imaplib2

Release 3.06

Piers Lauder

Dec 10, 2022

CONTENTS:

CHAPTER

ONE

LIBRARY REFERENCE

imaplib2.**version**(*use_tuple=False*)

Return the version of this module, either as a single string (the default) or a `tuple` of `major_version`, `minor_version` as integers

Parameters
use_tuple (`bool`) – Whether to return `tuple`

Returns
The version of this library, the format depends on the value of `use_tuple`

Return type
(`int`, `int`) if `use_tuple`

Return type
`str` otherwise

This module defines a class, `IMAP4`, which encapsulates a threaded connection to an IMAP4 server and implements the IMAP4rev1 client protocol as defined in RFC 3501 with several extensions. This module presents an almost identical API as that provided by the standard python library module `imaplib`, the main difference being that this version allows parallel execution of commands on the IMAP4 server, and implements the IMAP4rev1 `IDLE` extension. (`imaplib2` can be substituted for `imaplib` in existing clients with no changes in the code, but see the *caveat* below.)

An `IMAP4` instance is instantiated with an optional `host` and/or `port`. The defaults are `localhost` and `143` - the standard IMAP4 port number.

There are also five other optional arguments: `debug=level`, `debug_file=file`, `identifier=string`, `timeout=seconds`, `debug_buf_lvl=level`. Setting `debug` *level* (default: 0) to anything above `debug_buf_lvl` (default: 3) causes every action to be printed to *file* (default: *sys.stderr*). Otherwise actions are logged in a circular buffer and the last 20 printed on errors. The third argument provides a string to be prepended to thread names - useful during debugging (default: target host). The forth argument sets a timeout for responses from the server, after which the instance will abort. Note that this timeout is overridden by an IDLE timeout when active.

Caveat: Once an instance has been created, the invoker must call the `logout` method before discarding it, to shut down the threads.

There are two classes derived from `IMAP4` which provide alternate transport mechanisms:

IMAP4_SSL
IMAP4 client class over an SSL connection.

IMAP4_stream
IMAP4 client class over a stream.

There are also 2 utility methods provided for processing IMAP4 date strings:

Internaldate2Time(*datestr*)
Converts an IMAP4 `INTERNALDATE` string to Universal Time. Returns a `time` module tuple.

Time2Internaldate(*date_time*)
Converts `date_time` (a `time` module tuple, or an integer or float seconds) to an IMAP4 `INTERNALDATE` representation. Returns a string in the form: `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (including double-quotes).

And there is one utility method for parsing IMAP4 `FLAGS` responses:

ParseFlags(*response*)
Convert an IMAP4 flags response (a string of the form `"...FLAGS (flag ...)"`) to a python tuple..

CHAPTER

TWO

IMAP4 OBJECTS

All IMAP4rev1 commands are represented by methods of the same name

Each command returns a tuple: `(type, [data, ...])` where `type` is usually `'OK'` or `'NO'`, and `data` is either the text from the command response (always true when `type` is `'NO'`), or mandated results from the command. Each `data` is either a string, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: *literal* value).

Any logical errors raise the exception class `<instance>.error("<reason>")`. IMAP4 server errors raise `<instance>.abort("<reason>")`, which is a sub-class of `error`. Mailbox status changes from `READ-WRITE` to `READ-ONLY` raise `<instance>.readonly("<reason>")`, which is a sub-class of `abort`. Note that closing the instance and instantiating a new one will usually recover from an `abort`.

All commands take two optional named arguments: `callback` and `cb_arg`. If `callback` is provided then the command is asynchronous (the IMAP4 command is scheduled, and the call returns immediately), and the result will be posted by invoking `callback` with a single argument:

or, if there was a problem:

Otherwise the command is synchronous (waits for result). But note that state-changing commands will both block until previous commands have completed, and block subsequent commands until they have finished.

All (non-callback) arguments to commands are converted to strings, except for `authenticate`, and the last argument to `append` which is passed as an IMAP4 literal. If necessary (the string contains any non-printing characters or white-space and isn't enclosed with either parentheses or double quotes or single quotes) each string is quoted. However, the *password* argument to the `login` command is always quoted.

If you want to avoid having an argument string quoted (eg: the *flags* argument to `store`) then enclose the string in parentheses (eg: `(\Deleted)`). If you are using *sequence sets* containing the wildcard character '*', then enclose the argument in single quotes: the quotes will be removed and the resulting string passed unquoted.

To summarise the quoting rules:

- a string is automatically quoted if it contains at least one of the IMAP4 *atom-special* characters with the following exceptions:
- the password argument to the `login` command is always quoted;
- a string enclosed in `"..."` or `(...)` is passed as is;
- a string enclosed in `'...'` is stripped of the enclosing single quotes and the rest passed as is.

Note also that you can pass in an argument with a type that doesn't evaluate to *basestring* (eg: `bytearray`) and it will be converted to a string without quoting.

There is one instance variable, `state`, that is useful for tracking whether the client needs to login to the server. If it has the value `"AUTH"` after instantiating the class, then the connection is pre-authenticated (otherwise it will be `"NONAUTH"`). Selecting a mailbox changes the state to be `"SELECTED"`, closing a mailbox changes back to `"AUTH"`, and once the client has logged out, the state changes to `"LOGOUT"` and no further commands may be issued.

There is another instance variable, `capabilities`, that holds a list of the capabilities provided by the server (the same as the list returned by the IMAP4 `CAPABILITY` command).

An `IMAP4` instance has the following methods:

append(*mailbox, flags, date_time, message*)
Append message to named mailbox. All args except `message` can be None

authenticate(*mechanism, authobject*)
Authenticate command - requires response processing. *mechanism* specifies which authentication mechanism is to be used - it must appear in `<instance>.capabilities` in the form `AUTH=`*mechanism*. *authobject* must be a callable object:

It will be called to process server continuation responses. It should return data that will be encoded and sent to the server. It should return None if the client abort response * should be sent instead.

capability()
Return server IMAP4 capabilities.

check()
Checkpoint mailbox on server.

close()
Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before LOGOUT.

copy(*message_set, new_mailbox*)
Copy *message_set* messages onto end of *new_mailbox*.

create(*mailbox*)
Create new mailbox.

delete(*mailbox*)
Delete old mailbox.

enable(*capability*)
Send an RFC5161 enable string to the server. EG: ask the server to enable UTF-8 message encoding:

```
if 'ENABLE' in imapobj.capabilities:
    imapobj.enable("UTF8=ACCEPT")
```

enable_compression()
Ask the server to start compressing the connection. Should be called from user of this class after instantiation, as in:

```
if 'COMPRESS=DEFLATE' in imapobj.capabilities:
    imapobj.enable_compression()
```

examine(*mailbox*='INBOX')
Select a mailbox for READ-ONLY access. Flush all untagged responses. Returned *data* is count of messages in mailbox (EXISTS response). Mandated responses are `'FLAGS', 'EXISTS', 'RECENT', 'UIDVALIDITY'`, so other responses should be obtained by calling `response('FLAGS')` etc.

expunge()
Permanently remove deleted items from selected mailbox. Generates an EXPUNGE response for each deleted message. Returned *data* contains a list of EXPUNGE message numbers in order received.

fetch(*message_set, message_parts*)
Fetch (parts of) messages. `message_parts` should be a string of selected parts enclosed in parentheses, eg: `"(UID BODY[TEXT])"`. Returned *data* are tuples of message part envelope and data, followed by a string containing the trailer.

getacl(*mailbox*)
Get the Access Control Lists for a mailbox.

getannotation(*mailbox_name, entry_specifier, attribute_specifier*)
Retrieve ANNOTATIONS.

getquota(*root*)
Get the quota root's resource usage and limits. (Part of the IMAP4 QUOTA extension defined in RFC2087.)

getquotaroot(*mailbox*)
Get the list of quota roots for the named mailbox.

id(*field1, value1, ...*)
IMAP4 ID extension: exchange information for problem analysis and determination. NB: a single argument is assumed to be correctly formatted and is passed through unchanged (for backward compatibility with earlier version). The ID extension is defined in RFC 2971.

idle(*timeout*=None)
Put server into IDLE mode until server notifies some change, or *timeout* (secs) occurs [default: 29 minutes], or another IMAP4 command is scheduled.

list(*directory*='""', *pattern*='*')
List mailbox names in directory matching pattern. Returned *data* is list of LIST responses.

login(*user, password*)
Identify client using plaintext password. The *password* argument will be quoted.

login_cram_md5(*user, password*)
Force use of CRAM-MD5 authentication.

logout()
Shutdown connection to server. Returns server BYE response. NB: You must call this to shut down threads before discarding an instance.

lsub(*directory*='""', *pattern*='*')
List *subscribed* mailbox names in directory matching pattern. Returned *data* are tuples of message part envelope and data.

myrights(*mailbox*)
Show my Access Controll Lists for *mailbox* (i.e. the rights that I have on *mailbox*).

namespace()
Returns IMAP namespaces per RFC2342.

noop()
Send NOOP command.

partial(*message_num, message_part, start, length*)
Fetch truncated part of a message. Returned *data* is tuple of message part envelope and data. NB: obsolete.

pop_untagged_responses()
(Helper method.) Generator for obtaining untagged responses. Returns and removes untagged responses in order of reception. Use at your own risk! (Removing untagged responses required by outstanding commands may cause errors.)

proxyauth(*user*)
Assume authentication as *user*. (Allows an authorised administrator to proxy into any user's mailbox.)

recent()
(Helper method.) Return RECENT responses if any exist, else prompt server for an update using the NOOP command. Returned *data* is None if no new messages, else list of RECENT responses, most recent last.

rename(*oldmailbox, newmailbox*)
Rename old mailbox name to new.

response(*code*)
(Helper method.) Return data for response *code* if received, or None. Response value is cleared. Returns the given *code* in place of the usual *type*.

search(*charset, criterium, ...*)
Search mailbox for matching messages. Returned *data* contains a space separated list of matching message numbers.

select(*mailbox*='INBOX', *readonly*=False)
Select a mailbox. Flush all untagged responses. Returned *data* is count of messages in mailbox (EXISTS response). Mandated responses are `'FLAGS', 'EXISTS', 'RECENT', 'UIDVALIDITY'`, so other responses should be obtained by calling `response('FLAGS')` etc.

setacl(*mailbox, who, what*)
Set the Access Control Lists for a mailbox.

setannotation(*mailbox_name, entry, attribute_value[, entry, attribute_value]**)
Set ANNOTATIONS.

setquota(*root, limits*)
Set the quota root's resource limits.

sort(*sort_criteria, charset, search_criteria, ...*)
IMAP4rev1 extension SORT command.

starttls(*keyfile*, *certfile*, *ca_certs*, *cert_verify_cb*, *ssl_version*="ssl23", *tls_level*="tls_compat")
Start TLS negotiation as per RFC 2595. If non-null, *cert_verify_cb* will be called to verify the server certificate, with peer certificate and hostname as parameters. If *cert_verify_cb* returns a non-null response, an SSL exception will be raised with the response as reason. `starttls` should be called from user of the IMAP4 class after instantiation, as in:

```
if 'STARTTLS' in imapobj.capabilities:
    imapobj.starttls()
```

The recognized values for `tls_level` are:

tls_secure: accept only TLS protocols recognized as "secure" *tls_no_ssl*: disable SSLv2 and SSLv3 support *tls_compat*: accept all SSL/TLS versions

`status`(*mailbox, names*)

Request named status conditions for mailbox.

`store`(*message_set, command, flag_list*)

Alters flag dispositions for messages in mailbox.

subscribe(*mailbox*)

Subscribe to new mailbox.

thread(*threading_algorithm, charset, search_criteria, ...*)

IMAP4rev1 extension THREAD command.

uid(*command, arg, ...*)

Execute `command arg ...` with messages identified by UID, rather than message number. Returns response appropriate to *command*.

unsubscribe(*mailbox*)

Unsubscribe from old mailbox.

xatom(*command, arg, ...*)

Allow simple extension commands as notified by server in CAPABILITY response. Returns response appropriate to *command*.

`IMAP4` instances have a variable, `PROTOCOL_VERSION`, that is set to the most recent supported protocol in the CAPABILITY response.

CHAPTER

THREE

USAGE

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
def cb(cb_arg_list):
    response, cb_arg, error = cb_arg_list
    typ, data = response
    if not data:
        return
    for field in data:
        if type(field) is not tuple:
            continue
        print('Message %s:\n%s\n'
            % (field[0].split()[0], field[1]))

import getpass, imaplib2
M = imaplib2.IMAP4()
M.LOGIN(getpass.getuser(), getpass.getpass())
M.SELECT(readonly=True)
typ, data = M.SEARCH(None, 'ALL')
for num in data[0].split():
    M.FETCH(num, '(RFC822)', callback=cb)
M.CLOSE()
M.LOGOUT()
```

Note that IMAP4 message numbers change as the mailbox changes, so it is highly advisable to use UIDs instead via the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

CHAPTER

FOUR

REFERENCES

Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at http://www.washington.edu/imap.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

V