTMLBody("<p>Look at the <em>pretty kitty</em>!</p>\n")
html2 = HTMLBody("<p>Now look at this <strong>dog</strong>.</p>\n")
html3 = HTMLBody("<p>Which one is <span style='color: pink'>cuter</span>?</p>\n")

snuffles = BytesAttachment.from_file("snuffles.jpeg", inline=True)
rags = BytesAttachment.from_file("rags.jpeg", inline=True)

mixed_text = text1 & snuffles & text2 & rags & text3
mixed_html = html1 & snuffles & html2 & rags & html3

alternative = mixed_text | mixed_html
```

The `alternative` object can then be converted to an e-mail with the same *`compose()`* method that mixed objects have.

Tip: In this specific example, we can save on e-mail size by instead creating a mixed message containing alternative parts, like so:

```
mixed = (text1 | html1) & snuffles & (text2 | html2) & rags & (text3 | html3)
```

Tip: The parts of a *`multipart/alternative`* message should generally be placed in increasing order of preference, which means that the text part should be on the left of the `|` and the HTML part should be on the right.

1.9 *`multipart/related`* Messages

Mixing plain text and attachments is all well and good, but when it comes to HTML, it'd be better if we could reference attachments directly in, say, an `<img>` tag's `src` attribute. We can do this in three steps:

1. Assign each attachment's `content_id` attribute a unique ID generated with `email.utils.make_msgid`.
2. Within the HTML document, refer to a given attachment via the URI `cid:{content_id[1:-1]}` — that is, "`cid:`" followed by the attachment's `content_id` with the leading & trailing angle brackets stripped off.
3. Combine the HTML body with the attachments using the `^` operator to make a *`multipart/related`* object. The HTML body should be on the left end of the operator chain!

Example:

```
from email.utils import make_msgid
from eletter import BytesAttachment, HTMLBody

snuffles_cid = make_msgid()
rags_cid = make_msgid()

html = HTMLBody(f"""
    <p>Look at the <em>pretty kitty</em>!

    <div class="align: center;">
        <img src="cid:{snuffles_cid[1:-1]}" width="500" height="500"
             style="border: 1px solid blue;" />
    </div>

    <p>Now look at this <strong>dog</strong>.</p>

    <div class="align: center;">
        <img src="cid:{rags_cid[1:-1]}" width="500" height="200"
             style="border: 1px solid red;" />
    </div>

    <p>Which one is <span style="color: pink">cuter</span>?</p>
""")
```

(continues on next page)

(continued from previous page)

```
snuffles = BytesAttachment.from_file("snuffles.jpeg", inline=True, content_id=snuffles_
↪cid)
rags = BytesAttachment.from_file("rags.jpeg", inline=True, content_id=rags_cid)

related = html ^ snuffles ^ rags
```

Tip: You can remember the fact that *multipart/related* objects use `^` by association with *Content-ID*s, which are enclosed in <...>, which look like a sideways `^`!

Like mixed & alternative objects, `related` can then be converted to an e-mail with the *compose()* method. If you want, you can even use `|` to combine it with a mixed-text message before composing.

1.10 Sending E-mails

Once you've constructed your e-mail and turned it into an `EmailMessage` object, you can send it using Python's `smtplib`, `imaplib`, or `mailbox` modules or using a compatible third-party library. Actually doing this is beyond the scope of this tutorial & library, but may I suggest outgoing, by yours truly?

1.11 Decomposing Emails

Added in version 0.5.0.

If you have an `email.message.EmailMessage` instance (either composed using `eletter` or acquired elsewhere) and you want to convert it into an `eletter` structure to make it easier to work with, `eletter` provides a *decompose()* function for doing just that. Calling *decompose()* on an `EmailMessage` produces an *Eletter* instance that has attributes for all of the fields accepted by the *compose()* method plus a *content* field containing an `eletter` class.

Tip: If you want to decompose a message that is a plain `email.message.Message` instance but not an `EmailMessage` instance, you need to first convert it into an `EmailMessage` before passing it to *decompose()* or `decompose_simplify()`. This can be done with the `message2email()` function from the mailbits package.

If you want to decompose a message even further, you can call the *decompose_simple()* function on an `EmailMessage` or call the *simplify()* method of an *Eletter* to produce a *SimpleEletter* instance. In place of a `content` attribute, a *SimpleEletter* has *text*, *html*, and *attachments* attributes giving the original message's text and HTML bodies plus any attachments.

Once you've decomposed and/or simplified a message, you can examine its parts and do whatever you want with that information. You can also manually modify the *Eletter*/*SimpleEletter*'s various attributes and then call its *compose()* method (which takes no arguments) to recompose the instance into a modified `EmailMessage`. Note that the attributes are of stricter types than what is accepted by the corresponding arguments to the *compose()* function. In particular, addresses must be specified as *Address* instances, not as strings[1], the *from_* and *reply_to* attributes must always be lists, and the values of the *headers* attribute must always be lists.

Note: Most `EmailMessage` instances can be decomposed into *Eletter* instances; those that can't use *Content-Type*s not supported by `eletter`, i.e., *message* types other than *message/rfc822* or *multipart* types

[1] An e-mail address without a display name can be represented as an *Address* object by setting the display name to the empty string: `Address("", "user@domain.nil")`.

other than *multipart/alternative*, *multipart/mixed*, and *multipart/related*.

On the other hand, considerably fewer `EmailMessage` instances can be simplified into *SimpleEletter* instances. Messages that cannot be simplified include messages without plain text or HTML parts, mixed messages that alternate between plain text & HTML without supplying both types for every body part, *multipart/related* messages with more than one part, *multipart/mixed* messages containing *multipart/alternative* parts that do not consist of a plain text body plus an HTML body, and other unusual things. Trying to simplify such messages will produce *SimplificationError*s.

One category of messages can be simplified, but not without loss of information, and so they are not simplified by default: namely, *multipart/mixed* messages that alternate between bodies and attachments rather than placing all attachments at the end of the message. By default, trying to simplify such a message produces a *MixedContentError*; however, if the `unmix` argument to *decompose_simple()* or *Eletter.simplify()* is set to `True`, such messages will instead be simplified by separating the attachments from the bodies, which are then concatenated with no indication of where the attachments were located in the text.

CHAPTER

TWO

API

2.1 The `compose()` Function

eletter.**compose**(*, *to: Iterable[str | Address | Group]*, *from_: str | Address | Group | Iterable[str | Address | Group] | None = None*, *subject: str | None = None*, *text: str | None = None*, *html: str | None = None*, *cc: Iterable[str | Address | Group] | None = None*, *bcc: Iterable[str | Address | Group] | None = None*, *reply_to: str | Address | Group | Iterable[str | Address | Group] | None = None*, *sender: str | Address | None = None*, *date: datetime | None = None*, *headers: Mapping[str, str | Iterable[str]] | None = None*, *attachments: Iterable[*Attachment*] | None = None*) → EmailMessage

Construct an `EmailMessage` instance from a subject, *`From`* address, *`To`* addresses, and a plain text and/or HTML body, optionally accompanied by attachments and other headers.

All parameters other than `to` and at least one of `text` and `html` are optional.

Changed in version 0.2.0: `from_` and `reply_to` may now be passed lists of addresses.

Changed in version 0.4.0: `from_` may now be `None` or omitted.

Changed in version 0.4.0: All arguments are now keyword-only.

Changed in version 0.5.0: `subject` may now be `None` or omitted.

Parameters

- **subject** (*str*) – The e-mail's *`Subject`* line
- **to** (*iterable of addresses*) – The e-mail's *`To`* line
- **from_** (*address or iterable of addresses*) – The e-mail's *`From`* line. Note that this argument is spelled with an underscore, as "`from`" is a keyword in Python.
- **text** (*str*) – The contents of a *`text/plain`* body for the e-mail. At least one of `text` and `html` must be specified.
- **html** (*str*) – The contents of a *`text/html`* body for the e-mail. At least one of `text` and `html` must be specified.
- **cc** (*iterable of addresses*) – The e-mail's *`CC`* line
- **bcc** (*iterable of addresses*) – The e-mail's *`BCC`* line
- **reply_to** (*address or iterable of addresses*) – The e-mail's *`Reply-To`* line
- **sender** (*address*) – The e-mail's *`Sender`* line. The address must be a string or *`Address`*, not a *`Group`*.
- **date** (*datetime*) – The e-mail's *`Date`* line

- **attachments** (*iterable of attachments*) – A collection of *attachments* to append to the e-mail
- **headers** (*mapping*) – A collection of additional headers to add to the e-mail. A header value may be either a single string or an iterable of strings to add multiple headers with the same name. If you wish to set an otherwise-unsupported address header like *Resent-From* to a list of addresses, use the *format_addresses()* function to first convert the addresses to a string.

Return type
email.message.EmailMessage

Raises
ValueError – if neither `text` nor `html` is set

2.2 Addresses

Addresses in `eletter` can be specified in three ways:

- As an `"address@domain.com"` string giving just a bare e-mail address
- As an `eletter.Address("Display Name", "address@domain.com")` instance pairing a person's name with an e-mail address
- As an `eletter.Group("Group Name", iterable_of_addresses)` instance specifying a group of addresses (strings or `Address` instances)

Note: *eletter.Address* and *eletter.Group* are actually just subclasses of `Address` and `Group` from `email.headerregistry` with slightly more convenient constructors. You can also use the standard library types directly, if you want to.

---

class eletter.**Address**(*display_name: str*, *address: str*)

A combination of a person's name and their e-mail address

class eletter.**Group**(*display_name: str*, *addresses: Iterable[str | Address]*)

Added in version 0.2.0.

An e-mail address group

2.3 `MailItem` Classes

class eletter.**MailItem**

Added in version 0.3.0.

Base class for all `eletter` message components

compose(*, *to: Iterable[str | Address | Group]*, *from_: str | Address | Group | Iterable[str | Address | Group] | None = None*, *subject: str | None = None*, *cc: Iterable[str | Address | Group] | None = None*, *bcc: Iterable[str | Address | Group] | None = None*, *reply_to: str | Address | Group | Iterable[str | Address | Group] | None = None*, *sender: str | Address | None = None*, *date: datetime | None = None*, *headers: Mapping[str, str | Iterable[str]] | None = None*) → EmailMessage

Convert the *MailItem* into an `EmailMessage` with the item's contents as the payload and with the given subject, *From* address, *To* addresses, and optional other headers.

All parameters other than `to` are optional.

Changed in version 0.4.0: `from_` may now be `None` or omitted.

Changed in version 0.4.0: All arguments are now keyword-only.

Changed in version 0.5.0: `subject` may now be `None` or omitted.

Parameters

- **subject** (*str*) – The e-mail's *Subject* line
- **to** (*iterable of addresses*) – The e-mail's *To* line
- **from_** (*address or iterable of addresses*) – The e-mail's *From* line. Note that this argument is spelled with an underscore, as "`from`" is a keyword in Python.
- **cc** (*iterable of addresses*) – The e-mail's *CC* line
- **bcc** (*iterable of addresses*) – The e-mail's *BCC* line
- **reply_to** (*address or iterable of addresses*) – The e-mail's *Reply-To* line
- **sender** (*address*) – The e-mail's *Sender* line. The address must be a string or *Address*, not a *Group*.
- **date** (*datetime*) – The e-mail's *Date* line
- **headers** (*mapping*) – A collection of additional headers to add to the e-mail. A header value may be either a single string or an iterable of strings to add multiple headers with the same name. If you wish to set an otherwise-unsupported address header like *Resent-From* to a list of addresses, use the *format_addresses()* function to first convert the addresses to a string.

Return type
email.message.EmailMessage

2.3.1 Attachments

class eletter.**Attachment**

Base class for the attachment classes

class eletter.**BytesAttachment**(*content: bytes*, *filename: str | None*, ***, *content_id: str | None = None*, *content_type: str = NOTHING*, *inline: bool = False*)

A binary e-mail attachment. *content_type* defaults to "`application/octet-stream`".

content: `bytes`

The body of the attachment

content_id: `str | None`

Added in version 0.3.0.

Content-ID header value for the item

content_type: `str`

The *Content-Type* of the attachment

filename: `str | None`

The filename of the attachment

Changed in version 0.5.0: `filename` can now be `None`.

classmethod from_file(*path: bytes | str | PathLike[bytes] | PathLike[str]*, *content_type: str | None = None*, *inline: bool = False*, *content_id: str | None = None*) → *BytesAttachment*

Added in version 0.2.0.

Construct a *BytesAttachment* from the contents of the file at `path`. The filename of the attachment will be set to the basename of `path`. If `content_type` is `None`, the *Content-Type* is guessed based on `path`'s file extension.

Changed in version 0.3.0: `inline` and `content_id` arguments added

inline: `bool`

Whether the attachment should be displayed inline in clients

class eletter.**EmailAttachment**(*content: EmailMessage*, *filename: str | None*, ***, *content_id: str | None = None*, *inline: bool = False*)

Added in version 0.2.0.

A *message/rfc822* e-mail attachment

content: `EmailMessage`

The body of the attachment

content_id: `str | None`

Added in version 0.3.0.

Content-ID header value for the item

filename: `str | None`

The filename of the attachment

Changed in version 0.5.0: `filename` can now be `None`.

classmethod from_file(*path: bytes | str | PathLike[bytes] | PathLike[str]*, *inline: bool = False*, *content_id: str | None = None*) → *EmailAttachment*

Construct an *EmailAttachment* from the contents of the file at `path`. The filename of the attachment will be set to the basename of `path`.

inline: `bool`

Whether the attachment should be displayed inline in clients

class eletter.**TextAttachment**(*content: str*, *filename: str | None*, ***, *content_id: str | None = None*, *content_type: str = NOTHING*, *inline: bool = False*)

A textual e-mail attachment. `content_type` defaults to `"text/plain"` and must have a maintype of *text*.

content: `str`

The body of the attachment

content_id: `str | None`

Added in version 0.3.0.

Content-ID header value for the item

content_type: `str`

The *Content-Type* of the attachment

filename: `str | None`

The filename of the attachment

Changed in version 0.5.0: `filename` can now be `None`.

classmethod from_file(*path: bytes | str | PathLike[bytes] | PathLike[str]*, *content_type: str | None = None*, *encoding: str | None = None*, *errors: str | None = None*, *inline: bool = False*, *content_id: str | None = None*) → *TextAttachment*

Added in version 0.2.0.

Construct a `TextAttachment` from the contents of the file at `path`. The filename of the attachment will be set to the basename of `path`. If `content_type` is `None`, the *Content-Type* is guessed based on `path`'s file extension. `encoding` and `errors` are used when opening the file and have no relation to the *Content-Type*.

Changed in version 0.3.0: `inline` and `content_id` arguments added

inline: `bool`

Whether the attachment should be displayed inline in clients

2.3.2 Body Classes

class eletter.**HTMLBody**(*content: str*, *, *content_id: str | None = None*)

Added in version 0.3.0.

A *text/html* e-mail body

content: `str`

The HTML source of the body

content_id: `str | None`

Added in version 0.3.0.

Content-ID header value for the item

class eletter.**TextBody**(*content: str*, *, *content_id: str | None = None*)

Added in version 0.3.0.

A *text/plain* e-mail body

content: `str`

The plain text body

content_id: `str | None`

Added in version 0.3.0.

Content-ID header value for the item

2.3.3 Multipart Classes

class eletter.**Multipart**

Added in version 0.3.0.

Base class for all multipart classes. All such classes are mutable sequences of `MailItem`s supporting the usual methods (construction from an iterable, subscription, `append()`, `pop()`, etc.).

class eletter.**Alternative**(*content: Iterable[*MailItem*] = NOTHING*, *, *content_id: str | None = None*)

Added in version 0.3.0.

A *multipart/alternative* e-mail payload. E-mails clients will display the resulting payload by choosing whichever part they support best.

An `Alternative` instance can be created by combining two or more `MailItem`s with the `|` operator:

```
text = TextBody("This is displayed on plain text clients.\n")
html = HTMLBody("<p>This is displayed on graphical clients.<p>\n")

alternative = text | html
```

Likewise, additional *MailItem*s can be added to an *Alternative* instance with the |= operator:

```
# Same as above:
alternative = Alternative()
alternative |= TextBody("This is displayed on plain text clients.\n")
alternative |= HTMLBody("<p>This is displayed on graphical clients.<p>\n")
```

Using | to combine a *MailItem* with a `str` automatically converts the `str` to a *TextBody*:

```
# Same as above:

text = "This is displayed on plain text clients.\n"
html = HTMLBody("<p>This is displayed on graphical clients.<p>\n")

alternative = text | html

assert alternative.contents == [
    TextBody("This is displayed on plain text clients.\n"),
    HTMLBody("<p>This is displayed on graphical clients.<p>\n"),
]
```

When combining two *Alternative* instances with | or |=, the contents are "flattened":

```
# Same as above:
txtalt = Alternative([
    TextBody("This is displayed on plain text clients.\n")
])
htmlalt = Alternative([
    HTMLBody("<p>This is displayed on graphical clients.<p>\n")
])
alternative = txtalt | htmlalt
assert alternative.contents == [
    TextBody("This is displayed on plain text clients.\n"),
    HTMLBody("<p>This is displayed on graphical clients.<p>\n"),
]
```

Changed in version 0.4.0: Using | to combine a *MailItem* with a `str` now automatically converts the `str` to a *TextBody*

content: `list`[*MailItem*]

The *MailItem*s contained within the instance

content_id: `str` | `None`

Added in version 0.3.0.

Content-ID header value for the item

class eletter.**Mixed**(*content: Iterable*[MailItem] = *NOTHING*, *, *content_id: str* | *None* = *None*)

Added in version 0.3.0.

A *multipart/mixed* e-mail payload. E-mails clients will display the resulting payload one part after another, with attachments displayed inline if their `inline` attribute is set.

A *Mixed* instance can be created by combining two or more *MailItem*s with the & operator:

```
text = TextBody("Look at the pretty kitty!\n")
image = BytesAttachment.from_file("snuffles.jpeg", inline=True)
sig = TextBody("Sincerely, Me\n")

mixed = text & image & sig
```

Likewise, additional *MailItem*s can be added to a *Mixed* instance with the &= operator:

```
# Same as above:
mixed = Mixed()
mixed &= TextBody("Look at the pretty kitty!\n")
mixed &= BytesAttachment.from_file("snuffles.jpeg", inline=True)
mixed &= TextBody("Sincerely, Me\n")
```

Using & to combine a *MailItem* with a `str` automatically converts the `str` to a *TextBody*:

```
# Same as above:
image = BytesAttachment.from_file("snuffles.jpeg", inline=True)

mixed = "Look at the pretty kitty!\n" & image & "Sincerely, Me\n"

assert mixed.contents == [
    TextBody("Look at the pretty kitty!\n"),
    BytesAttachment.from_file("snuffles.jpeg", inline=True),
    TextBody("Sincerely, Me\n"),
]
```

When combining two *Mixed* instances with & or &=, the contents are "flattened":

```
part1 = Mixed()
part1 &= TextBody("Look at the pretty kitty!\n")
part1 &= BytesAttachment.from_file("snuffles.jpeg", inline=True)

part2 = Mixed()
part2 &= TextBody("Now look at this dog.\n")
part2 &= BytesAttachment.from_file("rags.jpeg", inline=True)
part2 &= TextBody("Which one is cuter?\n")

mixed = part1 & part2

assert mixed.contents == [
    TextBody("Look at the pretty kitty!\n"),
    BytesAttachment.from_file("snuffles.jpeg", inline=True),
    TextBody("Now look at this dog.\n"),
    BytesAttachment.from_file("rags.jpeg", inline=True),
    TextBody("Which one is cuter?\n"),
]
```

Changed in version 0.4.0: Using & to combine a *MailItem* with a `str` now automatically converts the `str` to a *TextBody*

content: list[*MailItem*]

The *MailItem*s contained within the instance

content_id: str | None

Added in version 0.3.0.

Content-ID header value for the item

class eletter.**Related**(*content: Iterable*[MailItem] = *NOTHING*, *start: str* | *None* = *None*, *, *content_id: str* | *None* = *None*)

Added in version 0.3.0.

A *multipart/related* e-mail payload. E-mail clients will display the part indicated by the *start* parameter, or the first part if *start* is not set. This part may refer to other parts (e.g., images or CSS stylesheets) by their *Content-ID* headers, which can be generated using email.utils.make_msgid.

Note: *Content-ID* headers begin & end with angle brackets (<...>), which need to be stripped off before including the ID in the starting part.

A *Related* instance can be created by combining two or more *MailItem*s with the ^ operator:

```
from email.utils import make_msgid

img_cid = make_msgid()

html = HTMLBody(
    "<p>Look at the pretty kitty!</p>"
    f'<img src="cid:{img_cid[1:-1]}"/>"
    "<p>Isn't he <em>darling</em>?</p>"
)

image = BytesAttachment.from_file("snuffles.jpeg", content_id=img_cid)

related = html ^ image
```

Likewise, additional *MailItem*s can be added to a *Related* instance with the ^= operator:

```
# Same as above:

img_cid = make_msgid()

related = Related()

related ^= HTMLBody(
    "<p>Look at the pretty kitty!</p>"
    f'<img src="cid:{img_cid[1:-1]}"/>"
    "<p>Isn't he <em>darling</em>?</p>"
)

related ^= BytesAttachment.from_file("snuffles.jpeg", content_id=img_cid)
```

Using ^ to combine a *MailItem* with a str automatically converts the str to a *TextBody*, though this is generally not all that useful, as you'll usually want to create *Related* instances from *HTMLBody*s instead.

When combining two *Related* instances with ^ or ^=, the contents are "flattened":

```
# Same as above:

img_cid = make_msgid()

htmlrel = Related([
    HTMLBody(
        "<p>Look at the pretty kitty!</p>"
        f'<img src="cid:{img_cid[1:-1]}"/>"
        "<p>Isn't he <em>darling</em>?</p>"
    )
])

imgrel = Related([
    BytesAttachment.from_file("snuffles.jpeg", content_id=img_cid)
])

related = htmlrel ^ imgrel

assert related.contents == [
    HTMLBody(
        "<p>Look at the pretty kitty!</p>"
        f'<img src="cid:{img_cid[1:-1]}"/>"
        "<p>Isn't he <em>darling</em>?</p>"
    ),
    BytesAttachment.from_file("snuffles.jpeg", content_id=img_cid),
]
```

Changed in version 0.4.0: Using `^` to combine a *MailItem* with a `str` now automatically converts the `str` to a *TextBody*

content: `list`[*MailItem*]

The *MailItem*s contained within the instance

content_id: `str` | `None`

Added in version 0.3.0.

Content-ID header value for the item

get_root() → *MailItem*

Added in version 0.5.0.

Retrieves the root part, i.e., the part whose `content_id` equals *start*, or the first part if *start* is not set.

Raises

`ValueError` – if the instance is empty or no part has a `content_id` equaling *start*

start: `str` | `None`

The *Content-ID* of the part to display (defaults to the first part)

2.4 Decomposition

eletter.**decompose**(*msg: EmailMessage*) → *Eletter*

Added in version 0.5.0.

Decompose an `EmailMessage` into an *`Eletter`* instance containing a *`MailItem`* and a collection of headers. Only structures that can be represented by `eletter` classes are supported.

All message parts that are not *`text/plain`*, *`text/html`*, *`multipart/*`*, or *`message/*`* are treated as attachments. Attachments without filenames or an explicit "attachment" *`Content-Disposition`* are treated as inline.

Any information specific to how the message is encoded is discarded (namely, "charset" parameters on *`text/*`* parts, *`Content-Transfer-Encoding`* headers, and *`MIME-Version`* headers).

Headers on message sub-parts that do not have representations on *`MailItem`*s are discarded (namely, everything other than *`Content-Type`*, *`Content-Disposition`*, and *`Content-ID`*).

Raises

- **`TypeError`** – if any sub-part of `msg` is not an `EmailMessage` instance
- ***`DecompositionError`*** – if `msg` contains a part with an unrepresentable *`Content-Type`*

eletter.**decompose_simple**(*msg: EmailMessage*, *unmix: bool = False*) → *SimpleEletter*

Added in version 0.5.0.

Decompose an `EmailMessage` into a *`SimpleEletter`* instance consisting of a text body and/or HTML body, some number of attachments, and a collection of headers. The `EmailMessage` is first decomposed with *`decompose()`* and then simplified by calling *`Eletter.simplify()`*.

By default, a *`multipart/mixed`* message can only be simplified if all of the attachments come after all of the message bodies; set `unmix` to `True` to separate the attachments from the bodies regardless of what order they come in.

Raises

- **`TypeError`** – if any sub-part of `msg` is not an `EmailMessage` instance
- ***`DecompositionError`*** – if `msg` contains a part with an unrepresentable *`Content-Type`*
- ***`SimplificationError`*** – if `msg` cannot be simplified

class eletter.**Eletter**

Added in version 0.5.0.

A decomposed e-mail message

content: *`MailItem`*

The message's body

subject: `str | None`

The message's subject line, if any

from_: `list[Address | Group]`

The message's *`From`* addresses

to: `list[Address | Group]`

The message's *`To`* addresses

cc: `list[Address | Group]`

The message's *`CC`* addresses

bcc: list[Address | Group]

The message's *BCC* addresses

reply_to: list[Address | Group]

The message's *Reply-To* addresses

sender: Address | None

The message's *Sender* address, if any

date: datetime | None

The message's *Date* header, if set

headers: dict[str, list[str]]

Any additional headers on the message. The header names are lowercase.

compose() → EmailMessage

Convert the *Eletter* back into an EmailMessage

simplify(*unmix: bool = False*) → *SimpleEletter*

Simplify the *Eletter* into a *SimpleEletter*, breaking down *Eletter.content* into a text body, HTML body, and a list of attachments.

By default, a *multipart/mixed* message can only be simplified if all of the attachments come after all of the message bodies; set unmix to True to separate the attachments from the bodies regardless of what order they come in.

Raises

SimplificationError – if msg cannot be simplified

class eletter.**SimpleEletter**

Added in version 0.5.0.

A decomposed simple e-mail message, consisting of a text body and/or HTML body plus some number of attachments and headers

text: str | None

The message's text body, if any

html: str | None

The message's HTML body, if any

attachments: list[*Attachment*]

Attachments on the message

subject: str | None

The message's subject line, if any

from_: list[Address | Group]

The message's *From* addresses

to: list[Address | Group]

The message's *To* addresses

cc: list[Address | Group]

The message's *CC* addresses

bcc: list[Address | Group]

The message's *BCC* addresses

reply_to: list[Address | Group]

The message's *Reply-To* addresses

sender: Address | None

The message's *Sender* address, if any

date: datetime | None

The message's *Date* header, if set

headers: dict[str, list[str]]

Any additional headers on the message. The header names are lowercase.

compose() → EmailMessage

Convert the *SimpleEletter* back into an EmailMessage

2.5 Exceptions

exception eletter.**Error**

Bases: Exception

Added in version 0.5.0.

The superclass of all custom exceptions raised by eletter

exception eletter.**DecompositionError**

Bases: *Error*, ValueError

Added in version 0.5.0.

Raised when eletter is asked to decompose an EmailMessage with an unrepresentable *Content-Type*

exception eletter.**SimplificationError**

Bases: *Error*, ValueError

Added in version 0.5.0.

Raised when eletter is asked to simplify a message that cannot be simplified

exception eletter.**MixedContentError**

Bases: *SimplificationError*

Added in version 0.5.0.

Subclass of *SimplificationError* raised when a *multipart/mixed* is encountered in which one or more attachments precede a message body part; such messages can be forced to be simplified by setting the unmix argument of *simplify()* or *decompose_simple()* to True.

2.6 Utility Functions

eletter.**assemble_content_type**(*maintype: str*, *subtype: str*, ***params: str*) → str

Added in version 0.2.0.

Construct a *Content-Type* string from a maintype, subtype, and some number of parameters

Raises

ValueError – if f"{maintype}/{subtype}" is an invalid *Content-Type*

eletter.**format_addresses**(*addresses: Iterable[str | Address | Group]*, *encode: bool = False*) → str

Convert an iterable of e-mail address strings (of the form "foo@example.com", without angle brackets or a display name), Address objects, and/or Group objects into a formatted string. If encode is False (the default), non-ASCII characters are left as-is. If it is True, non-ASCII display names are converted into **RFC 2047** encoded words, and non-ASCII domain names are encoded using Punycode.

eletter.**reply_quote**(*s: str*, *prefix: str = '> '*) → str

Added in version 0.2.0.

Quote a text following the *de facto* standard for replying to an e-mail; that is, prefix each line of the text with "> " (or a custom prefix), and if a line already starts with the prefix, omit any trailing whitespace from the newly-added prefix (so "> already quoted" becomes ">> already quoted").

If the resulting string does not end with a newline, one is added. The empty string is treated as a single line.

CHAPTER

THREE

CHANGELOG

3.1 v0.6.0 (in development)

- Support Python 3.10, 3.11, and 3.12
- Drop support for Python 3.6
- Migrated from setuptools to hatch

3.2 v0.5.0 (2021-03-27)

- Attachments' filenames can now be `None`
- Added a *`decompose()`* function for decomposing an `EmailMessage` into a *`MailItem`* plus headers
- Added a *`decompose_simple()`* function for decomposing an `EmailMessage` into a text body, HTML body, attachments, and headers
- The `subject` argument to the *`compose()`* function & method can now be `None`/omitted
- If an address argument to *`compose()`* is set to an empty list, the corresponding header will no longer be present in the resulting e-mail
- Gave *`Related`* a *`get_root()`* method

3.3 v0.4.0 (2021-03-13)

- Using `|`, `&`, or `^` on a *`MailItem`* and a `str` now automatically converts the `str` to a *`TextBody`*
- The `from_` argument to the *`compose()`* function & method can now be `None`/omitted
- *`format_addresses()`* has been moved to mailbits but is still re-exported from this library for the time being.
- **Breaking**: All arguments to the *`compose()`* function & method are now keyword-only

3.4 v0.3.0 (2021-03-11)

- Gave the `from_file()` classmethods `inline` and `content_id` arguments
- Gave all classes optional `content_id` attributes
- Added *TextBody*, *HTMLBody*, *Alternative*, *Mixed*, and *Related* classes for constructing complex e-mails

3.5 v0.2.0 (2021-03-09)

- Gave *BytesAttachment* and `FileAttachment` each a `from_file()` classmethod
- The `from_` and `reply_to` arguments to *compose()* may now be passed lists of addresses
- Support address groups
- Added *assemble_content_type()*, *format_addresses()*, and *reply_quote()* utility functions
- Added an *EmailAttachment* class

3.6 v0.1.0 (2021-03-09)

Initial release

`eletter` provides functionality for constructing & deconstructing `email.message.EmailMessage` instances without having to touch the needlessly complicated `EmailMessage` class itself. A simple function enables composition of e-mails with text and/or HTML bodies plus attachments, and classes are provided for composing more complex multipart e-mails.

CHAPTER

FOUR

INSTALLATION

`eletter` requires Python 3.7 or higher. Just use pip for Python 3 (You have pip, right?) to install `eletter` and its dependencies:

```
python3 -m pip install eletter
```

CHAPTER

FIVE

EXAMPLES

Constructing an e-mail with the *compose()* function:

```
import eletter

TEXT = (
    "Oh my beloved!\n"
    "\n"
    "Wilt thou dine with me on the morrow?\n"
    "\n"
    "We're having hot pockets.\n"
    "\n"
    "Love, Me\n"
)

HTML = (
    "<p>Oh my beloved!</p>\n"
    "<p>Wilt thou dine with me on the morrow?</p>\n"
    "<p>We're having <strong>hot pockets</strong>.</p>\n"
    "<p><em>Love</em>, Me</p>\n"
)

with open("hot-pocket.png", "rb") as fp:
    picture = eletter.BytesAttachment(
        content=fp.read(),
        filename="enticement.png",
        content_type="image/png",
    )

msg = eletter.compose(
    subject="Meet Me",
    from_="me@here.qq",
    to=[eletter.Address("My Dear", "my.beloved@love.love")],
    text=TEXT,
    html=HTML,
    attachments=[picture],
)
```

`msg` can then be sent like any other `EmailMessage`, say, by using outgoing.

For more complex e-mails, a set of classes is provided. Here is the equivalent of the HTML-with-image e-mail with alternative plain text version from the `email` examples page in the Python docs:

```
from email.utils import make_msgid
import eletter

text = eletter.TextBody(
    "Salut!\n"
    "\n"
    "Cela ressemble à un excellent recipie[1] déjeuner.\n"
    "\n"
    "[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718\n"
    "\n"
    "--Pepé\n"
)

asparagus_cid = make_msgid()

html = eletter.HTMLBody(
    "<html>\n"
    "  <head></head>\n"
    "  <body>\n"
    "    <p>Salut!</p>\n"
    "    <p>Cela ressemble à un excellent\n"
    '        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-'
    'Epicurious-203718">\n'
    "            recipie\n"
    "        </a> déjeuner.\n"
    "    </p>\n"
    f'    <img src="cid:{asparagus_cid[1:-1]}" />\n'
    "  </body>\n"
    "</html>\n"
)

image = eletter.BytesAttachment.from_file(
    "roasted-asparagus.jpg",
    inline=True,
    content_id=asparagus_cid,
)

msg = (text | (html ^ image)).compose(
    subject="Ayons asperges pour le déjeuner",
    from_=eletter.Address("Pepé Le Pew", "pepe@example.com"),
    to=[
        eletter.Address("Penelope Pussycat", "penelope@example.com"),
        eletter.Address("Fabrette Pussycat", "fabrette@example.com"),
    ],
)
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX

R

S

T