

---

**outgoing**  
*Release 0.6.1.dev1*

**John Thorvald Wodder II**

**2024 Apr 23**



# CONTENTS

<b>1</b>	<b>Configuration</b>	<b>3</b>
1.1	The Configuration File . . . . .	3
1.2	Sending Methods . . . . .	3
1.3	Passwords . . . . .	7
<b>2</b>	<b>Core Python API</b>	<b>9</b>
2.1	Functions . . . . .	9
2.2	Sender Objects . . . . .	10
2.3	Exceptions . . . . .	10
<b>3</b>	<b>Command-Line Program</b>	<b>13</b>
3.1	Options . . . . .	13
<b>4</b>	<b>Available Extensions</b>	<b>15</b>
4.1	Sending Methods . . . . .	15
4.2	Password Schemes . . . . .	15
<b>5</b>	<b>Writing Extensions</b>	<b>17</b>
5.1	Writing Sending Methods . . . . .	17
5.2	Writing Password Schemes . . . . .	18
<b>6</b>	<b>Utilities for Extension Authors</b>	<b>19</b>
6.1	Pydantic Types & Models . . . . .	20
<b>7</b>	<b>Changelog</b>	<b>23</b>
7.1	v0.6.1 (in development) . . . . .	23
7.2	v0.6.0 (2023-10-30) . . . . .	23
7.3	v0.5.0 (2023-02-09) . . . . .	23
7.4	v0.4.0 (2022-10-25) . . . . .	23
7.5	v0.3.2 (2022-09-03) . . . . .	24
7.6	v0.3.1 (2022-01-02) . . . . .	24
7.7	v0.3.0 (2021-10-31) . . . . .	24
7.8	v0.2.5 (2021-09-27) . . . . .	24
7.9	v0.2.4 (2021-08-02) . . . . .	24
7.10	v0.2.3 (2021-07-04) . . . . .	24
7.11	v0.2.2 (2021-07-02) . . . . .	24
7.12	v0.2.1 (2021-05-12) . . . . .	25
7.13	v0.2.0 (2021-03-14) . . . . .	25
7.14	v0.1.0 (2021-03-06) . . . . .	25
<b>8</b>	<b>Installation</b>	<b>27</b>

<b>9</b>	<b>Examples</b>	<b>29</b>
<b>10</b>	<b>Indices and tables</b>	<b>31</b>
	<b>Python Module Index</b>	<b>33</b>
	<b>Index</b>	<b>35</b>

[GitHub](#) | [PyPI](#) | [Documentation](#) | [Issues](#) | *[Changelog](#)*



## CONFIGURATION

### 1.1 The Configuration File

`outgoing` reads information on what sending method and parameters to use from a **TOML** or JSON configuration file. The default configuration file is TOML, and its location depends on your OS:

Linux	~/.config/outgoing/outgoing.toml or \$XDG_CONFIG_HOME/outgoing/outgoing.toml
macOS	~/Library/Application Support/outgoing/outgoing.toml
Windows	%USERPROFILE%\AppData\Local\jwodder\outgoing\outgoing.toml

Changed in version 0.5.0: Due to an upgrade to v3 of `platformdirs`, the default configuration path on macOS changed from `~/Library/Preferences/outgoing/outgoing.toml` to `~/Library/Application Support/outgoing/outgoing.toml`.

To find the exact path on your system, after installing `outgoing`, run:

```
python3 -c "from outgoing import get_default_configpath; print(get_default_configpath())"
```

Within the configuration file, all of the `outgoing` settings are contained within a table named “`outgoing`”. This table must include at least a `method` key giving the name of the sending method to use. The rest of the table depends on the method chosen (see below). Unknown or inapplicable keys in the table are ignored.

File & directory paths in the configuration file may start with a tilde (`~`) to refer to a path in the user’s home directory. Any relative paths are resolved relative to the directory containing the configuration file.

### 1.2 Sending Methods

#### 1.2.1 `command`

The `command` method sends an e-mail by passing it as input to a command (e.g., **sendmail**, sold separately).

Configuration fields:

**`command`**

[string or list of strings (optional)] Specify the command to run to send e-mail. This can be either a single command string that will be interpreted by the shell or a list of command arguments that will be executed directly without any shell processing. The default command is `sendmail -i -t`.

---

**Note:** Relative paths in the command will not be resolved by `outgoing` (unlike other paths in the configuration file), as it is not possible to reliably determine what is a path and what is not.

---

Example `command` configuration:

```
[outgoing]
method = "command"
command = ["/usr/local/bin/mysendmail", "-i", "-t"]
```

Another sample configuration:

```
[outgoing]
method = "command"
# A single string will be interpreted by the shell, so metacharacters like
# pipes have their special meanings:
command = "my-mail-munger | ~/some/dir/mysendmail"
```

## 1.2.2 smtp

The `smtp` method sends an e-mail to a server over SMTP.

Configuration fields:

**host**

[string (required)] The domain name or IP address of the server to connect to

**ssl**

[boolean or "starttls" (optional)]

- `true`: Use SSL/TLS from the start of the connection
- `false` (default): Don't use SSL/TLS
- `"starttls"`: After connecting, switch to SSL/TLS with the STARTTLS command

**port**

[integer (optional)] The port on the server to connect to; the default depends on the value of `ssl`:

- `true` — 465
- `false` — 25
- `"starttls"` — 587

**username**

[string (optional)] Username to log into the server with

**password**

[password (optional)] Password to log into the server with; can be given as either a string or a password specifier (see “[Passwords](#)”)

**netrc**

[boolean or filepath (optional)] If `true`, read the username & password from `~/.netrc` instead of specifying them in the configuration file. If a filepath, read the credentials from the given netrc file. If `false`, do not use a netrc file.

Example `smtp` configuration:



```
[outgoing]
method = "smtp"
host = "mx.example.com"
ssl = "starttls"
username = "myname"
password = { "file" = "~/secrets/smtp-password" }
```

Another sample configuration:

```
[outgoing]
method = "smtp"
host = "mail.nil"
port = 1337
ssl = true
# Read username & password from the "mail.nil" entry in this netrc file:
netrc = "~/secrets/net.rc"
```

### 1.2.3 mbox

The mbox method appends e-mails to an mbox file on the local machine.

Configuration fields:

**path**

[filepath (required)] The location of the mbox file. If the file does not exist, it will be created when the sender object is entered.

Example mbox configuration:

```
[outgoing]
method = "mbox"
path = "~/MAIL/inbox"
```

### 1.2.4 maildir

The maildir method adds e-mails to a Maildir mailbox directory on the local machine.

Configuration fields:

**path**

[directory path (required)] The location of the Maildir mailbox. If the directory does not exist, it will be created when the sender object is entered.

**folder**

[string (optional)] A folder within the Maildir mailbox in which to place e-mails

### 1.2.5 mh

The `mh` method adds e-mails to an MH mailbox directory on the local machine.

Configuration fields:

**path**

[directory path (required)] The location of the MH mailbox. If the directory does not exist, it will be created when the sender object is entered.

**folder**

[string or list of strings (optional)] A folder within the Maildir mailbox in which to place e-mails; can be either the name of a single folder or a path through nested folders & subfolders

Example configuration:

```
[outgoing]
method = "mh"
path = "~/mail"
# Place e-mails inside the "work" folder inside the "important" folder:
folder = ["important", "work"]
```

### 1.2.6 mmdf

The `mmdf` method adds e-mails to an MMDF mailbox file on the local machine.

Configuration fields:

**path**

[filepath (required)] The location of the MMDF mailbox. If the file does not exist, it will be created when the sender object is entered.

### 1.2.7 babyl

The `babyl` method adds e-mails to a Babyl mailbox file on the local machine.

Configuration fields:

**path**

[filepath (required)] The location of the Babyl mailbox. If the file does not exist, it will be created when the sender object is entered.

### 1.2.8 null

Goes nowhere, does nothing, ignores all configuration keys.

Example `null` configuration:

```
[outgoing]
# Just send my e-mails into a black hole
method = "null"
```

## 1.3 Passwords

When a sending method calls for a password, API key, or other secret, there are several ways to specify the value.

Using a string, naturally, supplies the value of that string as the password:

```
password = "hunter2"
```

Alternatively, passwords may instead be looked up in external resources. This is done by setting the value of the password field to a table with a single key-value pair, where the key identifies the password lookup scheme and the value is either a string or a sub-table, depending on the scheme.

The builtin password schemes are as follows. Extension packages can define additional password schemes.

### 1.3.1 base64

For slightly more security than a plaintext password, a password can be stored in base64 by specifying a table with a single base64 key and the encoded password as the value:

```
password = { base64 = "aHVudGVyMg==" }
```

Base64 passwords must decode to UTF-8 text.

### 1.3.2 file

A password can be read from a file by specifying a table with a single file key and the filepath as the value:

```
password = { file = "path/to/file" }
```

The entire contents of the file, minus any leading or trailing whitespace, will then be used as the password. As with paths elsewhere in the configuration file, the path may start with a tilde, and relative paths are resolved relative to the directory containing the configuration file.

### 1.3.3 env

A password can be read from an environment variable by specifying a table with a single env key and the name of the environment variable as the value:

```
password = { env = "PROTOCOL_PASSWORD" }
```

### 1.3.4 dotenv

Passwords can be read from a key in a .env-style file as supported by `python-dotenv` like so:

```
password = { dotenv = { key = "NAME_OF_KEY_IN_FILE", file = "path/to/file" } }
```

The file path is resolved following the same rules as other paths. If the file field is omitted, the given key will be looked up in a file named .env in the same directory as the configuration file.

### 1.3.5 keyring

Passwords can be retrieved from the system keyring using [keyring](#). The basic format is:

```
password = { keyring = { service = "host_or_service_name", username = "your_username" } }
```

If the `service` key is omitted, the value will default to the sending method's host value, if it has one; likewise, an omitted `username` will default to the username for the sending method, if there is one. A specific keyring backend can be specified with the `backend` key, and the directory from which to load the backend can be specified with the `keyring-path` key.

## CORE PYTHON API

### 2.1 Functions

outgoing provides the following functions for constructing e-mail sender objects. Once you have a sender object, simply use it in a context manager to open it up, and then call its `send()` method with each `email.message.EmailMessage` object you want to send. See *Examples* for examples.

`outgoing.from_config_file(path: str | bytes | PathLike[str] | PathLike[bytes] | None = None, section: str | None = 'outgoing', fallback: bool = True) → Sender`

Read configuration from the table/field section (default “outgoing”) in the file at `path` (default: the path returned by `get_default_configpath()`) and construct a sender object from the specification. The file may be either TOML or JSON (type detected based on file extension). If `section` is `None`, the entire file, rather than only a single field, is used as the configuration. If `fallback` is true, the file is not the default config file, and the file either does not exist or does not contain the given section, fall back to reading from the default section of the default config file.

**Raises**

- `InvalidConfigError` – if the configuration is invalid
- `MissingConfigError` – if no configuration file or section is present

`outgoing.from_dict(data: Mapping[str, Any], configpath: str | bytes | PathLike[str] | PathLike[bytes] | None = None) → Sender`

Construct a sender object using the given `data` as the configuration. If `configpath` is given, any paths in the `data` will be resolved relative to `configpath`’s parent directory; otherwise, they will be resolved relative to the current directory.

`data` should not contain a “configpath” key; such an entry will be discarded.

**Raises**

- `InvalidConfigError` – if the configuration is invalid

`outgoing.get_default_configpath() → Path`

Returns the location of the default config file (regardless of whether it exists) as a `pathlib.Path` object

## 2.2 Sender Objects

**class** outgoing.**Sender**

*Sender* is a *Protocol* implemented by sender objects. The protocol requires the following behavior:

- Sender objects can be used as context managers, and their `__enter__` methods return `self`.
- Within its own context, calling a sender's `send(msg: email.message.EmailMessage)` method sends the given e-mail.

`__enter__()` → *Self*

`__exit__(exc_type: type[BaseException] | None, exc_val: BaseException | None, exc_tb: TracebackType | None)` → *bool* | *None*

`send(msg: EmailMessage)` → *Any*

Send *msg* or raise an exception if that's not possible

In addition to the base protocol, outgoing's built-in senders are *reentrant* and *reusable* as context managers, and their `send()` methods can be called outside of a context.

## 2.3 Exceptions

**exception** outgoing.**Error**

Bases: *Exception*

The superclass for all exceptions raised by outgoing

**exception** outgoing.**InvalidConfigError**(*details: str*, *configpath: str* | *bytes* | *PathLike[str]* | *PathLike[bytes]* | *None* = *None*)

Bases: *Error*

Raised on encountering an invalid configuration structure

**configpath:** *str* | *bytes* | *PathLike[str]* | *PathLike[bytes]* | *None*

The path to the config file containing the invalid configuration

**details:** *str*

A message about the error

**exception** outgoing.**InvalidPasswordError**(*details: str*, *configpath: str* | *bytes* | *PathLike[str]* | *PathLike[bytes]* | *None* = *None*)

Bases: *InvalidConfigError*

Raised on encountering an invalid password specifier or when no password can be determined from a specifier

**exception** outgoing.**MissingConfigError**(*configpaths: Sequence[str* | *bytes* | *PathLike[str]* | *PathLike[bytes]]*)

Bases: *Error*

Raised when no configuration section can be found in any config files

**configpaths:** *list*[*str* | *bytes* | *PathLike[str]* | *PathLike[bytes]]*

Paths to the configfiles searched for configuration

**exception** `outgoing.NetrcLookupError`

Bases: *Error*

Raised by `lookup_netrc()` on failure to find a match in a netrc file

**exception** `outgoing.UnsupportedEmailError`

Bases: *Error*

Raised by sender objects when asked to send an e-mail that uses features or constructs that the sending method does not support





## COMMAND-LINE PROGRAM

```
outgoing [<options>] [<msg-file> ...]
```

You can use **outgoing** to send fully-composed e-mails directly from the command line with the **outgoing** command. Save your e-mail as a complete *message/rfc822* document and then run **outgoing path/to/email/file** to send it using the configuration in the default config file (or specify another config file with the **--config** option). Multiple files can be passed to the command at once to send multiple e-mails. If no files are specified on the command line, the command reads an e-mail from standard input.

### 3.1 Options

**-c <file>, --config <file>**

Specify a *configuration file* to use instead of the default configuration file

**-E <file>, --env <file>**

Added in version 0.2.0.

Load environment variables from the given *.env* file before reading the configuration file. By default, environment variables are loaded from the first file named “.env” found by searching from the current directory upwards.

**-l <level>, --log-level <level>**

Added in version 0.2.0.

Set the *logging level* to the given value; default: INFO. The level can be given as a case-insensitive level name or as a numeric value.

**-s <key>, --section <key>**

Added in version 0.2.0.

Read the configuration from the given table or key in the configuration file; defaults to “outgoing”

**--no-section**

Added in version 0.2.0.

Read the configuration fields from the top level of the configuration file instead of expecting them to all be contained below a certain table/key



## AVAILABLE EXTENSIONS

It is possible to write packages for extending `outgoing` with support for further sending methods and password schemes. See *Writing Extensions* for how to do so.

If you develop an extension package, please submit a PR so it can be listed on this page!

### 4.1 Sending Methods

- `outgoing-mailgun` — Supports sending e-mail via [Mailgun](#)

### 4.2 Password Schemes

None yet. Be the first!



## WRITING EXTENSIONS

### 5.1 Writing Sending Methods

A sending method is implemented as a callable (usually a class) that accepts the fields of a configuration structure as keyword arguments and returns a *sender object*. The keyword arguments include the `method` field and also include a `configpath` key specifying a `pathlib.Path` pointing to the configuration file (or `None` if `from_dict()` was called without setting a `configpath`). Callables should accept any keyword argument and ignore any that they do not recognize.

For example, given the following configuration:

```
[outgoing]
method = "foobar"
server = "www.example.nil"
password = { env = "SECRET_TOKEN" }
comment = "I like e-mail!"
```

the callable registered for the “foobar” method will be called with the following keyword arguments:

```
**{
    "method": "foobar",
    "server": "www.example.nil",
    "password": {"env": "SECRET_TOKEN"},
    "comment": "I like e-mail!",
    "configpath": Path("path/to/configfile"),
}
```

If the configuration passed to a callable is invalid, the callable should raise an *InvalidConfigError*.

Callables can resolve password fields by passing them to `resolve_password()` or by using pydantic and the *Password* type. Callables should resolve paths relative to the directory containing `configpath` by using `resolve_path()` or by using pydantic and the *Path*, *FilePath*, and/or *DirectoryPath* types.

The last step of writing a sending method is to package it in a Python project and declare the callable as an entry point in the `outgoing.senders` entry point group so that users can install & access it. For example, if your project is built using `setuptools`, and the callable is a `FooSender` class in the `foobar.senders` module, and you want it to be usable by setting `method = "foo"`, add the following to your `setup.py`:

```
setup(
    ...
    entry_points={
        "outgoing.senders": [
```

(continues on next page)

(continued from previous page)

```
        "foo = foobar.senders:FooSender",
    ],
    },
    ...
)
```

## 5.2 Writing Password Schemes

A password scheme is implemented as a function that takes the value part of a `password = { scheme = value }` entry as an argument and returns the corresponding password as a `str`. If the function additionally accepts arguments named `host`, `username`, and/or `configpath` (either explicitly or via `**kwargs`), the corresponding values passed to `resolve_password()` will be forwarded to the scheme function.

If the value structure is invalid, or if no password can be found, the function should raise an `InvalidPasswordError`.

The last step of writing a password scheme is to package it in a Python project and declare the function as an entry point in the `outgoing.password_schemes` entry point group so that users can install & access it. For example, if your project is built using `setuptools`, and the function is `foo_scheme()` in the `foobar.passwords` module, and you want it to be usable by writing `password = { foo = some-value }`, add the following to your `setup.py`:

```
setup(
    ...
    entry_points={
        "outgoing.password_schemes": [
            "foo = foobar.passwords:foo_scheme",
        ],
    },
    ...
)
```

## UTILITIES FOR EXTENSION AUTHORS

```
outgoing.lookup_netrc(host: str, username: str | None = None, path: str | bytes | PathLike[str] |
    PathLike[bytes] | None = None) → tuple[str, str]
```

Look up the entry for `host` in the netrc file at `path` (default: `~/.netrc`) and return a pair of the username & password. If `username` is specified and it does not equal the username in the file, a `NetrcLookupError` is raised.

### Raises

- `NetrcLookupError` – if no entry for `host` or the default entry is present in the netrc file; or if `username` differs from the username in the netrc file
- `netrc.NetrcParseError` – if the `netrc` module encounters an error

```
outgoing.resolve_password(password: str | Mapping[str, Any], host: str | None = None, username: str | None
    = None, configpath: str | Path | None = None) → str
```

Resolve a configuration password value. If `password` is a string, it is returned unchanged. Otherwise, it must be a mapping with exactly one element; the key is used as the name of the password scheme, and the value is passed to the corresponding function for retrieving the password.

When resolving a password field in an `outgoing` configuration structure, the `configpath` and any `host/service` or `username` values from the configuration (or `host/service/username` constants specific to the sending method) should be passed into this function so that they can be made available to any password scheme functions that need them.

### Raises

`InvalidPasswordError` – if password is invalid or cannot be resolved

```
outgoing.resolve_path(path: str | bytes | PathLike[str] | PathLike[bytes], basepath: str | bytes | PathLike[str] |
    PathLike[bytes] | None = None) → Path
```

Convert a path to a `pathlib.Path` instance and resolve it using the same rules for as paths in outgoing configuration files: expand tildes by calling `Path.expanduser()`, prepend the parent directory of `basepath` (usually the value of `configpath`) to the path if given, and then resolve the resulting path to make it absolute.

### Parameters

- `path` (*path*) – the path to resolve
- `basepath` (*path*) – an optional path to resolve `path` relative to

### Return type

`pathlib.Path`

```
class outgoing.OpenClosable
```

Bases: `ABC`, `BaseModel`

An abstract base class for creating `reentrant` context managers. A concrete subclass must define `open()` and `close()` methods; `OpenClosable` will then define `__enter__` and `__exit__` methods that keep track of the depth of nested `with` statements, calling `open()` and `close()` only when entering & exiting the outermost `with`.

**abstract** `close()` → `None`

**model\_computed\_fields:** `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding `ComputedFieldInfo` objects.

**abstract** `open()` → `None`

## 6.1 Pydantic Types & Models

The senders built into `outgoing` make heavy use of `pydantic` for validating & processing configuration, and some of the custom types & models used are also of general interest to anyone writing an `outgoing` extension that also uses `pydantic`.

**class** `outgoing.Path`

Converts its input to `pathlib.Path` instances, including expanding tildes. If there is a field named `configpath` declared before the `Path` field and its value is non-`None`, then the value of the `Path` field will be resolved relative to the parent directory of the `configpath` field; otherwise, it will be resolved relative to the current directory.

**class** `outgoing.FilePath`

Like `Path`, but the path must exist and be a file

**class** `outgoing.DirectoryPath`

Like `Path`, but the path must exist and be a directory

**class** `outgoing.Password`

A subclass of `pydantic.SecretStr` that accepts `outgoing` password specifiers as input and automatically resolves them using `resolve_password()`. `Host`, `username`, and `configpath` values are passed to `resolve_password()` as follows:

- If `Password` is subclassed and given a `host` class variable naming a field, and if the subclass is then used in a model where a field with that name is declared before the `Password` subclass field, then when the model is instantiated, the value of the named field will be passed as the `host` argument to `resolve_password()`. (If the named field is not present on the model that uses the subclass, the `Password` field will fail validation.)
- Alternatively, `Password` can be subclassed with `host` set to a class callable (a classmethod or staticmethod), and when that subclass is used in a model being instantiated, the callable will be passed a `dict` of all validated fields declared before the password field; the return value from the callable will then be passed as the `host` argument to `resolve_password()`. (If the callable raises an exception, the `Password` field will fail validation.)
- If `Password` is used in a model without being subclassed, or if `host` is not defined in a subclass, then `None` will be passed as the `host` argument to `resolve_password()`.
- The `username` argument to `resolve_password()` can likewise be defined by subclassing `Password` and defining `username` appropriately.
- If there is a field named `configpath` declared before the `Password` field, then the value of `configpath` is passed to `resolve_password()`.

For example, if writing a `pydantic` model for a sender configuration where the host-analogue value is passed in a field named `"service"` and for which the username is always `"__token__"`, you would subclass `Password` like this:



```
class MyPassword(outgoing.Password):
    host = "service"

    @staticmethod
    def username(values: dict[str, Any]) -> str:
        return "__token__"
```

and then use it in your model like so:

```
class MySender(pydantic.BaseModel):
    configpath: Optional[outgoing.Path] = None
    service: str
    password: MyPassword # Must come after `configpath` and `service`!
    # ... other fields ...
```

Then, when `MySender` is instantiated, the input to the password field would be automatically resolved by doing (effectively):

```
my_sender.password = pydantic.SecretStr(
    resolve_password(
        my_sender.password,
        host=my_sender.service,
        username="__token__",
        configpath=my_sender.configpath,
    )
)
```

**Note:** As this is a subclass of `pydantic.SecretStr`, the value of a `Password` field is retrieved by calling its `get_secret_value()` method.

### class outgoing.StandardPassword

A subclass of `Password` in which `host` is set to "host" and `username` is set to "username"

### class outgoing.NetrcConfig

A pydantic model usable as a base class for any senders that wish to support both password fields and netrc files. The model accepts the fields `configpath`, `netrc` (a boolean or a file path; defaults to `False`), `host` (required), `username` (optional), and `password` (optional). When the model is instantiated, if `password` is `None` but `netrc` is true or a filepath, the entry for `host` is looked up in `~/.netrc` or the given file, and the `username` and `password` fields are set to the values found.

The model will raise a validation error if any of the following are true:

- `password` is set but `netrc` is true
- `password` is set but `username` is not set
- `username` is set but `password` is not set and `netrc` is false
- `netrc` is true or a filepath, `username` is non-`None`, and the `username` in the netrc file differs from `username`
- `netrc` is true or a filepath and no entry can be found in the netrc file

`configpath:` `Path` | `None`

`host:` `str`

```
model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}
```

A dictionary of computed field names and their corresponding ComputedFieldInfo objects.

```
netrc: pydantic.StrictBool | FilePath
```

```
password: StandardPassword | None
```

```
username: str | None
```

## CHANGELOG

### 7.1 v0.6.1 (in development)

- Support platformdirs v4.0
- Migrated from setuptools to hatch

### 7.2 v0.6.0 (2023-10-30)

- Support python-dotenv v1.0
- Always read JSON configuration files using UTF-8 encoding
- Always read files for the “file” password method using UTF-8 encoding
- Support Python 3.12
- Correct the default Linux config file location listed in the README
- Update pydantic to v2.0

### 7.3 v0.5.0 (2023-02-09)

- Update platformdirs dependency to v3. This is a **breaking** change on macOS, where the default configuration path changes from `~/Library/Preferences/outgoing/outgoing.toml` back to `~/Library/Application Support/outgoing/outgoing.toml`.

### 7.4 v0.4.0 (2022-10-25)

- Drop support for Python 3.6
- Support Python 3.11
- Use `tomllib` on Python 3.11

## 7.5 v0.3.2 (2022-09-03)

- Overload `Password.__eq__` so that instances continue to compare equal to `pydantic.SecretStr` instances even under `pydantic 1.10`

## 7.6 v0.3.1 (2022-01-02)

- Support `tomli 2.0`

## 7.7 v0.3.0 (2021-10-31)

- Support Python 3.10
- Replaced `entrypoints` dependency with `importlib-metadata`
- Replaced `appdirs` dependency with `platformdirs`. This is a **breaking** change on macOS, where the default configuration path changes from `~/Library/Application Support/outgoing/outgoing.toml` to `~/Library/Preferences/outgoing/outgoing.toml`.

## 7.8 v0.2.5 (2021-09-27)

- `outgoing.errors.UnsupportedEmailError` is now re-exported as `outgoing.UnsupportedEmailError` like all the other exception classes

## 7.9 v0.2.4 (2021-08-02)

- Update for `tomli 1.2.0`

## 7.10 v0.2.3 (2021-07-04)

- Read TOML files in UTF-8

## 7.11 v0.2.2 (2021-07-02)

- Switch from `toml` to `tomli`

## 7.12 v0.2.1 (2021-05-12)

- Support Click 8

## 7.13 v0.2.0 (2021-03-14)

- Require the port field of SMTPSender to be non-negative
- Mark *Sender* as runtime\_checkable and export it
- Gave the **outgoing** command --section, --no-section, and --log-level options
- Added logging to built-in sender classes
- The **outgoing** command now loads settings from .env files and has an --env option

## 7.14 v0.1.0 (2021-03-06)

Initial release

outgoing provides a common interface to multiple different e-mail sending methods (SMTP, sendmail, mbox, etc.). Just construct a sender from a configuration file or object, pass it an *EmailMessage* instance, and let the magical internet daemons take care of the rest.

outgoing itself provides support for only basic sending methods; additional methods are provided by *extension packages*.



## INSTALLATION

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

```
python3 -m pip install outgoing
```





## EXAMPLES

A sample configuration file:

```
[outgoing]
method = "smtp"
host = "mx.example.com"
ssl = "starttls"
username = "myname"
password = { file = "~/secrets/smtp-password" }
```

Sending an e-mail based on a configuration file:

```
from email.message import EmailMessage
import outgoing

# Construct an EmailMessage object the standard Python way:
msg = EmailMessage()
msg["Subject"] = "Meet me"
msg["To"] = "my.beloved@love.love"
msg["From"] = "me@here.qq"
msg.set_content(
    "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"
)

# Construct a sender object based on the default config file (assuming it's
# populated)
with outgoing.from_config_file() as sender:
    # Now send that letter!
    sender.send(msg)
```

As an alternative to using a configuration file, you can specify an explicit configuration by passing the configuration structure to the `outgoing.from_dict()` method, like so:

```
from email.message import EmailMessage
import outgoing
```

(continues on next page)

(continued from previous page)

```
# Construct an EmailMessage object using the eletter library
# <https://github.com/jwodder/eletter>:
from eletter import compose

msg1 = compose(
    subject="No.",
    to=["me@here.qq"],
    from_="my.beloved@love.love",
    text=(
        "Hot pockets?  Thou disgusteth me.\n"
        "\n"
        "Pineapple pizza or RIOT.\n"
    ),
)

msg2 = compose(
    subject="I'd like to place an order.",
    to=["pete@za.aa"],
    from_="my.beloved@love.love",
    text="I need the usual.  Twelve Hawaiian Abominations to go, please.\n",
)

SENDING_CONFIG = {
    "method": "smtp",
    "host": "smtp.love.love",
    "username": "my.beloved",
    "password": {"env": "SMTP_PASSWORD"},
    "ssl": "starttls",
}

with outgoing.from_dict(SENDING_CONFIG) as sender:
    sender.send(msg1)
    sender.send(msg2)
```

## INDICES AND TABLES

- `genindex`
- `search`



## PYTHON MODULE INDEX

### O

outgoing, ??



## Symbols

[\\_\\_enter\\_\\_\(\) \(outgoing.Sender method\), 10](#)  
[\\_\\_exit\\_\\_\(\) \(outgoing.Sender method\), 10](#)  
[-E](#)  
     outgoing command line option, [13](#)  
[--config](#)  
     outgoing command line option, [13](#)  
[--env](#)  
     outgoing command line option, [13](#)  
[--log-level](#)  
     outgoing command line option, [13](#)  
[--no-section](#)  
     outgoing command line option, [13](#)  
[--section](#)  
     outgoing command line option, [13](#)  
[-c](#)  
     outgoing command line option, [13](#)  
[-l](#)  
     outgoing command line option, [13](#)  
[-s](#)  
     outgoing command line option, [13](#)

## C

[close\(\) \(outgoing.OpenClosable method\), 20](#)  
[configpath \(outgoing.InvalidConfigError attribute\), 10](#)  
[configpath \(outgoing.NetrcConfig attribute\), 21](#)  
[configpaths \(outgoing.MissingConfigError attribute\), 10](#)

## D

[details \(outgoing.InvalidConfigError attribute\), 10](#)  
[DirectoryPath \(class in outgoing\), 20](#)

## E

[Error, 10](#)

## F

[FilePath \(class in outgoing\), 20](#)  
[from\\_config\\_file\(\) \(in module outgoing\), 9](#)  
[from\\_dict\(\) \(in module outgoing\), 9](#)

## G

[get\\_default\\_configpath\(\) \(in module outgoing\), 9](#)

## H

[host \(outgoing.NetrcConfig attribute\), 21](#)

## I

[InvalidConfigError, 10](#)  
[InvalidPasswordError, 10](#)

## L

[lookup\\_netrc\(\) \(in module outgoing\), 19](#)

## M

[MissingConfigError, 10](#)  
[model\\_computed\\_fields \(outgoing.NetrcConfig attribute\), 21](#)  
[model\\_computed\\_fields \(outgoing.OpenClosable attribute\), 20](#)  
[module](#)  
     outgoing, [1](#)

## N

[netrc \(outgoing.NetrcConfig attribute\), 22](#)  
[NetrcConfig \(class in outgoing\), 21](#)  
[NetrcLookupError, 10](#)

## O

[open\(\) \(outgoing.OpenClosable method\), 20](#)  
[OpenClosable \(class in outgoing\), 19](#)  
[outgoing](#)  
     module, [1](#)  
[outgoing \(command\), 11](#)  
[outgoing command line option](#)  
     [-E, 13](#)  
     [--config, 13](#)  
     [--env, 13](#)  
     [--log-level, 13](#)  
     [--no-section, 13](#)  
     [--section, 13](#)  
     [-c, 13](#)

-l, [13](#)  
-s, [13](#)

## P

Password (*class in outgoing*), [20](#)

password (*outgoing.NetrcConfig attribute*), [22](#)

Path (*class in outgoing*), [20](#)

## R

resolve\_password() (*in module outgoing*), [19](#)

resolve\_path() (*in module outgoing*), [19](#)

## S

send() (*outgoing.Sender method*), [10](#)

Sender (*class in outgoing*), [10](#)

StandardPassword (*class in outgoing*), [21](#)

## U

UnsupportedEmailError, [11](#)

username (*outgoing.NetrcConfig attribute*), [22](#)